DESIGN AND IMPLEMENTATION OF A PSEUDO LANGUAGE PROCESSOR

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment of

the requirement for the degree of

Master of Science

---

by

Yu-Ping William Sun

November 1979

ACKNOWLEDGMEMTS

I would like to express my sincere gratitude to Dr. Jayashree Ramanathan, my thesis advisor, who has patiently read the entire manuscript and offered many useful suggestions. Her advice and assistance led to the successful completion of this thesis project. My appreciation is also extended to Dr. J. C. Huang, Dr. Souhail El-Asfouri, and Dr. Liang C. Shen for serving as committee members.

A special thank goes to my wife Huey-Nan for her patient and careful typing of the manuscript. Without her encouragement and understanding, this thesis may never have been completed.

DESIGN AND IMPLEMENTATION OF A PSEUDO LANGUAGE PROCESSOR

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment of

the requirement for the degree of

Master of Science

---

by

Yu-Ping William Sun

November 1979

ABSTRACT

The two objectives in software development are:

- cost reduction, and
- the production of reliable software.

Structured, top-down design is the major technique currently used to achieve these objectives. Pseudo Language (PL) is presented in this thesis as a means for encouraging good design practices and functional programming. A Pseudo Language Processor (PLP) which analyses the PL program structurally is also described in this thesis. PLP is a software tool which enforces good design practices and prints out useful messages for validating programs written in Pseudo Language.

TABLE OF CONTENTS

Chapter 1
INTRODUCTION

Over the past few years, the rapidly increasing cost of software and the need for improving software reliability and maintainability has spurred a search for better methods of software production. Structured programming and top-down design have emerged as practical tools to the problem of developing reliable software systems. It has been observed by various researchers that programmer productivity can be vastly improved if the development of software is split into two equally important phases [RB]

1) The design phase
2) The implementation phase

Some of the recent design language systems are HIPO [S], PDL [CG], SEMI-CODE [C], WELLMADE [B]. These software tools have been suggested for use during the design phase. For instance, Hierarchy plus Input-Process-Output (HIPO) is a procedure for hierarchical functional design by which programming projects can be analysed into system, program, and moudule level. It consists of two basic components: a hierarchy chart, which shows how each function is divided into subfunctions; and a input-process-output chart, which expresses each function in the hierarchy in terms of its input and output. Program Design Language (PDL) can be thought of as "Structured English". Input to the PDL processor consists of control information plus designs for procedures. The output is a working design document which can be photoreduced and included in a project development workbook. SEMI-CODE is a means of describing software using English-like sentences. In other words, by using the notation of verbs and nouns in a top-down refinement of a sequence of English-like sentences which can be used to describe functions. This sequence of sentences is iteratively refined so that the verbs and nouns initially used could become an implemented program in a specific language. Finally, WELLMADE is a software design discipline which is based upon constructive approach and which is intended to be applicable to practical software development. It currently addresses the task of deriving provably correct programs from the functional specifications. The main theme of this methodology is a constrained and controlled process of designing software by constructing a correct design from careful considerations of its functional requirements.

All the design tools mentioned above - HIPO, PDL, SEMI-CODE, and WELLMADE - concentrate on the control logic of the programming task, and also have some structure in its design. But, not enough syntactic structure exists in programs, designed using these tools, so that they can not be extensively analysed by a program analyser.

Recent research by Ramanathan and Blattner [RB] introduces a Pseudo Language (PL) and a Pseudo Language Processor (PLP) - a translator which examines source programs in PL and provides a listing of these programs together with a variety of messages (like <u>symbol cross reference table</u>, <u>path expression</u>, and <u>possible path expression anomalies</u>). These messages can be used by the programmer both during the design and implementation phase. PL is to enforce structured programming, and it resembles Pidgin English therefore very readable. It is easy to program in PL since the programmer can ignore the messy details necessitated in actual implementation languages like FORTRAN, PL/I, COBOL, PASCAL etc. The programmer can concentrate on the logic of design. Another important characteristic of PL is that it requires the programmer to explicitly identify the functional components of the programming task at hand. A PL program can serve as the documentation for the implementation version of the program. A PL program also provides communications among the programmers in a team and contributes towards increased programmer productivity.

The PLP is based on the philosophy that it is the cheapest to correct errors during the earlier stages of program development. This is because of the fact that fewer corrective changes have to be made to debug a design program. In order for the PLP to provide the messages which can indicate errors in the PL program and list the functional components in the program, the PL program must have a recognizable structure. The fundamental components of a PL program are:

- nouns and their descriptions
- assignments
- commands
- control structures

A simple example of PL is shown in figure 1. It restricts the English sentences to be 'commands'. This restriction forces the programmer to identify the functional components of the programming problem. The PLP also performs extensive static analysis using a technique which is based on a symbolic and structural analysis of the source PL program. This analysis is used to print out messages that can be used by the programmer for validating and debugging the program [RB].

This thesis presents development and implementation steps of Pseudo Language and Pseudo Language Processor.

The author

- developed a context-free grammar to define the syntax of Pseudo Language,
- designed a scanner to recognize the input string (source program),
- generated a semantics to provide some useful messages (symbol cross reference table, path expression) for validating the source program, and
- designed a analyser to detect the possible data flow anomalies for the path expression resulting from the semantics.

The detailed discussions and examples are presented in the following chapters of this thesis. Chapter 2 presents some background materials - data flow analysis, context-free grammar, attribute grammar and so on. Chapter 3 introduces the concepts of PL and PLP. Finally, Chapter 4 presents the detailed development of this processor.

Figure 1 - EXAMPLE OF A PL SORT PROGRAM

```
BEGIN_INTRO
   PL PROGRAM FOR SORT ;
DICTIONARY
   SIZE_OF_TABLE, I          : INTEGER ;
   FIRST_ITEM, SECOND_ITEM   : POINTER ;
   FLAG_OF_CHANGE            : FLAG INITIAL 1 ;
   TABLE                     : ARRAY ;
END_INTRO
BEGIN
   IF          SIZE_OF_TABLE > 1    THEN
          WHILE      FLAG_OF_CHANGE = 1
          DO         FLAG_OF_CHANGE = 0 ;
                     FOR  I = 1 TO SIZE_OF_TABLE
                     DO   GET FIRST_ITEM IN TABLE ;
                          GET SECOND_ITEM IN TABLE ;
                          IF  FIRST_ITEM > SECOND_ITEM THEN
                          BEGIN
                            INTERCHANGE FIRST_ITEM AND
                            SECOND_ITEM ;
                            FLAG_OF_CHANGE = 1 ;
                         END ;
                     END;
           END ;
           PRINT TABLE ;
END ;
```

Chapter
BACKGROUND AND DEFINITIONS

The Pseudo Language, the Pseudo Language Processor, and some formal definitions are presented in this chapter.

2.1 Data Flow Analysis

As is usual, the control structure of a program will be modeled by a flow graph composed of nodes and edges. Each "collapsed" node is either a simple statement or a sequence of simple statements or expressions, S1, S2, S3,... S_n, such that the statements or expressions are all executed before any branch can be taken. Each edge in the flow graph models a possible transfer of control. A flow graph G is a triple G = (N,P,n_0) where

1) N is a finite set of nodes,
2) P, a subset of N x N, is a finite set of edges, and
3) n_0 is a unique program start node from which there is a "path" to every other node in the graph.

A sequence of nodes X1, X2,... X_k is a path of length K if $(X_i, X_{i+1}) \in E$ for $1 \leq i < k$. A path P = X1, X2,...X_k is an execution path if $X_1 = n_0$ and if all the nodes X1....X_k are executed in order. Note that a path in a flow graph may not necessarily be an execution path. The translation described in this thesis is based on local attributes associated with each node of the flow graph. More precisely, an attributed flow graph has a set of attributes S(X) associated with each node $X \in N$.

This thesis addresses the problem of generating a program analyzer for detecting a specific class of errors known as data flow anomalies. For the detection of anomalies, the local attributes in the set S(X) are:

- DEF(X): the set of variable(s) defined by the node X,
- REF(X): the set of variable(s) referenced by the node X.
- UNDEF(X): the set of all variable(s) in the program which initially have the undefined attributes associated with them.

A path $P = X_1, X_2, \ldots X_k$ of a flow graph G is said to have an anomaly with respect to variable A if the variable is imporperly used on the path, such as:

- $A \in \text{UNDEF}(X_g)$ and $A \in \text{REF}(X_i)$ and $A \notin \text{DEF}(X_j)$, $1 \leq g < j < i \leq k$. This is called an Undefine-Reference (UR) anomaly, i.e., the variable A is referenced but never been given a value, that is defined, at previous node(s).
- $A \in \text{DEF}(X_i)$ and $A \in \text{DEF}(X_j)$ and $A \notin \text{REF}(X_h)$, $1 < i < h < j < k$. This is called Define-Define (DD) anomaly, i.e., the

variable A was defined at node X_i and then node X_j without an intervening reference ($A \notin REF(X_h)$, $1<i<h<j\leq k$).

The following partial PL program will be used to illustrate the two anomalies mentioned above.

```
1) BEGIN
2) Z = 1 ;
3) SET X TO ZERO ;
4) IF FLAG  .EQ. 1 THEN DO
5)    Z =  Y + 3 ;
6)    OD ;
7) END ;
```

Each line in this program represents a statement. Note that the variable "FLAG" was used in line 4 to be a component of relational expression. Obviously, "FLAG" was referenced but never previously defined, i.e., a UR anomaly occurs in the program. If statement 5 was executed, then a DD anomaly would be caused by the definition of variable "Z" at line 2 and line 5 without an intervening reference.

There are several other types of anomalies such as defined but never used, defined but never declared and so forth. Some of these anomalies are dependent on the language specifications. As an example, a variable need not be declared in FORTRAN but not in PL/I or PASCAL. Work done in this thesis analyses Pseudo Language programs to detect the DD, UR, UU(declared and declared again) and defined but never referenced anomalies. The model for the PLP component which analyses the structure of input PL programs is a grammar.

2.2 Context Free Grammar (CFG)

A context free grammar (CFG) is a four-tuple (VN, VT, P,S) where:

VN: a finite set of non-terminal symbols,

VT: a finite set of terminal symbols,

P : a finite set of productions and,

S : a unique start symbol.

A production (rewriting rule) $p \in P$ is written as $p = X0 \rightarrow X_1, X_2, \ldots X_{np}$ where $n_p \geq 1$, $X0 \in VN$ and $X_k \in VN \cup VT$ for $1 \leq k \leq n_p$. The start symbol appears only on the left-hand side of the zeroth production. We say that W is a direct derivation of V ($V \rightarrow W$), if we can write V = xAy, W = xay for some string x and y, where $A \rightarrow a$ is a production in P. An example of derivation sequence is given below:

Let G be a CFG, where

G = (VN, VT, P,S) such that

VN = (P, E)

VT = (id)

S = (PROGRAM)

```
P : 1) PROGRAM → P
    2) P       → E
    3) E       → E + E
    4) E       → E * E
    5) E       → id
```

The derivation sequence PROGRAM → P → E → E+E → id+E → id+ E*E → id+E*id → id+id*id shows that "PROGRAM" derives id+id*id. Note that as long as there is a non-terminal in a string, one can derive a new string from it. However, a graphical representation can be used to describe the derivation sequence. This representation is called the parse tree. A parse tree of the grammar is a finite, ordered tree with its nodes appropriately indexed. Each interior node of the parse tree is labeled using symbols from VN and the leaf nodes labeled using symbols from VT. If the symbol X_0 labels node n and the labels of the immediate descendants of node n are X_1, X_2,... X_{np}, then we say the production P applies at node n. A production applies at each interior node of the parse tree. A parse tree can have more than one associated derivation. More detailed description of this problem will be given in Chapter 3. Static attributes maybe attached to the parse tree nodes by associating attributes to symbols in VN U VT. More details about attributes are described below.

2.3 Attribute Grammar

An attribute grammar is an ordinary CFG augmented with attributes and semantic functions. These attributes are associated with the various productions of $P \in G$ and may be attached to the parse tree nodes of a program.

The semantics for an attribute grammar are functions which are executed as productions are applied. These functions calculate attribute value for the nodes associated with the production.

Static Attributes

For each $X \in VN \cup VT$, there are disjoint finite sets $S(X)$ of synthesized attributes. The attributes of a symbol X identify the various components of its "meaning". A symbol X may occur more than once in a production and hence an attribute of X may have more than one realization in the same production. A production $P = X \rightarrow X_1, X_2, \ldots X_{np}$ has the attribute occurrence (a,k) if $a \in A$ (Xk) for $0 \leq k \leq np$. Attribute occurrences are to be thought of as variables which are used in writing the semantics for a production. The synthesized attributes pass information up the tree toward the root. The value of a terminal symbol's synthesized attributes are given initially. Usually, in a compiler, this is the job of the scanner. The term "attribute" is often used ambiguously to mean some a $\in$ S(X), as in "an attribute of a nonterminal;" to mean some occurrence (a,k), as in "an attribute of a production;" or to mean a value attached to the parse tree, as in "an attribute of a node." It should always be clear from the context which sense is intended.

Semantic Functions

For each production p $\in$ P, there is a set F(p) of semantic functions as follows: for every synthesized occurrence (a,k) with k 0, 1, 2, 3,...np there is a semantic function $f^p(a,k) \in F(p)$ mapping certain other attribute occurrences of p into a value for (a,k). The dependency set of $f^p(a,k)$ is denoted $D^p(a,k)$ and contains those attribute occurrences of p used in the semantic functions specify the meanings of parse trees locally, in terms of only a node and its immediate descendants. The entire set of semantic functions for the grammar denoted $F = \bigcup_{p \in P} F(p)$.

Example 2.3a is a CFG which illustrates how to define the meaning of signed binary integers. Figure 2.3b shows the typical parse tree for the binary integer -101. The CFG in Example 2.3c has been extended to an attribute grammar for signed binary integers. It may be noted that the notation "A.b" stands for the "b" attribute of non-terminal "A". We have realized that the meaning of a binary integer is the numerical value it represents.

Example 2.3a - DEFINITION OF SIGNED BINARY INTEGERS

Let G be a CFG where

G = (VN, VT, P, N) such that

VN = (NUMBER, SIGN, BINARY-STRING, BIT)

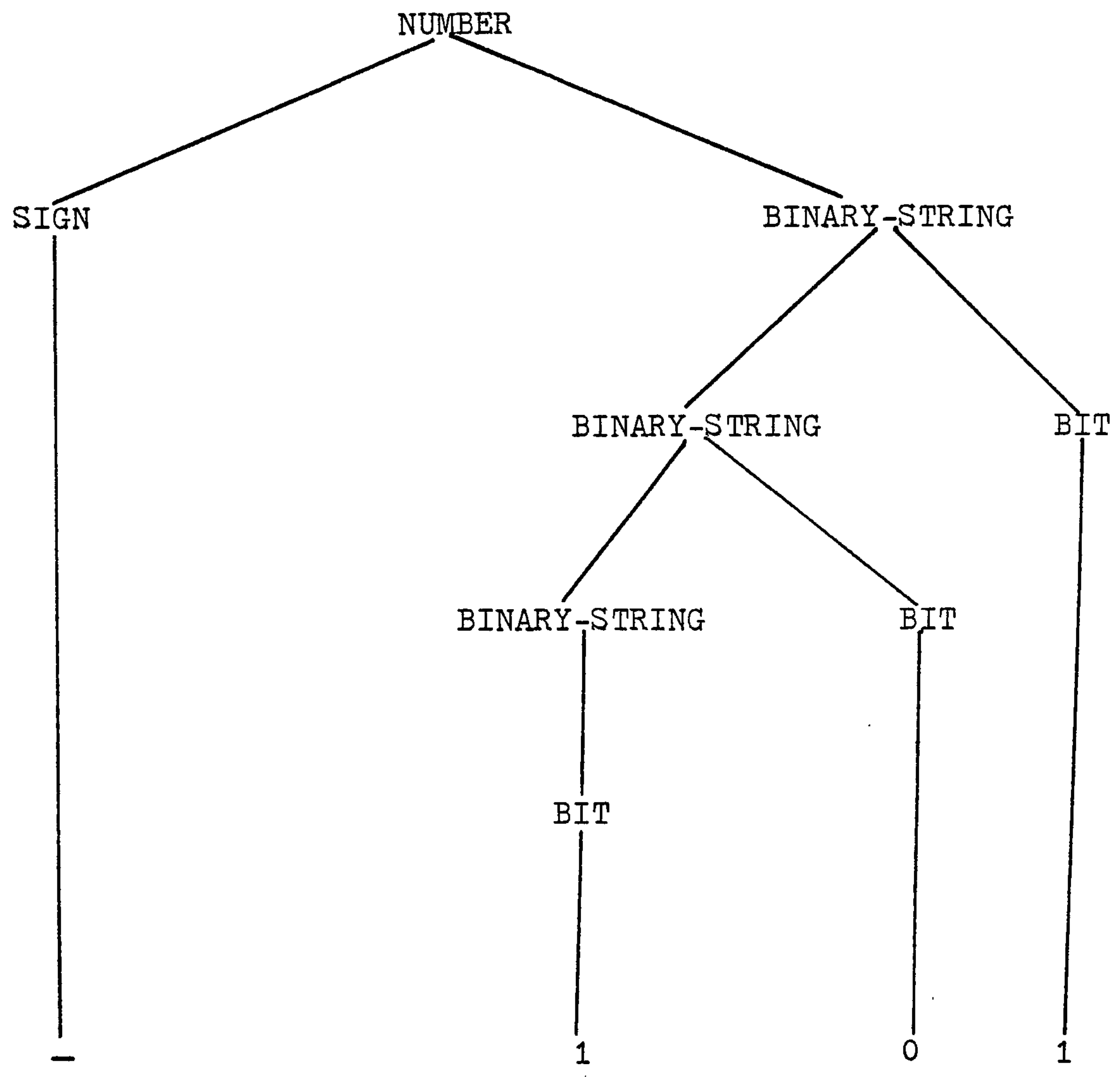
VT = (+, -, 0, 1)

START SYMBOL : NUMBER

PRODUCTIONS :

```
        1) NUMBER         → SIGN, BINARY-STRING
        2) BINARY-STRING  → BINARY-STRING, BIT
        3)                → BIT
        4) BIT            → 1
        5)                → 0
        6) SIGN           → +
        7)                → -
```

Figure 2.3b - TREE STRUCTURE FOR -101

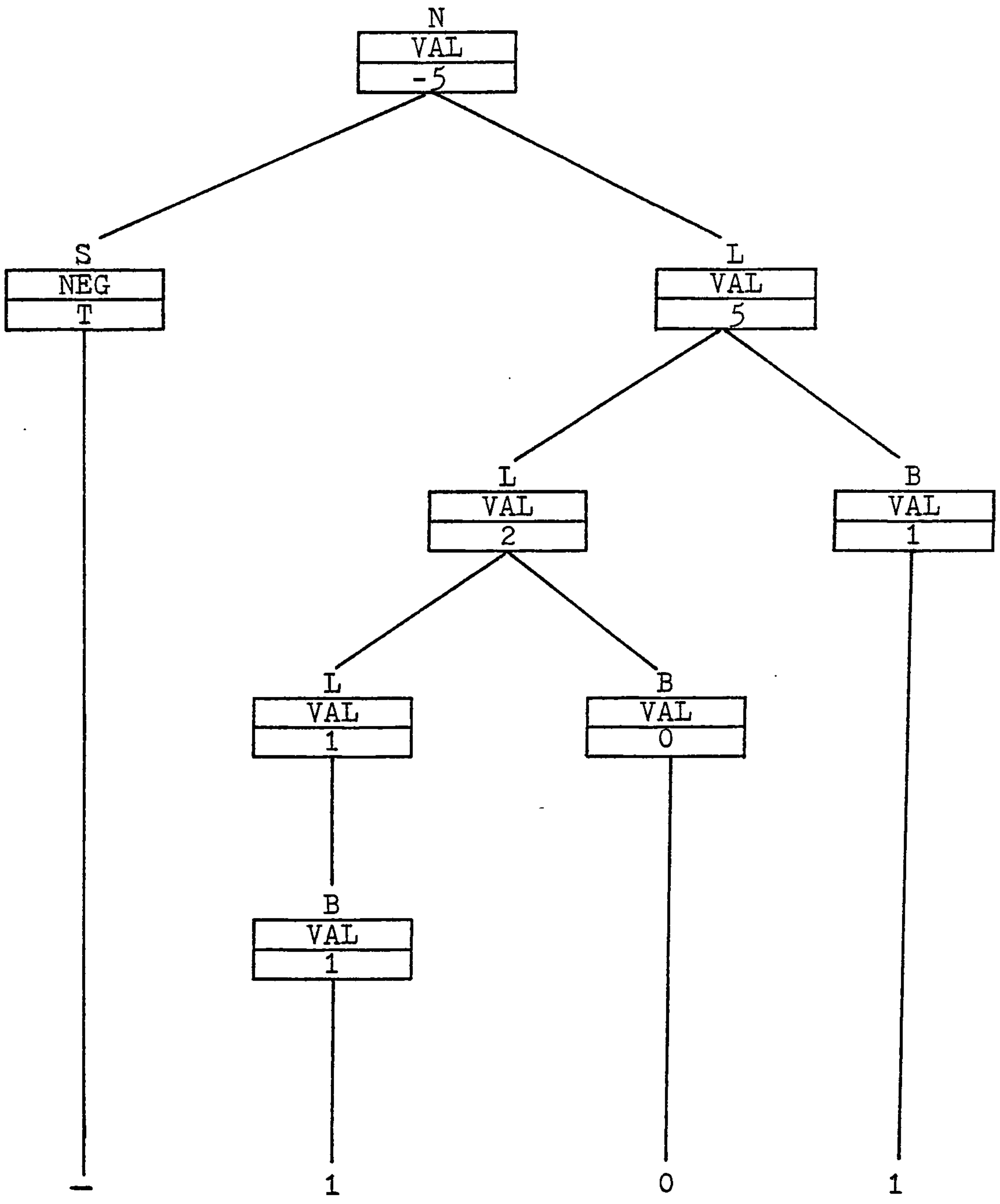


Example 2.3c - DEFINITION OF SYNTHESIZED GRAMMAR FOR SIGNED BINARY INTEGERS

Let G be the CFG defined in Example 2.3a

	PRODUCTION		SEMANTICS
1)	N	$\rightarrow$ S, L	N.VAL = <u>IF</u> S.NEG <u>THEN</u> - L.VAL <u>ELSE</u> L.VAL
2)	L	$\rightarrow$ L, B	L.VAL = 2L.VAL + B.VAL
3)		$\rightarrow$ B	L.VAL = B.VAL
4)	B	$\rightarrow$ 1	L.VAL = 1
5)		$\rightarrow$ 0	L.VAL = 0
6)	S	$\rightarrow$ +	S.NEG = FALSE
7)		$\rightarrow$ -	S.NEG = TRUE

Figure 2.3d - EVALUATED PARSE TREE FOR -101



Accordingly, we have invented the attribute "VAL" is also an attribute of the binary-string (L) and the bits (B), and S means sign. FIGURE 2.3d shows the parse tree for -101 associated with the attribute of each node. "VAL" is a synthesized attribute carrying information up the tree toward the root (start symbol N). In this figure, the effect of the attribute grammar specify that the meaning of -101 is -5.

2.4 Overview of the PLP (Pseudo Language Processor)

The research involving the programming and implemntation of the PLP presented in this thesis was accomplished in a five part sequence. The realization of each of these parts used information developed in prior parts. The first step was to tailor a contex-free grammar (CFG) which could dictate the structure of the programs it accepted and also have the properties to make analysis of PL programs quite easy. The next step was to determine if the input string (source program) was in the language defined by that CFG. Parsing was used to accomplish this task. The third step was to develop a scanner which could recognize the symbols that made up the programs. The fourth step was to produce the semantics that provided synthesized information like symbol cross reference and path expression (PE) for each symbol appearing in the input string. The fifth and last step was to analyse the path expression (local synthesized attribute) resulting from step 4 and generate all the possible anomalies for each variable involved in the source program. Each of these steps will be discussed in the remainder of this thesis. Figure 2.4a illustrates the PL processor with the integrated components.

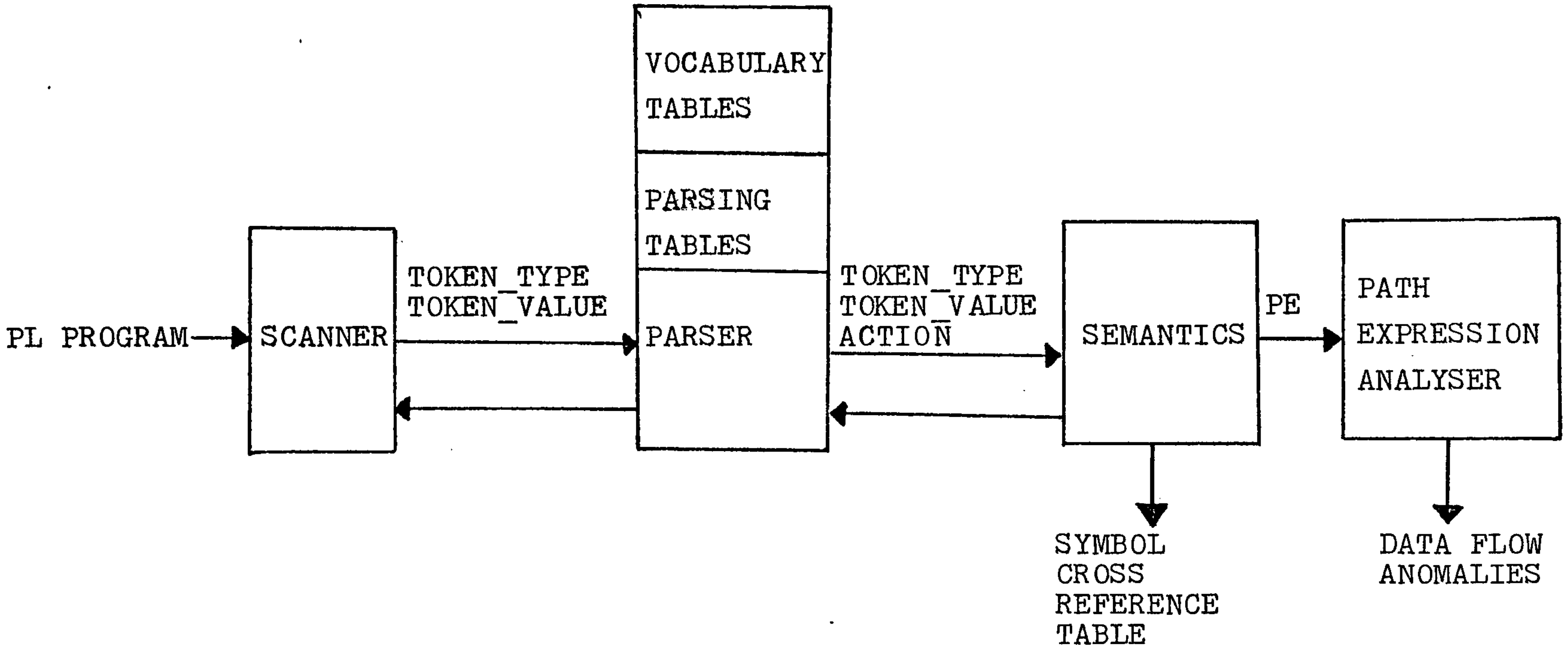


Figure 2.4a - OVERVIEW OF PSEUDO LANGUAGE PROCESSOR

Chapter 3

DESIGN OF PSEUDO LANGUAGE AND PSEUDO LANGUAGE GRAMMAR

A language is usually described by its two components

1) syntax (grammar)

2) semantics (meaning)

We will be mainly concerned with syntax of the language in this chapter. Some rules to be obeyed during the design and implementation of a well-structured language are given below:

- language must be designed so that the meanings of the program written in the language are clear.
- language definition should clearly imply its implementation, and must be complete, consistent.
- language definition should encourage program clarity and defensive programming, it can be accomplished by forbidding
  a) interference with the control variable, step, and size limit.
  b) GOTO's to an external label.

3.1 Design of PL (Pseudo Language) Grammar

A grammar is a finite description for possibly infinite sets of strings (languages). Once the characteristics of the language is determined, a grammar must be developed that defines these characteristics or structures of the program to be analysed.

The design of a context-free grammar for Pseudo language is based on PASCAL grammar[H] and EULER grammar (a generalization of ALGOL 60)[WW]. The reason for this is that PASCAL and ALGOL are structured programming languages which can cope with our requirements that PL allows "structured design", and "resembles the Pidgin English". A detailed discussion of PL grammar is next.

A complete PL grammar is given in Appendix A. As production 1 implies, all PL programs are seperated into two protions: the <u>introduction portion</u> and the <u>body portion</u>. P3 and P4 specify the two terminal symbols - "BEGIN_INTRO", "END_INTRO" to be the keywords that start and end the introduction portion. By analysing P5, we find that the components of "INTRODUCTION" are "EXPRPR", "FILENAME", "I/O", and "DICTIONARY". This production fully defines the components of the introduction portion. P6, P7, and P8 give the syntax for "FILENAME", while P9 - P15 define all "I/O" syntax which describes the input/output units used within the PL program. The productions associated with "DICTIONARYS" (P16 - P27) specify all nouns which are used in the body portion. Since PL allows for many different types, and each of these types can have many variable occurrences, infinite recursion was used in the productions. First of all, within a specific type, one or more variable can be declared. This sequence of variables was allowed by P20 - P27, and P74 - P85. Here, P25, P26, P27, and P124 send all variables (including nouns, numbers, and dimensional array) to "STRU_NOUN, P20 then sends "STRU_NOUN" to "PRIMARYPR". When P21, P22, and P23 are applied, then variable recursion is accomplished. Production 18 and production 19 therefore provide the type recursion. Finally, production 2 terminates the parsing action of the first part of PL. Note that the question mark (?) is to be used as a delimiter in this grammar. The reason is that any symbol delimited by "?" is treated as a grammar symbol by PARSER GENERATOR (PG). Without this delimiter, the PG would use those symbols as an instruction implying the separation of two grammar symbols. Example 3.1a has shown the acceptance of multiple variables and types.

Production 28 to production 123 fully define the syntax of PL body portion. We shall think of the body of PL as being composed of the statement forms which specify the control flow in the implementation program and statement forms which specify the executable, functional components of the implementation program. The fundamental executable

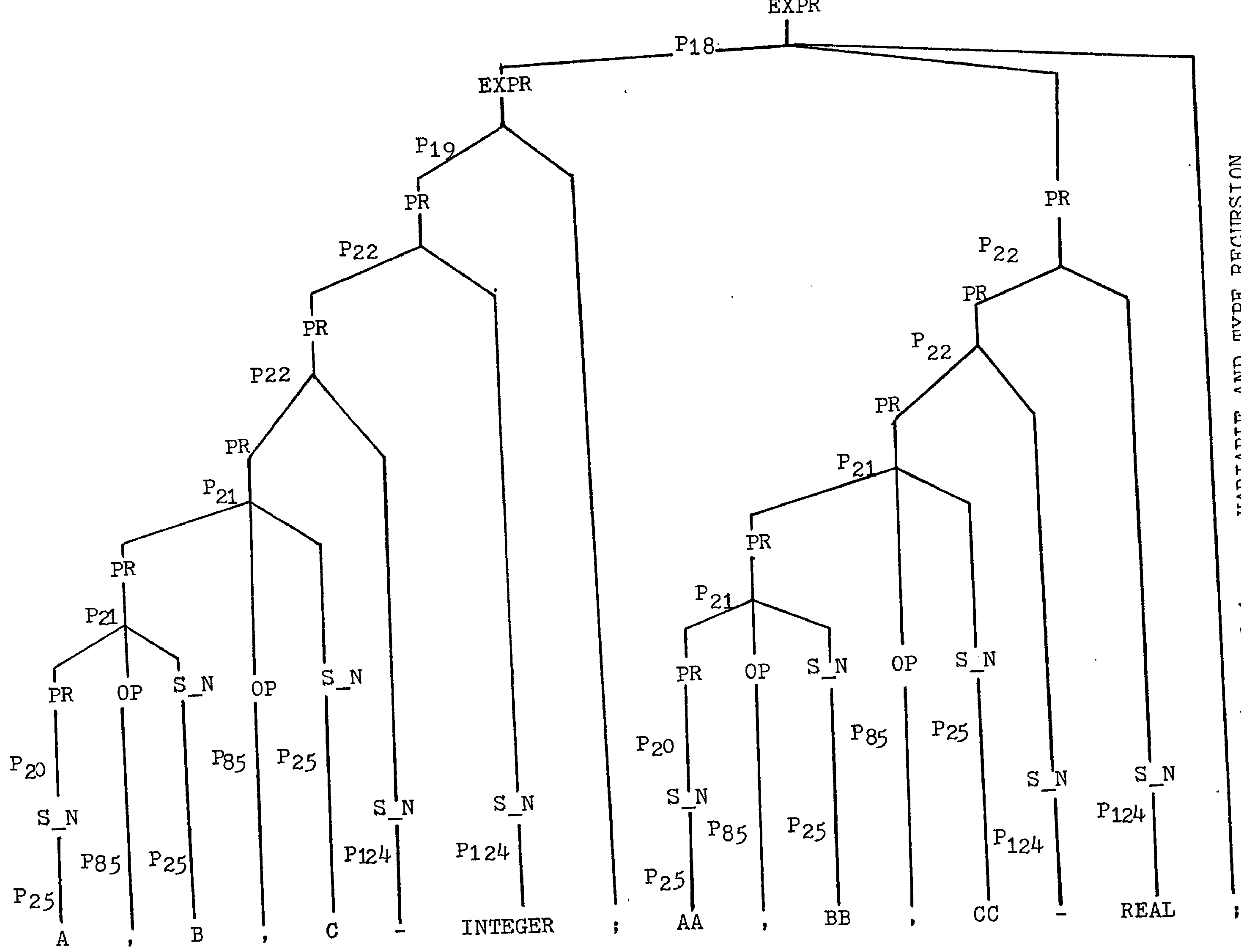


Figure 3.1a - VARIABLE AND TYPE RECURSION

components are assignments, commands, and control structures. The following paragraph will illustrate the construction and usage of these important program structures.

