August 9, 1976

To the Graduate Council:

I am submitting herewith a thesis written by William Roy Haynes, Jr., entitled "ORACLE: A Pedagogic Compiler." I recommend that it be accepted in partial fulfillment of the requirements for Master of Science, with a major in Computer Science.

Charles P. Pfleeger

Charles P. Pfleeger, Major Professor

Charles E. Hughes

Charles E. Hughes, Major Professor

We have read this thesis
and recommend its acceptance:

Michael G Thomason

CRANES CREST

Accepted for the Council:

Hilton A. Smith

Vice Chancellor
Graduate Studies and Research

ORACLE: A PEDAGOGIC COMPILER

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

William Roy Haynes, Jr.

August 1976

1287971

DEDICATION

ORACLE: A Pedagogic Compiler is dedicated to my wife, Faye, whose love and patience made this thesis worthwhile.

ACKNOWLEDGMENTS

The author is grateful for the guidance and patience of

Dr. Charles E. Hughes, Associate Professor of Computer Science and

Dr. Charles P. Pfleeger, Assistant Professor of Computer Science.

ABSTRACT

ORACLE is a pedagogic compiler. The design offers an environment conducive to the examination of the actions performed by a simple compiler. Two features are provided to assist the student. The first is a trace option which permits source statements to initiate or terminate a selective monitoring of the actions of the compiler. The second provides the necessary conditions to simulate the replacement of three compiler components: symbol table management, lexical analysis, and syntax analysis. Each replacement module is monitored by ORACLE in order to detect errors and to verify correct operation.

TABLE OF CONTENTS

CHAPTER	PAGE
INTRODUCTION	1
I. THE LANGUAGE OF ORACLE	4
Features	4
Implementation Restrictions	6
II. THE GRAMMAR	8
Background of LL(1) Grammar	8
Definition of the Grammar	9
Discussion of LL(1) Conditions	13
III. THE COMPILER	19
Lexical Analysis	19
Technique and background	19
Error analysis	21
Symbol Table Management	22
Technique and background	22
Error analysis	24
Syntax Analysis	24
Technique and background	24
Error analysis and recovery	25
Semantic Analysis and Code Generation	28
Technique and background	28
Error analysis	32

CHAPTER PAGE

Compiler Options 32

Trace Option 32

Debug Option 34

IV. REPLACEMENT MODE 35

Overview 35

Replacement Mode 1 - Symbol Table Management 37

Activation and input requirements 37

Communication 37

Evaluation 38

Replacement Mode 2 - Lexical Analysis 40

Activation and input requirements 40

Communication 41

Evaluation 41

Replacement Mode 3 - Syntax Analysis 41

Activation and input requirements 41

Communication 43

Evaluation 45

V. ORACLE IN RETROSPECTION AND THOUGHTS FOR THE FUTURE 46

BIBLIOGRAPHY 50

APPENDIXES 53

APPENDIX A. PROGRAM LOGIC MANUAL 54

APPENDIX B. MESSAGES AND CODES 95

VITA 99

INTRODUCTION

A compiler is a program which translates a source language into a target language. The basic functions of a compiler include lexical analysis, syntax analysis, semantic analysis and code generation, and symbol table management.

The examination of the compilation process presents several non-trivial problems. First, the computer science student needs to develop some familiarity with a source language. Obviously complex commercial languages which tend to offer many options and features to the programmer are not suited to close scrutiny within a short period of time. Therefore a subset of some existing language, or a unique language, must be defined for this purpose. This definition process may produce a source language which contains only simpler language structures. Hence, the actions performed by a compiler which recognizes an overly simplified language may leave the student with little or no understanding of some of the more interesting problems involved in compiling.

After the student is familiar with the source language, the compilation process can be examined. An ideal approach would be to have the student design and implement all phases of a compiler. However, the time constraints imposed upon a normal classroom environment would make it difficult for such projects to be realized. Thus, the procedure usually adopted is to isolate phases of compilation into separate program projects.

The components of a compiler--lexical analysis, syntax analysis, semantic analysis and code generation, as well as symbol table management--lend themselves quite readily to this method of examination because of the

relatively well defined purpose of each component. However, one fundamental factor may diminish the productivity of these projects. This factor is the environment in which the student's module will ultimately exist. An artificial environment may support the module or the module may be free standing. In neither situation will the student realize the interdependencies between the phases of compilation. Thus, a student could complete a series of subcomponents of a compiler, but not have a practical understanding of how the actions performed by one module interact or relate to the functions performed by other modules in a compiler.

In order to provide an environment conducive to the examination of each of the basic phases in compiling, the author has developed a compiler called ORACLE. This is a simple compiler which allows the student to monitor the process of compilation through a trace facility. ORACLE also provides an option which allows the student to simulate the replacement of any combination of three basic components of the compiler: lexical analysis, syntax analysis, and symbol table management.

The source language of ORACLE is designed to hasten the familiarity phase and to provide language features which would pose interesting problems in compilation. Some of these features are:

1. Multiple Data Types
2. Block Structure
3. IF-THEN-ELSE
4. DO Statement

The language accepted by ORACLE is an LL(1) language. This class of languages makes modifications easy and compilation fast and transparent.

The components of ORACLE also contain interesting applications of formal language theory. Lexical analysis employs a finite state recognizer. Syntax analysis is performed by a table driven push down automaton. Semantic analysis and code generation are performed by syntax-directed translation. The symbol table management routines employ multiple single purpose functions to manipulate a block structured symbol table.

The implementation language of ORACLE is PL/I. It is designed for use on an IBM 360/370 series computer. The structure of the compiler is completely modular. The only global variables are a set of flags and the external input and output files. ORACLE was developed over a period of one year. It is presented as an example of the benefits derived from the modular programming philosophy and in-line documentation practices.

ORACLE has not been developed as an end product. It does not yet generate executable code, but the provisions for an interpretive module have been made. ORACLE is a basic foundation from which future educational endeavors in the field of compiler design can be initiated. Furthermore, it provides the student with insight into many of the phases of compilation.

CHAPTER I

THE LANGUAGE

I. FEATURES

The first decision which was made in the development of ORACLE was the selection of the language. The intent of the project dictated that the language not be too complex; otherwise, it could not be realized within a reasonable period of time. This eliminated most commercial languages. The project also called for a language with features which could be easily understood by the student. Finally, the language needed sufficient power to permit the development of non-trivial programs.

The initial step in defining the language was to select a set of features which would meet the criteria of the language. Some of the features selected were:

Block Structure - This creates interesting problems in accessing the symbol table.

Multiple data types, integer and string - This provides enough variety to require checking for type compatibility and provides variety in the storage allocation algorithm.

IF-THEN-ELSE - Besides being a good construct for programming, this illustrates problems in syntax-directed translation; specifically, in deciding with a one token look-ahead whether it will be IF-THEN, or IF-THEN-ELSE.

UNTIL-REPEAT - This is the DO-TYPE construct of ORACLE. It offers more flexibility than the usual iteration scheme. It is patterned

after the REPEAT-UNTIL structure of PASCAL (Wirth, 1971). However, it provides for examining the logical expression before execution of the block.