Production 71 and production 72 define the syntax of "ASSIGNMENT" to be the same format as in FORTRAN or PASCAL. A "COMMAND" is a very important feature of PL program. It helps programmers easily specify functional logic without involving too much details necessary in an implementation language. P47, P48, P50, P51, P52, P53, P54, and P55 gave the syntax to this construct. The grammar production for "COMMAND"

```
a) COMMAND       →  VERB PART, RETURN PART,;
b)               →  VERB PART,;
c)               →  RETURN PART,;
d) VERB PART     →  VERB CLAUSE, COMMENTS
e) VERB CLAUSE   →  VERB*
f) RETURN PART   →  RETURN_KEY, COMMENTS
g) COMMENTS      →  COMMENTS, GARBAGEPR
h)               →  GARBAGEPR
i) GARBAGEPR     →  NOUN_GARBAGE*
```

Here, the terminal symbol "VERB*" is to be semantically interpreted as any English language verb. "NOUN_GARBAGE*" is to be interpreted as an English word (it can be a noun, a proposition, or a conjunction). The noun-terminal "COMMENTS" can be thought of as a sequence of English words. Production a and production b terminate the "COMMAND". A simple example of "COMMAND" is shown as following:

(verb)	(noun)	(garbage)	(noun)	(garbage)
INTERCHANGE	FIRST_ITEM	IN	TABLE	WITH
	(noun)	(garbage)	(noun)	
	SECOND_ITEM	IN	TABLE ;	

Finally, we discuss the productions that define control structures. The syntactic construct called "CONTROL STRUCTURES" consists of the "CASE STAT", "WHILE STAT", "FOR STAT", "IF STAT", "CYCLE STAT", "REPEAT STAT", "EXIT STAT", "WITH STAT", "COMPOUND STAT", "DO STAT", and "CALL STAT". These structures in turn may be sent to any of the structures supported by the various languages currently in use. "P39 - P43", "P56 - P70", and"P74- P122" give the syntax for these constructs. Again, the following productions will recursively generate all possible sentences (except keywords) in control structures.

a)	EXPR	→	PRIMARY
b)		→	OPERATOR, PRIMARY
c)		→	EXPR, PRIMARY
d)	PRIMARY	→	LEFT PAREN, EXPR, RIGHT PAREN
e)		→	NUMBER*
f)		→	IDENTIFIER*

```
g) OPERATOR  →  +
h)           →  -
i)           →  *
j)           →  /
k)           →  **
l)           →   =
m)           →  >
n)           →  <
o)           →  <=
p)           →  >=
q)           →  ?.?
r)           →  ?,?
```

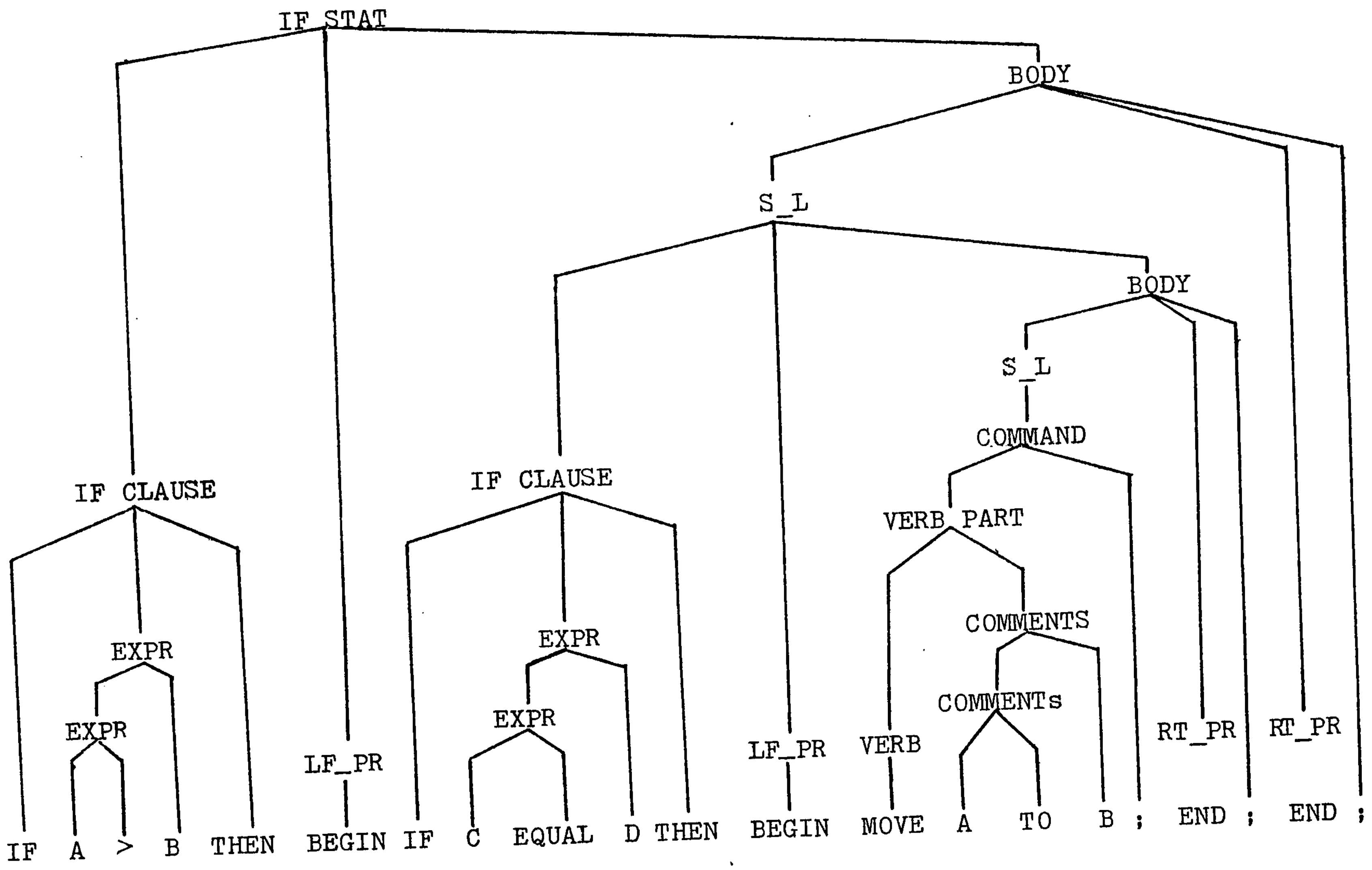
These productions look quite similar to those productions mentioned in the introduction portion (P18 - P24). Actually, these two structures almost give the same syntax of the language. The reason for this duplication is to avoid the "ambiguity" problem during parsing. More specifically a word in the introduction was treated as a "NOUN" whereas the same word in the body is treated as an "IDENTIFIER*" or a "NOUN_GARBAGE". "Ambiguity" is a typical problem encountered during the design of PL grammar. This subject will be discussed later in this section.

As was mentioned in Chapter 2, the data flow analysis is based on the knowledge that input programs are "well-structured". Structured transfer of control is accomplished by the use of "IF", "WHILE", "CASE", and "REPEAT" structures. Production 106 to production 113 give the definition of the "IF" as follows:

```
a) IF STAT     →  IF CLAUSE, LEFT PAREN, BODY
b)             →  IF CLAUSE, LEFT PAREN, BODY, ELSE PART
c) ELSE PART  →  ELSE_KEY, LEFT PAREN, BODY
d) ELSE_KEY   →  ELSE
e) IF CLAUSE  →  IF_KEY, EXPR, THEN
f) IF_KEY     →  IF
g) THEN_KEY   →  THEN
h) BODY       →  STAT LIST, RIGHT PAREN,;
```

The ability to transfer control to the block of codes is represented by "LEFT PAREN, BODY" when "IF CLAUSE" is true or to the block of codes represented by "ELSE PART" when "IF CLAUSE" is false. Sometimes, the "ELSE PART" is not needed. In other words, "IF" can be just a conditional check statement followed by a true part statement. This may be accomplished by the productions in the partial grammar described above. Note that the "nesting" ability of "IF" statement is also defined by the grammar. This is because of that the P63 sends "IF STAT" back to "CONTROL STAT" , and "CONTROL STAT" may be parsed to "STAT" (P42) then "STAT LIST" (P40). Therefore, "STAT LIST" itself may be a "IF" structure. By applying P39, it is seen that a "nesting" is recursively defined. An example is shown in Figure 3.1b.

Figure 3.1b - NESTING OF " IF " STATEMENT



Another nested structure, the "WHILE" loop, was defined in the grammar by production 93 and production 94. The ability to have "nesting" can be generated by the same procedure as described in the "IF" productions above. Production 62 sends "WHILE STAT" back to "CONTROL STAT". The reminding steps are exactly the same. "REPEAT" loop is another transfer control structure. Production 99, and 100 give the syntax. From the syntax, it is easy to see that "WHILE" loop and "REPEAT" loop have the same feature to transfer control flow except that the former checks the conditional expression first and then executes the loop body when condition is true, or jump out of the loop when the condition is false; while the latter executes the loop's body first and then checks the conditional expression. Either of these two constructs will fulfill the requirement of transferability without using "GOTO"s. Note that concurrent execution of statements may be specified by the PL grammar. The syntax of "CYCLE STAT" is given by P97. Hence, PL allows the design of program forms for a wide variety of applications. The rest of the structures in the body portion like "WITH", "CALL", and "EXIT" statements, are all "straight-line composition", so that the constructs of syntax are straightforward, and defined respectively by P104, P105, P122, and P102, P103. It should be noted that "EXIT" is allowed in the PL grammar. The reason is because PL tries to give the users the flexibility to develop their logic for a variety of applications. But, one should always keep in mind - "GOTO'S" are not encouraged in structured programming work. Production 28 requires the entire body portion to be surrounded by a "LEFT PAREN" and "RIGHT PAREN" which are seperately defined by P29 - P33, and P34 - P38. Finally, production 1 defines the PL program.

The features of the PL grammar were not all implemented at one time. As described in chapter 1, PL was designed to accept structured constructs and also resemble Pidgin English - allow English verbs and sentences to appear in a PL program. Obviously, it is not a trivial contex-free grammar, therefore,the SLR (simple left - right) parser construction method initially used is not enough to produce an adequate parser. In other words, SLR parser just cannot remember enough left context to decide what action the parser should take on some particular input string. Unfortunately, a more complicated and powerful LALR(K) parser (a lookahead LR parser) could not be generated by the existing parser generator. Hence, the PL grammar was designed to fulfill the input requirements of currently existing PARSER GENERATOR. The grammar designed was SLR(1). Many different techniques were used to solve the problems which were encountered during the design phase.

The frequent problem is due to "ambiguity". A grammer is called "ambiguous" if there exists a string in the language for which there is more than one derivation sequence. Example 3.1c shows that the string "babab" has two derivation trees corresponding to the grammar. When an ambiguous grammar is fed to the parser generator, the generator is unable to construct a deterministic parsing table for the grammar. Certain input strings (programs) may have more than one translation. This problem is usually caused when trying to define the grammar productions to generate a sequence of strings or a sequence of statements recursively. Fortunately, the "ambiguity", usually, can be fixed by inserting in "hard-token" (terminal symbols like keywords, semi-colon, and ?;?).

One other frequently encountered problem is that the "left recursion" and "right recursion" occur simultaneously causing ambiguity. For example, if production 21 is redefined as:

PRIMARYPR $\rightarrow$ PRIMARYPR, OPERATOR, PRIMARYPR

Then a "left and right recursion" will generate two derivations. However, this problem can be solved by changing the level of these "trouble" items to another level (either upper or lower level). A typical example is shown in the PL grammar (Appendix A, Production 18 - Production 27). Most of the problems discussed above

Example 3.1c - TWO DERIVATION TREES FOR "babab"

Let G be a CFG where

G = (VN, VT, P, N) such that

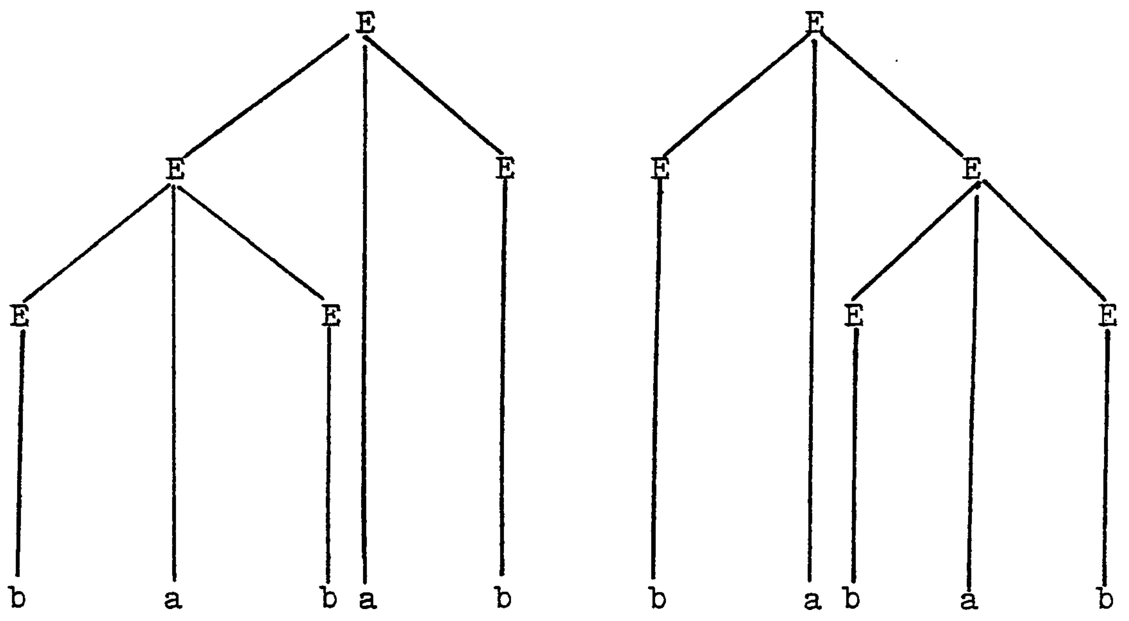
VN = (E)

VT = (a, b)

PRODUCTIONS :

1) E $\rightarrow$ E, a, E

2) $\rightarrow$ b



are implicit, which are very hard to detect at the first "trial run". A lot of tests and refinements had to be made before the "bugs" were exterminated.

The final grammar of Appendix A was developed after many changes and corrections. The PL grammar has its limitations as well as its virtures. In addition to the acceptance of well-structured constructs and resemblance of Pidgin English, the primary design of
to associate these features with a parse tree which could be easily analyzed. Note that the partial grammar of appendix A (production 125 to production 150) was developed by William R. Ledbetter[L] in order to provide the following features:

a) Providing a multiple-processing capability.
b) Providing a procedure file management capability for use within the PLP.
c) Providing the capability to redefine default values of indicators which implement some functions within the PLP.

Details see Chapter 4 of Ledbetter's thesis - "A Pseudo Language Processor for Design Validation and Implementation of Systems"

3.2 Description of Pseudo Language (PL)

A PL program is a program form which represents a broad class of possible implementation in any of the standard programming languages. The fundamental components of a PL program are

1) Introduction section
   - nouns and their descriptions
2) Body section
   - assignments
   - commands
   - control structures

All keywords used in the following text are underlined. Constructs are parenthesized using "<" and ">". A <word> is a construct with a string of characters of length up to twenty. The characters in a word may be digits, an alphabet symbol, or underscore symbol "_". The first character in a word must be a letter (A - Z).

3.2.1 Introduction Section of a PL Program

Every PL porgram must begin with an introduction portion. In the <introduction> the programmer specifies:

- the interface with other programs
- the descriptions of all nouns needed in the body of the program.

Figure 3.2.1a provides a complete frame of PL introduction section. <Sentence> in the <introduction> can be

Figure 3.2.1a - DESCRIPTION OF PL INTRODUCTION STRUCTURES

```
BEGIN_INTRO       ______________  Required Keyword
                                 \
   <sentence> ;                   |
        .                         |
        .                          > must use at least
        .                         |  one <sentence>
                                  |
   <sentence> ;                  /

FILES             ______________  Optional Keyword

   <sentence> ;
        .
        .
        .

   <sentence> ;

INPUT_PARAMETERS  ______________  Optional Keyword

   <sentence> ;
        .
        .
        .

   <sentence>;

OUTPUT_PARAMETERS ______________  Optional Keyword
   <sentence> ;
        .
        .
        .

   <sentence> ;

DICTIONARY        ______________  Required Keyword
                                 \
   <sentence> ;                   > must use at least
        :                         |  one <sentence>
   <sentence> ;                  /
END_INTRO         ______________  Required Keyword
```

any English sentence except if it appears after the keyword dictionary. It must begin with a sequence of <noun> which are seperated by commas. In the dictionary a <sentence> may include the optional keyword initial followed by any sequence of <constants>.

The keywords used in the introduction are:

begin_intro	(required)
files	(optional)
input_parameters	(optional)
output_parameters	(optional)
dictionary	(required)
initial	(optional)
end_intro	(required)

Some restrictions on <sentence> are as follows. If <sentence> follows keyword dictionary then:

a) A <noun> may not be a keyword used either in the <introduction> or in the <body>. (keywords used in the body section will be defined later).

b) A <sentence> must be followed by a ";" and must contain at least a single noun.

c) After the keyword initial there must be at least one <constant>. A <constant> can be any number defined in any of the existing programming languages.

d) The general form of <sentence> after dictionary is:

a) <noun 1>, <noun 2>, ... <noun $_n$> garbage ; or

b) <noun 1>, <noun 2>, ... <noun $_n$> <garbage> initial <constant 1> ... <constant $_n$> ;

where <garbage> is any s quence of words or empty.

There are no restrictions on sentences which follow other keywords in the <introduction> except that they must have at least one word and end with a semi-colon (;). An example of PL <introduction> is shown in Example 3.2.1b.

3.2.2 Body Section of a PL Program

The body of a PL program must be surrounded by a <left parenthesis> and a <right parenthesis> and followed by a semi-colon (;).

- <left parenthesis> is a do, begin, cobegin or "(".
- <right parenthesis> is a od, end, coend or ")".

The general form of body section in a PL program has shown below:

```
<left parenthesis>
      <body sentence>
            .
            .
            .
            .
      <body sentence>
<right parenthesis>;
```

Example 3.2.1b - EXAMPLE OF PL INTRODUCTION

```
BEGIN_INTRO
        SCANNER ;
   INPUT_PARAMETERS
      TOKEN    :  SYMBOL OF INPUT STRING ;
   OUTPUT_PARAMETERS
      TOKEN_TYPE   :  ENTRY POINT OF THE NEW TOKEN IN
                      VCBLRY_TABLE ;
      TOKEN_VALUE  :  VALUE POINTS TO THE ENTRY OF NEW
                      TOKEN IN SYMBOL_TABLE ;
   DICTIONARY
      FIRST_TIME   : INTEGERS ;
      SYMBOL_TABLE : SEQUENCE OF INTRIES ;
      VCBLRY_TABLE : TABLE OF TERMINAL AND NON-TERMINAL
                     SYMBOLS ;
      END_OF_FLAG  : FLAG INITIAL 0 ;
END_INTRO
```

A <body sentence> can be

- assignment
- command
- control structure

An <assignment> is just any standard assignment statement in FORTRAN or PASCAL. For instance,

```
A =   (B+C)*D
A :=  (B+C)*D
```

A <command> is an English sentence that must begin with a verb and followed by nouns and acted by the verb. A general form of a <command> is:

<verb> <garbage> <noun 1> <garbage> <noun 2>...<noun $_n$> <garbage> ;

or

<verb> <garbage> <noun 1> <garbage>....

return <garbage> <noun 1>....

The definition of <garbage> is the same as in the <introduction>. For instance:

```
MOVE RECORD TO TABLE;
```

Here, MOVE is a <verb> and possibly a routine name, RECORD and TABLE are <nouns> if they are declared in the <introduction>. TO is a <garbage> (since it is not declared).

To summarize:

1) A <command> must begin with a <verb> which can also be thought of as a routine name. A <verb> is just a <word>. A <command> must end with a ";".

2) A <verb> <u>cannot</u> be a keyword which is used either in the <introduction> or <body>.

3) In a command, any <word> following <verb> which also appears in the <introduction> is treated as as a <noun>.

4) A <word> must be followed by at least one <word> prior to return or semi-colon (;).

5) All <nouns> following the keyword <u>return</u> are considered by the Pseudo Language Processor (PLP) as being defined in the routine named by the <verb>.

The control structures in PL allow the programmer to specify a variety of branches without jumping to an unconditional branch (i.e. GOTO'S). <u>Figure 3.2.2a</u> shows a complete description of the control sturctures. A program of PL body is shown in <u>Example 3.2.2b</u>.

To summarize, the restrictions in body sentences are:

1) Keywords used in body sentences are - <u>begin</u>, <u>begincase</u>, <u>call</u>, <u>case</u>, <u>cobegin</u>, <u>coend</u>, <u>continue</u>, <u>cycle</u>, <u>do</u>, <u>exit</u>, <u>else</u>, <u>end</u>, <u>endcase</u>, <u>for</u>, <u>if</u>, <u>od</u>, <u>print</u>, <u>read</u>, <u>repeat</u>, <u>then</u>, <u>until</u>, <u>with</u>, <u>while</u>, <u>write</u>.

2) A <test> is any sequence of words and special symbols. The words may not be the keywords begin, begincase, cobegin, do, then, and the symbol "[".

3) Every <body sentence> must terminate with a semicolon (;).

4) <label> is a word that is an integer or is not a keyword.

Note that the <body sentence> is recursively defined. Details concerning recursion were discussed in Section 3.1.

A <comment> in the PL program is also processed by the PLP in order to space and document it appropriately in the output listing. If a <sentence> or a <body sentence> begins with a "//" then the rest of the line is ignored. Generally speaking, a sequence of programs in PL has the following form:

```
<introduction 1>                            \
<body 1>                                    |
<introduction 2>                            |   PL Program
                     PL Program              >
<body 2>                                    |   List
      .                                     |
      .                                     |
      .                                     |
<introduction n>                            |
<body n>                                    /
```

Figure 3.2.2a - DESCRIPTION OF PL CONTROL STRUCTURES

```
CALL       <sentence> ;


CASE  <test>  BEGINCASE
   <label 1> :  <body sentence>  END ;
       .
       .
       .
   <label k> :  <body sentence>  END ;
ENDCASE ;


FOR   <test>
   <left parenthesis>
   <body sentence>
       .
       .
   <body sentence>
   <right parenthesis> ;


IF    <test>  THEN
   <left parenthesis>
   <body sentence>
       .
       .
   <body sentence>
   <right parenthesis>
ELSE
   <left parenthesis>                  )
   <body sentence>                     |
       :                               } Optional
   <body sentence>                     |
   <right parenthesis> ;               )
```

```
WHILE  <test>
   <left parenthesis>
   <body sentence>
       .
       .
   <body sentence>
   <right parenthesis> ;


REPEAT
   <body sentence>
       .
       .
   <body sentence>
UNTIL    <test> ;


WITH  <test>
   <left parenthesis>
   <body sentence>
       .
       .
   <body sentence>
   <right parenthesis> ;


CYCLE
   <body sentence>
       .
       .
   <body sentence>
   <right parenthesis> ;
```

```
DO     <number> <sentence>   <sentence>
       <body sentence>
           .
           .
           .
       <body sentence>
       <number>   CONTINUE ;


EXIT      <sentence> ;

READ      <sentence> ;

PRINT     <sentence> ;

WRITE     <sentence> ;
```

Example 3.2.2b - EXAMPLE OF PL BODY

```
BEGIN
  IF  FIRST_TIME THEN  BEGIN
      GET SYMBOL AND RETURN SYMBOL ;
      SET FIRST_TIME ;
  WHILE  SYMBOL = END_OF_FILE  BEGIN
     CURRENT_STATE (J) = NEXT_STATE (I) ;
     CASE  NEXT_STATE IS  BEGINCASE
         KEYWORD : DETERMINE TOKEN_TYPE ;
                   SET TOKEN_VALUE TO 0 ;
                   END ;
         IDENTIFIER : INSTALL IDENTIFIER IN SYMBOL TABLE ;
                      END ;
         SPECIAL_SYMBOL : DETERMINE TOKEN_TYPE ;
                          SET TOKEN_VALUE TO 0 ;
                          END ;
         NUMBER   :  CONVERT TO INTEGER NUMBER ;
                    END ;
      ENDCASE ;
      GET_SYMBOL AND RETURN SYMBOL ;
   END ;
END ;
```

Chapter 4
PROGRAMMING IMPLEMENTATION OF PLP

The implementation work of Pseudo Language Processor (PLP) consists of five steps. These steps could be grouped into two parts. The first part includes defining a PL grammar, writing a Skeleton Parser and Scanner, and designing the Semantics. The second part contains the design of a path expression Analyser. The steps in each part actually implemented a pass of the PLP. Hence, PLP is a two-pass processor. The entire work for both portions is based on the parsing techniques using a Parser Generator to generate parsers. The context-free grammar which defines the syntax of PL is the input to the Parser Generator. The Parser Generator generates a parsing table (consisting of a VOCABULARY table, a READ action table, a APPLY action table, and a LOOK action table) as output. These tables contain all the information needed by the parser to determine the appropriate parsing action for any input string (source program). A scanner is used to scan each input string, and return the proper information to the parser. With this information, and using the information provided by the parsing table, the parser is able to parse any source program. During the parsing process, if a symbol is misspelled or left out, then a diagnostic message is generated.

Meanwhile, when a production is being applied, the semantic functions associated with that production are called. When the parsing process terminates, the Semantics provides some messages useful for validating the source program. The Analyser then analyses the path expression (PE) resulting from the Semantics of the earlier pass for generating all possible data flow anomalies. Chapter 3 describes the grammar development for PL, the remaining steps will be discussed in their entirety in this chapter.

4.1 Parser

A parser for a context-free grammar, G, is a program that takes as input a string W and produces as output either a parse tree for W, if W is a sentence of G, or an error message indicating that W is not a sentence of G [AU]. The two basic types of parser for context-free grammar are "bottom-up" and "top-down". As indicated by their name, "top-down" parser starts with the top (root) and work down to the bottom (leaves), while "bottom-up" parser builds parse trees from leaves to the root. The "bottom-up" method is also know as "shift-reduce" parser, because it consists of shifting input symbols onto a stack until the right side of a production appears on top of the stack. Note that the input stream to the parser is being scanned from left to right one symbol at a time. This thesis uses "shift-reduce" parsing method to construct the Skeleton Parser.

There are four possible actions in the " shift-reduce " parsing sequence :

1) Shift - Shift next symbol to the top of the stack.
2) Reduce - When the parser determines a sequence of symbols on the stack is the same as the right-hand side of a production in the specified grammar, it may then replace those symbols with the left-hand side non-terminal symbol.
3) Error - The parser determines that a syntax error has occured and takes appropriate action.
4) Accept - The parser announces successful completion of the parsing.

There are several different " shift-reduce " parsers. Precedence parsing probably is the easiest one to implement. <u>Example 4.1a</u> is a good illustration to this parsing method. However, there are many grammar constructs which can not be handled by this method. LR parsing techniques can solve the problems encountered by simple precedence parsing. The LR parser resolves the problems by examining the contents of the stack to obtain the left-context information of the input string it has already seen and looking ahead in the input string to get right-context information of next symbol. By doing so, most of the problems of " ambiguity " can be overcome. LR(0) is the simplest method of LR parsing. This

Example 4.1a - PRECEDENCE PARSING ACTION

GRAMMAR :

P :	1)	$E \rightarrow E + E$
	2)	$E \rightarrow E * E$
	3)	$E \rightarrow (E)$
	4)	$E \rightarrow id$

INPUT STRING : id + id * id $

STACK	INPUT	ACTION
1) \$	$id_1 + id_2 * id_3$ \$	shift
2) \$ id_1	$+ id_2 * id_3$ \$	reduce by P_4
3) \$ E	$+ id_2 * id_3$ \$	shift
4) \$ E +	$id_2 * id_3$ \$	shift
5) \$ E + id_2	$* id_3$ \$	reduce by P_4
6) \$ E + E	$* id_3$ \$	shift
7) \$ E + E *	id_3 \$	shift
8) \$ E + E * id_3	\$	reduce by P_4
9) \$ E + E * E	\$	reduce by P_2
10) \$ E + E	\$	reduce by P_1
11) \$ E	\$	accept

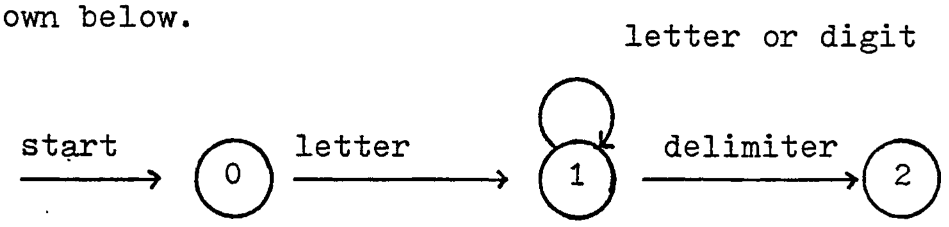
method uses no-context information to complete the parsing action. However, for some grammars which can not be parsed using LR(0), SLR(1) (simple LR) technique may be the good solution to the problem. SLR(1) uses the right-context information (look ahead in the input string) to get rid of the "ambiguity". Sometimes, SLR methods are not sufficient to solve the difficulties and LALR (lookahead LR) techniques are employed. LALR examines both left and right context information to overcome the problems encountered in LR(0), or in SLR. As was mentioned in Chapter 3, the powerful LALR Parser Generator was not implemented, so, a SLR(1) parsing technique was used in this thesis. Actually, the Skeleton Parser simulated a push down automata (a finite state machine with a stack). This push down automata can recognize the valid PL program and go to an accept state. Invalid PL programs may send the push down automata to a recovery routine.

A more precise discussion about parsing techniques and Skeleton Parser was presented in Jame D. Arthur's thesis "A Unified Model For Constructing Automatic Analyzers Which Perform Static And Dynamic Program Validation" [A].

4.2 Scanner

The function of a scanner is to read the source program, one symbol at a time, and to translate it into a sequence of units called tokens. Examples of tokens are keywords, identifiers, constants, operators, etc.[AU]. The scanner and parser are coupled. That is, the scanner is a subroutine which is called by the parser whenever it needs a new token. The requirements of Skeleton Parser are that for each new symbol read by the scanner, an integer value, token-type, and a token-value are to be returned. The token-type is the pointer which indicates where that symbol appears in the Vocabulary Table (VT). The "Vocabulary Table" is a part of the parsing table and is composed of all terminal and non-terminal symbols in the PL grammar. For instance, if the left-parenthesis "(" is read by the scanner, then the value returned for token-type would be 1 (see Appendix B). Furthermore, if the symbol read in is a variable (i.e., symbol is not in the list of terminal symbols), and met the specification for "VERB*", then the position of where "VERB*" appears in the VT (68) is returned as the token-type value. Similar logic is used for other variables such as "NOUN*", "JUNK*", "NOUN_GARBAGE*", and "IDENTIFIER*". However, the value returned for token-value is 0 for all symbols except "NOUN*"s, "VERB*"s, and "IDENTIFIER*"s. As these variables are read, they are placed in a symbol table by the scanner if they are not already in there. Thus, the value returned in token-value for a variable is a unique position in the symbol table. The token-value is used to distinguish between instances of a token. For example, if a token is read and its token-type, returned by scanner, is a "NOUN*" then the token-value can indicate whether the "NOUN*" found is A, B, C, or D, etc. Another primary function of the scanner is to inform the parser when the input is completed. The Parser Generator has appended a terminal symbol "EOF SYMBOL" to production 125 of PL grammar (see Appendix A). As a matter of fact, the original PL grammar read by PG does not contain this symbol. In other words, this symbol is not part of the source input program. When the scanner finds there is no more input , then the token-type returned to the parser is a value of the position corresponding to "EOF SYMBOL" in the VT. To summarize, the total information needed by the Skelton Parser about the symbols read by the scanner is obtained from the values passed in token-type and token-value. The implementation details are given below.

We begin with the discussion of designing a program for scanner. Here, a very useful "flow chart" called <u>transition diagram</u> is introduced. Consider an example of identifiers. In the ordinary programming language, an identifier is defined to be a letter followed by zero or more letters or digits. The transition diagram for an identifier is shown below.



In this diagram, the circles are called states. The states are connected by arrows, called edges. The starting state of this diagram is state 0, the edge from state 0 indicates that the first input symbol must be a letter. If this is the case, then the transition enters state 1 and gets the next input symbol. If another letter or digit is obtained, then the transition re-enters state 1 (cycle) and looks at the next symbol. It continues these steps until the input symbol is a delimiter for identifier, then enters state 2 and terminates the process. This approach is very helpful for constructing the PL Scanner. It is because the action taken by this scanner is highly dependent on what item has been seen recently. The precise specification for the scanner is called Finite State Automata (FSA). There are two different types of finite automatas - Non-determinstic Finite Automata (NFA) and Determininstic Finite Automata (DFA). The definition of NFA is :

A finite automata M over an alphabet P is a system (K, P, S, Q, F) where

1) K is a finite, non-empty set of states.
2) P is a finite, non-empty set of states.
3) S is a mapping of $K \times P$ into subset of K.
4) Q is a distinguished initial state.
5) F is a non-empty subset of K and is the set of final states.,

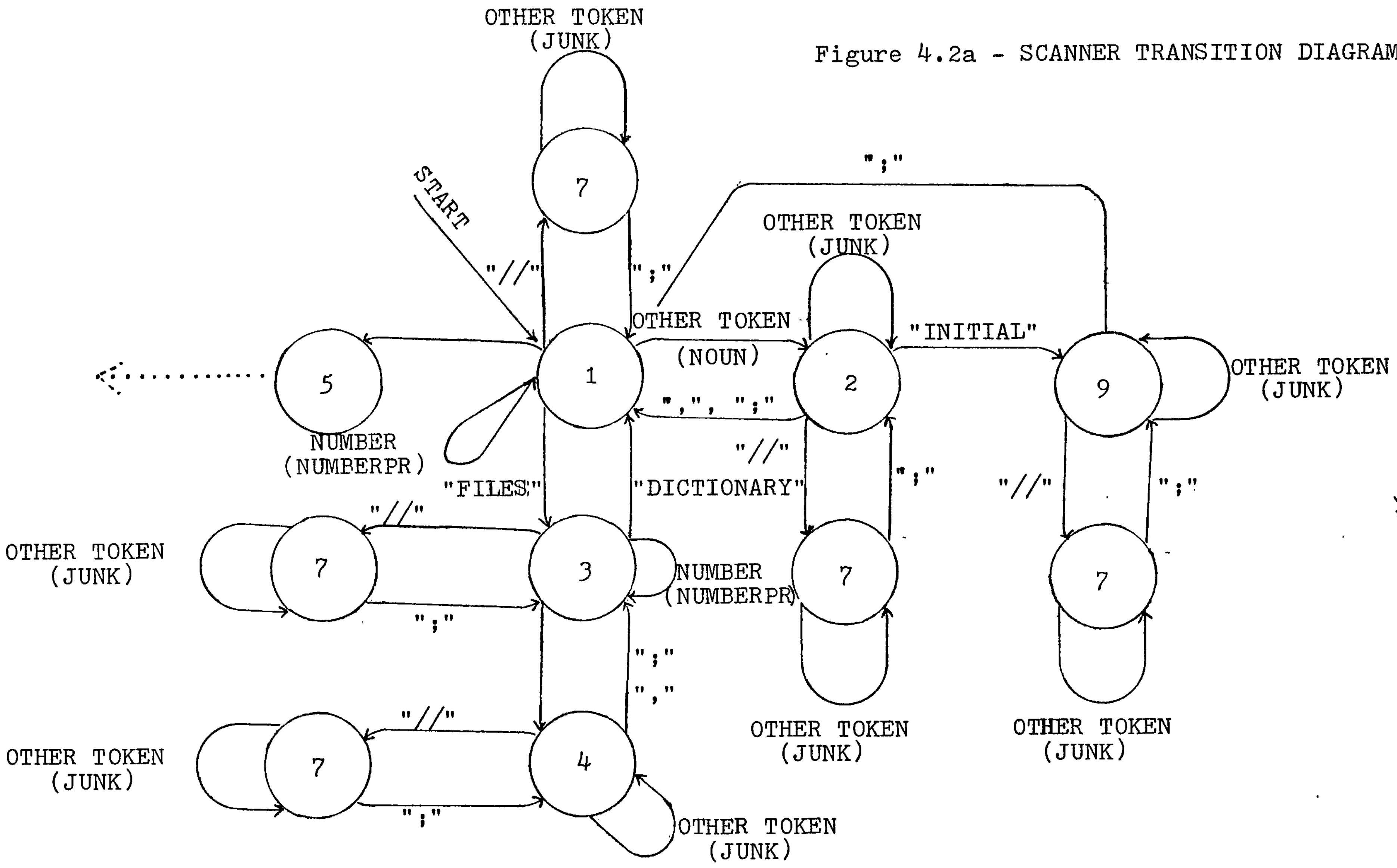
While the definition of DFA are :

A finite automata M over an alphabet P is a system (K, P, S, Q, F) where

K, P, Q, F are the same as was defined in NFA, but there is no transition on input ϵ (empty string) and S is a mapping of $K \times P$. In other words, for each state s_i and input symbol p_i there is at most one edge labeled p_i leaving s_i.

Since there is at most one transition out of any state on any input symbol, obviously, DFA is easier to simulate by a program than a NFA. Therefore, the auther implements the scanner by simulating a DFA. The implementation of the scanner has a program fragment for each state. The program fragment can determine the proper transition to make on the current input symbol. Figure 4.2a is a complete transition diagram for the scanner. The beauty of using this state diagram is that the scanner can examine the current state to get the left-context information and look ahead at the next symbol to get the right-c ntext information in order to determine the token-type, token-value and the appropriate transition action. Many complicated problems can be overcome by applying this method. For example, if the scanner reads a token, and this token is neither a terminal symbol nor a number, then the syntax of PL grammar will tell us that the token could be the type of "NOUN*", "JUNK*", "VERB*", "NOUN_GARBAGE*",

Figure 4.2a - SCANNER TRANSITION DIAGRAM



- Starting state is 1
- Edge X → Y denotes transfer of control from X to Y on reading token a and return type b if it appears in parenthesis, else return a.
- Rest of the machine on the following pages

```
1:  if TOKEN IS OTHER THAN "NUMBER", "//", "FILES",
    "END_INTRO" then return TOKEN AS NOUN.

2:  if TOKEN IS OTHER THAN "//", "INITIAL", ";", ","
    then return TOKEN AS JUNK.

3:  if TOKEN IS OTHER THAN "//", "DICTIONARY", ";",
    "TERMINAL SYMBOL" then return TOKEN AS NOUN.

4:  if TOKEN IS OTHER THAN "//", ";" return TOKEN AS JUNK.

9:  if TOKEN IS OTHER THAN "//", ";", then return TOKEN AS
    JUNK.

7:  // COMMENT SYMBOL
    // ENCOUNTERED IN
    // INTRODUCTION PART
    SIMULATE THE EXPR→  PRIMARYPR, ";" PRODUCTION
```

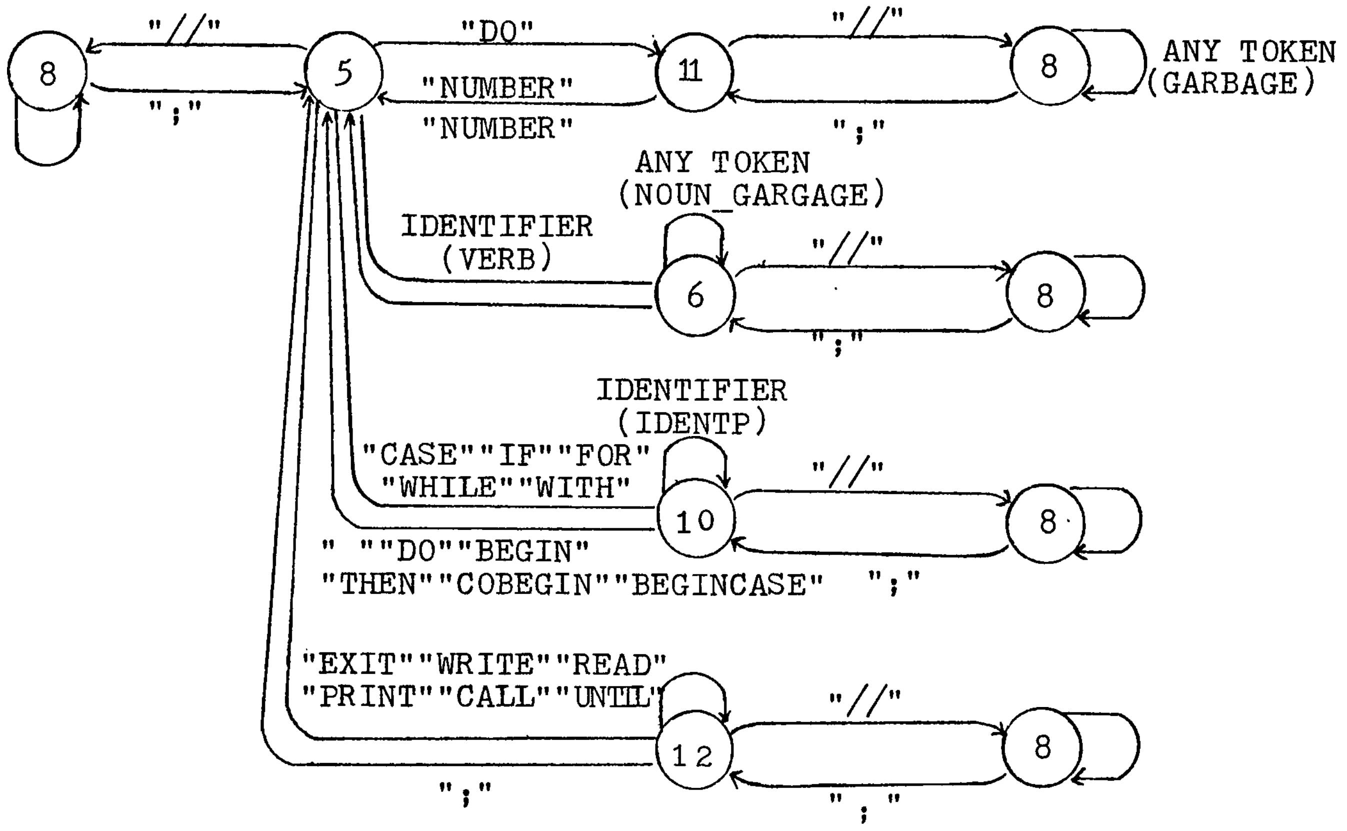


Figure 4.2ab - SCANNER TRANSITION DIAGRAM

```
5 :  if TOKEN IS NOT A TERMINAL AND NOT A NUMBER AND
        NOT "//", "DO"
        if TOKEN IS IN NOUN_TABLE
        then return TOKEN  AS A IDENTIFIER ;
        else
             if LOOKAHEAD TOKEN IS A OPERATOR
             then return TOKEN AS A IDENTIFIER ;
                  UPDATE NOUN_TABLE AND NOUN_VALUE TABLE ;
             else return TOKEN AS A VERB ; UPDATE VERB_TABLE
                  AND VERB_VALUE TABLE ;
6 :  if TOKEN IS OTHER THAN "//", ".", ";"
     then return TOKEN AS A NOUN_GARBAGE ;
     if TOKEN IS IN NOUN_TABLE
     then return VALUE POINTS TO THE SYMBOL TABLE ;
     else
         if TOKEN IS A TERMINAL OR A SPECIAL_KEYS
         then return VALUE AS 0 ;
         else UPDATE NOUN_TABLE AND NOUN_VALE TABLE ;
8 :  SIMULATE THE PRODUCTION  48;
     //  COMMAND  ———  VERB PART ;
10 : if TOKEN IS OTHER THAN "//", "DO", "BEGIN", "COBEGIN",
        "THEN", "BEGINCASE"
     then return TOKEN AS A IDENTIFIER ;
11 : if TOKEN IS OTHER THAN "//" AND TOKEN = NUMBER
     then return TOKEN AS A LABEL_KEY ;
     else GO TO STATE 5;
12 : if TOKEN IS OTHER THAN "//", ";"
     then return TOKEN AS AN IDENTIFIER;
```

or "IDENTIFIER*". The problem is how to decide between these types. In Figure 4.2a, if current state is 1, then the value for token-type is returned as the position of "NOUN*" and transition enters state 2. If current state is 2, then token-type is returned as a "NOUN_GARBAGE*" and transition still re-enters the same state. However, if the current state is 5, then the scanner will return token-type either as a "VERB*" or as a "IDENTIFIER*" depending on the right-context information and transition will enter state 6 or state 10 respectively. Generally speaking, the type for a token in a language recognized by its scanner usually contains only the terminal symbols (keywords, operators, etc.), identifiers and numbers. However, the PL Scanner must have the ability to distinguish between various type of the identifiers. The reason is that the PL grammar is designed to fulfill the requirements for SLR(1) Parser. As a matter of fact, the SLR(1) grammar is not powerful enough to handle all the features described in the Pseudo Language. Hence, the scanner has to do the work which is normally handled by a more powerful grammar. Finally, a calling structure for all routines called by the scanner main routine (SCAN) is shown in Figure 4.2b, and the scanner itself is given in Appendix C.

4.3 Semantics for PL

As was mentioned in Chapter 1.5, the fourth step of

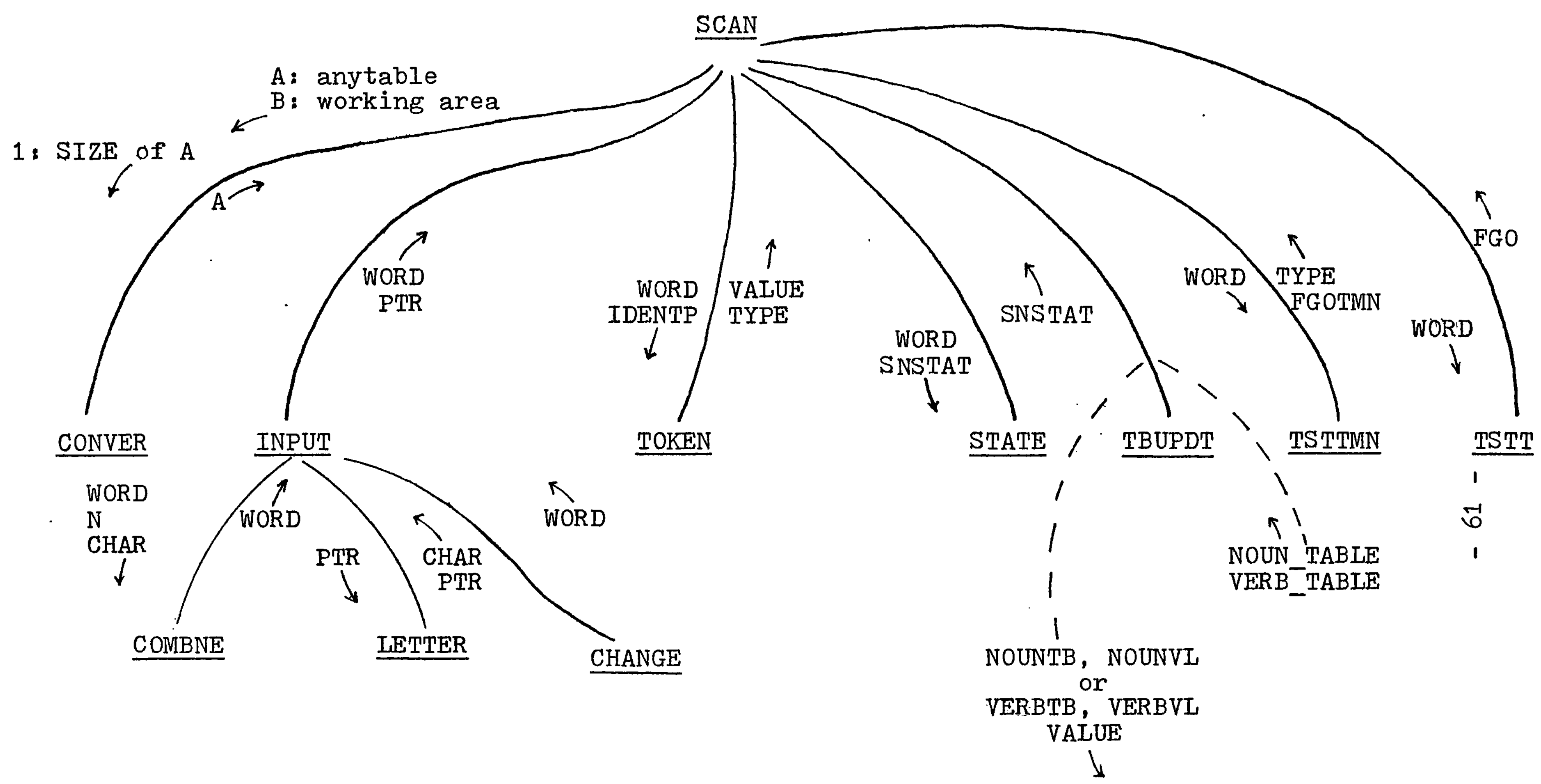


Figure 4.2b - CALLING STRUCTURE OF SCAN

PLP implementation is to construct the semantic functions for the PL grammar. In other words, the Skeleton parser should pass information regarding the derivation sequence to the semantic routines. In reality the derivation sequence is a node by node representation of the parse tree. In this thesis, the auther uses synthesized attribute values to evaluate the attribute of each node (token). An illustration and discussion of the implementation of PL semantics will be presented in this section.

For each interior node of a parser tree, there is an associated production. The semantics is called by Skeleton Parser whenever a production is about to be applied, i.e., the application of a production is the reduction of one or more symbols to a single non-terminal symbol. The parser passes information to semantics by three arguments used in the call statement - action, token-type, token-value. The action argument is the value of the production that the parser is ready to apply. The token-type and token-value contain the same information described in the previous section (4.2). These two arguments are those of the last variable read by the scanner. The other useful information such as "noun table", "verb table" are referenced and modified in both the scanner and the semantic routines. The transfer of information is done by using a FORTRAN COMMON statement. The function of semantics is to analyse the parse tree and generate a variety of messages for validating the source PL program. These messages are listed below together with a brief description of their characteristics.

1) Cross Reference Tables -

   Tables indicating variable in the PL program consisting of :

   a) A list of declared nouns together with the statement numbers where they appear.

   b) A list of undeclared nouns together with the statement numbers where they appear.

   c) A list of verbs together with the statement numbers where they appear.

2) Path Expression (PE) -

   A list of synthesized PE attributes for the declared and undeclared nouns. The symbols used in PE are :

   a) U <number> : variable is declared in line <number> in the <introduction>.

   b) R <number> : variable is referenced in line <number>.

   c) D <number> : variable is defined in line <number>.

   d) (--_---)* : variable is used zero or more times within a while loop as described by the parenthesized sequence of symbols. Variable is referenced in a test, at least once, as described by symbols on the left of the underscore.

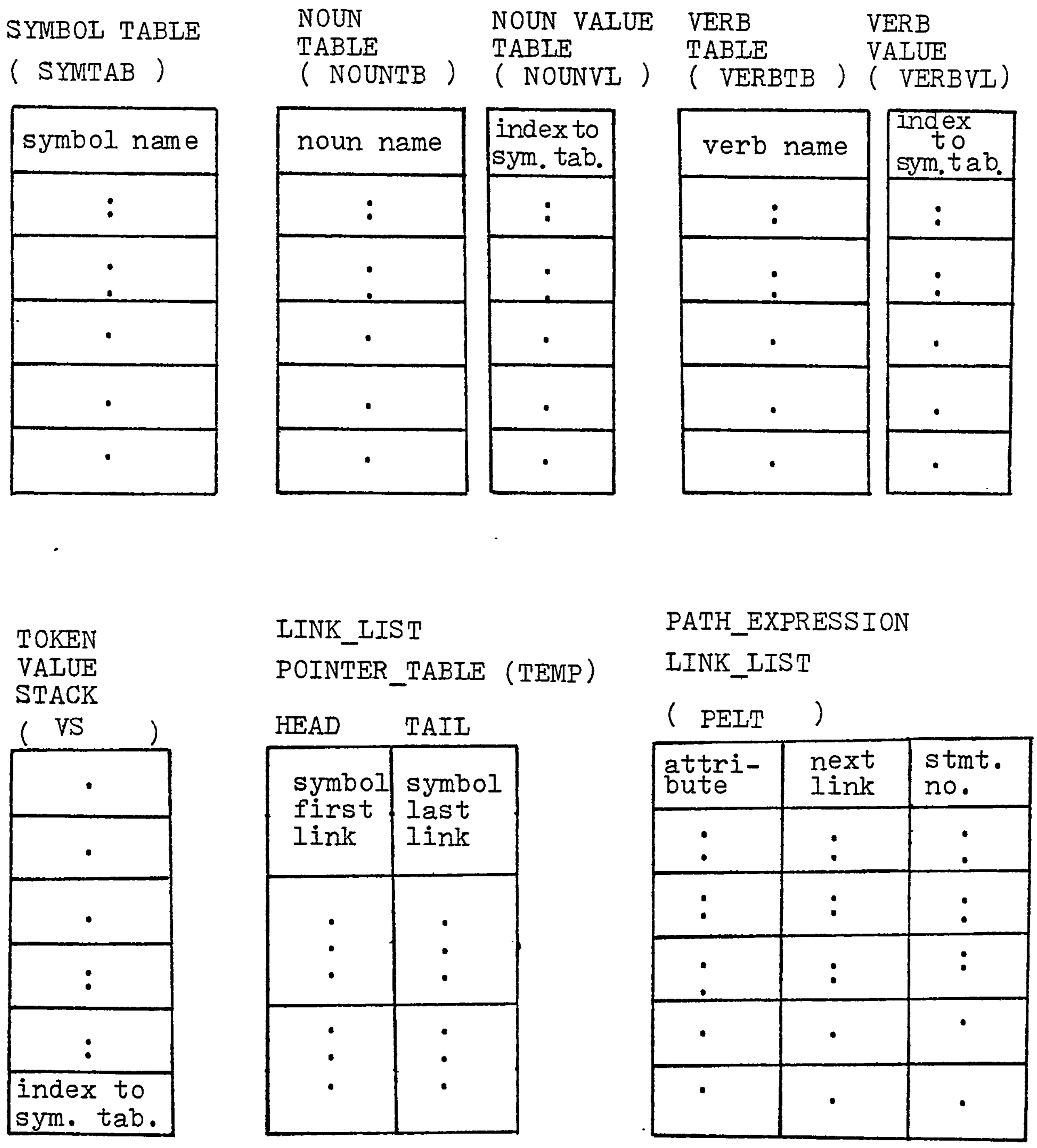
e) (---) # : variable is used within a repeat as described by the parenthesized sequence of symbols.

f) ---+--- : variable has alternative usage depending on the execution.

g) (---)↑ : variable is referenced unlimited number of times within a cycle as described by the parenthesized sequence of symbols.

The semantics program is shown in Appendix D. Note that all actions taken by semantics are dependent on the value of "action". The implementation details of semantics program is discussed as below. The main data structures of semantics program is shown in Figure 4.3a. The "symbol table", "noun table", and "verb table" are used to build up the cross reference information for all nouns and verbs. The "symbol table" is a buffer which contains the variables such as nouns, verbs, and files. In other words, the "symbol table" is just like a dictionary and contains all information needed, through the entire PLP, about the source PL program. The "noun table" and "verb table" are just subsets of "symbol table". "Verb table" is used to construct the verb directory file for use in the "file management of procedures" [L]. However, the "noun value table" (NOUNVL) and "verb value table" (VERBVL) are the index table for nouns and verbs.

Figure 4.3a - DATA STRUCTURES OF SEMANTICS



Each index points to the symbol table entry where the symbol is located. The other data structure that is useful for the semantic routine is the path expression (PE) consisting of three components:

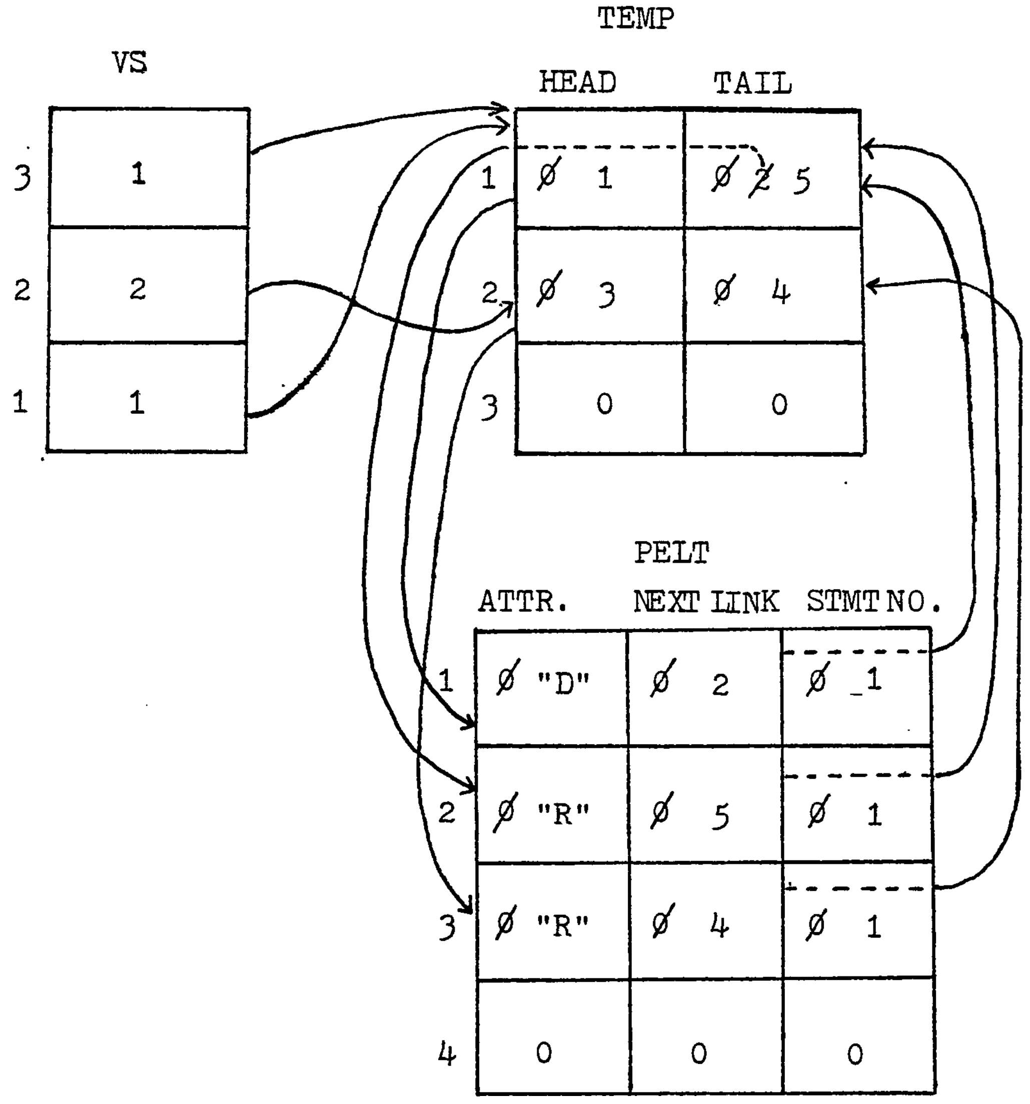
1) A value_stack (VS) - a one-dimensional array - with each value in the stack treated as the entry point to the linked list pointer table. Actually, stack VS is an index directory.
2) A linked_list_pointer_table (LLPT) - a two-dimensional array in which the first field contains the symbol first index (HEAD), while the second field contains the symbol last index (TAIL).
3) A path expression linked list (PELT) - a three-dimensional array in which the first field contains the attribute for each token, while the second field contains the value for the next available location in PELT (next link), and the third field contains the statement number for the token in the first field.

Figure 4.3b shows the basic chain-relation between the data structures. This figure also shows a statement of PL program read by the scanner. On reading X, the scanner translates it to 1 - the position in the symbol table. During the parsing phase, a call is issued to the semantics while production 116 (PRIMARYPR → IDENTIFIER*) is applied.

Figure 4.3b - CHAIN RELATION BETWEEN THE DATA STRUCTURES OF SEMANTICS

INPUT STRING :

1) X = Y - X ; $\longrightarrow$ 1 = 2 - 1 ;



The semantic action is to push the token-value into the value stack (VS). Using the same procedure, the token-value 2 (value for Y) is also pushed onto the VS. When the symbol ";" is read, the parser applies production 71 to produce "ASSIGNIMENT", then transfers control to semantics. The corresponding semantic function is to construct the attribute "D" (define) for the bottom token stacked in the VS and attribute "R" (reference) for other token(s) stacked in the same VS. Therefore, the attributes for statement "X = Y - X ;" should be PE(X) = DR, and PE(Y) = R. The attributes for these two tokens are moved to a linked list (PELT) for later use.

The algorithm for constructing the path expression is:

```
begin
    while VS is not empty do
  1. pick up the "TOP" token-value (V) from VS;
  2.       if the content of HEAD(V) is 0,
             then
                 a. put the "next link" onto HEAD(V);
                 b. put the appropriate attribute onto the
                    first field of PELT(HEAD(V));
                 c. put the statement number onto the third
                    field of PELT(HEAD(V));
                 d. get another "next link" in PELT;
                 e. put the new "next link" onto the second
                    field of PELT(HEAD(V));
```

```
            f. put the new "next link" onto TAIL(V).
        else
            a. put the appropriate attribute onto
               the first field of  PELT(TAIL(V));
            b. put the statement number onto the third
               field of  PELT(TAIL(V));
            c. get another "new link" in PELT;
            d. put the "new link" onto the second
               field of PELT(TAIL(V));
            e. put the "new link" onto TAIL(V).
    3. TOP = TOP - 1.
end
```

The detailed semantic functions corresponding to each action are shown below.

ACTION	SEMANTIC
1	a. Build up and print out the cross reference table for declared nouns;
	b. Build up and print out the cross reference table for undeclared nouns;
	c. Build up and print out the cross reference table for all verbs.
	e. Print out the path expression for each declared noun;
	f. Output the path expression and the name of each declared noun to a temporary disc file.

```
       2            Set up the starting pointer for undeclared
                    nouns.

  18 or 19          if flag is ture
                    then  build up the attribute "U" for all
                          token which are already stacked in
                          the value_stack (VS).
                    else
                          turn on flag.

      23            Build up the attribute "D" for all token
                    which already stacked in the VS.

  25 or 27       a. if  scan_state equals 3
                    then  update the file table.
                    else
                          update the noun table.
                 b. Stack the current token_value into the VS.

  29,30,31,32       if  production 101 has been applied
                    then  insert the underscore symbol "_"
                          into the link list of path expression
                          (PELT).  And turn off the production
                          number.
                    else
                          if  production 112 already has been
                              applied
                          then  turn off production number.
```

45	Build up the attribute "D" for all tokens which are already stacked in the VS.
46	Build up the attribute "R" for all tokens which are already stacked in the VS.
49	Same as Action 45.
50	<u>if</u> comments_flag is false <u>then</u> turn on the comments_flag.
51	Insert the minus sign "-" into the PELT.
52	Same as Action 46.
56,59	<u>if</u> the flag of production 95, 101, 108, 112, 118 is on <u>then</u> build up the attribute "R" for all tokens which are already stacked in the VS.
71	Build up attribute "D" for the first token stacked in the VS and attribute "R" for the rest.
86	Stick two ")" symbols into the PELT.