Two dimensional arrays - This confronts a basic problem in storage allocation.

In addition to these basic features, the language of ORACLE has other characteristics which are similar to those offered by commercial languages. A list of these characteristics is provided for comparison:

1. The declaration of variables is mandatory.
2. There are only two data types, INTEGER and STRING.
3. There are two data structures, scalar and array.
4. Arrays may have a total of two dimensions. They have a lower bound of one. Storage for the elements is row-ordered.
5. All data are statically allocated.
6. Mixed mode operations are not permitted.
7. The set of keywords is reserved and cannot be used as identifiers. These are : BEGIN, ELSE, END, GOTO, IF, INPUT, INTEGER, OUTPUT, REPEAT, STRING, THEN, UNTIL, and the relational operators, LT, LE, EQ, GE, GT, NE.
8. A compound statement corresponds to a BEGIN-END block in PL/I. New declarations may be made upon entry into the block; however, the storage is statically allocated and is not released upon exiting the block.
9. A program is simply a labeled compound statement.
10. There is no CALL verb; i.e., no subroutines or subfunctions.
11. There is a GOTO statement.
12. Labels are identifiers followed immediately by a colon. This constitutes the declaration of a label.

13. A label does not have to be declared before it is referenced.

14. A label cannot be referenced outside of the block in which it is declared.

15. The language is free form; i.e., statements may span card boundaries. However, variables and constants may not span card boundaries.

16. The semicolon (;) is a statement separator as in PASCAL (Wirth & Jensen, 1975, p. 7).

17. The scope of a variable decends into inner blocks. In other words, a variable declared in an outer block is accessible by a block contained within the outer block.

18. Input and output are limited to one file each.

19. Blanks are not compressed. However, they have no grammatical meaning. In lexical analysis they are included in the sets of characters which delimit identifiers and integer constants.

20. A comment may appear anywhere a blank can appear. The text of a comment is enclosed by < > brackets.

II. IMPLEMENTATION RESTRICTIONS

The last requirement in the language definition involved the implementation restrictions imposed on specific tokens, as well as the limitations on the size of the programs which would be accepted by the compiler. A summary of these restrictions is provided below:

Token limitations

1. Distinct identifiers must be unique in the first eight characters.

2. Integer constants cannot exceed eight digits.

3. String constants, defined as a string of printable characters enclosed within single quotation marks, may not exceed seventy characters.

Program limitations

1. One thousand source cards.
2. One hundred distinct identifiers.
3. Fifty integer constants.
4. Fifty string constants.
5. Twenty levels of nested blocks open at one time.

CHAPTER II

THE GRAMMAR

I. BACKGROUND OF LL(1) GRAMMAR

After having defined the language to be recognized, the next phase was the definition of a grammar for the language. There were several criteria to be met.

First, a parsing strategy had already been chosen: the parser was to be table driven. This feature provided ORACLE with a capability to grow with relative ease. Essentially, any changes in the grammar would simply require changes to the parse table. Thus, a natural evolution in the grammar would have a minimal effect on the parsing algorithm of the compiler.

A second criterion was one character look-ahead. This would permit syntax errors to be immediately recognized in parsing.

Because of these two criteria, the choice was narrowed to two subclasses of Context Free grammars. One class was the LR(1) grammars, initially defined by Knuth (1965). The other was the LL(1) grammars which were first defined by Lewis and Stearns (1968). The LR(1) grammars theoretically recognize a larger group of languages than the LL(1) grammars. However, the relatively small size of the parse table of an LL(1) grammar as compared with the parse table of an LR(1) grammar made LL(1) the more desirable choice.

II. DEFINITION OF THE GRAMMAR

The next step in the development of a grammar was the formal definition. To provide better readability, the terminals of the grammar were delimited by the # symbol, and %EMPTY% was used to denote the string of length zero. The grammar was defined as follows:

GRAMMAR = (NON-TERMINALS, TERMINALS, PRODUCTIONS, PROGRAM) WHERE

NON-TERMINALS = (PROGRAM , BODY , DECLARE-BLOCK , DECLARE-VARIABLE , SIZE , 1SIZE , 2SIZE , 2DECLARE-VARIABLE , EXECUTE-BLOCK , LABEL , BLOCK , SIMPLE-STATEMENT , IF-STATEMENT , RELATION , INTEGER-EXPRESSION , SIGNED-INTEGER-TERM , INTEGER-TERM , INTEGER-WORD , INTEGER-VARIABLE , SUBSCRIPT , 2SUBSCRIPT , 2INTEGER-TERM , 2INTEGER-RELATION , STRING-EXPRESSION , STRING-VARIABLE , 2STRING-RELATION , THEN-CLAUSE , ELSE-CLAUSE , NOT-IF-STATEMENT , ASSIGNMENT , REPEAT-BLOCK , LIST , 2LIST , 2LABEL , 2INTEGER-WORD)

TERMINALS = (#UNKNOWN# , #,# , #(# , #;# , #=# , #)# , #ADDITION OPERATOR# , #MULTIPLICATION OPERATOR# , #RELATION OPERATOR# , #LABEL CONSTANT# , #INTEGER CONSTANT# , #STRING CONSTANT# , #NEW IDENTIFIER# , #LABEL IDENTIFIER# , #INTEGER IDENTIFIER# , #STRING IDENTIFIER# , #BEGIN# , #ELSE# , #END# , #GOTO# , #IF# , #INPUT# , #INTEGER# , #OUTPUT# , #REPEAT# , #STRING# , #THEN# , #UNTIL#)

PRODUCTIONS:

1. PROGRAM ::= #LABEL CONSTANT# BODY #;#

2. BODY ::= #BEGIN# DECLARE-BLOCK EXECUTE-BLOCK #END#

3. DECLARE-BLOCK ::= #INTEGER# DECLARE-VARIABLE #;# DECLARE-BLOCK

4. DECLARE-BLOCK ::= #STRING# DECLARE-VARIABLE #;# DECLARE-BLOCK

5. DECLARE-BLOCK ::= %EMPTY%

6. DECLARE-VARIABLE ::= #NEW IDENTIFIER# SIZE 2DECLARE-VARIABLE

7. SIZE ::= #(# 1SIZE 2SIZE #)#

8. SIZE ::= %EMPTY%

9. 1SIZE ::= #INTEGER CONSTANT#

10. 2SIZE ::= #,# 1SIZE

11. 2SIZE ::= %EMPTY%

12. 2DECLARE-VARIABLE ::= #,# DECLARE-VARIABLE

13. 2DECLARE-VARIABLE ::= %EMPTY%

14. EXECUTE-BLOCK ::= LABEL BLOCK EXECUTE-BLOCK

15. EXECUTE-BLOCK ::= %EMPTY%

16. LABEL ::= #LABEL CONSTANT#

17. LABEL ::= %EMPTY%

18. BLOCK ::= BODY #;#

19. BLOCK ::= SIMPLE-STATEMENT

20. SIMPLE-STATEMENT ::= IF-STATEMENT

21. SIMPLE-STATEMENT ::= NOT-IF-STATEMENT #;#

22. IF-STATEMENT ::= #IF# RELATION #THEN# THEN-CLAUSE

23. RELATION ::= INTEGER-EXPRESSION 2INTEGER-RELATION

24. RELATION ::= STRING-EXPRESSION 2STRING-RELATION

25. INTEGER-EXPRESSION ::= SIGNED-INTEGER-TERM 2INTEGER-TERM

26. SIGNED-INTEGER-TERM ::= #ADDITION OPERATOR# INTEGER-TERM

27. SIGNED-INTEGER-TERM ::= INTEGER-TERM

28. INTEGER-TERM ::= INTEGER-WORD 2INTEGER-WORD

29. INTEGER-WORD ::= #(# INTEGER-EXPRESSION #)#

30. INTEGER-WORD ::= INTEGER-VARIABLE

31. INTEGER-WORD ::= #INTEGER CONSTANT#

32. INTEGER-VARIABLE ::= #INTEGER IDENTIFIER# SUBSCRIPT

33. SUBSCRIPT ::= #(# INTEGER-EXPRESSION 2SUBSCRIPT #)#

34. SUBSCRIPT ::= %EMPTY%

35. 2SUBSCRIPT ::= #,# INTEGER-EXPRESSION

36. 2SUBSCRIPT ::= %EMPTY%

37. 2INTEGER-WORD ::= #MULTIPLICATION OPERATOR# INTEGER-WORD 2INTEGER-WORD

38. 2INTEGER-WORD ::= %EMPTY%

39. 2INTEGER-TERM ::= #ADDITION OPERATOR# INTEGER-TERM 2INTEGER-TERM

40. 2INTEGER-TERM ::= %EMPTY%

41. 2INTEGER-RELATION ::= #RELATION OPERATOR# INTEGER-EXPRESSION

42. STRING-EXPRESSION ::= STRING-VARIABLE

43. STRING-EXPRESSION ::= #STRING CONSTANT#

44. STRING-VARIABLE ::= #STRING IDENTIFIER# SUBSCRIPT

45. 2STRING-RELATION ::= #RELATION OPERATOR# STRING-EXPRESSION

46. THEN-CLAUSE ::= BODY #;# ELSE-CLAUSE

47. THEN-CLAUSE ::= NOT-IF-STATEMENT #;# ELSE-CLAUSE

48. THEN-CLAUSE ::= IF-STATEMENT

49. ELSE-CLAUSE ::= #ELSE# BLOCK

50. ELSE-CLAUSE ::= %EMPTY%

51. NOT-IF-STATEMENT ::= ASSIGNMENT

52. NOT-IF-STATEMENT ::= #UNTIL# REPEAT-BLOCK #REPEAT#

53. NOT-IF-STATEMENT ::= #INPUT# LIST

54. NOT-IF-STATEMENT ::= #OUTPUT# LIST

55. NOT-IF-STATEMENT ::= #GOTO# 2LABEL

56. ASSIGNMENT ::= INTEGER-VARIABLE #=# INTEGER-EXPRESSION

57. ASSIGNMENT ::= STRING-VARIABLE #=# STRING-EXPRESSION

58. REPEAT-BLOCK ::= RELATION #;# EXECUTE-BLOCK

59. LIST ::= INTEGER-VARIABLE 2LIST

60. LIST ::= STRING-VARIABLE 2LIST

61. 2LIST ::= #,# LIST

62. 2LIST ::= %EMPTY%

63. 2LABEL ::= #NEW IDENTIFIER#

64. 2LABEL ::= #LABEL IDENTIFIER#

III. DISCUSSION OF LL(1) CONDITIONS

After the formal definition of the grammar was established, it was necessary to show that the grammar was LL(1). The necessary conditions for a grammar to be LL(1) have been recently discussed by Aho and Ullman (1972, pp. 333-367) and independently by Griffiths (1975). Essentially, these conditions require that in a left-to-right, top-down scan any non-terminal in the grammar can be deterministically expanded by a one character look-ahead in the input string.

To show that the grammar recognized by ORACLE satisfies the LL(1) conditions, two functions must be defined. The notation used in these definitions is that of Aho and Ullman (1972, p. 343).

Let G = (N, T, P, S) where

N = set of non-terminals

T = set of terminals

P = set of productions

and S = the start symbol.

Suppose $A \in N$ has the productions:

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n .$$

Define

$$\text{FIRST } (\alpha_i) = \{t \in T \mid \alpha_i \overset{*}{\rightarrow} t\beta,\ \beta \in (N \cup T)^*\}.$$

Also define

$$\text{FOLLOW } (A) = \{t \in T \mid S \overset{*}{\rightarrow} \gamma A t \omega;\ \gamma, \omega \in (N \cup T)^*\}.$$

Then a grammar, G is said to be LL(1) if and only if the following conditions are satisfied:

1. $\text{FIRST } (\alpha_i) \cup \text{FIRST } (\alpha_j) = \phi$, for all $i \neq j$.
2. If $\alpha_i \overset{*}{\rightarrow} e$, then

   $\text{FIRST } (\alpha_j) \cap \text{FOLLOW } (A) = \phi$, for all $j \neq i$.

Thus, a mechanical inspection of the productions of the grammar recognized by ORACLE will suffice to show that it is LL(1). The conditions are obviously satisfied for those non-terminals in the grammar which have only one production whose left-most element is a terminal. The following discussion provides for the inspection of the more complex cases. For convenience, these cases are divided into two categories.

First, the non-terminals which do not contain the empty string production are examined using the FIRST and FOLLOW functions. For notational purposes Pi represents the right-hand side of the i^{th} production. These non-terminals are:

BLOCK

FIRST (P18) = (#BEGIN#)

FIRST (P19) = (#IF# , #INTEGER IDENTIFIER# , #STRING IDENTIFIER# , #INPUT# , #OUTPUT# , #GOTO# , #UNTIL#)

SIMPLE-STATEMENT

FIRST (P20) = (#IF#)

FIRST (P21) = (#NTEGER IDENTIFIER# , #STRING IDENTIFIER# , #INPUT# , #OUTPUT# , #GOTO# , #UNTIL#)

RELATION

FIRST(P23) = (#ADDITION OPERATOR# , #(# , #INTEGER IDENTIFIER# , #INTEGER CONSTANT#)

FIRST (P24) = (#STRING IDENTIFIER# , #STRING CONSTANT#)

INTEGER-EXPRESSION

FIRST (P25) = (#ADDITION OPERATOR# , #(# , #INTEGER IDENTIFIER# , #INTEGER CONSTANT#)

SIGNED-INTEGER-TERM

FIRST (P26) = (#ADDITION OPERATOR#)

FIRST (P27) = (#(# , #INTEGER IDENTIFIER# , #INTEGER CONSTANT#)

INTEGER-TERM

FIRST (P28) = (#(# , #INTEGER IDENTIFIER# , #INTEGER CONSTANT#)

INTEGER-WORD

FIRST (P29) = (#(#)

FIRST (P30) = (#INTEGER IDENTIFIER#)

FIRST (P31) = (#INTEGER CONSTANT#)

STRING-EXPRESSION

FIRST (P42) = (#STRING IDENTIFIER#)

FIRST (P43) = (#STRING CONSTANT#)