87 a. Stick two symbols, "(" into the PELT.

b. Turn off the flag of any production number.

88 Turn on the flag of production number 88.

90 a. Stick three symbols, ")", "+", "(" into the PELT.

b. Build up the attribute "D" for all token(s) which are already stacked into the VS.

94 a. Stick "(" symbol into the PELT.

b. Turn on the flag of production number 101.

96 Turn on the flag of production 96.

97 Stick two symbols, ")", "↑" into the PELT.

98 Stick "9" symbol into the PELT.

99 a. Stick two symbols, ")", "#" into the PELT.

b. Turn off the flag of production number.

100 Same as Action 105.

101 Turn on the flag of production number 108.

102 Same as Action 46.

105	Turn on the flag of production number 105.
106,108	Same as Action 94.
109	Same as Action 97.
110	a. Stick two "(" symbols into the PELT. b. Turn off the flag or production number.
111	Turn on the flag of production number 118.
116	Stack current token_value into the VS.
118	Same as Action 71.
120	Same as Action 116.

Finally, the call structure of routines called by SEMNTC is shown in Figure 4.3c.

4.4 PE (Path Expression) Analyser

The last part presented in this thesis is the analysis of the data flow. In other words, an analyser has been implemented for examining the PE for each declared noun to indicate the possible errors known as data flow anomalies.

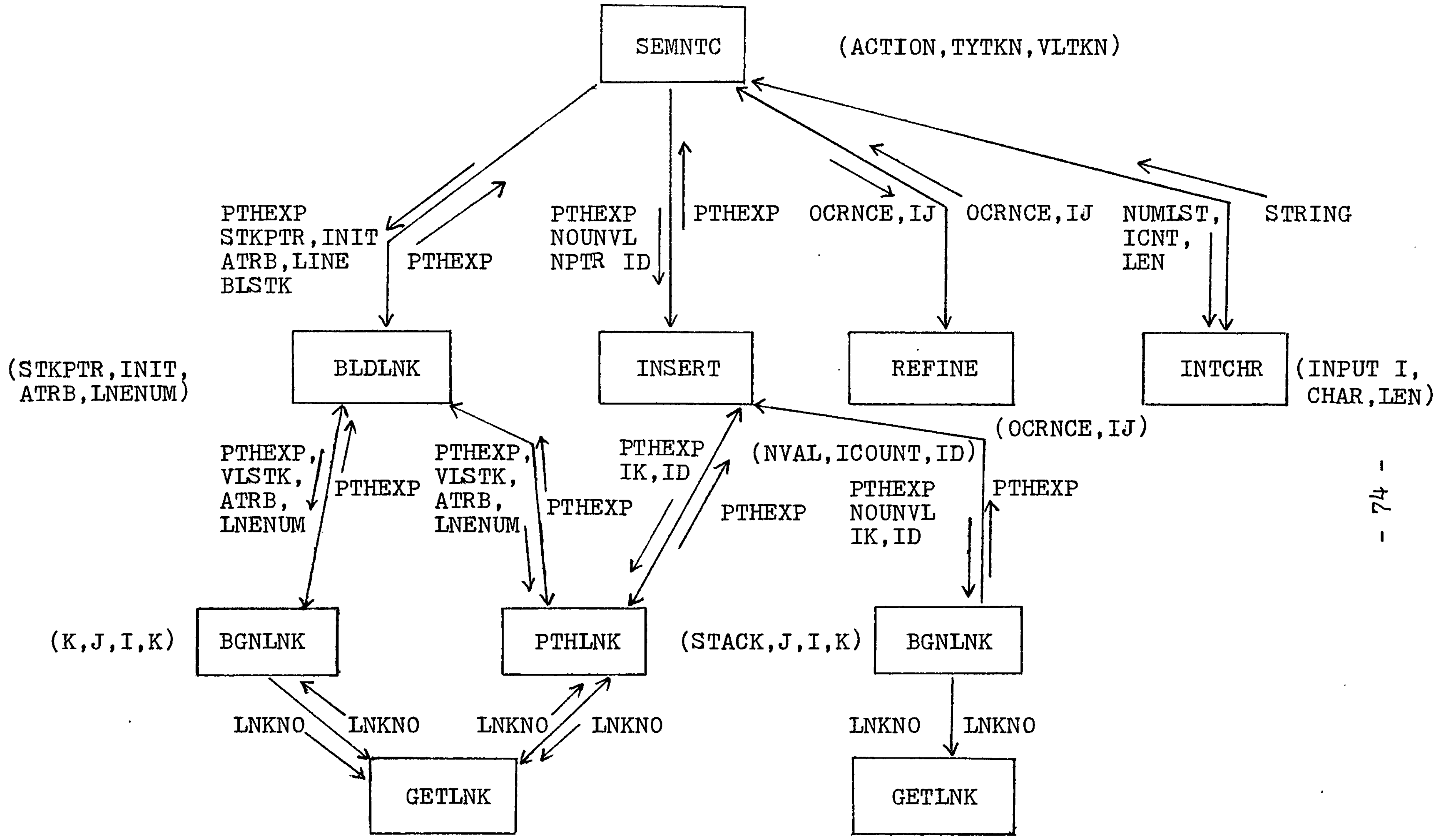


Figure 4.3c - CALLING STRUCTURES OF SEMANTICS

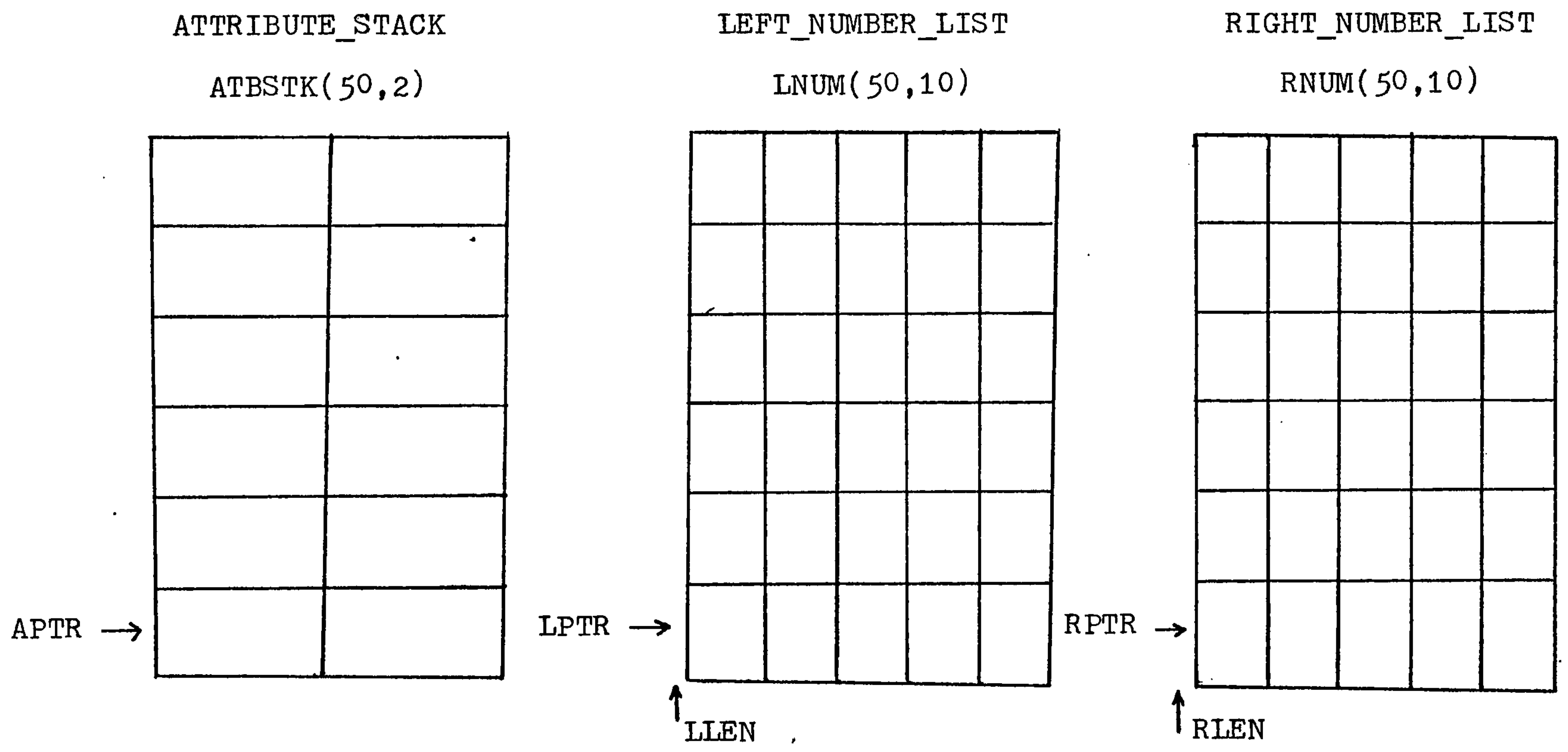
The PE Analyser is just like another independent processor which has its own parser, scanner, and semantic actions. As is usual, a context-free grammar which can accept all possible path expressions must be defined in advance, and run through the Parser Generator to generate a parsing table. A grammar for accepting the PE is shown in Appendix E. Again, a scanner is used to scan the input string (the path expression for each declared noun generated by the Semantics in the previous execution pass) from a disk file, and return the proper information to the Skeleton Parser. The parsing actions are the same as was mentioned in the previous section (4.1). The Semantics for the PE Analyser implements the algorithm to detect the data flow anomalies. In this section, the author will concentrate on the development of Semantics.

The semantics program for PE Analyser is shown in Appendix F. The functions of Semantics is to analyse the parse tree and detect various anomalies of PE associated with the parse tree. When an anomaly is found, information is printed out displaying what type (DD, UR, etc.) of anomaly has occured, also which statement(s) caused them. In order to determine this information, the Semantics has to compute the value of two attributes - PE and Statement number associated with each parse tree node. The detailed implementation work for evaluating those attributes will be discussed later. Here, several basic data structures of PE Analyser are shown in Figure 4.4a, where

1) attribute_value_stack (ATBSTK) is a two-dimensional array which is used to store all the values of the PE attributes; each value in this stack represents the PE for variables at a specific node; symbol "APTR" is used as the stack pointer.
2) left_number_list (LNUM) is a multiple-dimensional array which contains the statement number(s) of left PE attribute for each variable; symbol "LPTR" is used as the table pointer, while "LLEN" is used to indicate the length for those left PE attributes.
3) right_number_list (RNUM) is the same definition as those of LNUM unless RNUM is used for the right PE attributes.
4) while_relation_flag_stack (WLREFG) is a one-dimensional array which is used to indicate if a "while relation"is encountered in the different "nesting" levels.
5) counter (COUNT) is a one-dimensional array which is used as a table to bookkeep the parenthesis within a "WHILE" loop.

Note that the size of these stacks and tables can be adjusted by changing the size parameters.

Figure 4.4a - DATA STRUCTURES OF PE ANALYSER



Now we examine the evaluation of the PE attributes. There are four different types of anomalies to be dectected by the Analyser. The anomalies are:

- UU : declared and declared again
- UR : referenced without defined
- DD : defined and redefined
- defined but never referenced

First, consider a simple example -

```
1) X = 1 ;
2) Y = 1 ;
3) X = Y + 2 ;
```

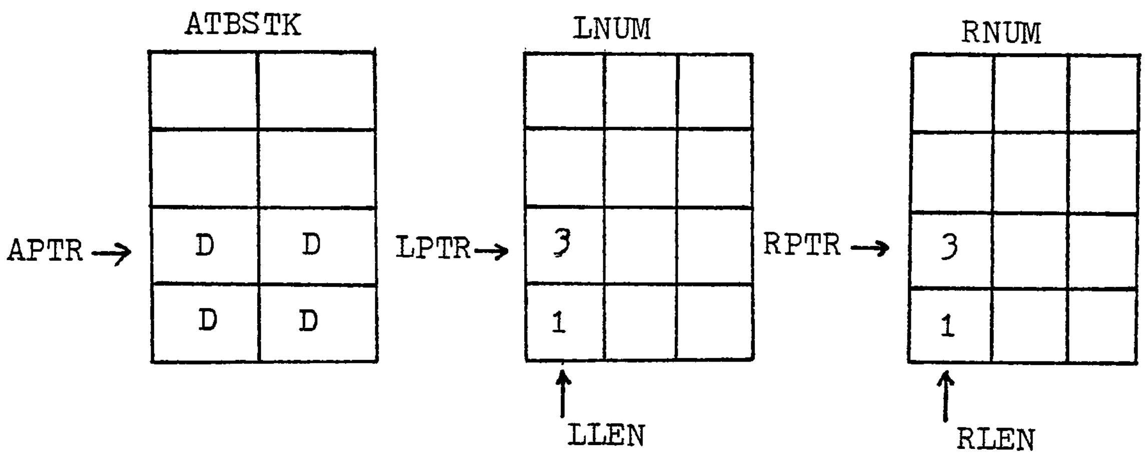
The path expression, PE(X) is D1D3. An obvious DD anomaly occurs in this block of code. The error is detected at the time the first two attributes collapsed to one attribute. However, an example 1) X = 1 ; 2) Y = 1 ; 3) X = X + 2 ; does not have the DD anomaly, because of an intervening reference to X. The semantic steps for detecting those anomalies are given as follows:

```
begin
   1. initialize all stacks, tables, and pointers;
   2. if ACTION = 8, 9, or 10
      then
         stack the current attribute value onto ATBSTK;
         // Note that the attribute value was dispatched
         // into two field of ATBSTK.
         stack statement number onto LNUM;
         stack statement number onto RNUM;
```

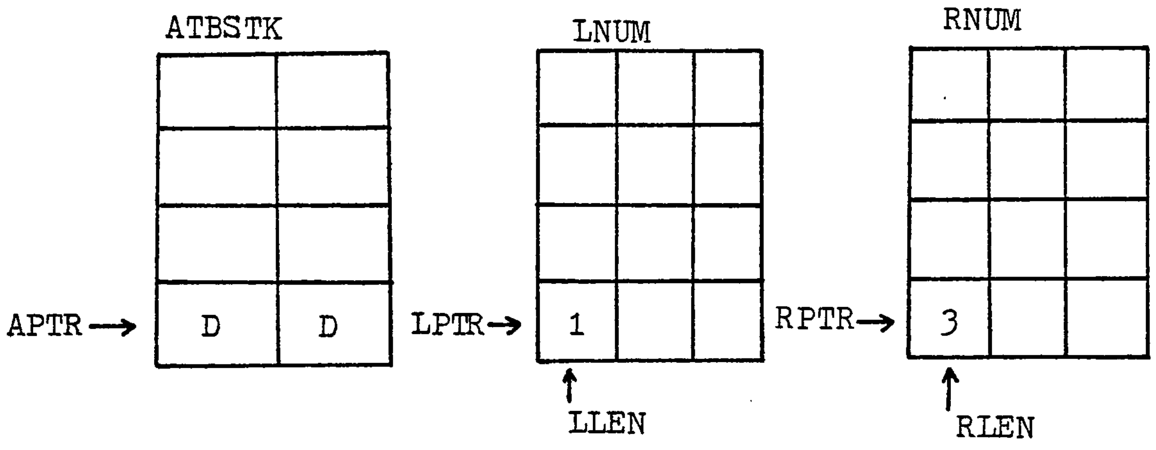
```
  3. if  ACTION = 2
     then
          a. set up the left attribute (LATB) ;
             set up the right attribute (RATB) ;
             // This semantic routine always deal with the
             // top two attributes in the ATBSTK.  Since
             // the parser parses input string from left
             // to right, so the lower attribute is LATB,
             // the upper attribute is RATB.
          b. build up the statement number list for LATB
             and RATB ;
             // This information is used for the error
             // analysis routine.
          c. employ the error analyse routine ;
             // This routine examines the LATB and RATB
             // to indicate the possible errors for asso-
             // ciated node.
          d. employ a collapsing routine ;
             // This routine combine the LATB and RATB,
             // the result  is pushed back onto ATBSTK.
          e. re-initialize all stacks, tables and pointers.
end
```

Using the example mentioned above (X = 1; Y = 1; X = Y + 2;) and applying the above steps, a DD anomaly will be detected automatically. The resulting stack for steps 2 and 3 is shown below.

PE STACK FOR STEP 2 :

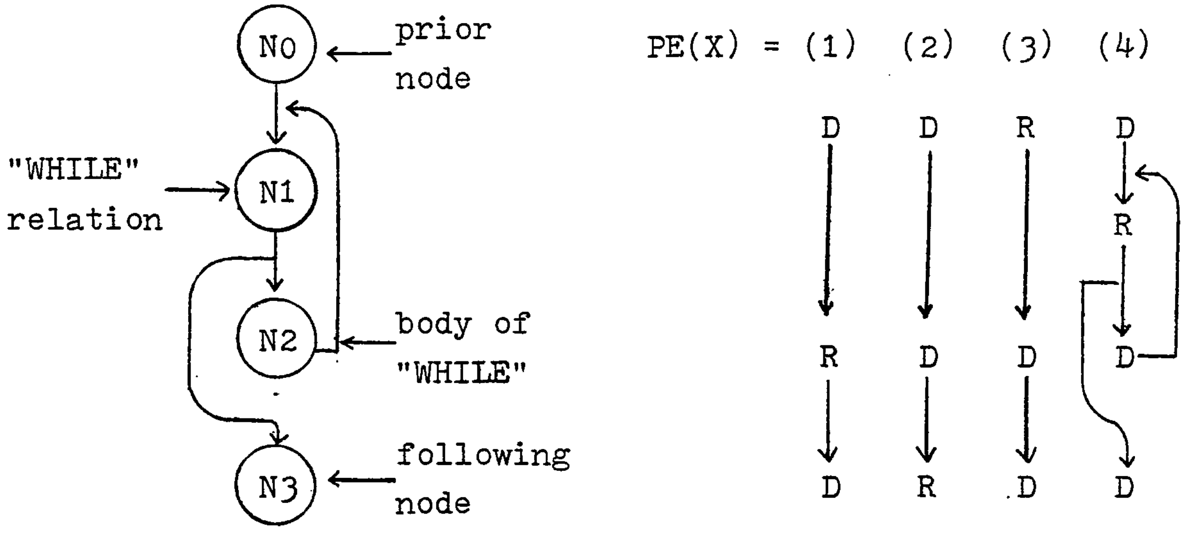


PE STACK FOR STEP 3 :



The meaning of the figure for step 3 is that node X is first defined at statement 1 and last defined at statement 3. In step 2.d a collapsing routine was used to propagate the attributes in the stack. Further discussion will be given in later of this section. However, the error analysis routine is quite simple and straightforward. The program for the error analysis routine is shown in Appendix F.

The analysis of attribute is relatively easy and straightforward as long as there are no loops and branches. However, with the "IF", "WHILE", "CASE" and "REPEAT" statements, the analytical work become more complicated. The discussion here starts with the "WHILE" statement. First consider the flow graph representing a "WHILE" loop:



Note that the variable X can be either in N_0, N_1, N_2, or N_3.

Here, N0 represents all statements before the "WHILE" loop. N1 represents the codes for "WHILE" relation. N2 represents the statements in the body portion of "WHILE" loop. N3 is the node following the "WHILE" loop. Each of these nodes has a set of PE attribute values. Consider four different cases of PE values for a variable X in these nodes:

1) D (R) * D
2) D (D) * R
3) R (D) * D
4) D (R _ D) * D

The first case means that variable X is defined before and after the "WHILE" loop. It is also referenced in the body portion of the "WHILE" loop. Intuitively, one would say no possible anomaly exists, because if X is defined first at N0 then referenced at N2 and defined at N3. However, if the "WHILE" relation was false then X would be defined at N0 and then again N3 without intervening reference. This is a DD anomaly. Now, try another approach - ignore the body portion of "WHILE" loop. Consider the second case, if the attributes of N2 are ignored, then a DD anomaly does not exist. Same suitation will happen to case 3.

The solution to the "WHILE" loop lies in considering what attribute values should be computed for the variables used in the body and relation part of "WHILE" loop. The algorithm shown below can handle the cases we just mentioned.

Algorithm for Evaluating "WHILE" Body Attributes

Assume $PE_C(X)$ is the attribute for variable X in the "WHILE" body. PE_{CL} represents the left PE attribute stacked in the ATBSTK for X. $PE_{CR}(X)$ represents the right PE attribute stacked in the ATBSTK for X.

```
begin
        if  ACTION = 12 or 13
        then
              if  $PE_{CR}(X)$ = " R "
              then  $PE_{CR}(X)$ = " ␢ "
end
```

Note that blank (" ␢ ") means empty attribute. This attribute allows " D " or " R " to override it during the parsing of attributes from left to right. Consequently, an algorithm for the node collapsing routine is shown below:

Algorithm for Node Collapsing

First, assume there are two nodes X_1, and X2 to be collapsed into node X_3. The PE attributes associated with each node are represented as PE_1, PE_2 and PE_3 repectively.

```
begin
    if    $PE_{1L}$  =  " ⊅ "
    then
          $PE_{3L}$  =  $PE_{2L}$ ;
    else
          $PE_{3L}$  =  $PE_{1L}$ ;
    if    $PE_{2R}$  =  " ⊅ "
    then
          $PE_{3R}$  =  $PE_{1R}$ ;
    else
          $PE_{3R}$  =  $PE_{2R}$ ;
end
```

An example would be the best illustration for this algorithm. Consider the first case : D (R) * D. As action equals 7 or 8 (See Appendix E - PE grammar), then the attribute "D" or "R" will be pushed onto the attribute stack. In other words, PE_{1L} = "D", PE_{1R} = "D", PE_{2L} = "R", PE_{2R} = "R". While action equal 12 or 13, then PE_{2R} equal "⊅". Collapsing routine is employed, when action is 2. This routine pops off the top two attributes from ATBSTK and causes the following result : PE_{3L} = "D", PE_{3R} = "D". Again as action equals 7,another "D" is pushed onto ATBSTK. Hence, a left attribute "D" and a right attribute "D" pass to error analysis routine when action 2 is reached. Therefore, a DD anomaly is detected. This is just what we want ! Case 2 and case 3 can be handled with the same approach. Now, consider the "WHILE" relation. That is whether or not the body portion of the "WHILE" loop is executed, the first and last statement executed is the relation expression. Hence, if a variable is referenced in relation expression before entering the "WHILE" body, thus it will be also referenced after the body is executed. Consider the case 4 : D (R_D) * D. This path expression means variable X was defined at prior node N_0 and referenced in the relation expression of "WHILE" loop (N_1), defined at body portion N_2, and again defined at the following node N_3. From the point of view of straight-line composition, it looks like a DD anomaly will occur as collapsing function takes place between N_2 and N_3. Actually, there is no such error. The philosophy to prevent this anomaly from happening is to stick a "R" attribute just after "WHILE" loop. In other words, the PE analyser needs to insert a "R" into the ATBSTK just before the next attribute is pushed into the stack. The following algorithm can correctly handle the "WHILE" relation construct.

Algorithm for Computing "WHILE" Relation Attribute

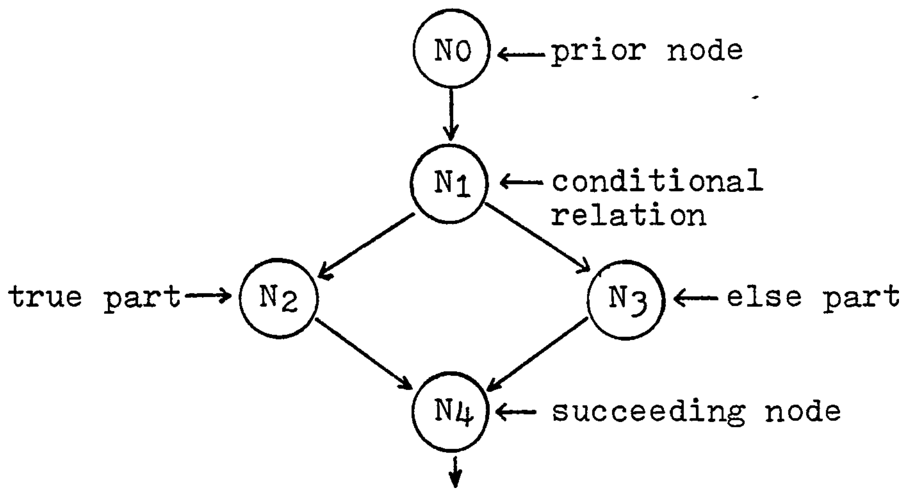
The definition of PE is the same as that described in the previous algorithm. "WLREFG" and "COUNT" were defined in the early discussion of this section. "I" is used as a flag counter.

```
begin
    if ACTION = 12
    then
        if WLREFG(I) = 1 AND COUNT(I) ≠ -1
        then
            $PE_{2L} = PE_{1L}$ ;
            $PE_{2R} = PE_{1L}$ ;
end
```

Noting that "WLREFG(I) = 1" means the "WHILE" relation expression has been encountered for a "WHILE" nesting level. "COUNT(I) $\neq$ -1"means a appropriate position in the attribute stack <u>is ready</u> for sticking in a "R" attribute for the corresponding "WHILE" nesting level. The beauty of using the technique of flag-stack is that the flag-stack can handle a group of nested "WHILE" loop. In other words, this flag-stack can accurately point out which level of "WHILE" relation expression is encountered or past. Now, apply this algorithm to illustrate case 4. $PE_{WHILE}(X)$ will equal "RR" The PE for N_0 is "DD". By collapsing N_0 with N_{WHILE}, PE(X) equal "DR". Again, collapse this attribute with the attribute of N_3 ("DD"). Therefore, the error analyse routine takes the left attribute (LATB) as "R", and right attribute (RATB) as "D", so there is no anomaly.

The approach for the "IF" statement is quite similar to that of "WHILE" statement. A flow graph for "IF" is given as follows:

"IF" Flow Graph



N_0 represents all codes before "IF".

N_1 represents the codes for conditional relation.

N_2 represents the codes for ture part of "IF" body.

N_3 represents the codes for else part of "IF" body.

N_4 represents the codes following the "IF".

Consider the following PE types for a variable X in these nodes:

1) D (D + R) R

2) D (D + D) R

3) R (D + R) D

4) R (D + D) D

The first case represents a variable X defined before the "IF" and then redefined in the true part of "IF" , while second case means that X is defined before and redefined in both the true and else parts. Case 4, and Case 3 are quite similar to Cases 1 and 2, except they are defined in the "IF" body and again defined in the following nodes. The algorithm given below computes the "IF" attributes.

The central idea in this algorithm is to calculate the occurrance of "D" attribute within the "IF" body portion. Actually, the algorithm gives the details for another special collapsing routine. As before,

$PE_1(X)$ represents the attributes for X in first node.
$PE_2(X)$ represents the attributes for X in second node.
$PE_3(X)$ represents the attributes for X after being collapsed of N_1 and N_2.

```
if  ACTION = 4  then
begin
        if   PE1L  =  "D"
        then
             PE3L  =  "D" ;
        if   PE2L  =  "D"
        then
             PE3L  =  "D" ;
        if   PE1R  =  "D"
        then
             PE3R  =  "D" ;
        if   PE2R  =  "D"
        then
             PE3R  =  "D" ;
end
```

Now, examine the algorithm to illustrate ase 2, the PE of "IF" body is equal to "DD" and the PE for NO is "DD". Collapse these two attributes to automatically detect a DD. Besides, a list of statement number(s) is generated to point out where the "error" occurred. For example, if the anomaly of case 2 was detected, then the numbers are printed out to indicate that the DD anomaly occurred either in the true part or in the else part.

The approach and solution to the "CASE" statement are just the same as those of "IF" statement, so no detailed discussion will be given.

Since the execution path of "REPEAT" loop will go through the body portion and relation at least once before exiting the loop, this construct is treated as those of "Straight-line" composition.

Finally, the functions employed by each action is described as follows:

ACTION	SEMANTIC FUNCTION
1	a. *if* the right node of attribute_stack (ATBSTK) is "D" *then* print out the message of "Defined before and never referenced".
	b. Initialize all tables and pointers.

2	a. Set up the left attribute and right attribute for ERROR routine. b. Call error analyse routine. c. Call collapse routine.
4	a. Build up the left attribute and right attribute for first node and second node. b. Build up "D" attribute and corresponding line number for third node. c. Initialize all tables and pointers.
7	Turn on the flag of while_relation for each occurance (WLREFG(INT)).
8,9,10	Stack the current token value into ATBSTK.
11	Stack the current token_value into number stack (LNUM,RNUM).
12,13	a. <u>if</u> the flag of while_relation is on, <u>then</u> insert a attribute "R" before next attribute is pushed into the ATBSTK. b. Rearrange the number list for new attribute. c. Call collapse routine. d. Turn off the flag of while_relation.

```
16          if   the flag of while_relation is on,
            then increment the parenthesis counter
                 (COUNT(INT)).

19          if   the flag of while_relation is on,
            then decrement the parenthesis counter.
```

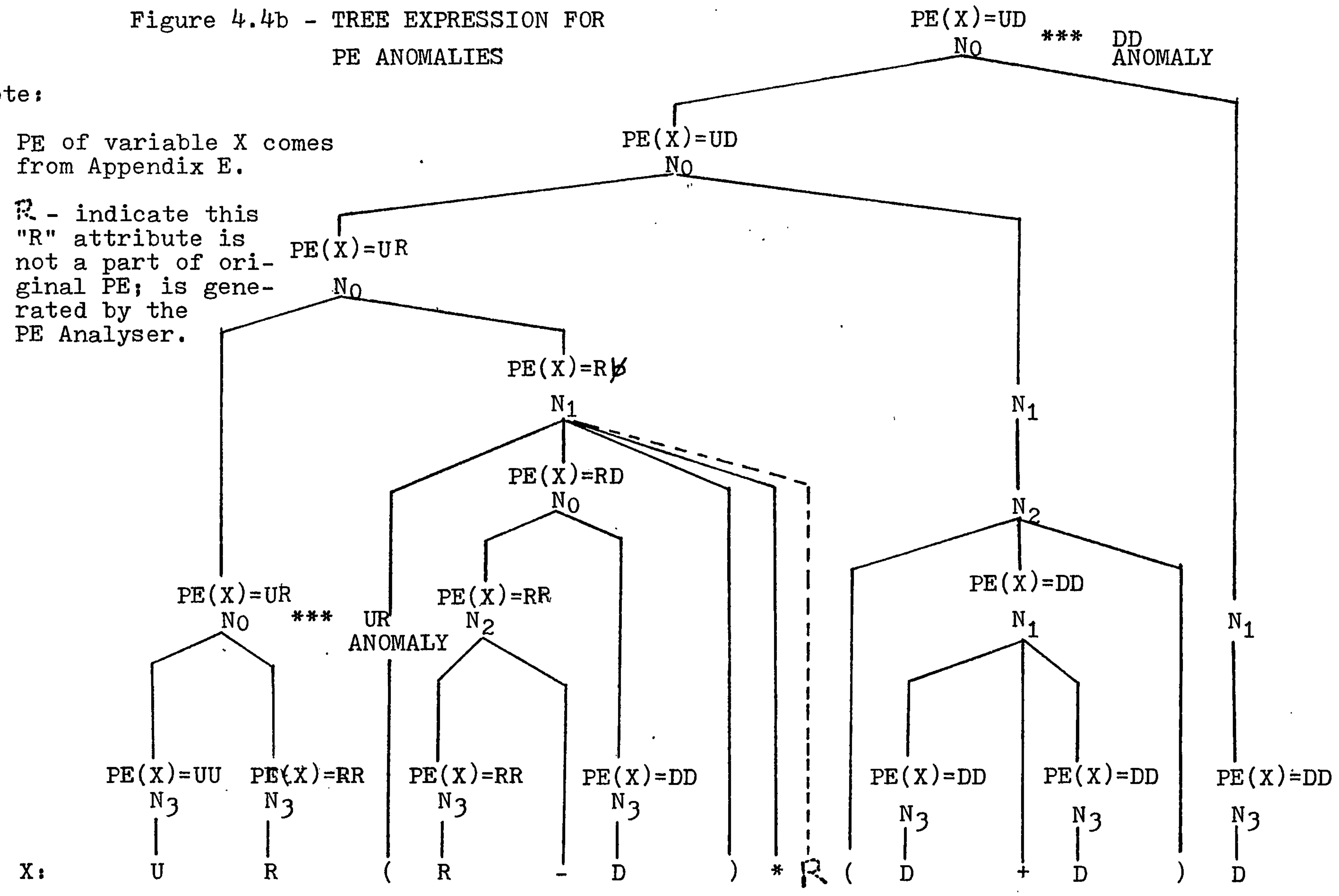
A list of all possible anomalies for the input source program (See Appendix E) is given in Appendix H. Figure 4.4b illustrates the path expression attributes for a specific program.

Figure 4.4b - TREE EXPRESSION FOR PE ANOMALIES

Note:

1) PE of variable X comes from Appendix E.

2) R - indicate this "R" attribute is not a part of original PE; is generated by the PE Analyser.



Chapter 5
CONCLUSION

The major expense in developing computer systems is in writing software - software costs are expected to rise even further. By 1985, computer software expenses will constitute about ninety percent of the total system cost[B1]. However, the cost of finding an error in software increases as the software development comes nearer to completion. Errors found during the early stages of design and specification are relatively inexpensive to correct as compared with errors found during total system integration [R]. Another factor resulting the cost of large systems is the problem of communication between the different programmers in a team.

In this thesis, the author has presented a design language, called Pseudo Language (PL), which improves communications between programmers and thereby improves the chances of detecting errors. The reason is that

1) PL programs resemble Pidgin English, and

2) PL encourages top-down, structured design practices.

Programs written in PL are called program forms. Program forms avoid implementation details and are therefore easily readable. PL also forces the programmer to identify the control structures as well as the functional components of the program system during the design phase [RB].

A Pseudo Language Processor (PLP) has also been presented. This processor is an automatic tool for analysing a PL program and print out messages that include

1) a list of nouns and verbs together with the statement numbers where they appear.
2) a list of path expression for nouns.
3) a list of self explanatory warning messages of certain conditions detected by the PLP for nouns.

These message are used to indicate the violations of good design practices and possible errors in the source program.

In the future, PLP can also be designed as an interactive system which aids program form validation and implementation program synthesis. The variety of messages that can be generated more than those described in thesis. However, the techniques for generating all messages will be the same as described in this thesis. Another important research effort should be in the automatic translation of PL programs to the current implementation languages (FORTRAN, PASCAL, etc.) All this work is certainly possible. Software engineering, however, is quite a new area in computer science

Lot of research work still needs to be done in this field in developing tools useful in the early stages of design.

APPENDIX A

PL GRAMMAR

```
/*       1   PROGRAM : INTRODUCTION STAT, COMPOUND STAT                            */
/*       2   INTRODUCTION STAT : START, INTRODUCTION, FINISH                       */
/*       3   START : BEGIN+INTRO                                                   */
/*       4   FINISH : END+INTRO                                                    */
/*       5   INTRODUCTION : EXPRPR, FILENAME, I/O, DICTIONARYS                     */
/*       6   FILENAME : FILE+KEY, EXPRPR                                           */
/*       7            ; EMPTY                                                      */
/*       8   FILE+KEY : FILES                                                      */
/*       9   I/O : INPUT, OUTPUT                                                   */
/*      10   INPUT : INPUT+KEY, EXPRPR                                             */
/*      11         ; EMPTY                                                         */
/*      12   INPUT+KEY : INPUT+PARAMETERS                                          */
/*      13   OUTPUT : OUTPUT+KEY, EXPRPR                                           */
/*      14          ; EMPTY                                                        */
/*      15   OUTPUT+KEY : OUTPUT+PARAMETERS                                        */
/*      16   DICTIONARYS : DICTIONARY+KEY, EXPRPR                                  */
/*      17   DICTIONARY+KEY : DICTIONARY                                           */
/*      18   EXPRPR : EXPRPR, PRIMARYPR, ;                                       */
/*      19          ; PRIMARYPR, ;                                               */
/*      20   PRIMARYPR : STRU+NOUN                                                 */
/*      21             ; PRIMARYPR, OPERATOR, STRU+NOUN                            */
/*      22             ; PRIMARYPR, STRU+NOUN                                      */
/*      23             ; PRIMARYPR, INITIAL PART                                   */
/*      24   INITIAL PART : INITIAL, STRU+NOUN                                     */
/*      25   STRU+NOUN : NOUN*                                                     */
/*      26             ; NUMBERPR*                                                 */
/*      27             ; NOUN*, (, PRIMARYPR, )                                    */
/*      28   COMPOUND STAT : LEFT PAREN, STAT LIST, RIGHT PAREN,                   */
/*      28                 ;                                                     */
/*      29   LEFT PAREN : DO                                                       */
/*      30              ; COBEGIN                                                  */
/*      31              ; BEGIN                                                    */
/*      32              ; [                                                        */
/*      33              ; (                                                        */
/*      34   RIGHT PAREN : OD                                                      */
/*      35               ; COEND                                                   */
/*      36               ; END                                                     */
/*      37               ; ]                                                       */
/*      38               ; )                                                       */
/*      39   STAT LIST : STAT LIST, STAT                                           */
/*      40             ; STAT                                                      */
/*      41   STAT : LABEL, STAT                                                    */
/*      42        ; CONTROL STAT                                                   */
/*      43        ; COMMAND                                                        */
/*      44   COMMAND : ASSIGNMENT                                                  */
/*      45           ; READ, EXPR, ;                                             */
/*      46           ; PRINT, EXPR, ;                                            */
/*      47           ; VERB PART, RETURN PART, ;                                 */
/*      48           ; VERB PART, ;                                              */
/*      49           ; WRITE, EXPR, ;                                            */
/*      50   VERB PART : VERB CLAUSE, COMMENTS                                     */
/*      51   VERB CLAUSE : VERB*                                        */
/*      52   RETURN PART : RETURN+KEY, COMMENTS                         */
/*      53   RETURN+KEY : RETURN                                        */
/*      54   COMMENTS : COMMENTS, GARBAGEPR                             */
/*      55              ; GARBAGEPR                                     */
/*      56   EXPR : PRIMARY                                             */
/*      57        ; OPERATOR, PRIMARY                                   */
/*      58        ; EXPR, OPERATOR                                      */
/*      59        ; EXPR, PRIMARY                                       */
/*      60   CONTROL STAT : CASE STAT                                   */
/*      61                ; WHILE STAT                                  */
/*      62                ; FOR STAT                                    */
/*      63                ; IF STAT                                     */
/*      64                ; CYCLE STAT                                  */
/*      65                ; REPEAT STAT                                 */
/*      66                ; EXIT STAT                                   */
/*      67                ; WITH STAT                                   */
/*      68                ; COMPOUND STAT                               */
/*      69                ; DO STAT                                     */
/*      70                ; CALL STAT                                   */
/*      71   ASSIGNMENT : EXPR, ASSIGNMENT SYMBOL, EXPR, ;            */
/*      72   ASSIGNMENT SYMBOL : :=                                   */
/*      73                     ; =                                      */
/*      74   OPERATOR : +                                               */
/*      75            ; -                                               */
/*      76            ; *                                               */
/*      77            ; /                                               */
/*      78            ; **                                              */
/*      79            ; ↑=                                              */
/*      80            ; >                                               */
/*      81            ; <                                               */
/*      82            ; <=                                              */
/*      83            ; >=                                              */
/*      84            ; .                                             */
/*      85            ; ,                                             */
/*      86   CASE STAT : CASE CLAUSE, UNITS, ENDCASE, ;               */
/*      87   CASE CLAUSE : CASE+KEY, EXPR, BEGINCASE                    */
/*      88   CASE+KEY : CASE                                            */
/*      89   UNIT : LABEL, STAT LIST, END, ;                          */
/*      90   LABEL : EXPR, :                                          */
/*      91   UNITS : UNITS, UNIT                                        */
/*      92         ; UNIT                                               */
/*      93   WHILE STAT : WHILE+KEY, EXPR, LEFT PAREN, BODY             */
/*      94   WHILE+KEY : WHILE                                          */
/*      95   FOR STAT : FOR+KEY, EXPR, LEFT PAREN, BODY                 */
/*      96   FOR+KEY : FOR                                              */
/*      97   CYCLE STAT : CYCLE+KEY, BODY                               */
/*      98   CYCLE+KEY : CYCLE                                          */
/*      99   REPEAT STAT : REPEAT+KEY, STAT LIST, UNTIL+KEY, EXPR,      */
/*      99               ;                                            */
/*     100   REPEAT+KEY : REPEAT                                        */
/*      101   UNTIL←KEY : UNTIL                                                  */
/*      102   EXIT STAT : EXIT, EXPR, ;                                        */
/*      103               ; EXIT, ;                                            */
/*      104   WITH STAT : WITH←KEY, EXPR, LEFT PAREN, BODY                       */
/*      105   WITH←KEY : WITH                                                    */
/*      106   IF STAT : IF CLAUSE, LEFT PAREN, BODY                              */
/*      107           ; IF CLAUSE, LEFT PAREN, BODY, ELSE PART                   */
/*      108   ELSE PART : ELSE←KEY, LEFT PAREN, BODY                             */
/*      109   ELSE←KEY : ELSE                                                    */
/*      110   IF CLAUSE : IF←KEY, EXPR, THEN←KEY                                 */
/*      111   IF←KEY : IF                                                        */
/*      112   THEN←KEY : THEN                                                    */
/*      113   BODY : STAT LIST, RIGHT PAREN, ;                                 */
/*      114   PRIMARY : LEFT PAREN, EXPR, RIGHT PAREN                            */
/*      115           ; NUMBER*                                                  */
/*      116           ; IDENTIFIER*                                              */
/*      117   DO STAT : DO1, DO LIST                                             */
/*      118   DO1 : DO, LABEL←KEY*, EXPR, ASSIGNMENT SYMBOL, EXPR,               */
/*      118         ;                                                          */
/*      119   DO LIST : STAT LIST, NUMBER*, CONTINUE←KEY, ;                    */
/*      120   GARBAGEPR : NOUN←GARBAGE*                                          */
/*      121   CONTINUE←KEY : CONTINUE                                            */
/*      122   CALL STAT : CALL, EXPR, ;                                        */
/*      123   COMMAND : RETURN PART                                              */
/*      124   STRU←NOUN : JUNK*                                                  */
/*      125   JOB : PROG←LIST, EOF SYMBOL                                        */
/*      126   PROG←LIST : PROG←LIST, DIR←BLOCK                                   */
/*      127             ; PROG←LIST, PROGRAM←PRIME                               */
/*      128             ; DIR←BLOCK                                              */
/*      129             ; PROGRAM←PRIME                                          */
/*      130   DIR←BLOCK : BEGIN←DIR, DIR←LIST, END←DIR                           */
/*      131   PROGRAM←PRIME : PROGRAM                                            */
/*      132   DIR←LIST : DIR←LIST, DIR←ARGS                                      */
/*      133            ; DIR←ARGS                                                */
/*      134   DIR←ARGS : DIRECTIVE, DIR←NAME, NOUN*, JUNK*, ;                  */
/*      135            ; DIRECTIVE, DIR←NAME, NOUN*, ;                         */
/*      136            ; DIRECTIVE, DIR←NAME, ;                                */
/*      137            ; OPT←ARGS, ;                                           */
/*      138   DIR←NAME : GET←VERB                                                */
/*      139            ; SAVE←VERB                                               */
/*      140            ; PRINT←VERB←LIB                                          */
/*      141            ; INIT←VERB←LIB                                           */
/*      142            ; DEL←VERB                                                */
/*      143            ; ZERO←USE                                                */
/*      144   OPT←ARGS : OPTION, OPT←LIST                                        */
/*      145   OPT←LIST : OPT←LIST, OPT←NAME                                      */
/*      146            ; OPT←NAME                                                */
/*      147   OPT←NAME : NO←STRUC                                                */
/*      148            ; PRINT←30                                                */
/*      149            ; COMMENTS←T                                              */
/*      150            ; NO←PATH                                                 */
```

APPENDIX B

VOCABULARY TABLE FOR PL GRAMMAR

```
      DATA ((VCBLRY(I,J),J=1,10),I=1,151)
     & / 2H( ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H) ,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H* ,2H  ,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H**,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2H+ ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H- ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H/ ,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H< ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2H  ,2H<=,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2H= ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H> ,2H  ,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H>=,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2H, ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2H  ,2H. ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H:=,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H: ,2H  ,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H; ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2HBE,2HGI,2HN ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HBE,
     & 2HGI,2HNC,2HAS,2HE ,2H  ,2H  ,2H  ,2H  ,2H  ,2HBE,2HGI,2HN←,2HDI,
     & 2HR ,2H  ,2H  ,2H  ,2H  ,2H  ,2HBE,2HGI,2HN←,2HIN,2HTR,2HO ,2H  ,
     & 2H  ,2H  ,2H  ,2HCA,2HLL,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2HCA,2HSE,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HCO,2HBE,2HGI,
     & 2HN ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HCO,2HEN,2HD ,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2HCO,2HMM,2HEN,2HTS,2H←T,2H  ,2H  ,2H  ,2H  ,
     & 2H  ,2HCO,2HNT,2HIN,2HUE,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HCY,2HCL,
     & 2HE ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HDE,2HL←,2HVE,2HRB,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2HDI,2HCT,2HIO,2HNA,2HRY,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2HDI,2HRE,2HCT,2HIV,2HE ,2H  ,2H  ,2H  ,2H  ,2H  ,2HDO,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HEL,2HSE,2H  ,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HEN,2HD ,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2H  ,2HEN,2HDC,2HAS,2HE ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2HEN,2HD←,2HDI,2HR ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HEN,2HD←,2HIN,
     & 2HTR,2HO ,2H  ,2H  ,2H  ,2H  ,2H  ,2HEO,2HF ,2HSY,2HMB,2HOL,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2HEX,2HIT,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2H  ,2HFI,2HLE,2HS ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HFO,2HR ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HGE,2HT←,2HVE,2HRB,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2HID,2HEN,2HTI,2HFI,2HER,2H* ,2H  ,2H  ,
     & 2H  ,2H  ,2HIF,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HIN,
     & 2HIT,2HIA,2HL ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HIN,2HIT,2H←V,2HER,
     & 2HB←,2HLI,2HB ,2H  ,2H  ,2H  ,2HIN,2HPU,2HT←,2HPA,2HRA,2HME,2HTE,
     & 2HRS,2H  ,2H  ,2HJU,2HNK,2H* ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2HLA,2HBE,2HL←,2HKE,2HY*,2H  ,2H  ,2H  ,2H  ,2H  ,2HNO,2HUN,2H* ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HNO,2HUN,2H←G,2HAR,2HBA,2HGE,
     & 2H* ,2H  ,2H  ,2H  ,2HNO,2H←P,2HAT,2HH ,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2H  ,2HNO,2H←S,2HTR,2HUC,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HNU,2HMB,
     & 2HER,2H* ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HNU,2HMB,2HER,2HPR,2H* ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2HOD,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2HOP,2HTI,2HON,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HOU,
     & 2HTP,2HUT,2H←P,2HAR,2HAM,2HET,2HER,2HS ,2H  ,2HPR,2HIN,2HT ,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HPR,2HIN,2HT←,2HBO,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2H  ,2HPR,2HIN,2HT←,2HVE,2HRB,2H←L,2HIB,2H  ,2H  ,2H  ,
     & 2HRE,2HAD,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HRE,2HPE,2HAT,
     & 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HRE,2HTU,2HRN,2H  ,2H  ,2H  ,
     & 2H  ,2H  ,2H  ,2H  ,2HSA,2HVE,2H←V,2HER,2HB ,2H  ,2H  ,2H  ,2H  ,
& 2H  ,2HTH,2HEN,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HUN,2HTI,
& 2HL ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HVE,2HRB,2H* ,2H  ,2H  ,
& 2H  ,2H  ,2H  ,2H  ,2H  ,2HWH,2HIL,2HE ,2H  ,2H  ,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2HWI,2HTH,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HWR,
& 2HIT,2HE ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HZE,2HRO,2H←J,2HSE,
& 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H[ ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2H  ,2H] ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
& 2H↑=,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HAS,2HSI,2HGN,
& 2HME,2HNT,2H  ,2H  ,2H  ,2H  ,2H  ,2HAS,2HSI,2HGN,2HME,2HNT,2H S,
& 2HYM,2HBO,2HL ,2H  ,2HBO,2HDY,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
& 2H  ,2HCA,2HLL,2H S,2HTA,2HT ,2H  ,2H  ,2H  ,2H  ,2H  ,2HCA,2HSE,
& 2H C,2HLA,2HUS,2HE ,2H  ,2H  ,2H  ,2H  ,2HCA,2HSE,2H S,2HTA,2HT ,
& 2H  ,2H  ,2H  ,2H  ,2H  ,2HCA,2HSE,2H←K,2HEY,2H  ,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2HCO,2HMM,2HAN,2HD ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HCO,
& 2HMM,2HEN,2HTS,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HCO,2HMP,2HOU,2HND,
& 2H S,2HTA,2HT ,2H  ,2H  ,2H  ,2HCO,2HNT,2HIN,2HJE,2H←K,2HEY,2H  ,
& 2H  ,2H  ,2H  ,2HCO,2HNT,2HRO,2HL ,2HST,2HAT,2H  ,2H  ,2H  ,2H  ,
& 2HCY,2HCL,2HE ,2HST,2HAT,2H  ,2H  ,2H  ,2H  ,2H  ,2HCY,2HCL,2HE←,
& 2HKE,2HY ,2H  ,2H  ,2H  ,2H  ,2H  ,2HDI,2HCT,2HIO,2HNA,2HRY,2HS ,
& 2H  ,2H  ,2H  ,2H  ,2HDI,2HCT,2HIO,2HNA,2HRY,2H←K,2HEY,2H  ,2H  ,
& 2H  ,2HDI,2HR←,2HAR,2HGS,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HDI,2HR←,
& 2HBL,2HOC,2HK ,2H  ,2H  ,2H  ,2H  ,2H  ,2HDI,2HR←,2HLI,2HST,2H  ,
& 2H  ,2H  ,2H  ,2H  ,2H  ,2HDI,2HR←,2HNA,2HME,2H  ,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2HDO,2H L,2HIS,2HT ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HDO,
& 2H S,2HTA,2HT ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HDO,2H1 ,2H  ,2H  ,
& 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HEL,2HSE,2H P,2HAR,2HT ,2H  ,2H  ,
& 2H  ,2H  ,2H  ,2HEL,2HSE,2H←K,2HEY,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
& 2HEX,2HIT,2H S,2HTA,2HT ,2H  ,2H  ,2H  ,2H  ,2H  ,2HEX,2HPR,2H  ,
& 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HEX,2HPR,2HPR,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2H  ,2H  ,2HFI,2HLE,2HNA,2HME,2H  ,2H  ,2H  ,2H  ,2H  ,
& 2H  ,2HFI,2HLE,2H←K,2HEY,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HFI,2HNI,
& 2HSH,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HFO,2HR ,2HST,2HAT,2H  ,
& 2H  ,2H  ,2H  ,2H  ,2H  ,2HFO,2HR←,2HKE,2HY ,2H  ,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2HGA,2HRB,2HAG,2HEP,2HR ,2H  ,2H  ,2H  ,2H  ,2H  ,2HI/,
& 2HC ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HIF,2H C,2HLA,2HUS,
& 2HE ,2H  ,2H  ,2H  ,2H  ,2H  ,2HIF,2H S,2HTA,2HT ,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2H  ,2HIF,2H←K,2HEY,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
& 2HIN,2HIT,2HIA,2HL ,2HPA,2HRT,2H  ,2H  ,2H  ,2H  ,2HIN,2HPU,2HT ,
& 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HIN,2HPL,2HT←,2HKE,2HY ,2H  ,
& 2H  ,2H  ,2H  ,2H  ,2HIN,2HTR,2HOD,2HUC,2HTI,2HON,2H  ,2H  ,2H  ,
& 2H  ,2HIN,2HTR,2HOD,2HUC,2HTI,2HON,2H S,2HTA,2HT ,2H  ,2HJO,2HB ,
& 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HLA,2HBE,2HL ,2H  ,2H  ,
& 2H  ,2H  ,2H  ,2H  ,2H  ,2HLE,2HFT,2H P,2HAR,2HEN,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2HOP,2HER,2HAT,2HOR,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HOP,
& 2HT←,2HAR,2HGS,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HOP,2HT←,2HLI,2HST,
& 2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HOP,2HT←,2HNA,2HME,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2H  ,2HOU,2HTP,2HUT,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
& 2HOU,2HTP,2HUT,2H←K,2HEY,2H  ,2H  ,2H  ,2H  ,2H  ,2HPR,2HIM,2HAR,
& 2HY ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HPR,2HIM,2HAR,2HYP,2HR ,2H  ,
& 2H  ,2H  ,2H  ,2H  ,2HPR,2HOG,2HRA,2HM ,2H  ,2H  ,2H  ,2H  ,2H  ,
& 2H  ,2HPR,2HOG,2HRA,2HM←,2HPR,2HIM,2HE ,2H  ,2H  ,2H  ,2HPR,2HOG,
& 2H←L,2HIS,2HT ,2H  ,2H  ,2H  ,2H  ,2H  ,2HRE,2HPE,2HAT,2H S,2HTA,
& 2HT ,2H  ,2H  ,2H  ,2H  ,2HRE,2HPE,2HAT,2H←K,2HEY,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2HRE,2HTJ,2HRN,2H P,2HAR,2HT ,2H  ,2H  ,2H  ,2H  ,2HRE,
& 2HTU,2HRN,2H←K,2HEY,2H  ,2H  ,2H  ,2H  ,2H  ,2HRI,2HGH,2HT ,2HPA,
& 2HRE,2HN ,2H  ,2H  ,2H  ,2H  ,2HST,2HAR,2HT ,2H  ,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2H  ,2HST,2HAT,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
& 2HST,2HAT,2H L,2HIS,2HT ,2H  ,2H  ,2H  ,2H  ,2H  ,2HST,2HRU,2H←N,
& 2HOU,2HN ,2H  ,2H  ,2H  ,2H  ,2H  ,2HTH,2HEN,2H←K,2HEY,2H  ,2H  ,
& 2H  ,2H  ,2H  ,2H  ,2HUN,2HIT,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,
& 2H  ,2HUN,2HIT,2HS ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2H  ,2HUN,2HTI,
& 2HL←,2HKE,2HY ,2H  ,2H  ,2H  ,2H  ,2H  ,2HVE,2HRB,2H C,2HLA,2HUS,
& 2HE ,2H  ,2H  ,2H  ,2H  ,2HVE,2HRB,2H P,2HAR,2HT ,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2HWH,2HIL,2HE ,2HST,2HAT,2H  ,2H  ,2H  ,2H  ,2H  ,2HWH,
& 2HIL,2HE←,2HKE,2HY ,2H  ,2H  ,2H  ,2H  ,2H  ,2HWI,2HTH,2H S,2HTA,
& 2HT ,2H  ,2H  ,2H  ,2H  ,2H  ,2HWI,2HTH,2H←K,2HEY,2H  ,2H  ,2H  ,
& 2H  ,2H  ,2H  /
```