THEN-CLAUSE

FIRST (P46) = (#BEGIN#)

FIRST (P47) = (#INTEGER IDENTIFIER# , #STRING IDENTIFIER# , #INPUT# , #OUTPUT# , #GOTO# , #UNTIL#)

FIRST (P48) = (#IF#)

NOT-IF-STATEMENT

FIRST (P51) = (#INTEGER IDENTIFIER# , #STRING IDENTIFIER#)

FIRST (P52) = (#UNTIL#)

FIRST (P53) = (#INPUT#)

FIRST (P54) = (#OUTPUT#)

FIRST (P55) = (#GOTO#)

ASSIGNMENT

FIRST (P56) = (#INTEGER IDENTIFIER#)

FIRST (P57) = (#STRING IDENTIFIER#)

REPEAT-BLOCK

FIRST (P58) = (#ADDITION OPERATOR# , #(# , #INTEGER IDENTIFIER# , #INTEGER CONSTANT# , #STRING IDENTIFIER# , #STRING CONSTANT#)

LIST

FIRST (P59) = (#INTEGER IDENTIFIER#)

FIRST (P60) = (#STRING IDENTIFIER#)

2LABEL

FIRST (P63) = (#NEW IDENTIFIER#)

FIRST (P64) = (#LABEL IDENTIFIER#)

An examination of the remaining non-terminals will be presented in a similar manner. The following is an inspection of the non-terminals which contain the empty string production in their sets of productions:

DECLARE-BLOCK

FIRST (P3) = (#INTEGER#)

FIRST (P4) = (#STRING#)

FOLLOW (DECLARE-BLOCK) = (#LABEL CONSTANT# , #BEGIN# , #IF# , #INTEGER IDENTIFIER#, #STRING IDENTIFIER# , #INPUT# , #OUTPUT# , #GOTO# , #UNTIL#)

SIZE

FIRST (P7) = (#(#)

FOLLOW (SIZE) = (#,# , #;#)

2SIZE

FIRST (P10) = (#, #)

FOLLOW (2SIZE) = (#)#)

2DECLARE-VARIABLE

FIRST (P12) = (#, #)

FOLLOW (2DECLARE-VARIABLE) = (#;#)

EXECUTE-BLOCK

FIRST (P14) = (#LABEL CONSTANT# , #BEGIN# , #INTEGER IDENTIFIER# , #STRING IDENTIFIER# , #INPUT# , #OUTPUT# , #GOTO# , #UNTIL#)

FOLLOW (EXECUTE-BLOCK) = (#END# , #REPEAT#)

SUBSCRIPT

FIRST (P33) = (#(#)

FOLLOW (SUBSCRIPT) = (#MULTIPLICATION OPERATOR# , #ADDITION OPERATOR# , #RELATION OPERATOR# , #THEN# , #)# , #;# , #=# , #, #)

2SUBSCRIPT

FIRST (P35) = (#, #)

FOLLOW (2SUBSCRIPT) = (#)#)

2INTEGER-WORD

FIRST (P37) = (#MULTIPLICATION OPERATOR#)

FOLLOW (2INTEGER-WORD) = (#ADDITION OPERATOR# , #RELATION OPERATOR# , #THEN# , #)# , #;#)

2INTEGER-TERM

FIRST (P39) = (#ADDITION OPERATOR#)

FOLLOW (2INTEGER-TERM) = (#RELATION OPERATOR# , #THEN# , #)# , #;#)

ELSE-CLAUSE

FIRST (P49) = (#ELSE#)

FOLLOW (ELSE-CLAUSE) = (#LABEL CONSTANT# , #BEGIN# , #IF# , #INTEGER IDENTIFIER# , #STRING IDENTIFIER# , #UNTIL# , #INPUT# , #OUTPUT# , #GOTO# , #END# , #REPEAT#)

2LIST

FIRST (P61) = (#, #)

FOLLOW (2LIST) = (#;#)

Thus, this rather tedious examination shows conclusively that the grammar recognized by ORACLE is LL(1).

CHAPTER III

THE COMPILER

I. LEXICAL ANALYSIS

Technique and Background

After the language and the grammar were defined, the design of the compiler was begun. The four basic components of a compiler--lexical analysis, syntax analysis, semantic analysis and code generation, as well as symbol table management, were separated into distinct design modules to expedite the development of ORACLE.

The first module developed was lexical analysis. The primary function of the lexical phase is recognizing the terminals of the grammar. These terminals are usually referred to as tokens in a compiler.

The first step in this recognition process was defining the tokens in terms of their representation in the input string. For example, an identifier token was defined as a string of alphabetic and numeric characters which begins with an alphabetic character. (See Appendix A for a definition of each token recognized by ORACLE.)

The next step was to design a recognizer. This was done by adapting the finite state recognizer developed by Gries (1971, pp. 52-83) to the token definitions. The adaptation was itself rather simple. First, the finite state machine was diagrammed (Figure 1) and then it was implemented in PL/I code.