APPENDIX C

SCANNER PROGRAM

```
      SUBROUTINE SCAN(TYPE,VALUE)
C ********************************************************************
C * BEGIN+INTRO
C *   PLP SCANNER; // PSEUDO LANGUAGE PROCESSOR SCANNER
C *                     WRITTEN BY YU-PING SUN,  DATE: 2-6-79
C *   INPUT+PARATERS - CARD+ IMAGE ;
C *   OUTPUT+PARAMETERS - TOKEN+TYPE, TOKEN+VALUE ;
C *   INTERFACE - CALLED BY ROUTINE INTPSR,NEXTKN;
C *               CALLING ROUTINE CONVER, INPUT, TSTTMN,
C *               TSTNUM, TOKEN, STATE, TBUPDT;
C *   DICTIONARY
C *   SNSTAT - 1: STARTING FROM LABEL 1000;//DETAILS SEE USER'S
C *                                          MANUAL- SCAN DIAGRAM
C *            2: STARTING FROM LABEL 2000; // SEE USER'S MANUAL
C *            3: STARTING FROM LABEL 3000; // SEE USER'S MANUAL
C *            4: STARTING FROM LABEL 4000; // SEE USER'S MANUAL
C *            5: STARTING FROM LABEL 5000; // SEE USER'S MANUAL
C *            6: STARTING FROM LABEL 6000; // SEE USER'S MANUAL
C *            7: STARTING FROM LABEL 7000; // SEE USER'S MANUAL
C *            8: STARTING FROM LABEL 8000; // SEE USER'S MANUAL
C *            9: STARTING FROM LABEL 9000; // SEE USER'S MANUAL
C *           10: STARTING FROM LABEL 10000; // SEE USER'S MANUAL
C *           11: STARTING FROM LABEL 11000; // SEE USER'S MANUAL
C *           12: STARTING FROM LABEL 12000; // SEE USER'S MANUAL
C *   SYMTAB - SYMBOL TABLE , SIZE DEPENDENT,
C *            CURRENT SIZE CAN HANDLE 300 DIFFERENT SYMBOLS;
C *   NOUNVL - VALUE TABLE OF NOUNS, SIZE DEPENDENT,
C *            CURRENT SIZE CAN HANDLE 200 DIFFERENT NOUNS;
C *   VERBVL - VALUE TABLE OF VERBS, SIZE DEPENDENT,
C *            CURRENT SIZE CAN HANDLE 200 DIFFERENT VERBS;
C *   FILETB - TABLE OF FILES, SIZE DEPENDENT,
C *            CURRENT SIZE CAN CONTAIN 50 DIFFERENT FILES;
C *   NOUNTB - TABLE OF NOUNS, SIZE DEPENDENT,
C *            CURRENT SIZE CAN CONTAIN 200 DIFFERENT NOUNS;
C *   VERBTB - TABLE OF VERBS, SIZE DEPENDENT,
C *            CURRENT SIZE CAN CONTAIN 200 DIFFERENT VERBS;
C *   FLAG   - FLAG FOR DETECTING COMMENTS,
C *            INITIALIZE TO ".TRUE.".
C *   FGONUM - FLAG FOR DETECTING NUMBER,INITIALIZE TO ".FALSE."
C *   FGOTMN - FLAG FOR DETECTING TERMINAL SYMBOL,
C *            INITIALIZE TO ".FALSE."
C *   FST80  - FLAG FOR DETECTING IF THE FIRST TIME REACH 80TH
C *            COLUMN, INITIALIZE TO ".TRUE.";
C *   TC     - FLAG FOR DETECTING TRANS+COMMENT,
C *             INITIALIZE TO ".FALSE.";
C * END+INTRO
C *******************************************************************
C
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH,FLAG,FGONUM,FGOTMN,TRACNG
      LOGICAL PP,P80,TC,ANAL,FST80
      DIMENSION NUMBER(12),TYPES(9,10),GRABGE(30,10)
      DIMENSION VERKEY(13,10),TMPVCB(151,10)
      DIMENSION KEYWRD(4,10),TMPWRD(10),DOKEY(10)
      DIMENSION INITAL(10),RETURN(10),ENDEXP(6,10)
      DIMENSION MACVAR(10)
C
      COMMON WORD(10)
      COMMON /BLOCK/ NOTERM,VCBLRY(151,10),SYMTAB(300,10),SYMS,IVOCSZ
      COMMON /COMMENT/ FLAG
      COMMON /DEVICE/ INUSE,SAVUSE
      COMMON /E/ TRACNG
      COMMON /FLAGS/ SNSTAT,CHECK,EOF
      COMMON /POINTR/ FLPTR,VLPTR,VBPTR,NPTR,LNKNO, STKPTR,IJ,IR
      COMMON /RECMSG/ CARD(80),PTR
      COMMON /REPLAC/ MACIN(100),MACOUT(100),MACSUB
      COMMON /SIZE/SYMSZ,FILESZ
      COMMON /SWITCH/ PP,P80,TC,ANAL
      COMMON /TABLE/ MAXPET,FILETB(50,10),NOUNTB(200,10),VERBTB(200,11),
     &               NOUNVL(200),VERBVL(200),PTHEXP(3000,3),
     &               VLSTK(200),TEMP(200,2)
      COMMON /TYPE/ NUMTP,NOUNTP,NUMTPP,VERBTP,IDENTP,NGRBTP,LABLTP
     &              ,JUNKTP,COMATP
      COMMON /Z/ PATH
C
      DATA FGONUM,FGOTMN/.FALSE.,.FALSE./
      DATA FST80/.TRUE./
      DATA PTR,TIME/80,1/
      DATA SEMCLN/2H; /
      DATA GRABGE/2HA ,9*2H  ,2HAN,9*2H  ,2HAN,2HD ,8*2H  ,2HAS,
     $            9*2H  ,2HEQ,2HUA,2HL  ,7*2H  ,2HFR,2HOM  ,
     $            8*2H  ,2HGE,9*2H  ,2HGR,2HEA,2HTE,2HR ,6*2H  ,
     $            2HGT,9*2H  ,2HHA,2HS ,8*2H  ,2HHA,2HVE,8*2H  ,
     $            2HIN,9*2H  ,2HIS,9*2H  ,2HIT,9*2H  ,2HLE,2HSS,
     $            8*2H  ,2HLE,9*2H  ,2HLT,9*2H  ,2HNO,2HT ,8*2H  ,
     $            2HOF,9*2H  ,2HOF,2HF ,8*2H  ,2HOR,9*2H  ,2HOU,
     $            2HT ,8*2H  ,2HTH,2HAN,8*2H  ,2HTH,2HE ,8*2H  ,
     $            2HTO,9*2H  ,2HUS,2HIN,2HG ,7*2H  ,2HIN,2HTO,8*2H  ,
     $            2HBY,9*2H  ,2HON,9*2H  ,2HON,2HTO,8*2H  /
      DATA TYPES/2HNU,2HMB,2HER,2H* ,6*2H    ,
     $2HNO,2HUN,2H* ,7*2H  ,2HNU,2HMB,2HER,2HPR,2H* ,5*2H  ,
     &2HVE,2HRB,2H* ,7*2H  ,2HID,2HEN,2HTI,2HFI,2HER,2H* ,4*2H  ,
     &2HNO,2HUN,2H←G,2HAR,2HBA,2HGE,2H* ,3*2H  ,2HLA,2HBE,2HL←,
     &2HKE,2HY*,5*2H  ,2HJU,2HNK,2H* ,7*2H  ,2H; ,9*2H  /
      DATA NUMBER/2H0 ,2H1 ,2H2 ,2H3 ,2H4 ,2H5 ,2H6 ,2H7 ,2H8 ,2H9 ,
     &2H  ,2H. /
      DATA COMMENT/2H///
      DATA INITAL/2HIN,2HIT,2HIA,2HL ,6*2H  /
      DATA RETURN/2HRE,2HTU,2HRN,7*2H  /
      DATA BLANK/2H  /
      DATA COMMA/2H, /
      DATA VERKEY/2H:=,9*2H  ,2H= ,9*2H  ,2H+ ,9*2H  ,2H- ,9*2H  ,
     $            2H* ,9*2H  ,2H/ ,9*2H  ,2H**,9*2H  ,2H( ,9*2H  ,
     $            2H; ,9*2H  ,2H) ,9*2H  ,2H: ,9*2H  ,2H. ,9*2H  ,
     $            2H, ,9*2H  /
      DATA ENDEXP/2HTH,2HEN,8*2H  ,2HDO,9*2H  ,2HBE,2HGI,2HN ,
     &         7*2H  ,2HCO,2HBE,2HGI,2HN ,6*2H  ,2H[ ,9*2H  ,
     &         2HBE,2HGI,2HNC,2HAS,2HE ,5*2H  /
      DATA MACVAR /
     & 2HBE,2HGI,2HN+,2HMA,2HC , 5*2H   /
      DATA MACIN,MACOUT / 100*0, 100*0 /
C
C
      IF(PATH)WRITE(6,10)
   10 FORMAT(10X,10HENTER SCAN)
   20 IF(TIME.EQ.0)GO TO 150
C
C ***  BY USING CONVER ROUTINE TO TRANSFER VCBLRY TABLE
C ***   TYPES TABLE, AND VERKEY TABLE FROM COLUMN-WISE TO ROW-WISE.
C
      CALL CONVER(VCBLRY,TMPVCB,IVOCSZ,10)
      CALL CONVER(TYPES,TMPVCB,9,10)
      CALL CONVER(VERKEY,TMPVCB,13,10)
      CALL CONVER(GRABGE,TMPVCB,30,10)
      CALL CONVER(ENDEXP,TMPVCB,6,10)
C
C ***  SEARCH VOCABULARY TABLE RETURN
C ***  TOKEN+TYPE FOR NUMBER,
C ***  TOKEN+TYPE FOR NOUN,
C ***  TOKEN+TYPE FOR NUMBERPR,
C ***  TOKEN+TYPE FOR VERB,
C ***  TOKEN+TYPE FOR IDENTIFIER,
C ***  TOKEN+TYPE FOR NOUN+GARBAGE,
C ***  TOKEN+TYPE FOR LABEL,
C ***  TOKEN+TYPE FOR JUNK;
C ***  TOKEN+TYPE FOR COMATP(;)
C
C
C
      DO 140 K=1,9
      DO 130 I=1,NOTERM
      DO 30 J=1,10
      IF(VCBLRY(I,J).NE.TYPES(K,J))GO TO 130
   30 CONTINUE
      GO TO (40,50,60,70,80,90,100,110,120),K
   40 NUMTP=I
      GO TO 140
   50 NOUNTP=I
      GO TO 140
   60 NUMTPP=I
      GO TO 140
   70 VERBTP=I
      GO TO 140
   80 IDENTP=I
      GO TO 140
   90 NGRBTP=I
      GO TO 140
  100 LABLTP=I
      GO TO 140
  110 JUNKTP=I
      GO TO 140
  120 COMATP=I
      GO TO 140
  130 CONTINUE
  140 CONTINUE
      TIME=0
C
C ***  BY USING ROUTINE INPUT TO GET A TOKEN
C
  150 CALL INPUT(WORD,PTR)
C
C ***   CHECK COMMENTS FLAG (TC)
C
      IF(WORD(1).NE.COMMENT)GO TO 155
      IF(TC)GO TO 155
      PTR=80
      GO TO 150
C
C ****************************************************************
C *     VERSION 2. WRITTEN BY Y.P. SUN DATE 5-15-79             *
C ****************************************************************
C
  155 CONTINUE
C
C           SEE IF MACRO SHOULD BE INVOKED
C
      DO 156 J = 1,10
      IF (WORD(J) .NE. MACVAR(J)) GO TO 158
  156 CONTINUE
      SAVSTA = SNSTAT
      SNSTAT = 13
      MACSUB = 0
      GO TO 150
C
C
C ***    STARTING FINITE STATE MACHINE (FSM)
C
C
  158 GO TO (1000,2000,3000,4000,5000,6000,7000,8000,9000,
     &      10000,11000,12000,13000,14000,15000,16000),SNSTAT
C
 1000 CONTINUE
C
      IF(TRACNG)WRITE(6,700)SNSTAT
  700 FORMAT(5X,"ENTER SCAN, SCAN+STAT= ",I4/)
      IF(WORD(1).NE.COMMENT)GO TO 160
      TYPE=JUNKTP
      VALUE=0
      GO TO 190
  160 CALL TSTTMN(WORD,TYPE,FGOTMN)
      IF(.NOT.FGOTMN)GO TO 170
      VALUE=0
      GO TO 190
  170 CALL TSTNUM(WORD,NUMBER,FGONUM)
      IF(.NOT.FGONUM)GO TO 180
      TYPE=NUMTPP
      VALUE=0
      GO TO 190
  180 CALL TOKEN(WORD,TYPE,VALUE,IDENTP)
      TYPE=NOUNTP
  190 CALL STATE(WORD,SNSTAT,RLSTAT,TYPE)
      RETURN
C
 2000 CONTINUE
      IF(TRACNG)WRITE(6,700)SNSTAT
C
      IF(WORD(1).NE.COMMENT)GO TO 200
      TYPE=JUNKTP
      VALUE=0
      GO TO 220
  200 CALL TSTTMN(WORD,TYPE,FGOTMN)
      IF(.NOT.FGOTMN)GO TO 215
      VALUE=0
      IF(WORD(1).NE.COMMA.AND.WORD(1).NE.SEMCLN)GO TO 205
      GO TO 220
  205 DO 210 I=1,10
      IF(WORD(I).NE.INITAL(I))GO TO 215
  210 CONTINUE
      GO TO 220
  215 TYPE=JUNKTP
  220 CALL STATE(WORD,SNSTAT,RLSTAT,TYPE)
      RETURN
C
 3000 CONTINUE
      IF(TRACNG)WRITE(6,700)SNSTAT
C
      IF(WORD(1).NE.COMMENT)GO TO 230
      TYPE=JUNKTP
      VALUE=0
      GO TO 260
  230 CALL TSTTMN(WORD,TYPE,FGOTMN)
      IF(.NOT.FGOTMN)GO TO 240
      VALUE=0
      GO TO 260
  240 CALL TSTNUM(WORD,NUMBER,FGONUM)
```

```
      IF(.NOT.FGONUM)GO TO 250
      TYPE=NUMTPP
      VALUE=0
      GO TO 260
  250 CALL TOKEN(WORD,TYPE,VALUE,IDENTP)
      TYPE=NOUNTP
  260 CALL STATE(WORD,SNSTAT,RLSTAT,TYPE)
      RETURN
C
 4000 CONTINUE
      IF(TRACNG)WRITE(6,700)SNSTAT
C
      IF(WORD(1).NE.COMMENT)GO TO 270
      TYPE=JUNKTP
      VALUE=0
      GO TO 290
  270 CALL TSTTMN(WORD,TYPE,FGOTMN)
      IF(.NOT.FGOTMN)GO TO 290
      VALUE=0
      IF(WORD(1).NE.COMMA.AND.WORD(1).NE.SEMCLN)GO TO 280
      GO TO 290
  280 TYPE=JUNKTP
  290 CALL STATE(WORD,SNSTAT,RLSTAT,TYPE)
      RETURN
C
 5000 CONTINUE
      IF(TRACNG)WRITE(6,700)SNSTAT
C
      IF(WORD(1).NE.COMMENT)GO TO 300
      TYPE=VERBTP
      VALUE=0
      GO TO 370
  300 CALL TSTTMN(WORD,TYPE,FGOTMN)
      IF(.NOT.FGOTMN)GO TO 310
      VALUE=0
      GO TO 370
  310 CALL TSTNUM(WORD,NUMBER,FGONUM)
      IF(.NOT.FGONUM)GO TO  320
      TYPE=NUMTP
      VALUE=0
      GO TO 370
  320 CALL TOKEN(WORD,TYPE,VALUE,IDENTP)
      DO 330 I=1,NPTR
      IF(VALUE.NE.NOUNVL(I))GO TO 330
      GO TO 370
  330 CONTINUE
C
C *** LOOKAHEAD KEYS (=, :=, +, -, *, /,(, ), **, ;, :, ",", ".")
C
      RLPTR=PTR
      CALL INPUT(TMPWRD,PTR)
      DO 360 I=1,13
      DO 340 J=1,10
      IF(TMPWRD(J).NE.VERKEY(I,J))GO TO 360
  340 CONTINUE
      PTR=RLPTR
C
C *** UPDATE NOUN←TABLE AND NOUN←VALUE←STACK
C
      CALL TBUPDT(1,VALUE)
C
      GO TO 370
  360 CONTINUE
      PTR=RLPTR
C
C ***  UPDATE VERB←TABLE AND VER←VALUE←STACK
C
      CALL TBUPDT(2,VALUE)
      TYPE=VERBTP
  370 CALL STATE(WORD,SNSTAT,RLSTAT,TYPE)
      RETURN
C
 6000 CONTINUE
      IF(TRACNG)WRITE(6,700)SNSTAT
C
      IF(WORD(1).NE.COMMENT)GO TO 380
      TYPE=VERBTP
      VALUE=0
      GO TO 450
  380 CALL TSTTMN(WORD,TYPE,FGOTMN)
      IF(.NOT.FGOTMN)GO TO 410
      VALUE=0
      IF(WORD(1).NE.SEMCLN)GO TO 390
      GO TO 450
  390 DO 400 I=1,10
      IF(WORD(I).NE.RETURN(I))GO TO 405
  400 CONTINUE
      GO TO 450
  405 TYPE=NGRBTP
      GO TO 450
  410 TYPE=NGRBTP
      CALL TSTNUM(WORD,NUMBER,FGONUM)
      IF(.NOT.FGONUM)GO TO 415
      VALUE=0
      GO TO 450
C
C *** SPECIAL KEY (A,AN,AND,AS,BY,EQUAL,FROM,GE,GREATER,GT,HAS,HAVE
C ***                   IN,INTO,IS,IT,LESS,LE,LT,NOT,OF,OFF,ON,ONTO,OR,
C ***                   OUT,THAN,THE,TO,USING)
C
  415 DO 430 J=1,30
      DO 420 I=1,10
      IF(WORD(I).NE.GRABGE(J,I))GO TO 430
  420 CONTINUE
      VALUE=0
      GO TO 450
  430 CONTINUE
      CALL TOKEN(WORD,TYPE,VALUE,IDENTP)
      TYPE=NGRBTP
C
C ***  UPDATE NOUN+TABLE AND NOUN+VALUE+STACK
C
      CALL TBUPDT(1,VALUE)
  450 CALL STATE(WORD,SNSTAT,RLSTAT,TYPE)
      RETURN
C
 7000 CONTINUE
      IF(TRACNG)WRITE(6,700)SNSTAT
C
      IF(PTR.NE.80)GO TO 470
C
C ***   IF POINTER EQUAL 80 THEN RETURN TWO PSEUDO TOKENS AS
C ***   JUNK AND ";"
C
      IF(.NOT.FST80)GO TO 455
C
C ***   IF SEEN THE 80TH COLUMN THEN TURN OFF THE FLAG (FST80)
C
      FST80=.FALSE.
C
      PTR=PTR-1
      DO 452 I=1,10
      WORD(I)=BLANK
  452 CONTINUE
      GO TO 470
C
  455 TYPE=COMATP
      FST80=.TRUE.
      VALUE=0
      SNSTAT=RLSTAT
C
C ***   IF SEEN A COMMENTS SYMBOL(//)
C ***   THEN TURN OFF THE THE FLAG  (FLAG) FOR SEMANTICS.
C
      FLAG=.FALSE.
      DO 460 I=1,10
      WORD(I)=BLANK
  460 CONTINUE
      RETURN
C
  470 TYPE=JUNKTP
      VALUE=0
      RETURN
```

```
C
 8000 CONTINUE
      IF(TRACNG)WRITE(6,700)SNSTAT
C
      IF(PTR.NE.80)GO TO 490
C
C ***   IF POINTER EQUAL 80 THEN RETURN TWO TOKENS AS
C ***   NOUN+GARBAGE AND ";"
C
      IF(.NOT.FST80)GO TO 475
C
C ***   TURN OFF FST80
C
      FST80=.FALSE.
C
      PTR=PTR-1
      DO 472 I=1,10
      WORD(I)=BLANK
  472 CONTINUE
      GO TO 490
C
  475 TYPE=COMATP
      FST80=.TRUE.
      VALUE=0
      SNSTAT=RLSTAT
C
C ***   TURN OFF SEMANTICS FLAG (FLAG)
C
      FLAG=.FALSE.
      DO 480 I=1,10
      WORD(I)=BLANK
  480 CONTINUE
      RETURN
C
  490 TYPE=NGRBTP
      VALUE=0
      RETURN
C
 9000 CONTINUE
      IF(TRACNG)WRITE(6,700)SNSTAT
C
      IF(WORD(1).NE.COMMENT)GO TO 500
      TYPE=JUNKTP
      VALUE=0
      GO TO 520
  500 CALL TSTTMN(WORD,TYPE,FGOTMN)
      IF(.NOT.FGOTMN)GO TO 510
      IF(WORD(1).NE.SEMCLN)GO TO 510
      VALUE=0
      GO TO 520
  510 TYPE=JUNKTP
```

```
      VALUE=0
  520 CALL STATE(WORD,SNSTAT,RLSTAT,TYPE)
      RETURN
C
10000 CONTINUE
      IF(TRACNG)WRITE(6,700)SNSTAT
C
      IF(WORD(1).NE.COMMENT)GO TO 530
      TYPE=VERBTP
      VALUE=0
      GO TO 600
  530 CALL TSTTMN(WORD,TYPE,FGOTMN)
      IF(.NOT.FGOTMN)GO TO 555
      DO 550 I=1,7
      DO 540 J=1,10
      IF(WORD(J).NE.ENDEXP(I,J))GO TO 550
  540 CONTINUE
      VALUE=0
      GO TO 600
  550 CONTINUE
  555 TYPE=IDENTP
      DO 570 I=1,VLPTR
      DO 560 J=1,10
      IF(WORD(J).NE.NOUNTB(I,J))GO TO 570
  560 CONTINUE
      CALL TOKEN(WORD,TYPE,VALUE,IDENTP)
      GO TO 600
  570 CONTINUE
      DO 590 I=1,FLPTR
      DO 580 J=1,10
      IF(WORD(J).NE.FILETB(I,J))GO TO 590
  580 CONTINUE
      CALL TOKEN(WORD,TYPE,VALUE,IDENTP)
      GO TO 600
  590 CONTINUE
      VALUE=0
  600 CALL STATE(WORD,SNSTAT,RLSTAT,TYPE)
      RETURN
C
11000 CONTINUE
      IF(TRACNG)WRITE(6,700)SNSTAT
C
      IF(WORD(1).NE.COMMENT)GO TO 610
      TYPE=VERBTP
      VALUE=0
      GO TO 630
  610 CALL TSTNUM(WORD,NUMBER,FGONUM)
      IF(.NOT.FGONUM)GO TO 620
      TYPE=LABLTP
      VALUE=0
      GO TO 630
  620 CALL STATE(WORD,SNSTAT,RLSTAT,TYPE)
      GO TO 5000
  630 CALL STATE(WORD,SNSTAT,RLSTAT,TYPE)
      RETURN
C
12000 CONTINUE
C
      IF(TRACNG)WRITE(6,700)SNSTAT
C
      IF(WORD(1).NE.COMMENT)GO TO 640
      TYPE=VERBTP
      VALUE=0
      GO TO 710
  640 CALL TSTTMN(WORD,TYPE,FGOTMN)
      IF(.NOT.FGOTMN)GO TO 650
      IF(WORD(1).NE.SEMCLN)GO TO 650
      VALUE=0
      GO TO 710
  650 TYPE=IDENTP
      DO 670 I=1,VLPTR
      DO 660 J=1,10
      IF(WORD(J).NE.NOUNTB(I,J))GO TO 670
  660 CONTINUE
      CALL TOKEN(WORD,TYPE,VALUE,IDENTP)
      GO TO 710
  670 CONTINUE
C
      DO 690 I=1,FLPTR
      DO 680 J=1,10
      IF(WORD(J).NE.FILETB(I,J))GO TO 690
  680 CONTINUE
      CALL TOKEN(WORD,TYPE,VALUE,IDENTP)
      GO TO 710
  690 CONTINUE
      VALUE=0
C
  710 CALL  STATE(WORD,SNSTAT,RLSTAT,TYPE)
      RETURN
C
13000 CONTINUE
C
      IF (TRACNG) WRITE (6,700) SNSTAT
C
      CALL STATE (WORD,SNSTAT,RLSTAT,TYPE)
      IF (SNSTAT .EQ. 14) GO TO 150
      SNSTAT = SAVSTA
      GO TO 170
C
14000 CONTINUE
C
      IF (TRACNG) WRITE (6,700) SNSTAT
```

```
C
      CALL STATE (WORD,SNSTAT,RLSTAT,TYPE)
      IF (SNSTAT .EQ. 15) GO TO 750
      SNSTAT = SAVSTA
      GO TO 170
C
  750 MACSUB = MACSUB + 1
      IF (MACSUB .LE. 20) GO TO 770
      WRITE (6,760)
  760 FORMAT (1X,36HTOO MANY MACRO VARIABLE REPLACEMENTS)
      GO TO 150
  770 IF (MOD(MACSUB,2) .EQ. 0) GO TO 790
      ISUB = MACSUB/2 * 10
      DO 780 I = 1,10
  780 MACIN(ISUB+I) = WORD(I)
      GO TO 150
  790 ISUB = (MACSUB-1)/2 * 10
      DO 800 I = 1,10
  800 MACOUT(ISUB+I) = WORD(I)
      GO TO 150
C
15000 CONTINUE
C
      IF (TRACNG) WRITE (6,700) SNSTAT
C
      CALL STATE (WORD,SNSTAT,RLSTAT,TYPE)
      IF (SNSTAT .EQ. 16 .OR. SNSTAT .EQ. 14) GO TO 150
      MACSUB = 0
      SNSTAT = SAVSTA
      GO TO 170
C
16000 CONTINUE
C
      IF (TRACNG) WRITE (6,700) SNSTAT
C
      CALL STATE (WORD,SNSTAT,RLSTAT,TYPE)
      IF (SNSTAT .EQ. 17) GO TO 820
      MACSUB = 0
      SNSTAT = SAVSTA
      GO TO 170
C
  820 CALL MMAINT (0)
      SAVUSE = INUSE
      INUSE = 8
      PTR = 80
      SNSTAT = SAVSTA
      GO TO 150
C
C
      END
      SUBROUTINE CONVER(A,B, I1,J1)
C ***********************************************************
C *  ROUTINE WHICH CAN REARRANGE THE INPUT ARRAY            *
C *  FROM COLUMN-WISE TO ROW-WISE.                          *
C ***********************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH
      DIMENSION A(I1,J1),B(I1,J1)
      COMMON /Z/ PATH
C
      IF(PATH) WRITE(6,30)
   30 FORMAT(10X,"ENTER CONVER")
C
      I=0
      J=1
      M=1
      N=0
      DO 5 K=1,J1
      DO 5 L=1,I1
      IF(I.NE.I1)GO TO 10
      J=J+1
   10 I=MOD(I,I1)+1
      IF(N.NE.J1)GO TO 15
      M=M+1
   15 N=MOD(N,J1)+1
      B(M,N)=A(I,J)
    5 CONTINUE
      DO 20 JJ=1,J1
      DO 20 II=1,I1
      A(II,JJ)=B(II,JJ)
   20 CONTINUE
      RETURN
      END
      SUBROUTINE INPUT(WORD,PTR)
C ************************************************************
C *   ROUTINE WHICH CAN GET A TOKEN FROM INPUT CARD+IMIAGE *
C *   WHENEVER IT BEING CALLED.                              *
C *   DELIMETERS : +, -, *, /, **, >, >=, <, <=, (, [, ),    *
C *                ], ",", ;, ".", :, =, ↑=, " "             *
C ************************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH
      DIMENSION IA(10),IB(10),WORD(10),EOFSYB(10)
C
      COMMON /DEVICE/ INUSE,SAVUSE
      COMMON /FLAGS/SNSTAT,CHECK,EOF
      COMMON /Z/ PATH
C
      DATA EOFSYB / 2HEO,2HF ,2HSY,2HMB,2HOL,5*2H   /
      DATA BLANK/2H  /
      DATA IA/2H+ ,2H- ,2H( ,2H[ ,2H) ,2H] ,2H; ,2H, ,2H= ,2H. /
      DATA IB/2H> ,2H< ,2H: ,2H↑ ,6*2H   /
      DATA ISLASH/2H/ /
      DATA STAR/2H* /,EQUAL/2H= /
C
      IF(PATH) WRITE(6,50)
   50 FORMAT(10X,"ENTER INPUT")
C
      DO 3 I1=1,10
    3 WORD(I1)=BLANK
      CHAR=BLANK
      N=0
    2 IF(CHAR.NE.BLANK)GO TO 5
      CALL LETTER(CHAR,PTR)
C
      GO TO (6,6,6,6,6,6,7,7,6,6,6,6,6,6,6,6,6),SNSTAT
    7 IF(PTR.NE.80)GO TO 6
C
      RETURN
    6 CONTINUE
C
      IF(EOF.NE.1) GO TO 2
      DO 4 I1=1,10
      WORD(I1)=EOFSYB(I1)
    4 CONTINUE
      RETURN
C
    5 DO 10 I=1,10
      IF(CHAR.EQ.IA(I))GO TO 15
      IF(CHAR.EQ.IB(I))GO TO 35
   10 CONTINUE
C
      IF(CHAR.EQ.STAR)GO TO  45
      IF(CHAR.EQ.ISLASH)GO TO 45
   11 N=N+1
      CALL COMBNE(WORD,N,CHAR)
      CALL LETTER(CHAR,PTR)
   13 IF(EOF.EQ.1)CHAR=BLANK
      DO 20 II=1,10
      IF(CHAR.EQ.IA(II).OR.CHAR.EQ.IB(II))GO TO 25
   20 CONTINUE
      IF(CHAR.EQ.BLANK.OR.CHAR.EQ.ISLASH)GO TO 25
      IF(CHAR.EQ.STAR.OR.CHAR.EQ.EQUAL)GO TO 25
      GO TO 11
   15 N=N+1
      CALL COMBNE(WORD,N,CHAR)
      CALL LETTER(CHAR,PTR)
      CHAR=BLANK
      GO TO 25
   35 N=N+1
      CALL COMBNE(WORD,N,CHAR)
      CALL LETTER(CHAR,PTR)
      IF(EOF.EQ.1)CHAR=BLANK
      IF(CHAR.EQ.EQUAL)GO TO 15
      GO TO 25
   45 N=N+1
      CALL COMBNE(WORD,N,CHAR)
      CALL LETTER(CHAR,PTR)
      IF(CHAR.EQ.STAR)GO TO 15
      IF(CHAR.EQ.ISLASH)GO TO 15
   25 PTR=PTR-1
      IF (INUSE .EQ. SAVUSE) RETURN
      CALL CHANGE (WORD)
      RETURN
      END
      SUBROUTINE CHANGE (WORD)
C
C          THIS ROUTINE COMPARES THE CURRENT INPUT TOKEN WITH
C          A SET OF TOKENS IN ORDER TO CHANGE THE CURRENT TOKEN TO ONE IN
C          A SET OF OTHER TOKENS FOR MODIFICATION OF MACRO VARIABLES.
C
      IMPLICIT INTEGER (A-Z)
C
      DIMENSION WORD(10)
      LOGICAL PATH
C
      COMMON /REPLAC/ MACIN(100),MACOUT(100),MACSUB
      COMMON /Z/ PATH
C
      IF (PATH) WRITE (6,5)
    5 FORMAT (40X,12HENTER CHANGE)
      IF (MACSUB .LE. 0) RETURN
      DO 40 I = 1,MACSUB/2
      ISUB = (I-1) * 10
      DO 10 II = 1,10
      IF (WORD(II) .NE. MACIN(II+ISUB)) GO TO 30
   10 CONTINUE
      DO 20 II = 1,10
   20 WORD(II) = MACOUT(II+ISUB)
      GO TO 40
   30 CONTINUE
   40 CONTINUE
      RETURN
C
      END
      SUBROUTINE STATE(WORD,SNSTAT,RLSTAT,TYPE)
C ***********************************************************
C *   ROTINE WHICH BASE ON THE CURRENT STATE AND TOKEN      *
C *   TO DECIDE THE NEXT STATE.                             *
C * *********************************************************
      IMPLICIT INTEGER(A-Z)
      LOGICAL TRACNG,PATH
      DIMENSION VERKEY(8),EXPRKY(5,10),IDKEY(6,10),TMPEXP(6,10)
      DIMENSION INITAL(10),RETURN(10),FILEKY(10),ENDEXP(7,10)
      DIMENSION TMPEND(6,10),ENDKEY(10),WORD(10),DCTNRY(10)
      DIMENSION MACVAR(20)
C
      COMMON /E/TRACNG
      COMMON /Z/ PATH
      COMMON /TYPE/NUMTP,NOUNTP,NUMTPP,VERBTP,IDENTP,NGRBTP,LABLTP,
     &              JUNKTP,COMATP
      DATA TIME/1/
      DATA COMMENT/2H///
      DATA INITAL/2HIN,2HIT,2HIA,2HL ,6*2H  /
      DATA RETURN/2HRE,2HTU,2HRN,7*2H  /
      DATA COMMA/2H, /
      DATA SEMCLN/2H; /
      DATA BLANK/2H  /
      DATA DOKEY/2HDO/
      DATA FILEKY/2HFI,2HLE,2HS ,7*2H  /
      DATA VERKEY/2H:=,2H= ,2H+ ,2H- ,2H* ,2H/ ,2H: ,2H; /
      DATA EXPRKY/2HIF,9*2H  ,2HWH,2HIL,2HE ,7*2H  ,2HCA,2HSE,
     &            8*2H  ,2HWI,2HTH,8*2H  ,2HFO,2HR ,8*2H  /
      DATA IDKEY/2HRE,2HAD,8*2H  ,2HWR,2HIT,2HE ,7*2H  ,
     $           2HPR,2HIN,2HT ,7*2H  ,2HEX,2HIT,8*2H  ,
     $           2HCA,2HLL,8*2H  ,2HUN,2HTI,2HL ,7*2H  /
      DATA ENDKEY/2HEN,2HD←,2HIN,2HTR,2HO ,5*2H  /
      DATA DCTNRY/2HDI,2HCT,2HIO,2HNA,2HRY,5*2H  /
      DATA MACVAR /
     & 2HRE,2HPL,2H←M,2HAC,2H  , 5*2H  ,
     & 2HGE,2HT←,2HMA,2HC ,2H  , 5*2H   /
C
      IF(PATH)WRITE(6,5)
    5 FORMAT(10X,"ENTER STATE")
C
      IF(TIME.NE.1)GO TO 10
      CALL CONVER(EXPRKY,TMPEND,5,10)
C     CALL CONVER(ENDEXP,TMPEND,7,10)
      CALL CONVER(IDKEY,TMPEND,6,10)
      TIME=0
   10 RLSTAT=SNSTAT
C
C ***    STARTING FINITE STATE MACHINE (FSM)
C
C
      GO TO (1000,2000,3000,4000,5000,6000,7000,8000,9000,
     &        10000,11000,12000,13000,14000,15000,16000), SNSTAT
C
 1000 CONTINUE
      IF(TRACNG)WRITE(6,90)SNSTAT
   90 FORMAT(10X,"ENTER STATE ,SCAN+STAT=",I4)
C
      IF(WORD(1).NE.COMMENT)GO TO 100
      SNSTAT=7
      RETURN
  100 IF(TYPE.NE.NUMTPP)GO TO 110
      SNSTAT=1
      RETURN
C
C ***   CHECK FILE KEY (FILES)
C
  110 DO 120 I=1,10
      IF(WORD(I).NE.FILEKY(I))GO TO 130
  120 CONTINUE
      SNSTAT=3
      RETURN
C
  130 DO 140 I=1,10
      IF(WORD(I).NE.ENDKEY(I))GO TO 150
  140 CONTINUE
      SNSTAT=5
      RETURN
C
  150 IF(TYPE.NE.NOUNTP)GO TO 160
      SNSTAT=2
      RETURN
C
  160 SNSTAT=1
      RETURN
C
 2000 CONTINUE
      IF(TRACNG)WRITE(6,90)SNSTAT
C
      IF(WORD(1).NE.COMMENT)GO TO 170
      SNSTAT=7
      RETURN
C
  170 IF(WORD(1).NE.COMMA.AND.WORD(1).NE.SEMCLN)GO TO 180
      SNSTAT=1
      RETURN
C
  180 DO 1 90 I=1,10
      IF(WORD(I).NE.INITAL(I))GO TO 200
  190 CONTINUE
      SNSTAT=9
      RETURN
C
```

```
  200 IF(TYPE.NE.JUNKTP)SNSTAT=2
      SNSTAT=2
      RETURN
C
 3000 CONTINUE
C
      IF(WORD(1).NE.COMMENT)GO TO 210
      SNSTAT=7
      RETURN
C
C ***   CHECK DICTIONARY KEY (DICTIONARY)
C
  210 DO 220 I=1,10
      IF(WORD(I).NE.DCTNRY(I))GO TO 230
  220 CONTINUE
      SNSTAT=1
      RETURN
C
  230 IF(TYPE.NE.NUMTPP)GO TO 240
      SNSTAT=3
      RETURN
C
  240 IF(TYPE.NE.NOUNTP)GO TO 250
      SNSTAT=4
      RETURN
C
  250 SNSTAT=3
      RETURN
C
 4000 CONTINUE
C
      IF(WORD(1).NE.COMMENT)GO TO 260
      SNSTAT=7
      RETURN
C
  260 DO 270 I=1,10
      IF(WORD(I).NE.INITAL(I))GO TO 280
  270 CONTINUE
      SNSTAT=3
      RETURN
C
  280 IF(WORD(1).NE.COMMA.AND.WORD(1).NE.SEMCLN)GO TO 290
      SNSTAT=3
      RETURN
C
  290 IF(TYPE.NE.JUNKTP)SNSTAT=4
      SNSTAT=4
      RETURN
C
 5000 CONTINUE
C
```

```
      IF(WORD(1).NE.COMMENT)GO TO 300
      SNSTAT=8
      RETURN
C
C ***   CHECK DO KEY (DO)
C
  300 IF(WORD(1).NE.DOKEY)GO TO 310
      SNSTAT=11
      RETURN
C
C ***   CHECK EXPRESSION KEYS (IF, WHILE, CASE, WITH, FOR)
C
  310 DO 330 I=1,5
      DO 320 J=1,10
      IF(WORD(J).NE.EXPRKY(I,J))GO TO 330
  320 CONTINUE
      SNSTAT=10
      RETURN
C
  330 CONTINUE
C
C ***   CHECK I/O KEYS (READ, WRITE, PRINT, CALL, EXIT, UNTIL)
C
      DO 340 I=1,6
      DO 335 J=1,10
      IF(WORD(J).NE.IDKEY(I,J))GO TO 340
  335 CONTINUE
      SNSTAT=12
      RETURN
C
  340 CONTINUE
C
      IF(TYPE.NE.VERBTP)GO TO 345
      SNSTAT=6
      RETURN
C
  345 SNSTAT=5
      RETURN
C
 6000 CONTINUE
C
      IF(WORD(1).NE.COMMENT)GO TO 350
      SNSTAT=8
      RETURN
C
  350 IF(WORD(1).NE.SEMCLN)GO TO 360
      SNSTAT=5
      RETURN
C
  360 DO 370 I=1,10
      IF(WORD(I).NE.RETURN(I))GO TO 380
  370 CONTINUE
      SNSTAT=6
      RETURN
  380 IF(TYPE.NE.NGRBTP)SNSTAT=6
      SNSTAT=6
      RETURN
C
 7000 CONTINUE
C
      SNSTAT=7
      RETURN
C
 8000 CONTINUE
C
      SNSTAT=8
      RETURN
C
 9000 CONTINUE
C
      IF(WORD(1).NE.COMMENT)GO TO 390
      SNSTAT=7
      RETURN
C
  390 IF(WORD(1).NE.SEMCLN)GO TO 400
      SNSTAT=1
      RETURN
  400 IF(TYPE.NE.JUNKTP)SNSTAT=9
      SNSTAT=9
      RETURN
C
10000 CONTINUE
C
      IF(WORD(1).NE.COMMENT)GO TO 410
      SNSTAT=8
      RETURN
  410 IF(TYPE.NE.IDENTP)GO TO 420
      SNSTAT=10
      RETURN
C
  420 SNSTAT=5
      RETURN
C
11000 CONTINUE
C
      IF(WORD(1).NE.COMMENT)GO TO 430
      SNSTAT=8
      RETURN
  430 CONTINUE
      SNSTAT=5
      RETURN
C
```

```
12000 CONTINUE
C
      IF(WORD(1).NE.COMMENT)GO TO 440
      SNSTAT=8
      RETURN
C
  440 IF(TYPE.NE.IDENTP)GO TO 450
      SNSTAT=12
      RETURN
C
  450 SNSTAT=5
      RETURN
C
13000 CONTINUE
C
      DO 460 I = 1,10
      IF (WORD(I) .NE. MACVAR(I)) RETURN
  460 CONTINUE
      SNSTAT = 14
      RETURN
C
14000 CONTINUE
C
      SNSTAT = 15
      RETURN
C
15000 CONTINUE
C
      IF (WORD(1) .EQ. COMMA) GO TO 481
      DO 480 I = 1,10
      IF (WORD(I) .NE. MACVAR(I)) GO TO 490
  480 CONTINUE
  481 SNSTAT = 14
      RETURN
C
  490 DO 500 I = 11,20
      IF (WORD(I-10) .NE. MACVAR(I)) RETURN
  500 CONTINUE
      SNSTAT = 16
      RETURN
C
16000 CONTINUE
C
      SNSTAT = 17
      RETURN
C
      END
```

```
C
      SUBROUTINE TBUPDT(FLAG,VALUE)
C ********************************************************************
C *      ROUTINE WHICH CAN UPDATE THE INPUT TABLE                     *
C *      FLAG - 1 : UPDATE THE NOUN←TABLE AND NOUN←VALUE←STACK        *
C *             - 2 : UPDATE THE VERB←TABLE AND VERB←VALUE←STACK      *
C ********************************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH
C
      COMMON /Z/ PATH
      COMMON /BLOCK/ NOTERM,VCBLRY(151,10),SYMTAB(300,10),SYMS,VOCSIZ
      COMMON /POINTR/ FLPTR,VLPTR,VBPTR,NPTR,LNKNO, STKPTR,IJ,IR
      COMMON /SIZE/SYMSZ,FILESZ
      COMMON /TABLE/ MAXPET,FILETB(50,10),NOUNTB(200,10),VERBTB(200,11),
     &               NOUNVL(200),VERBVL(200),PTHEXP(3000,3),
     &               VLSTK(200),TEMP(200,2)
C
      IF(PATH) WRITE(6,5)
    5 FORMAT(10X,"ENTER TBUPDT")
C
      GO TO (10,40), FLAG
   10 DO 20 I1=1,NPTR
      IF(NOUNVL(I1).NE.VALUE)GO TO 20
      RETURN
   20 CONTINUE
      VLPTR=VLPTR+1
C
      IF(VLPTR.GT.SYMSZ)WRITE(6,70)
C
      DO 30 J1=1,10
      NOUNTB(VLPTR,J1)=SYMTAB(VALUE,J1)
   30 CONTINUE
C
      NPTR=NPTR+1
      NOUNVL(NPTR)=VALUE
      RETURN
C
   40 DO 50 I2=1,VBPTR
      IF(VERBVL(I2).NE.VALUE)GO TO 50
C *        INCREMENT REFERENCE COUNT FOR THIS VERB
      VERBTB(I2,11) = VERBTB(I2,11) + 1
      RETURN
C
   50 CONTINUE
      VBPTR=VBPTR+1
      IF(VBPTR.GT.SYMSZ)WRITE(6,80)
      DO 60 J2=1,10
      VERBTB(VBPTR,J2)=SYMTAB(VALUE,J2)
   60 CONTINUE
      VERBVL(VBPTR)=VALUE
```

```
C             SET REFERENCE COUNT TO 1 FOR THIS VERB
      VERBTB(VBPTR,11) = 1
C
   70 FORMAT(/10X,"*** NOUN←TABLE OVERFLOW IN ROUTINE TBUPDT ***")
   80 FORMAT(/10X,"*** VERB←TABLE OVERFLOW IN ROUTINE TBUPDT ***")
C
      RETURN
C
      END
```

```
C
      SUBROUTINE TSTTMN(WORD,TYPE,FGOTMN)
C ***********************************************************************
C *         ROUTINE WHICH CAN DISTINGUISH TERMINAL SYMBOL FROM         *
C *         NON-TERMINAL SYMBOL.                                        *
C ***********************************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL FGOTMN,PATH
      DIMENSION WORD(10)
      COMMON /Z/ PATH
C
      COMMON /BLOCK/ NOTERM,VCBLRY(151,10),SYMTAB(300,10),SYMS,VOCSIZ
C
      IF( PATH ) WRITE(6,5)
    5 FORMAT(10X,"ENTER TSTTMN")
      DO 20 I=1,NOTERM
      DO 10 J=1,10
      IF(WORD(J).NE.VCBLRY(I,J))GO TO 20
   10 CONTINUE
      TYPE=I
      FGOTMN=.TRUE.
      RETURN
C
   20 CONTINUE
      FGOTMN=.FALSE.
      RETURN
      END
```

```
C
      SUBROUTINE LETTER(CHAR,PTR)
C ******************************************************
C *    ROUTINE WHICH CAN GET A CHARACTER AND UPDATE    *
C *    THE SYMBOL TABLE POINTER WHENEVER IT BEING      *
C *    CALLED.                                         *
C ******************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH,PP,P80,TC,ANAL
C
      COMMON /Z/ PATH
      COMMON /DEVICE/ INUSE,SAVUSE
      COMMON /FLAGS/SNSTAT,CHECK,EOF
      COMMON /RECMSG/ CARD(80),IPTR
C
C
      IF(PATH) WRITE(6,60)
C
      IF(PTR.NE.80)GO TO 30
    5 PTR=0
      READ(INUSE,10,END=40)CARD
   10 FORMAT(80A1)
   20 FORMAT(5X,80A1,/)
   30 PTR=PTR+1
      IPTR = PTR
      CHAR=CARD(PTR)
      RETURN
   40 IF (INUSE .EQ. SAVUSE) GO TO 50
      CALL MMAINT (1)
      INUSE = SAVUSE
      GO TO 5
   50 EOF=1
   60 FORMAT(10X,"ENTER LETTER")
      RETURN
      END
      SUBROUTINE TSTNUM(WORD,NUMBER,TEST)
C *****************************************************************
C *     ROUTINE WHICH CAN DISTINGUISH THE TOKEN IS A              *
C *     NUMBER OR A VARIABLE.                                     *
C *                                                               *
C *****************************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL TEST,PATH
      DIMENSION WORD(10),NUMBER(12)
      COMMON /Z/ PATH
      DATA BLANK/2H  /
C
      IF( PATH ) WRITE(6,5)
    5 FORMAT(10X,"ENTER TSTNUM")
C
      TEST=.FALSE.
      NN=0
   10 NN=NN+1
      IP=BLANK
CSHEL LEN=(NN/2*2-NN+1)*8
      LEN=(NN/2*2-NN+1)*6
      I=(NN+1)/2
CSHEL CALL LFLD(0,8,IP,FLD(LEN,8,WORD(I)))
      FLD(0,6,IP)=FLD(LEN,6,WORD(I))
      DO 20 J=1,12
      IF(IP.EQ.NUMBER(J))GO TO 30
   20 CONTINUE
      TEST=.FALSE.
      RETURN
   30 IF(I.LT.10)GO TO 10
      TEST=.TRUE.
      RETURN
      END
      SUBROUTINE COMBNE(WORD,N,CHAR)
C ****************************************************
C *   ROUTINE WHICH CAN CONCATENATE CHARACTER TO A WORD *
C ****************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH
      DIMENSION WORD(10)
      COMMON/ Z / PATH
C
      IF(PATH) WRITE(6,10)
   10 FORMAT(10X,"ENTER COMBNE")
C
CSHEL LEN=(N/2*2-N+1)*8
      LEN=(N/2*2-N+1)*6
      I=(N+1)/2
C
      IF(I.GT.10) RETURN
C
CSHEL CALL LFLD(LEN,8,WORD(I),FLD(0,8,CHAR))
      FLD(LEN,6,WORD(I))=FLD(0,6,CHAR)
      RETURN
      END
      SUBROUTINE TOKEN(WORD,TYPE,VALUE,IDENTP)
C ******************************************************************
C *   ROUTINE WHICH SET TOKEN+VALUE(VALUE) TO THE ENTRY           *
C *   POINT IN SYMBOL+TABLE(SYMTAB) AND TOKEN+TYPE(TYPE)          *
C *   TO THE TYPE OF IDENTIFIER(IDENTP).                          *
C ******************************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH
      DIMENSION WORD(10)
C
      COMMON /BLOCK/ NOTERM,VCBLRY(151,10),SYMTAB(300,10),SYMS,VOCSIZ
      COMMON /SIZE/SYMSZ,FILESZ,SZOSYM
      COMMON /Z/ PATH
C
      IF(PATH) WRITE(6,5)
    5 FORMAT(10X,"ENTER TOKEN")
C
      IF(SYMS.LT.SZOSYM)GO TO 10
       TYPE=IDENTP
      WRITE(6,7)
    7 FORMAT(/10X,"** SYMBOL+TABLE OVERFLOW IN ROUTINE TOKEN **")
      VALUE=300
      RETURN
   10 DO 9 L=1,SYMS
      DO 8 J=1,10
      IF(WORD(J).NE.SYMTAB(L,J))GO TO 9
    8 CONTINUE
      VALUE=L
      TYPE=IDENTP
      RETURN
    9 CONTINUE
      SYMS=SYMS+1
      DO 15 I=1,10
      SYMTAB(SYMS,I)=WORD(I)
   15 CONTINUE
      VALUE=SYMS
      TYPE=IDENTP
      RETURN
      END
```

APPENDIX D

SEMANTICS PROGRAM

```
      SUBROUTINE SEMNTC(ACTION,TYTKN,VLTKN)
C ****************************************************************
C * BEGIN+INTRO
C *   PLP SEMANTICS;
C *   //  ROUTINE SEMNTC BASES ON THE PARSING ALGORITHM CHECKING
C *   //  ALL THE PRODUCTION NUMBERS WHILE THEM BEING APPLIED,
C *   //  AND BUILDS UP THE PATH EXPRESSION FOR EACH PROCEDURE.
C *              WRITTEN BY YU-PING SUN, DATE: 5-10-79
C *   INPUT+PARAMETERS - TYTKN,VLTKN; //TOKEN+TYPE,TOKEN+VALUE
C *   OUTPUT+PARAMETERS - PATHEXP;
C *   DICTIONARY
C *   OCRNCE - TABLE OF OCCURANCE FOR EACH SYMBOL
C *            ,  SIZE DEPENDENT, CURRENT SIZE CAN HANDLE 200
C *            NOTATIONS;
C *   NUMLST - TABLE OF STATEMENT NUMBERS FOR EACH SYMBOL
C *              BEING USED,SIZE DEPENDENT, CURRENT SIZE CAN
C *              HANDLE 200 DIFFERENT NOTATIONS;
C *   TEMP   - TABLE WHICH CONTAINS OF THE INDEX OF HEAD AND TAIL
C *            FOR EACH SYMBOL,  SIZE DEPENDENT, CURRENT SIZE CAN
C *            HANDLE 200 DIFFERENT SYMBOLS;
C *   PTHEXP - TABLE WHICH CONTAINS OF THE PATH EXPRESSION
C *            FOR ALL SYMBOLS ,  SIZE DEPENDENT, CURRENT CAN
C *             HANDLE 3000 NOTATIONS.
C *   OUTBUF - TABLE WHICH CONTAINS OF THE PATH EXPRESSION AND
C *            STATEMENT NUMBER FOR EACH NOUN , SIZE DEPENDENT
C *            , CURRENT SIZE CAN HANDLE 500 NOTATIONS;
C *   VLSTK  - STACK OF TOKEN+VALUE ,SIZE DEPEDENT,
C *            CURRENT SIZE CAN CONTAIN 200 NODES.
C *   FLPTR  - FILE TABLE POINTER;
C *   VLPTR  - NOUN TABLE POINTER;
C *   VBPTR  - VERB TABLE POINTER;
C *   NPTR   - NOUN+VALUE STACK POINTER;
C *   STKPTR - TOKEN+VALUE STACK POINTER;
C *   OUTSZ  - LENGTH OF OUPUT BUFFER (OUTBUF);
C * END+INTRO
C ****************************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH,FLAG,PP,P80,TC,ANAL
      DIMENSION WORD(10),SYMBOL(12),OCRNCE(200),NUMLST(200)
      DIMENSION OUTBUF(500),STRING(10)
C
      COMMON /BLOCK/ NOTERM,VCBLRY(151,10),SYMTAB(300,10),SYMS,VOCSIZ
      COMMON /SWITCH/PP,P80,TC,ANAL
      COMMON /COMMENT/FLAG
      COMMON /FLAGS/SNSTAT,CHECK,JDUMNY
      COMMON /POINTR/FLPTR,VLPTR,VBPTR,NPTR,LNKNO, STKPTR,IJ,IR
      COMMON /PP3/LNEBUF(121),LNENO
      COMMON /SIZE/SYMSZ,FILESZ
      COMMON /TABLE/ MAXPET,FILETB(50,10),NOUNTB(200,10),VERBTB(200,11),
     &               NOUNVL(200),VERBVL(200),PTHEXP(3000,3),
     &               VLSTK(200),TEMP(200,2)
```

```
      COMMON /Z/PATH
C
      DATA OUTSZ/500/
      DATA ENDORC/2H& /
      DATA ENDOFL/2H$$/
      DATA SPCIDT/1/
      DATA SYMBOL/2HN ,2HU ,2HD ,2HR ,2H+ ,2H* ,2H( ,
     &2H) ,2H← ,2H# ,2H↑ ,2H- /
      DATA COMMA/2H, /
      IF(PATH)WRITE(6,10)
   10 FORMAT(30X,"ENTER SEMANTIC"/)
C
C
C        PROGRAM : INTRO STAT, COMPOUND STAT
C
C
C 1000 CONTINUE
C
      IF(ACTION.NE.1)GO TO 2000
C
C *** BUILD UP THE PROCEDURE NAME
C
      WRITE(6,20) (SYMTAB(NOUNVL(1),J1),J1=1,10)
   20 FORMAT(15X,"SYMBOL CROSS REFERENCE TABLE FOR  ",
     $      10A2,/,15X,50(1H-),///)
C
C ***  BUILD UP REFERENCE TABLE FOR NOUNS
C
      WRITE(6,30)
   30 FORMAT(5X,"DECLARED NOUNS",22X,"USED IN STATEMENT",
     $      /,5X,14(1H-),22X,17(1H-),//)
C
      DO 90 J=2,NPTR
      IN = NOUNVL(J)
      IP=TEMP(IN,1)
   40 IF(PTHEXP(IP,3).EQ.0)GO TO 50
      IR=IR+1
C
      IF(IR.GE.SYMSZ)WRITE(6,500)
C
      NUMLST(IR)=PTHEXP(IP,3)
   50 IP=PTHEXP(IP,2)
      IF(PTHEXP(IP,1).NE.0)GO TO 40
      IF(J.NE.IENDPT)GO TO 65
      WRITE(6,60)
   60 FORMAT(1H1,5X,"UNDECLARED NOUNS",18X,"USED IN STATEMENT",
     $     /6X,16(1H-),18X,17(1H-),//)
   65 CONTINUE
C
      ICOUNT=IR-1
      IF(ICOUNT.NE.0)GO TO 70
      JJ1=J-1
      WRITE(6,80)JJ1,(SYMTAB(IN,J1),J1=1,10),NUMLST(IR)
      IR=0
      GO TO 90
   70 CONTINUE
      JJ1 = J - 1
      WRITE(6,80)JJ1,(SYMTAB(IN,J1),J1=1,10),(NUMLST(J2),COMMA
     $ ,J2=1,ICOUNT),NUMLST(IR)
      IR=0
   80 FORMAT(2X,I5,1X,10A2,10X,8(I4,A1),/9(38X,8(I4,A1)/))
   90 CONTINUE
C
C ***  BUILD UP REFERENCE TABLE FOR VERBS
C
       WRITE(6,100)
  100 FORMAT(1H1,5X,"ALL VERBS",25X,"USED IN STATEMENT",
     $        /5X,9(1H-),25X,17(1H-),//)
C
      IF(VBPTR.EQ.0)GO TO 145
      DO 140 J=1,VBPTR
      IM=VERBVL(J)
      IP=TEMP(IM,1)
  110 IF(PTHEXP(IP,3).EQ.0)GO TO 120
      IR=IR+1
C
      IF(IR.GT.SYMSZ)WRITE(6,500)
C
      NUMLST(IR)=PTHEXP(IP,3)
  120 IP=PTHEXP(IP,2)
      IF(PTHEXP(IP,1).NE.0)GO TO 110
C
      ICOUNT=IR-1
      IF(ICOUNT.NE.0)GO TO 130
      WRITE(6,80)J,(SYMTAB(IM,J1),J1=1,10),NUMLST(IR)
C
      GO TO 135
C
  130 CONTINUE
      WRITE(6,80)J,(SYMTAB(IM,J1),J1=1,10),(NUMLST(J2),COMMA,
     $ J2=1,ICOUNT),NUMLST(IR)
C
  135 CONTINUE
C
      IR=0
  140 CONTINUE
  145 CONTINUE
C
      WRITE(6,150)
  150 FORMAT(1H1,/,5X,"PATH EXPRESSION :",//)
      DO 250 I=2,NPTR
      IQ=NOUNVL(I)
      IP=TEMP(IQ,1)
  160 IJ=IJ+1
C
      IF(IJ.GT.SYMSZ)WRITE(6,600)
C
      OCRNCE(IJ)=PTHEXP(IP,1)
C
      IF(PTHEXP(IP,3).EQ.0)GO TO 170
      IR=IR+1
      NUMLST(IR)=PTHEXP(IP,3)
  170 CONTINUE
C
      IP=PTHEXP(IP,2)
      IF(PTHEXP(IP,1).NE.0)GO TO 160
C
C ***   CALL REFINEMENT ROUTINE
C
      CALL REFINE(OCRNCE,IJ)
C
C ***  CONVER NUMLST FROM INTEGER TYPE TO CHARACTER TYPE
C
      INT=0
      ICNT=0
      DO 200 J2=1,IJ
      INT=INT+1
      IF(INT.GT.OUTSZ)WRITE(6,180)
  180 FORMAT(/5X,"OUTPUT BUFFER OVERFLOW")
      OUTBUF(INT)=SYMBOL(OCRNCE(J2))
      IF(OUTBUF(INT).NE.SYMBOL(2).AND.OUTBUF(INT).NE.SYMBOL(3).
     $   AND.OUTBUF(INT).NE.SYMBOL(4))GO TO 200
      ICNT=ICNT+1
C
      CALL INTCHR(NUMLST,ICNT,STRING,LEN)
C
      DO 190 I2=1,LEN
      INT=INT+1
      OUTBUF(INT)=STRING(I2)
  190 CONTINUE
  200 CONTINUE
      WRITE(6,210)(SYMTAB(IQ,J1),J1=1,10),(OUTBUF(II),II=1,INT)
C 210 FORMAT(5X,10A2,2X,1H:,2X,100A1,/4(30X,100A1)/)
  210 FORMAT(5X,10A2,2X,1H:,2X,50A1,/9(30X,50A1)/)
C
      IF(I.GE.IENDPT)GO TO 240
C
C ***  OUTPUT THE PATH EXPRESSION AND NAME OF EACH
C ***  DECLARED NOUN TO A TEMPARY DISC FILE (11).
C
      WRITE(11,230)INT,(OUTBUF(II),II=1,INT),ENDORC
      WRITE(11,235)(SYMTAB(IQ,J1),J1=1,10)
  230 FORMAT(I5,500A1)
  235 FORMAT(10A2)
C
  240 IA=IA+1
      IJ=0
      IR=0
  250 CONTINUE
C
C ***  OUTPUT A END+OF+FILE SYMBOL TO DISC FILE (11)
C ***  AT THE END OF THE PROCEDURE.
C
      WRITE(11,230)SPCIDT,ENDOFL,ENDOFL
C
      RETURN
C
C
C        INTRO STAT:  START, INTRO, FINISH
C
C
 2000 CONTINUE
C
      IF(ACTION.NE.2)GO TO 1800
      IENDPT=NPTR+1
      RETURN
C
C
C        EXPRPR :  EXPRPR, PRIMYPR
C
 1800 CONTINUE
C
      IF(ACTION.NE.18.AND.ACTION.NE.19)GO TO 2300
C
C ***    IF FLAG IS TRUE THEN
C ***      BUILD UP ATTRIBUTE "U"
C ***      ELSE RESET FLAG
      IF(FLAG)GO TO 260
      FLAG=.TRUE.
      RETURN
  260 LINE=LNENO
      INIT=1
      ATRB=2
      IF(STKPTR.EQ.0)RETURN
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
      RETURN
C
C
C        PRIMAYPR : PRIMAYPR, INIT PART
C
 2300 CONTINUE
C
      IF(ACTION.NE.23)GO TO 2500
      LINE=LNENO+1
      INIT=1
      ATRB=3
      IF(STKPTR.EQ.0)RETURN
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
      RETURN
C
C         STRU+NOUN : NOUN*
C
C
 2500 CONTINUE
C
      IF(ACTION.NE.25.AND.ACTION.NE.27)GO TO 2900
      IF(SNSTAT.NE.3)GO TO 280
C
C ***     UPDATE FILE TABLE
C
      FLPTR=FLPTR+1
C
      IF(FLPTR.GT.FILESZ)WRITE(6,700)
C
      NPTR=NPTR+1
      DO 270 J=1,10
  270 FILETB(FLPTR,J)=SYMTAB(VLTKN,J)
      STKPTR=STKPTR+1
      VLSTK(STKPTR)=VLTKN
      NOUNVL(NPTR)=VLTKN
      RETURN
C
C *** UPDATE NOUN TABLE
C
  280 VLPTR=VLPTR+1
      DO 285 J=1,NPTR
      IF(VLTKN.NE.NOUNVL(J))GO TO 285
      GO TO 305
  285 CONTINUE
      NPTR=NPTR+1
C
      IF(NPTR.GT.SYMSZ)WRITE(6,800)
C
      DO 300 J=1,10
      NOUNTB(VLPTR,J)=SYMTAB(VLTKN,J)
  300 CONTINUE
      NOUNVL(NPTR)=VLTKN
C
  305 STKPTR=STKPTR+1
      VLSTK(STKPTR)=VLTKN
      RETURN
C
C
C        LEFT PAREN : DO
C                   : COBEGIN
```

```
C                     : END
C                     : [
C
 2900 CONTINUE
C
      IF(ACTION.NE.29.AND.ACTION.NE.30.AND.
     $   ACTION.NE.31.AND.ACTION.NE.32)GO TO 4500
      IF(PRDNUM.NE.94)GO TO 306
C
      CALL INSERT(VLSTK,NPTR,9)
      PRDNUM=0
      RETURN
C
  306 IF(PRDNUM.NE.105.AND.PRDNUM.NE.96)GO TO 4500
      PRDNUM=0
      RETURN
C
C
C         COMMAND : READ, EXPR
C
C
 4500 CONTINUE
C
      IF(ACTION.NE.45)GO TO 4600
      LINE=LNENO+1
      INIT=1
      ATRB=3
      IF(STKPTR.EQ.0)RETURN
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
      RETURN
C
C
C        COMMAND : PRINT, EXPR
C
C
 4600 CONTINUE
C
      IF(ACTION.NE.46)GO TO 4900
      LINE=LNENO+1
      INIT=1
      ATRB=4
      IF(STKPTR.EQ.0)RETURN
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
      RETURN
C
C
C        COMMAND : WRITE, EXPR
C
C
 4900 CONTINUE
C
```

```
      IF(ACTION.NE.49)GO TO 5000
      LINE=LNENO+1
      INIT=1
      ATRB=4
      IF(STKPTR.EQ.0)RETURN
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
      RETURN
C
C
C      VERB PART : VERB CL, COMMENTS
C
C
 5000 CONTINUE
C
      IF(ACTION.NE.50)GO TO 5100
      IF(FLAG)GO TO 310
      FLAG=.TRUE.
      RETURN
C
  310 LINE=LNENO+1
      INIT=1
      ATRB=4
      IF(STKPTR.EQ.0)GO TO 315
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
  315 PRDNUM=0
      RETURN
C
C
C      VERB CL : VERB*
C
C
 5100 CONTINUE
C
      IF(ACTION.NE.51)GO TO 5200
      LINE=LNENO+1
      INIT=1
      ATRB=12
      STKPTR=1
      VLSTK(STKPTR)=VLTKN
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
      RETURN
C
C
C      RETURN PART : RETURN+KEY, COMMENTS
C
C
 5200 CONTINUE
C
      IF(ACTION.NE.52)GO TO 5600
      LINE=LNENO+1
      INIT=1
      ATRB=3
      IF(STKPTR.EQ.0)RETURN
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
      RETURN
C
C
C       EXPR : PRIMARY
C            : EXPR, PRIMARY
C
C
 5600 CONTINUE
C
      IF(ACTION.NE.56.AND.ACTION.NE.59)GO TO 7100
C
C ***  CHECK CASE+KEY, WHILE+KEY, UNTIL+KEY, WITH+KEY
C ***  AND IF+KEY
C
      IF(PRDNUM.NE.88.AND.PRDNUM.NE.94.AND.
     $   PRDNUM.NE.101.AND.PRDNUM.NE.105.AND.
     $   PRDNUM.NE.111)GO TO 320
      ATRB=4
      GO TO 325
C
C *** CHECK FOR+KEY
C
  320 IF(PRDNUM.NE.96)GO TO 7100
      ATRB=3
  325 LINE=LNENO+1
      INIT=1
      IF(STKPTR.EQ.0)RETURN
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
      RETURN
C
C
C       ASSIGMNT : EXPR, ASSIGMNT SYMB, EXPR
C
C
 7100 CONTINUE
C
      IF(ACTION.NE.71)GO TO 8600
      LINE=LNENO+1
      IF(STKPTR.LE.1)GO TO 330
      INIT=2
      ATRB=4
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
  330 INIT=1
      ATRB=3
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
      RETURN
C
C
```

```
C         CASE STAT : CASE CL, UNITS, ENDCASE
C
C
 8600 CONTINUE
C
      IF(ACTION.NE.86)GO TO 8700
      CALL INSERT(NOUNVL,NPTR,8)
      CALL INSERT(NOUNVL,NPTR,8)
      RETURN
C
C
C        CASE CL : CASE+KEY, EXPR, BEGINCASE
C
C
 8700 CONTINUE
C
      IF(ACTION.NE.87)GO TO 8800
      CALL INSERT(NOUNVL,NPTR,7)
      CALL INSERT(NOUNVL,NPTR,7)
      PRDNUM=0
      RETURN
C
C
C         CASE+KEY : CASE
C
C
 8800 CONTINUE
C
      IF(ACTION.NE.88)GO TO 9000
      PRDNUM=88
      RETURN
C
C
C         LABEL : EXPR
C
C
 9000 CONTINUE
C
      IF(ACTION.NE.90)GO TO 9300
      IF(PRDNUM.NE.88)GO TO 350
      CALL INSERT(NOUNVL,NPTR,8)
      CALL INSERT(NOUNVL,NPTR,5)
      CALL INSERT(NOUNVL,NPTR,7)
C
  350 LINE=LNENO+1
      INIT=1
      ATRB=4
      IF(STKPTR.EQ.0)RETURN
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
C
      RETURN
```

```
C
C
C         WHILE STAT : WHILE+KEY, EXPR, LP, BODY
C
C
 9300 CONTINUE
C
      IF(ACTION.NE.93)GO TO 9400
      CALL INSERT(NOUNVL,NPTR,8)
      CALL INSERT(NOUNVL,NPTR,6)
      RETURN
C
C
C         WHILE+KEY : WHILE
C
C
 9400 CONTINUE
C
      IF(ACTION.NE.94)GO TO 9600
      CALL INSERT(NOUNVL,NPTR,7)
      PRDNUM=94
      RETURN
C
C
C        FOR+KEY : FOR
C
C
 9600 CONTINUE
C
      IF(ACTION.NE.96) GO TO 9700
      PRDNUM=96
      RETURN
C
C
C         CYCLE STAT : CYCLE+KEY, BODY
C
C
 9700 CONTINUE
C
      IF(ACTION.NE.97)GO TO 9800
      CALL INSERT(NOUNVL,NPTR,8)
      CALL INSERT(NOUNVL,NPTR,11)
      RETURN
C
C
C         CYCLE+KEY : CYCLE
C
C
 9800 CONTINUE
C
      IF(ACTION.NE.98)GO TO  9900
      CALL INSERT(NOUNVL,NPTR,7)
      RETURN
C
C
C       REPEAT STAT : REPEAT←KEY, SL, UNTIL←KEY, EXPR
C
C
 9900 CONTINUE
C
      IF(ACTION.NE.99)GO TO 10000
      CALL INSERT(NOUNVL,NPTR,8)
      CALL INSERT(NOUNVL,NPTR,10)
      PRDNUM=0
      RETURN
C
C
C       REPEAT←KEY : REPEAT
C
C
10000 CONTINUE
C
      IF(ACTION.NE.100)GO TO 10100
      CALL INSERT(NOUNVL,NPTR,7)
      RETURN
C
C
C      UNTIL←KEY : UNTIL
C
C
10100 CONTINUE
C
      IF(ACTION.NE.101)GO TO 10200
      PRDNUM=101
      RETURN
C
C      EXIT STAT : EXIT, EXPR
C
C
10200 CONTINUE
C
      IF(ACTION.NE.102)GO TO 10500
      LINE=LNENO
      INIT=1
      ATRB=4
      IF(STKPTR.EQ.0)RETURN
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
      RETURN
C
C
C       WITH←KEY : WITH
C
```

```
C
10500 CONTINUE
C
      IF(ACTION.NE.105)GO TO 10600
      PRDNUM=105
      RETURN
C
C
C         IF STAT : IF CL, LP, BODY
C
C
10600 CONTINUE
C
      IF(ACTION.NE.106)GO TO 10800
      CALL INSERT(NOUNVL,NPTR,8)
      CALL INSERT(NOUNVL,NPTR,8)
      RETURN
C
C
C        ELSE PART : ELSE←KEY, LP, BODY
C
C
10800 CONTINUE
C
      IF(ACTION.NE.108)GO TO 10900
      CALL INSERT(NOUNVL,NPTR,8)
      CALL INSERT(NOUNVL,NPTR,8)
      RETURN
C
C
C        ELSE←KEY : ELSE
C
C
10900 CONTINUE
C
      IF(ACTION.NE.109)GO TO 11000
      CALL INSERT(NOUNVL,NPTR,8)
      CALL INSERT(NOUNVL,NPTR,5)
      CALL INSERT(NOUNVL,NPTR,7)
      RETURN
C
C
C         IF CL : IF←KEY, EXPR, THEN←KEY
C
C
11000 CONTINUE
C
      IF(ACTION.NE.110)GO TO 11100
      CALL INSERT(NOUNVL,NPTR,7)
      CALL INSERT(NOUNVL,NPTR,7)
      PRDNUM=0
      RETURN
C
C
C        IF+KEY : IF
C
C
11100 CONTINUE
C
      IF(ACTION.NE.111)GO TO 11600
      PRDNUM=111
      RETURN
C
C
C        PRIMARY : ID*
C
C
11600 CONTINUE
C
      IF(ACTION.NE.116)GO TO 11800
      IF(VLTKN.EQ.0)RETURN
      STKPTR=STKPTR+1
      VLSTK(STKPTR)=VLTKN
      RETURN
C
C
C        DO1 : DO, LABEL+KEY*, EXPR, ASSIGNT, SYMB, EXPR
C
C
11800 CONTINUE
C
      IF(ACTION.NE.118)GO TO 12000
      LINE=LNENO
      IF(STKPTR.LE.1)GO TO 340
      INIT=2
      ATBR=4
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
  340 INIT=1
      ATRB=3
      CALL BLDLNK(STKPTR,INIT,ATRB,LINE)
      RETURN
C
C
C
C       GARBGE : NOUN+GARBGE*
C
C
12000 CONTINUE
C
      IF(ACTION.NE.120)RETURN
      IF(VLTKN.EQ.0)RETURN
      STKPTR=STKPTR+1
      VLSTK(STKPTR)=VLTKN
C
  500 FORMAT(/10X,"** NUMBER LIST(NUMLST) OVERFLOW IN ROUTINE
     $           SEMNTC **")
  600 FORMAT(/10X,"** OCCURANCE TABLE OVERFLOW IN ROUTINE SEMNTC **")
  700 FORMAT(/10X,"** FILE TABLE OVERFLOW IN ROUTINE SEMNTC **")
  800 FORMAT(/10X,"** NOUN TABLE OVERFLOW IN ROUTINE SEMNTC **")
      RETURN
C
      END
      SUBROUTINE BLDLNK(STKPTR,INIT,ATRB,LNENUM)
C *******************************************************************
C *       ROUTINE WHICH CAN BUILD UP THE LINK LIST OF               *
C *       THE ATTRIBUTE FOR  EACH TOKEN STACKED IN THE VLSTK        *
C *******************************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH
      COMMON /Z/ PATH
      COMMON /TABLE/ MAXPET,FILETB(50,10),NOUNTB(200,10),VERBTB(200,11),
     &               NOUNVL(200),VERBVL(200),PTHEXP(3000,3),
     &               VLSTK(200),TEMP(200,2)
C
      IF(PATH) WRITE(6,5)
    5 FORMAT(10X,"ENTER BLDLNK")
C
      DO 20 I=INIT,STKPTR
      VAL=VLSTK(I)
      IK=I
      IF(TEMP(VAL,1).NE.0)GO TO 10
      CALL BGNLNK(VLSTK,IK,ATRB,LNENUM)
      GO TO 20
   10 CALL PTHLNK(VLSTK,IK,ATRB,LNENUM)
   20 CONTINUE
      STKPTR=0
      RETURN
      END
      SUBROUTINE BGNLNK(STACK,J,I,K)
C
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH
      DIMENSION STACK(100)
C
      COMMON /Z/ PATH
      COMMON /POINTR/ FLPTR,VLPTR,VBPTR,NPTR,LNKNO, STKPTR,IJ,IR
      COMMON /TABLE/ MAXPET,FILETB(50,10),NOUNTB(200,10),VERBTB(200,11),
     &               NOUNVL(200),VERBVL(200),PTHEXP(3000,3),
     &               VLSTK(200),TEMP(200,2)
C
      IF (PATH) WRITE(6,10)
   10 FORMAT(10X,"ENTER BGNLNK")
C
C ************************************************************
C *   ROUTINE WHICH CAN BUILD UP THE LINK TABLE
C *   FOR EACH OCCURANCE FROM BEGINNING
C *************************************************************
      CALL GETLNK
      TEMP(STACK(J),1)=LNKNO
      PTHEXP(LNKNO,1)=I
C
      PTHEXP(LNKNO,3)=K
      PPTR=LNKNO
      CALL GETLNK
      PTHEXP(PPTR,2)=LNKNO
      TEMP(STACK(J),2)=LNKNO
      RETURN
      END
      SUBROUTINE PTHLNK(STACK,J,I,K)
C
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH
      DIMENSION STACK(100)
      COMMON /Z/ PATH
C
      COMMON /POINTR/ FLPTR,VLPTR,VBPTR,NPTR,LNKNO, STKPTR,IJ,IR
      COMMON /TABLE/ MAXPET,FILETB(50,10),NOUNTB(200,10),VERBTB(200,11),
     &               NOUNVL(200),VERBVL(200),PTHEXP(3000,3),
     &               VLSTK(200),TEMP(200,2)
C
      IF (PATH) WRITE(6, 10)
   10 FORMAT(10X,"ENTER PTHLNK")
C
C ***********************************************************
C *     ROUTINE TRY TO BUILD UP LINK TABLE
C *     FOR EACH OCCURANCE
C ***********************************************************
      PTHEXP(TEMP(STACK(J),2),1)=I
      PPTR=TEMP(STACK(J),2)
      PTHEXP(PPTR,3)=K
      CALL GETLNK
      PTHEXP(PPTR,2)=LNKNO
      TEMP(STACK(J),2)=LNKNO
      RETURN
      END
      SUBROUTINE INSERT(NVAL,ICOUNT,ID)
C
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH
      DIMENSION NVAL(100)
C
      COMMON /Z/ PATH
      COMMON /TABLE/ MAXPET,FILETB(50,10),NOUNTB(200,10),VERBTB(200,11),
     &               NOUNVL(200),VERBVL(200),PTHEXP(3000,3),
     &               VLSTK(200),TEMP(200,2)
C
      IF (PATH) WRITE(6,5)
    5 FORMAT(10X,"ENTER INSERT")
C
C *********************************************************************
C *     THIS ROUTINE CAN INSERT SPECIAL SYMBOL INTO
C *     PATH EXPRESSION TABLE
C *********************************************************************
C
      DO 20 I=1,ICOUNT
       IK=I
      IF(TEMP(I,1).NE.0)GO TO 10
      CALL BGNLNK(NVAL,IK,ID,0)
      GO TO 10
   10 CALL PTHLNK(NVAL,IK,ID,0)
   20 CONTINUE
      RETURN
      END
      SUBROUTINE REFINE(OCRNCE,IJ)
C *****************************************************
C *    ROUTINE WHICH REFINE THE PATH EXPRESSION
C *    FOR EACH NOUN
C *****************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH
      DIMENSION OCRNCE(100),SYMBOL(11)
      COMMON /Z/ PATH
      DATA SYMBOL/1,2,3,4,7,8,5,6,11,10,9/
C
      IF(PATH) WRITE(6,5)
    5 FORMAT(10X,"ENTER REFINE")
C
      DO 50 M=1,50
      I=1
      J=1
   10 IF(I.GT.IJ)GO TO 40
   15 IF(OCRNCE(I).NE.SYMBOL(5))GO TO 25
C
      IF(OCRNCE(I+1).NE.SYMBOL(11).AND.OCRNCE(I+1).NE.SYMBOL(7))
     &   GO TO 11
      OCRNCE(J)=OCRNCE(I)
      I=I+2
      J=J+1
      GO TO 10
C
C
   11 IF(OCRNCE(I+1).NE.SYMBOL(6))GO TO 30
C
      DO 20 K=7,10
      IF(OCRNCE(I+2).NE.SYMBOL(K))GO TO 20
      I=I+3
      GO TO 10
   20 CONTINUE
      I=I+2
      GO TO 10
   25 IF(OCRNCE(I).NE.SYMBOL(7))GO TO 30
      IF(OCRNCE(I+1).NE.SYMBOL(6))GO TO 30
      I=I+1
      GO TO 10
   30 IF(OCRNCE(I).NE.SYMBOL(11))GO TO 35
      IF(OCRNCE(I+1).NE.SYMBOL(11))GO TO 35
      I=I+1
      GO TO 10
   35 OCRNCE(J)=OCRNCE(I)
C
      I=I+1
      J=J+1
      GO TO 10
   40 IJ=J-1
   50 CONTINUE
      RETURN
      END
      SUBROUTINE INTCHR(INPUT,I,CHAR,LEN)
C *******************************************************************
C *      ROUTINE WHICH CAN CONVER INPUT ARRAY FROM INTEGER          *
C *      TYPE(I - FORMAT ) TO CHARACTER TYPE (A - FORMAT )          *
C *                                                                 *
C *******************************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH
      DIMENSION INPUT(100),CHAR(10),TEMP(10),DIGIT(10)
      COMMON /Z/ PATH
      DATA DIGIT/2H0 ,2H1 ,2H2 ,2H3 ,2H4 ,2H5 ,2H6 ,
     &           2H7 ,2H8 ,2H9 /
      DATA BLANK/2H  /
C
      IF (PATH) WRITE(6,20)
   20 FORMAT(10X,"ENTER INTCHR")
C
      DO 1 I1=1,10
      CHAR(I1)=BLANK
      TEMP(I1)=BLANK
    1 CONTINUE
C
      LEN=1
      NUM=INPUT(I)
    2 IF(NUM.GE.10)GO TO 10
      CHAR(LEN)=DIGIT(NUM+1)
      DO 5 IP=1,LEN
      II=LEN-IP+1
      TEMP(II)=CHAR(IP)
    5 CONTINUE
      DO 8 J=1,10
      CHAR(J)=TEMP(J)
    8 CONTINUE
      RETURN
   10 RMNDR=MOD(NUM,10)
      NUM=NUM/10
      CHAR(LEN)=DIGIT(RMNDR+1)
      LEN=LEN+1
      GO TO 2
      END
      SUBROUTINE GETLNK
C
      IMPLICIT INTEGER (A-Z)
      LOGICAL PATH
C
      COMMON /Z/ PATH
      COMMON /POINTR/ FLPTR,VLPTR,VBPTR,NPTR,LNKNO, STKPTR,IJ,IR
      COMMON /TABLE/ MAXPET,FILETB(50,10),NOUNTB(200,10),VERBTB(200,11),
     &               NOUNVL(200),VERBVL(200),PTHEXP(3000,3),
     &               VLSTK(200),TEMP(200,2)
C
C          THIS SUBROUTINE PASSES TO THE CALLING ROUTINE THE INDEX
C          OF THE NEXT AVAILABLE NODE IN THE PATH EXPRESSION LINKED
C          LIST.  A CHECK IS MADE TO DETERMINE IF THE AVAILABLE
C          NODES ARE EXHAUSTED AND IF SO A MESSAGE TO THAT EFFECT
C          IS PRINTED AND EXECUTION IS STOPPED.
C
      DATA LNKNO / 0 /
C
      IF(PATH) WRITE(6,5)
    5 FORMAT(10X,"ENTER GETLNK")
C
      LNKNO = LNKNO + 1
      IF (LNKNO .LE. MAXPET) RETURN
      WRITE (6,10)
   10 FORMAT (10X,"THE PATH EXPRESSION LINKED LIST IS EXHAUSTED
     $          IN ROUTINE GETLNK. UPDATE THE SIZE OF PTHEXP
     $          AND THE VALUE OF MAXPET")
      STOP
      END
```