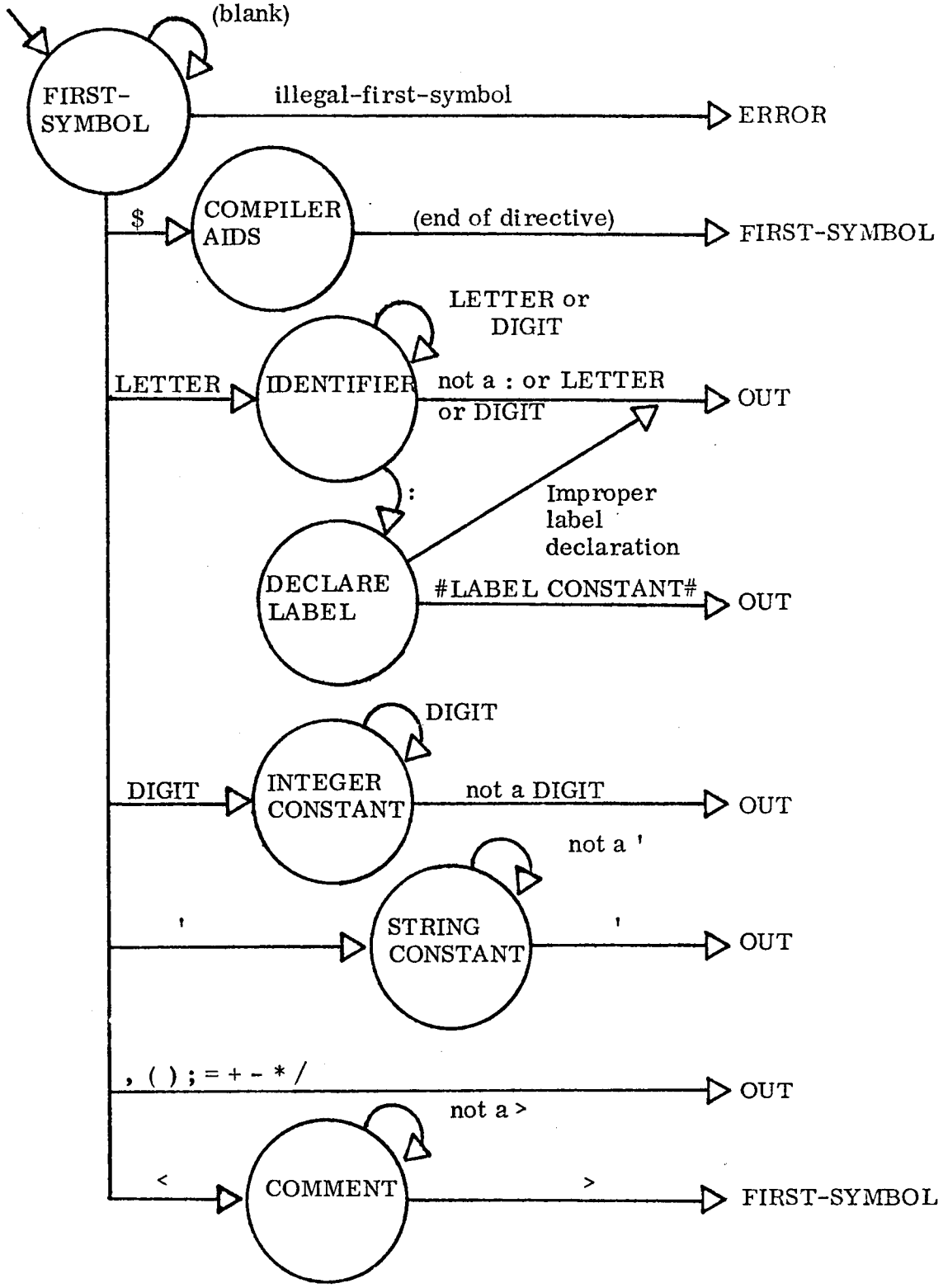


FIGURE 1

DIAGRAM OF FINITE STATE RECOGNIZER

The efficiency of the lexical phase is a basic by-product of its simplicity. However, it was necessary to include several non-lexical activities in this module. One of these is the initiation of a symbol table search.

The semantics of the language is embedded in the syntax. Therefore, type characteristics associated with identifiers and constants are required by the parser. The lexical phase recognizes the general classes of tokens: identifiers, integer constants, etc.; however, it is in the symbol table that the attributes associated with each unique variable are maintained. Thus, the responsibility for initiating a symbol table search for token classes lies with lexical analysis.

Another non-lexical responsibility of this module is the resolution of improper label declarations. This may include the multiple declaration of a label within a block or the declaration of an identifier as a label which has previously been declared in the same block.

Error Analysis

The lexical analysis module includes a certain amount of checking for error conditions. For example, the classification of a token as #UNKNOWN# requires providing a trap state for tokens which cannot be recognized. Upon the recognition of an error, a message is printed. (See Appendix B for a complete listing of error messages generated by lexical analysis.)

There is one special error trap in lexical analysis. This is the recognition of the end of the source program. This results in an end of text message and the generation of a special token which signifies that the input string is empty.

II. SYMBOL TABLE MANAGEMENT

Technique and Background

The next phase in the design of ORACLE was the symbol table management module. This course was chosen for two reasons. First, the symbol table development could itself be modularized to allow the addition of new features as they were needed. And second, the symbol table module would provide a good companion for the lexical module in an initial test environment. Thus, the interconnections between the first stages could be solidified early in development.

The foundation for the design of the symbol table management module was suggested by McKeeman (1975). This scheme associates entry points in the module with singular functions on the symbol table. For example, the entry point INSERT initiates the insertion of an identifier into the symbol table at the current level. (See Appendix A for a detailed discussion of the functions associated with each entry point in the symbol table module.)

The initial step in the design of the symbol table management module was the definition of the structure of the data which would be manipulated by the management routines. The basis for developing this definition was the search technique which was employed.

A simple sequential search algorithm was chosen. Two primary factors contributed to this decision. One was the relatively small size of the symbol table. The largest list contains only one hundred entries. The other was the frequency with which the entire symbol table would be searched. This was to be done only once for each token after it was recognized in lexical analysis. Thus, a more complex search technique was not dictated.

The search algorithm is aided by the separation of the symbol table into three LAST-IN-FIRST-OUT (LIFO) lists. These lists are generally defined by the classes of tokens recognized in lexical analysis: identifiers and keywords, integer constants, and string constants.

The identifier list has block structure superimposed on the LIFO list. This was a major feature provided by the language. The management of this list departs from the stack technique discussed by McKeeman (1975). Pointers are maintained to define the beginning and the ending of each block within the symbol table. These blocks correspond directly with the block structure of the source program. When a block is exited, the pointers are removed from access. Thus, a search or a query on the symbol table can access only active blocks. This adaptation leaves the entire symbol table available for post compilation activities, such as a postmortem dump. This also provides some flexibility for future additions to the symbol table management routines. For example, attribute listings could be generated and access to the symbol table could be provided for run-time diagnostics. (See Appendix A for a complete definition of each list which comprises the symbol table.)

The final stage of the development of the symbol table management module was the definition of the functions which would be performed. These functions are separated into five categories. The first two--search and insert--have been discussed previously. A set of routines to update and to examine the depth of the block structured identifier list make up the third category. The remaining categories define or fetch the value assigned to an attribute. (See Appendix A for a detailed examination of each category of symbol table functions.)

Error Analysis

Each phase in the development of the symbol table management module involved the recognition of error situations. Many of the implementation restrictions imposed on the user are based on the actions in this module. For example, distinct identifiers must be unique in the first eight characters because the symbol table stores only these characters. The symbol table also has size limitations which were imposed to hasten the implementation of the project. (See Appendix B for a complete listing of the diagnostics generated by the symbol table module.)

III. SYNTAX ANALYSIS

Technique and Background

The next phase in the design of ORACLE was the development of the syntax analysis module. This module is the heart of ORACLE. It is responsible for the initiation and termination of compilation. The primary function of this module is the syntactic evaluation of the input string. The syntax is provided by the language defined in Chapter I.

The evaluation process is performed by a table driven parser which utilizes a one token look-ahead to effect a deterministic, top down parse. The theoretical foundation for this class of LL(1) parsers is discussed extensively by Aho and Ullman (1972, pp. 333-368). The choice of an LL(1) technique was based on the fact that the parser tends to be small and fast, and also provides good error detection.

The development of the parser was first concerned with the construction of the parse table. As discussed in Chapter II, a table driven approach was chosen to provide a degree of flexibility in the design of

ORACLE. It allows a change to be made in the language with a minimal effect on the parser, because changes in syntax are incorporated by simply updating the parse table. Thus, the parser can evolve simultaneously with the language.

The second stage of the design of the parser was the development of the parsing actions. The essential actions dictated by the parse table are as follows:

POP - Pop the top of the parse stack.

PUSH - Push the right hand side of a specified production onto the parse stack.

ERROR - Trap state for syntax errors.

Hence, with the exception of the ERROR action, parsing was reduced to simple stack manipulations.