APPENDIX E

PL SOURCE PROGRAM AND SEMANTICS OUTPUT

```
 1                      BEGIN_INTRO
 2                              SAMPLE PL PROGRAM ;
 3                          DICTIONARY
 4                              X , Y - ARRAY OF INTEGEP NUMBER ;
 5                              FLAG , I : INTEGEP INITIAL 0 ;
 6                              TABLE - ARRAY CCTAINS OF STUDENT RECORD ;
 7                              01 NAME , 02 ADDRESS , 03 ID ;
 8                      END_INTRO

 9                          BEGIN
10        1                     I = 0 ;
11        1                     FLAG = 0 ;
12        1                     Y := X + 10 ;
13        1                     WHILE FLAG GREATER THAN 0 DO
14        1  2                      SEARCH M FOR TAELE RETURN M ;
15        1  2                      OD ;
16        1                     WHILE X ( 1 ) <= 10 DO
17        1  2                      X ( I ) = Y ( I ) + 1 ;
18        1  2                      END ;
19        1                     IF I > 10 THEN BEGIN
20        1  2                      X ( I ) = Y ( I ) * 2 + 1 ;
21        1  2                      I = 1 ;
22        1  2                      END ;
23        1                     ELSE BEGIN
24        1  2                      X = 1 ;
25        1  2                      Y = 3 ;
26        1  2                      END ;
27        1                     CASE I BEGINCASE
28        1  2                      1 : X := 0 ;
29        1  2  3                       END ;
30        1  2                      2 : Y := 10 ;
31        1  2  3                       END ;
32        1  2                      ENDCASE ;
33                          END ;
```

```
           SYMBOL CROSS REFERENCE TABLE FOR  SAMPLE
           ------------------------------------------------


  DECLARED NOUNS                          USED IN STATEMENT
  --------------                          -----------------

    1 X                                   4,  12,  16,  17,  20,  24,  28
    2 Y                                   4,  12,  17,  20,  25,  30
    3 FLAG                                5,  11,  13
    4 I                                   5,  10,  17,  17,  19,  20,  20,  21,
                                         27
    5 TABLE                               6,  14
    6 NAME                                7
    7 ADDRESS                             7
    8 ID                                  7
```

UNDECLARED NOUNS	USED IN STATEMENT
9 M	14, 14

ALL VERBS	USED IN STATEMENT
1 SEARCH	14

```
PATH EXPRESSION :

X                           :  U4R12(R16_D17)*((D20)+(D24))((D28))
Y                           :  U4D12(R17)*((R20)+(D25))((D30))
FLAG                        :  D5D11(R13_)*
I                           :  D5D10(R17R17)*R19((R20R20D21))R27
TABLE                       :  U6(R14)*
NAME                        :  U7
ADDRESS                     :  U7
ID                          :  U7
M                           :  R14D14)*
```

APPENDIX F

PE GRAMMAR

APPENDIX G

PE ANALYSER PROGRAM

```
C***********************************************************************
      SUBROUTINE ANALYSE(ACTION,TYPE,VALUE)
C **********************************************************************
C * BEGIN←INTRO                                                          *
C *   PLP ANALYSER;                                                      *
C *    //ROUTINE WHICH CAN ANALYZE THE POSSIBLE ANOMALIES FOR            *
C *    //THE PATH EXPRESSION OF EACH VARIABLE                            *
C *   DICTIONARY                                                         *
C *   ATBSTK - TOKEN←ATTRIBUTE←STACK,  50 BY 10 ARRAY  ;                 *
C *   LNUM  - LEFT←NUMBER←STACK, 50 BY 10 ARRAY;                         *
C *   RNUM - RIGHT←NUMBER←STACK, 50 BY 10 ARRAY;                         *
C *   WLREFG - FLAG STACK FOR DETECTING CONTROL STRUCTURE                *
C *            "WHILE" STATEMENT;                                        *
C *   COUNT - PARENTHESIS COUNTER;                                       *
C *   APTR  - ATTRIBUTE←STACK   POINTER;                                 *
C *   LPTR  - LEFT←NUMBER←STACK POINTER;                                 *
C *   RPTR  - RIGHT←NUMBER←STACK POINTER;                                *
C *   LLEN  - LENGTH POINTER FOR LEFT←NUMBER←STACK;                      *
C *   RLEN  - LENGTH POINTER FOR RIGHT←NUMBER←STACK;                     *
C *   INT   - LENTH POINTER FOR WLREFG;                                  *
C *   SYMSZ - SIZE  OF  ATBSTK,LNUM,RNUM, CURRENT                        *
C *           LENGTH IS 50;                                              *
C *   LENGTH - SIZE OF NUMBER←STACK, CURRENT LENGTH IS 10;               *
C *   LEVOFG - SIZE OF WLREFG STACK, CURRENT LENGTH IS 10;               *
C * END←INTRO                                                            *
C **********************************************************************
      IMPLICIT INTEGER (A-Z)
      LOGICAL RDFLAG
      DIMENSION VCBLRY(21,10),SYMTAB(100,10)
      COMMON /VAR/VARBLE(10)
      COMMON /BLOCK/DUMNY,VCBLRY,SYMTAB
      COMMON /TKNVAL/VLODCL,VLODEF,VLOREF
      COMMON /TABLES/ATBSTK(50,2),LNUM(50,10),RNUM(50,10)
     $              ,WLREFG(10),COUNT(10)
      COMMON /POINTR/APTR,LPTR,RPTR,LLEN,RLEN,LLOAB1,LLOAB2,
     $               RLOAB1,RLOAB2,PTR,INT
      COMMON /READD/RDFLAG
      DATA ATBSTK/100*0/
      DATA LNUM,RNUM/500*0,500*0/
      DATA APTR,LPTR,RPTR,LLEN,RLEN/5*0/
      DATA LLOAB1,LLOAB2,RLOAB1,RLOAB2/4*0/
      DATA LEVOFG/10/
      DATA LENGTH/10/
      DATA SYMSZ /50/
      DATA BLANK/2H  /
C
C ***   IF READ←FLAG IS "ON" THEN READS IN  THE VARIABLE NAME
C ***   FROM DISK FILE (11).
C
      IF(.NOT.RDFLAG)GO TO 1000
      READ(11,5,END=19000)VARBLE
```

```
    5 FORMAT(10A2)
      RDFLAG=.FALSE.
C
C
C
C        PATH : NO, EOF SYMBOL
C
 1000 CONTINUE
C
      IF(ACTION.NE.1)GO TO 2000
C
C *** IF THE RIGHT NODE OF ATTRIBUTE+STACK(ATBSTK) IS "D"
C *** THEN PRINT OUT THE ERROR MESSAGE.
C
      RATB=ATBSTK(APTR,2)
C
      IF(RATB.NE.VLODEF)GO TO 20
   10 RLEN=RLEN+1
C
      IF(RLEN.GT.LENGTH)WRITE(6,320)
C
      IF(RNUM(RPTR,RLEN+1).NE.0)GO TO 10
      WRITE(6,300)VARBLE,((SYMTAB(RNUM(RPTR,I),J),J=1,10),
     $                  I=1,RLEN)
      WRITE(6,310)
C
C *** INITIALIZE ALL STACKS AND POINTERS
C
   20 CALL INALZE
      RLEN=0
C
C
  300 FORMAT(///2X,10HVARIABLE: ,10A2,2X,25HWAS DEFINED IN STATEMENT ,
     $      3(10A2))
  310 FORMAT(34X,24HBUT WAS NEVER REFERENCED)
  320 FORMAT(/5X,"***  RIGHT NUMBER LIST OVERFLOW IN ROUTINE
     $          ANALYSE ***")
  330 FORMAT(/5X,"*** LEFT NUMLST LIST OVERFLOW IN ROUTINE
     $       ANALYSE ***")
  340 FORMAT(/5X,"***  STACK OF WHILE+FLAG OVERFLOW IN
     $          ROUTINE ANALYSE ***")
  350 FORMAT(/5X,"***  STACK OF ATTRIBUTE OVERFLOW IN
     $          ROUTINE ANALYSE ***")
  360 FORMAT(/5X,"***  STACK OF TOKEN+VALUE OVERFLOW IN
     $               ROUTINE ANALYSE ***")
C
      RETURN
C
C
C        NO : NO, N1
C
```

```
      C
       2000 CONTINUE
      C
            IF(ACTION.NE.2)GO TO 4000
      C
      C *** SET UP THE LEFT AND RIGHT ATTRIBUTE FOR ERROR ROUTINE
      C
            LATB=ATBSTK(APTR-1,2)
            RATB=ATBSTK(APTR,1)
         30 RLEN=RLEN+1
      C
            IF(RLEN.GT.LENGTH) WRITE(6,320)
      C
            IF(RNUM(RPTR-1,RLEN+1).NE.0)GO TO 30
         40 LLEN=LLEN+1
      C
            IF(LLEN.GT.LENGTH) WRITE(6,330)
      C
            IF(LNUM(LPTR,LLEN+1).NE.0)GO TO 40
      C
      C *** CALLING ERROR ANALYSE ROUTINE
      C
            CALL ERROR(LATB,RATB)
      C
      C ***  REDIFINE THE NUMBER LENGTH OF RIGHT ATTRIBUTE
      C
            RLEN=0
         45 RLEN=RLEN+1
            IF(RNUM(RPTR,RLEN+1).NE.0)GO TO 45
      C
      C
      C *** CALLING COLLAPSE ROUTINE
      C
            CALL COLLAPSE
            RETURN
      C
      C
      C        N1 : N1, N2
      C
      C
       4000 CONTINUE
      C
            IF(ACTION.NE.4)GO TO 7000
      C
      C ***  BUILD UP THE LEFT AND RIGHT ATTRIBUTE FOR
      C ***  FIRST NODE AND SECOND NODE
      C
            LATB1=ATBSTK(APTR-1,1)
            LATB2=ATBSTK(APTR,1)
            RATB1=ATBSTK(APTR-1,2)
            RATB2=ATBSTK(APTR,2)
```

```
C
C ***  BUILD UP "D" ATTRIBUTE FOR THIRD NODE
C
      IF(LATB1.NE.VLODEF)GO TO 60
      ATBSTK(APTR-1,1)=VLODEF
   50 LLOAB1=LLOAB1+1
      IF(LNUM(LPTR-1,LLOAB1+1).NE.0)GO TO 50
C
   60 IF(LATB2.NE.VLODEF)GO TO 80
      ATBSTK(APTR-1,1)=VLODEF
   70 LLOAB2=LLOAB2+1
      IF(LNUM(LPTR,LLOAB2+1).NE.0)GO TO 70
C
   80 IF(RATB1.NE.VLODEF)GO TO 100
      ATBSTK(APTR-1,2)=VLODEF
C
   90 RLOAB1=RLOAB1+1
      IF(RNUM(RPTR-1,RLOAB1+1).NE.0)GO TO 90
  100 IF(RATB2.NE.VLODEF)GO TO 120
      ATBSTK(APTR-1,2)=VLODEF
  110 RLOAB2=RLOAB2+1
      IF(RNUM(RPTR,RLOAB2+1).NE.0)GO TO 110
C
C
  120 IF(LLOAB2.EQ.0)GO TO 140
      DO 130 I=1,LLOAB2
      J=LLOAB1+I
      LNUM(LPTR-1,J)=LNUM(LPTR,I)
  130 CONTINUE
  140 IF(RLOAB2.EQ.0)GO TO 160
      DO 150 I=1,RLOAB2
      J=RLOAB1+I
      RNUM(RPTR-1,J)=RNUM(RPTR,I)
  150 CONTINUE
C
C
  160 LLOAB1=0
      LLOAB2=0
      RLOAB1=0
      RLOAB2=0
C
C *** INITIALIZE ALL STACKS AND POINTERS
C
      CALL INALZE
      RETURN
C
C
C       N2 : N3, NUMBERPR, ←
C
C
 7000 CONTINUE
```

```
      C
            IF(ACTION.NE.7)GO TO 8000
      C
      C *** TURN THE FLAG OF WHILE+ RELATION (WLREFG) FOR EACH OCCURANCE
      C
            INT=INT+1
      C
            IF(INT.GT.LEVOFG)WRITE(6,340)
      C
            WLREFG(INT)=1
            RETURN
      C
      C
      C        N3 : U
      C           : R
      C           : D
      C
      C
       8000 CONTINUE
      C
            IF(ACTION.NE.8.AND.ACTION.NE.9.AND.ACTION.NE.10)GO TO 11000
      C
      C ***  PUT TOKEN+VALUE INTO ATTRIBUTE STACK
      C
            APTR=APTR+1
      C
            IF(APTR.GT.SYMSZ)WRITE(6,350)
      C
            ATBSTK(APTR,1)=VALUE
            ATBSTK(APTR,2)=VALUE
      C
            RETURN
      C
      C
      C      NUMBERPR : NUMBER
      C
      C
      11000 CONTINUE
      C
            IF(ACTION.NE.11)GO TO 12000
      C
      C ***  PUT TOKEN+VALUE INTO NUMBER STACK
      C
            LPTR=LPTR+1
      C
            IF(LPTR.GT.SYMSZ)WRITE(6,360)
      C
            RPTR=RPTR+1
      C
      C
            LNUM(LPTR,1)=VALUE
      IF(RPTR.GT.SYMSZ) WRITE(6,360)
C
      RNUM(RPTR,1)=VALUE
      RETURN
C
C
C       N2 : LEFTPR, NO, RIGHTPR
C
C
12000 CONTINUE
C
      IF(ACTION.NE.12.AND.ACTION.NE.13)GO TO 16000
C     WRITE(6,850)ACTION
C
C ***  IF WHILE←RELATION←FLAG IS TRUE THEN
C ***  INSERT A "R" ATTRIBUTE BEFORE NEXT ATTRIBUTE
C
C     IF(ATBSTK(APTR-1,2).EQ.VLOREF)ATBSTK(APTR-1,2)=BLANK
      IF(ATBSTK(APTR,2).EQ.VLOREF)ATBSTK(APTR,2)=BLANK
C
      IF(WLREFG(INT).NE.1)RETURN
      IF(COUNT(INT).NE.-1)RETURN
C
      APTR=APTR+1
      LPTR=LPTR+1
      RPTR=RPTR+1
      ATBSTK(APTR,1)=ATBSTK(APTR-1,1)
      ATBSTK(APTR,2)=ATBSTK(APTR-1,1)
C
      DO 170 I=1,LLEN
      LNUM(LPTR,I)=LNUM(LPTR-1,I)
      RNUM(RPTR,I)=LNUM(LPTR-1,I)
  170 CONTINUE
C
      CALL COLLAPSE
C
C ***  RESET WHILE←RELATION←FLAG
C
C     WLREFG=.FALSE.
      WLREFG(INT)=0
      COUNT(INT)=0
      INT=INT-1
      RETURN
C
C
C       LEFTPR : (
C
C
16000 CONTINUE
C
      IF(ACTION.NE.16)GO TO 19000
      C
      C ***  IF THE FLAG OF WILE+RELATION (WLREFG) IS ON
      C ***  THEN INCREMENT THE PARENTHSES COUNTER(COUNT).
      C
      C
      C ***  IF THE FLAG OF WHILE+RELATION IS ON
      C ***  THEN DECREMENT THE PARENTHSES COUNTER.
      C
             IF(WLREFG(INT).EQ.0)RETURN
             COUNT(INT)=COUNT(INT)+1
             RETURN
      C
      C
      C        RIGHTPR : )
      C
      C
      19000 CONTINUE
      C
             IF(ACTION.NE.19)RETURN
             IF(WLREFG(INT).EQ.0)RETURN
             DO 200 I=1,INT
             COUNT(I)=COUNT(I)-1
        200 CONTINUE
             RETURN
      C
             END
      SUBROUTINE ERROR(LATB,RATB)
      IMPLICIT INTEGER (A-Z)
      DIMENSION VCBLRY(21,10),SYMTAB(100,10)
      COMMON /TABLES/ATBSTK(50,2),LNUM(50,10),RNUM(50,10)
      COMMON /BLOCK/IDUMNY,VCBLRY,SYMTAB
      COMMON /VAR/VARBLE(10)
      COMMON /TKNVAL/VLODCL,VLODEF,VLOREF
      COMMON /POINTR/APTR,LPTR,RPTR,LLEN,RLEN
C
      IF(LATB.NE.VLODCL.OR.RATB.NE.VLODCL)GO TO 10
      WRITE(6,500)VARBLE,((SYMTAB(RNUM(RPTR-1,I),J),J=1,10),
     $                          I=1,RLEN)
      WRITE(6,510)((SYMTAB(LNUM(LPTR,I),J),J=1,10),I=1,LLEN)
C
      RETURN
C
   10 CONTINUE
C
      IF(LATB.NE.VLODEF.OR.RATB.NE.VLODEF)GO TO 20
      WRITE(6,520)VARBLE,((SYMTAB(RNUM(RPTR-1,I),J),J=1,10),
     $                          I=1,RLEN)
      WRITE(6,530)((SYMTAB(LNUM(LPTR,I),J),J=1,10),I=1,LLEN)
C
      RETURN
C
   20 CONTINUE
C
      IF(LATB.NE.VLODCL.OR.RATB.NE.VLOREF)GO TO 30 .
      WRITE(6,540)VARBLE,((SYMTAB(LNUM(LPTR,I),J),J=1,10),
     $                          I=1,LLEN)
C
   30 CONTINUE
      RETURN
C
  500 FORMAT(///2X,"VARIABLE: ",10A2,2X,"WAS DECLARED IN STATEMENT "
     $           ,3(10A2)/34X,3(10A2)/)
  510 FORMAT(34X,"AND DECLARED AGAIN IN STATEMENT  ",3(10A2)/
     $          34X,4(10A2))
  540 FORMAT(///2X,"VARIABLE: ",10A2,2X,"WAS NOT DEFINED BEFORE "
     $           /34X,"AND REFERENCED IN STATEMENT  ",3(10A2))
  520 FORMAT(///2X,"VARIABLE: ",10A2,2X,"WAS DEFINED IN STATEMENT   "
     $          3(10A2)/57X,3(10A2))
  530 FORMAT(34X,"AND REDIFINED AGAIN IN STATEMENT ",3(10A2)
     $          /57X,3(10A2))
      END
```

```
C
      SUBROUTINE COLLAPSE
C *********************************************************************
C *       ROUTINE WHICH CAN PROPAGATE THE LEFT AND RIGHT              *
C *       ATTRIBUTE OF FIRST NODE AND SECOND NODE INTO                *
C *       THE THIRD NODE.                                             *
C *********************************************************************
      IMPLICIT INTEGER (A-Z)
      DIMENSION VCBLRY(21,10),SYMTAB(100,10)
      COMMON /TKNVAL/VLODCL,VLODEF,VLOREF
      COMMON /BLOCK/DUMNY,VCBLRY,SYMTAB
      COMMON /TABLES/ATBSTK(50,2),LNUM(50,10),RNUM(50,10)
      COMMON /POINTR/APTR,LPTR,RPTR,LLEN,RLEN,LLOAB1,LLOAB2
     $               ,RLOAB1,RLOAB2
      DATA BLANK/2H  /
C
      LATB1=ATBSTK(APTR-1,1)
      RATB2=ATBSTK(APTR,2)
C
      IF(LATB1.NE.BLANK)GO TO 20
      ATBSTK(APTR-1,1)=ATBSTK(APTR,1)
      DO 10 I=1,LLEN
      LNUM(LPTR-1,I)=LNUM(LPTR,I)
   10 CONTINUE
C
   20 IF(RATB2.EQ.BLANK)GO TO 35
      ATBSTK(APTR-1,2)=ATBSTK(APTR,2)
      DO 30 J=1,RLEN
      RNUM(RPTR-1,J)=RNUM(RPTR,J)
   30 CONTINUE
C
   35 CALL INALZE
C
      RETURN
      END
```

```
C
      SUBROUTINE INALZE
C **********************************************************************
C *        ROUTINE WHICH INITIALIZE THE STACK VALUE TO ZERO             *
C *        POINTING BY THE APTR(CURRENT ATTRIBUTE POINTER).            *
C **********************************************************************
      IMPLICIT INTEGER (A-Z)
      COMMON /TABLES/ATBSTK(50,2),LNUM(50,10),RNUM(50,10)
      COMMON /POINTR/APTR,LPTR,RPTR,LLEN,RLEN
C
      ATBSTK(APTR,1)=0
      ATBSTK(APTR,2)=0
      DO 10 I=1,10
      LNUM(LPTR,I)=0
   10 CONTINUE
C
      DO 20 I=1,10
      RNUM(RPTR,I)=0
   20 CONTINUE
C
      APTR=APTR-1
      LPTR=LPTR-1
      RPTR=RPTR-1
      LLEN=0
      RLEN=0
C
      RETURN
      END
```

APPENDIX H

<u>PE ANALYSER OUTPUT</u>

```
                    ALL POSSIBLE ANOMALIES FOR EACH VARIABLE OF  PROCEDURE   1


ARIABLE: X                       WAS NOT DEFINED BEFORE
                                 AND REFERENCED IN STATEMENT  12


ARIABLE: X                       WAS DEFINED IN STATEMENT  20                  2
                                 AND REDIFINED AGAIN IN STATEMENT 28


ARIABLE: X                       WAS DEFINED IN STATEMENT 28                  24
                                 BUT WAS NEVER REFERENCED


ARIABLE: Y                       WAS DEFINED IN STATEMENT  12
                                 AND REDIFINED AGAIN IN STATEMENT 25


ARIABLE: Y                       WAS DEFINED IN STATEMENT  25
                                 AND REDIFINED AGAIN IN STATEMENT 30


ARIABLE: Y                       WAS DEFINED IN STATEMENT 30
                                 BUT WAS NEVER REFERENCED


ARIABLE: FLAG                    WAS DEFINED IN STATEMENT  5
                                 AND REDIFINED AGAIN IN STATEMENT 11


ARIABLE: I                       WAS DEFINED IN STATEMENT  5
                                 AND REDIFINED AGAIN IN STATEMENT 10


ARIABLE: TABLE                   WAS NOT DEFINED BEFORE
                                 AND REFERENCED IN STATEMENT  14
```

BIBLIOGRAPHY

[A] ARTHUR J. D., "A Unified Model for Constructing Automatic Analysers Which Performs Static and Dynamic Program Validation", Master Thesis, Department of Computer Science, University of Houston, (may, 1979).

[AU] AHO, A. V., ULLMAN, J. D., Principles of Compiler Design, Addison-Weseley Publishing Company, (1977).

[B] BOYD, D. L., PIZZARELLO, "Introduction to the WELLMADE Design Methodology", IEEE Transactions on Software Engineering, Vol SE-4, No. 4, (July, 1978).

[B1] BOEHM, B., "Software and Its Impact: A Quantitative Assessment", Datamation, 19, (1973).

[C] CHAPIN, N., "Semi-Code in Design and Maintenance". Computer and People, Vol. 27, No. 6, (June 1978).

[CG] CAINE, S.H., GORDON, E. K., "PDL - A Tool for Software Design", Tutorial on Software Design Techniques, IEEE Catalog No. 76CH1145-2C, (October, 1976).

[H] HARTMAN, A. C., A Concurrent Pascal Compiler for Minicomputers, Springer-Verlag Berlin Heidlberg, (1977)

[L] LEDBETTER, W. R., "A Pseudo Language Processor for Design Validation and Implementation of Systems", Master Thesis, Department of Computer Science, University of Houston, (June 1979).

[R] RAMAMOORTHY, C. V., "Testing Large Software with Automated Software Evaluation Systems", IEEE Transactions on Software Engineering, Vol SE-1, No. 1, (March, 1978).

[RB] RAMANATHAN, J., BLATTNER, M., "Program Forms and Program Form Analysers for High Level Structured Design", Technical Report UH-CS-79-1, Department of Computer Science, University of Houston, (February, 1979).

[S] STAY, J. F., "HIPO and Integerated Program Design", Tutorial on Software Design Techniques, IEEE Catalog No. 76CH1145-2C, (October, 1976).

[WW] WIRTH, N., WEBER, H., "EULER : A Generalization of ALGOL and Its Formal Definition", Comm. ACM Vol. 9, No. 1, (1966).