Error Analysis and Recovery

After the parse table and the general parsing actions were defined, the development of syntax error recovery actions was begun. As noted previously, the LL(1) parsing technique provides good error detection capabilities, a result of its deterministic characteristics. Since a one token look-ahead is all that is required to make a decision, many error situations can be recognized immediately. However, a generalized scheme of syntax error recovery has not yet been developed. Therefore, syntax error recovery is at best ad hoc. An excellent survey of current syntax error recovery strategies is provided by Horning (1975).

In an effort to structure syntax error recovery within ORACLE, four syntax error recovery categories were established. These were: TOKEN, STATEMENT, BLOCK, and UNRECOVERABLE. This structuring of

recovery actions is based on the methods developed by Holt (1973) for the SP/k compiler. Each category is related to a specific set of circumstances which would arise as error situations in parsing.

The first category, TOKEN, recognizes error situations in which a change in the token value will allow parsing to continue. Several strategies were chosen to implement this capability. One well-documented strategy employed by ORACLE is the use of a set of productions called "error productions" to augment the grammar (Aho & Johnson, 1974; Lenius, 1970; Lenius, 1976; Wirth, 1968).

This category also employes several trivial recovery strategies. Essentially these change or assume a token value, or they request a new token from lexical analysis. For example, a token value of #UNKNOWN# will generally be ignored in parsing; i.e., another token will be requested from lexical analysis. This request is initiated by the TOKEN error routine.

A more severe class of errors is handled in the second category, STATEMENT. The circumstances which generally describe this class of errors are outlined below:

1. The actions provided by TOKEN may not permit parsing to resume in a meaningful manner.

2. A "solid" token, such as a semicolon, is assured to be on the parse stack.

This class of syntax error recovery employs the strategy "PANIC MODE," a phrase coined by Graham and Rhodes (1973). This strategy is divided into two steps. First, the input string is searched until a "solid" token, such as a semicolon, is detected. Then the parse stack is flushed until a corresponding "solid" token is found.

This mode has some undesirable consequences. First, some of the input string remains unscanned. And second, the semantic activities must be synchronized with the parse stack because some semantic actions may have been performed for the unparsed portion of the syntax stack which was flushed. There is one favorable feature of "PANIC MODE." It assures that error recovery will terminate at some point, because the end of the input string is guaranteed to be a "solid" token.

Equally severe errors are handled in category three, BLOCK. The general circumstances which describe this class of error recovery are outlined below:

1. The actions provided by TOKEN may not permit parsing to resume in a meaningful manner.

2. The existence of a "solid" token near the top of the parse stack cannot be assured.

The strategy employed in this category simply searches the input string for a "solid" token.

The most severe errors are handled by category four, UNRECOVERABLE. This category presents no real attempt at recovery; however, it is offered for completeness. The errors handled by this category are compiler errors and they are not related to syntax errors in the source program. For example, a token value of #UNKNOWN# on the parse stack would result in an UNRECOVERABLE error. The actions provided are a parse stack dump and termination. (See Appendix A for details on error recovery.)

IV. SEMANTIC ANALYSIS AND CODE GENERATION

Technique and Background

The final phase of the design of ORACLE was the development of the semantic analysis and code generation module. The primary function of this module is the generation of intermediate code. The form of the intermediate code is Polish suffix; a parenthesis-free notation originally developed by Lukasiewicz (1929). The generation strategy is syntax-directed translation. The theory of this strategy is discussed by Aho and Ullman (1972, pp. 215-223). This technique was chosen because of the simplicity and the efficiency with which syntax-directed translation effects the generation of the intermediate code. This was done by associating a syntax-directed translation element with each production in the grammar.

In the design of this module, the activities were subdivided into two program modules. One was the generative module, called semantic analysis, which performs the syntax-directed translation. The other was essentially a service module, called code generation, which provides for the collection of the Polish suffix code into a code stream suitable for processing by an interpreter.

The first stage in the development of semantic analysis was the definition of the syntax-directed translation elements. These elements were defined as follows:

1. #DCL LABEL# BODY

2. #GEN BEGIN OP# DECLARE-BLOCK EXECUTE-BLOCK #GEN END OP#

3. #DCL INTEGER# DECLARE-VARIABLE #;# DECLARE-BLOCK

4. #DCL STRING# DECLARE-VARIABLE #;#

DECLARE-BLOCK

5. %EMPTY%

6. #DCL ADDRESS# SIZE #DCL ID#

2DECLARE-VARIABLE

7. 1SIZE #DCL COLUMN# 2SIZE #DCL ROW#

8. %EMPTY%

9. #DCL DIMENSION VALUE#

10. 1SIZE

11. %EMPTY%

12. DECLARE-VARIABLE

13. %EMPTY%

14. LABEL BLOCK EXECUTE-BLOCK

15. %EMPTY%

16. #DCL LABEL#

17. %EMPTY%

18. BODY #;#

19. SIMPLE-STATEMENT

20. IF-STATEMENT

21. NOT-IF-STATEMENT #;#

22. RELATION #THEN# THEN-CLAUSE

23. INTEGER-EXPRESSION 2INTEGER-RELATION

24. STRING-EXPRESSION 2STRING-RELATION

25. SIGNED-INTEGER-TERM 2INTEGER-TERM

26. #DCL SIGN# INTEGER-TERM #GEN OP#

27. INTEGER-TERM

28. INTEGER-WORD 2INTEGER-WORD

29. INTEGER-EXPRESSION

30. INTEGER-VARIABLE #GEN VALUE-OF-INTEGER OP#

31. #GEN OPERAND# #GEN VALUE-OF-INTEGER OP#

32. #GEN OPERAND# SUBSCRIPT

33. INTEGER-EXPRESSION 2SUBSCRIPT

34. #GEN SCALAR#

35. INTEGER-EXPRESSION #GEN ROW MAXIMUM#
#GEN SUBSCRIPT2-OF OP#

36. #GEN SUBSCRIPT1-OF OP#

37. #DCL OP TYPE = INTEGER# INTEGER-WORD #GEN OP#
2INTEGER-WORD

38. %EMPTY%

39. #DCL OP TYPE = INTEGER# INTEGER-TERM #GEN OP#
2INTEGER-TERM

40. %EMPTY%

41. #DCL OP TYPE = INTEGER# INTEGER-EXPRESSION
#GEN OP#

42. STRING-VARIABLE #GEN VALUE-OF-STRING OP#

43. #GEN OPERAND# #GEN VALUE-OF-STRING OP#

44. #GEN OPERAND# SUBSCRIPT

45. #DCL OP TYPE = STRING# STRING-EXPRESSION #GEN OP#

46. BODY #;# ELSE-CLAUSE

47. NOT-IF-STATEMENT #;# ELSE-CLAUSE

48. IF-STATEMENT

49. BLOCK #GEN IF-THEN-ELSE OP#

50. #GEN IF-THEN OP#

51. ASSIGNMENT

52. #GEN UNTIL OP# REPEAT-BLOCK #GEN REPEAT OP#

53. #DCL IO OP = INPUT# LIST

54. #DCL IO OP = OUTPUT# LIST

55. 2LABEL #GEN GOTO OP#

56. INTEGER-VARIABLE INTEGER-EXPRESSION #GEN INTEGER-ASSIGN OP#

57. STRING-VARIABLE STRING-EXPRESSION #GEN STRING-ASSIGN OP#

58. RELATION #;# EXECUTE-BLOCK

59. #DCL IO TYPE = INTEGER# INTEGER-VARIABLE #GEN OP# 2LIST

60. #DCL IO TYPE = STRING# STRING-VARIABLE #GEN OP# 2LIST

61. LIST

62. %EMPTY%

63. #GEN LABEL#

64. #GEN LABEL#

These elements defined a set of semantic routines (Feldman, 1966). These routines were separated into two general categories based upon the actions which were performed. The first category was concerned with the processing of declarations. The second category was concerned with the actual generation of the intermediate code. (See Appendix A for a detailed discussion of the semantic routines.) As previously discussed, the simplification of the semantic routines was aided by the semantic action

performed in syntax analysis. For example, type checking was provided by syntax analysis.

Error Analysis

As in other modules of the compiler a certain amount of error checking was provided in the semantic analysis and code generation module.

In the program module called semantic analysis, this facility was primarily a function which was initiated by syntax analysis in syntax error recovery. These actions permitted the semantic stack to be synchronized with the parse stack in a flush operation. For convenience a special NO-OP operator was included in the syntax-directed translation elements. This operator corresponds to the "solid" tokens recognized in the parse stack. They are denoted in the syntax-directed translation elements as #;# and #THEN#.

The code generation program module provides error checking to insure that the intermediate code stream does not exceed the implementation restriction of 16-K words.

V. COMPILER OPTIONS

Trace Option

Concurrent with the design of the compiler modules was the development of the features in ORACLE which would realize the intent of the project. One of these features was called the TRACE OPTION. This facility would allow the student to selectively monitor parts of the compilation process. A request for TRACE output is provided by source statements. (See Appendix A for a description of the format of the TRACE request statement.) An outline of the output produced by this option is provided below:

A. Lexical Analysis

1. Module entry and exit
2. Input to and decisions of the finite state recognizer
3. Initial token classification
4. Construction of the token recognized in the input string
5. Final token classification, symbol table address, and input string values returned to Syntax Analysis

B. Syntax Analysis

1. Module entry and exit
2. Values returned by Lexical Analysis (same as A.5 above - these values are displayed only once if both Lexical and Syntax trace flags are active)
3. Productions applied in PUSH operation
4. Non-terminals and terminals popped in a POP operation
5. Dump of parse stack on abnormal termination of compilation

C. Semantic Analysis

1. Routine entry and exit
2. Syntax-directed translation elements applied
3. Semantic action begin and end
4. Semantic stack POP operations
5. Dump of semantic stack on abnormal termination

D. Code Generation

1. Routine entry and exit

E. Symbol Table

1. Routine entry and exit
2. Initial definition of new symbol table entries

3. Current level on scope query
4. Updated entry on set attribute query
5. Dump of symbol table on abnormal termination

Debug Option

Another feature of ORACLE was developed primarily as an implementation aid for the author. This feature was called the DEBUG OPTION. Like the TRACE OPTION, output generated by the DEBUG OPTION is requested by statements in the source program. (See Appendix A for a description of the format of a DEBUG request statement.) Essentially the output generated by the DEBUG OPTION augments the output generated by the TRACE OPTION to provide a complete label trace within the compiler.

CHAPTER IV

REPLACEMENT MODE

I. OVERVIEW

The second feature of ORACLE which was developed as a pedagogic aid was called Replacement Mode. This feature allows ORACLE to proceed one step beyond the usual compiler configuration. Replacement Mode permits the student to simulate the replacement of any combination of three basic compiler components of ORACLE: symbol table management, lexical analysis, and syntax analysis. This operation is provided through the CALL verb in PL/I. Upon recognition of a request for replacement operations, the student's replacement module will be called for initialization and during specific phases of compilation it will be invoked to perform its function.

The student never actually replaces a module within the compiler. Thus, the student's replacement module and ORACLE's standard one are present at the same time. This allows ORACLE to check the actions of the student's module for correctness and consistency. In general, the criterion for checking the replacement module is provided by values returned to ORACLE in a parameter list.

A request for Replacement Mode is provided by two keywords which are passed to ORACLE via the PARM field. One of these keywords specifies the level of replacement and the other keyword specifies a termination code. (See Appendix A for the values which may be coded in the PARM field.)

The level of replacement is defined by the hierarchy which was imposed upon the replacement activities. The lowest level in the hierarchy is called Replacement Mode 1. This mode allows the student to replace symbol table management.

The next level is called Replacement Mode 2. This mode allows the student to replace lexical analysis. An option of Replacement Mode 2 allows the student to replace symbol table management simultaneous with the replacement of lexical analysis. However, consistency checking for the student's symbol table management module is not provided by Replacement Mode 2.

The highest level in the hierarchy is called Replacement Mode 3. This allows the student to replace syntax analysis. The options provided by Replacement Mode 3 allow the student to provide lexical analysis and symbol table management modules for simultaneous replacement. As previously noted, the optional replacement modules are not checked by ORACLE. In addition to the optional modules for Replacement Mode 3, ORACLE supplies the semantic analysis and code generation components as a non-replacable module. Thus, two complete copies of a compiler are active in Replacement Mode 3.

The termination code provides ORACLE with a criterion to terminate Replacement Mode operations. For modes 1 and 2, this code is ignored and replacement is unconditional. However, for Replacement Mode 3, this code allows ORACLE to terminate the replacement of syntax analysis upon the recognition of a syntax error. The student may override this by requesting unconditional replacement. This termination criterion was developed because of the ad hoc nature of syntax error recovery. In other words, the unique way

in which ORACLE handles syntax error recovery and the way in which a student's replacement module performs this function may make the evaluation of the student's syntax analysis module meaningless.

In the development and the implementation of this feature it was necessary to impose some restrictions on the usage of Replacement Mode. The following is a list of these restrictions:

1. The replacement modules must be coded in PL/I. This is due to the PL/I environment established by ORACLE.

2. A standardized set of entry names and parameter list attributes must be used. (See Appendix A.) This is primarily due to the activation of replacement through the PL/I CALL verb.

3. The input and output files should be named and described as those defined by ORACLE. (See Appendix A.) This permits ORACLE to open these files.

II. REPLACEMENT MODE 1 - SYMBOL TABLE MANAGEMENT

Activation and Input Requirements

The lowest level of Replacement Mode activity is called Replacement Mode 1. This mode allows the student to simulate the replacement of symbol table management. Replacement Mode 1 is activated by specifying M=1 in the PARM field. In addition, the student should supply a replacement module which could be processed by the Linkage Editor, possibly in a prepass.

Communication

After Replacement Mode 1 has been activated, a call is made to the student's module for initialization. Subsequent activation is made during compilation by ORACLE's symbol table management module. The general flow of

these subsequent calls is as follows:

1. Entry into ORACLE's symbol table management module will establish a compatible entry point and initial parameter values for the replacement module.

2. Before returning to the invoking module within ORACLE, the replacement module will be invoked.

3. Upon return from the replacement module, an evaluation of its actions will be performed.

For a visual perspective of these activities, see Figure 2.

In the design of Replacement Mode 1, the restrictions on the structure of the student's module were minimized. The student has complete freedom in the design of the symbol table and in the selection of management strategies which will manipulate the data in the symbol table. Also, a student does not need to implement all of the management routines at one time. Thus, these routines can evolve in a gradual and natural way.

Evaluation

The checking procedures applied by Replacement Mode 1 are primarily concerned with the consistency with which the replacement module accesses its symbol table entries. In the context of Replacement Mode 1 consistency is defined in the following manner: Access of a symbol table is said to be consistent if after an entry is inserted into the symbol table for the first time, then subsequent searches on the symbol table for the same entry will find it at the address assigned by the initial insertion. Otherwise, access is said to be inconsistent. Thus, any technique which moves entries after the initial insertion is made, such as direct chaining as discussed by Morris (1968), cannot be evaluated by ORACLE.

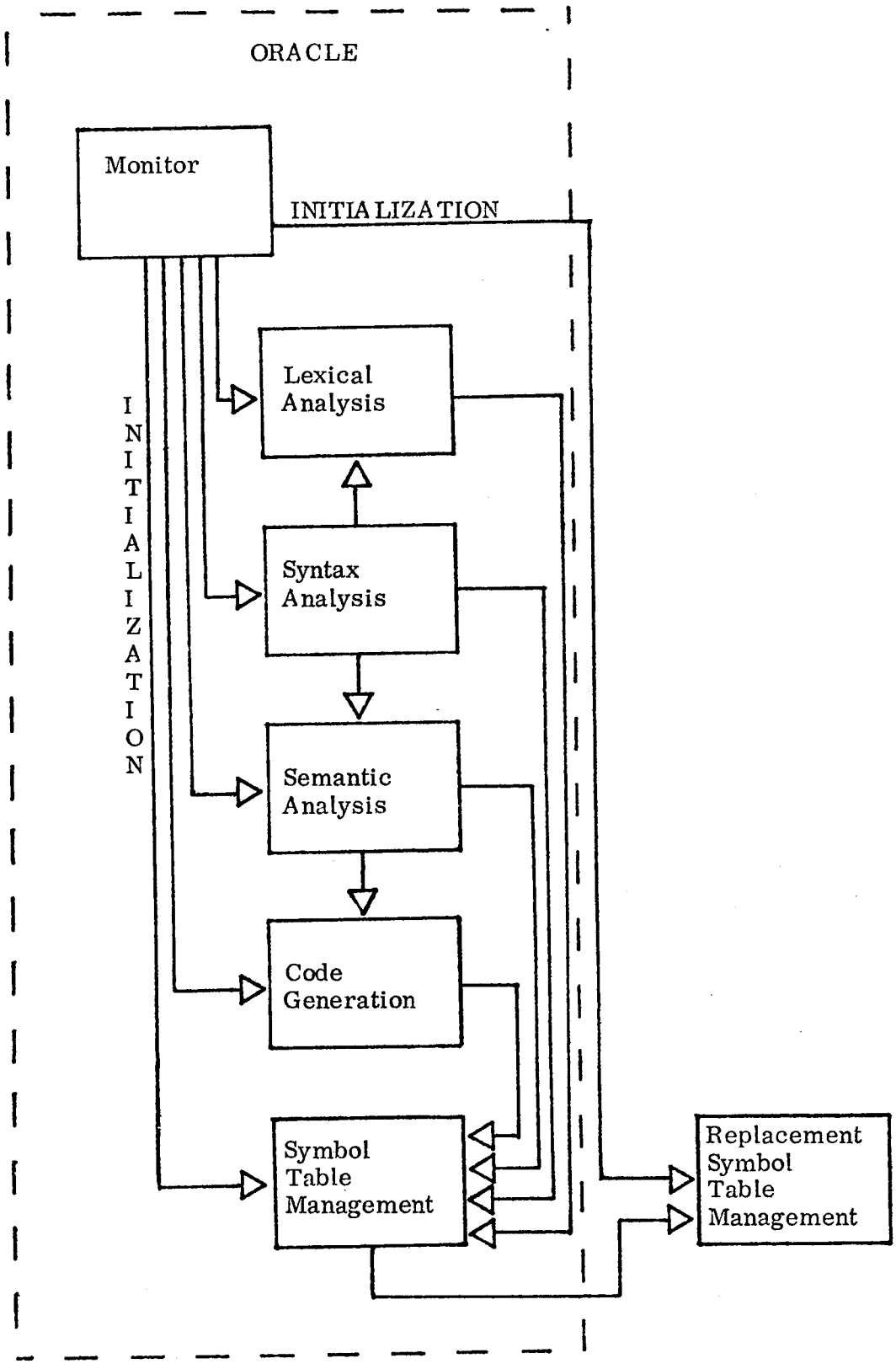


FIGURE 2

GRAPHIC VIEW OF REPLACEMENT MODE 1

III. REPLACEMENT MODE 2 - LEXICAL ANALYSIS

Activation and Input Requirements

The next level in the hierarchy of Replacement Mode is called Replacement Mode 2. This mode allows the student to simulate the replacement of lexical analysis. Replacement Mode 2 is activated by specifying M=2 in the PARM field. In addition, the student should supply a replacement module which can be processed by the Linkage Editor and a second copy of the source program. The student may also supply a replacement module for symbol table management. If the student does not supply a replacement module for symbol table management, ORACLE will provide a copy of its own symbol table management module for use in this mode. As noted previously, ORACLE does not provide checking for symbol table management in Replacement Mode 2.

The availability of two copies of the source program and a second symbol table management module allows ORACLE to resolve a possible synchonization problem in a relatively simple manner.

This problem arises because the source program and the symbol table are resources. It would be difficult to control the use of these resources if only one copy were available. For example, in processing the source program the student may use a look-ahead in lexical analysis which is greater than the one character look-ahead employed by ORACLE. Thus, the student could request a new card before ORACLE had completed processing the present card. Symbol table management could have similar problems. For example, lexical analysis could request an insertion to be made in the symbol table. If only one symbol table were available, it could be the case that two locations in the symbol table would contain identical values. Thus,

symbol table access would appear to be inconsistent in subsequent searches.

Communication

After Replacement Mode 2 has been recognized, a call is made to the student's module for initialization. Subsequent activations of the lexical analysis replacement module are made by ORACLE's lexical analysis module. The general flow of these subsequent calls is as follows:

1. Before returning to syntax analysis ORACLE's lexical analysis module will invoke the replacement module.

2. Upon return from the lexical analysis replacement module, an evaluation of the actions performed by the lexical analysis replacement module will be made by ORACLE.

These operations are graphically presented in Figure 3.

Evaluation

The evaluation process used by ORACLE in Replacement Mode 2 is a simple comparison of the token recognized by the student's replacement module with the token recognized by its own lexical routine. This process assumes that the token values are the same as those defined by ORACLE. (See Appendix A for a complete list of the token values and the input strings which they represent.)

IV. REPLACEMENT MODE 3 - SYNTAX ANALYSIS

Activation and Input Requirements

The highest level in the hierarchy of Replacement Mode is called Replacement Mode 3. This mode allows the student to simulate the replacement of syntax analysis. Replacement Mode 3 is activated by specifying M=3

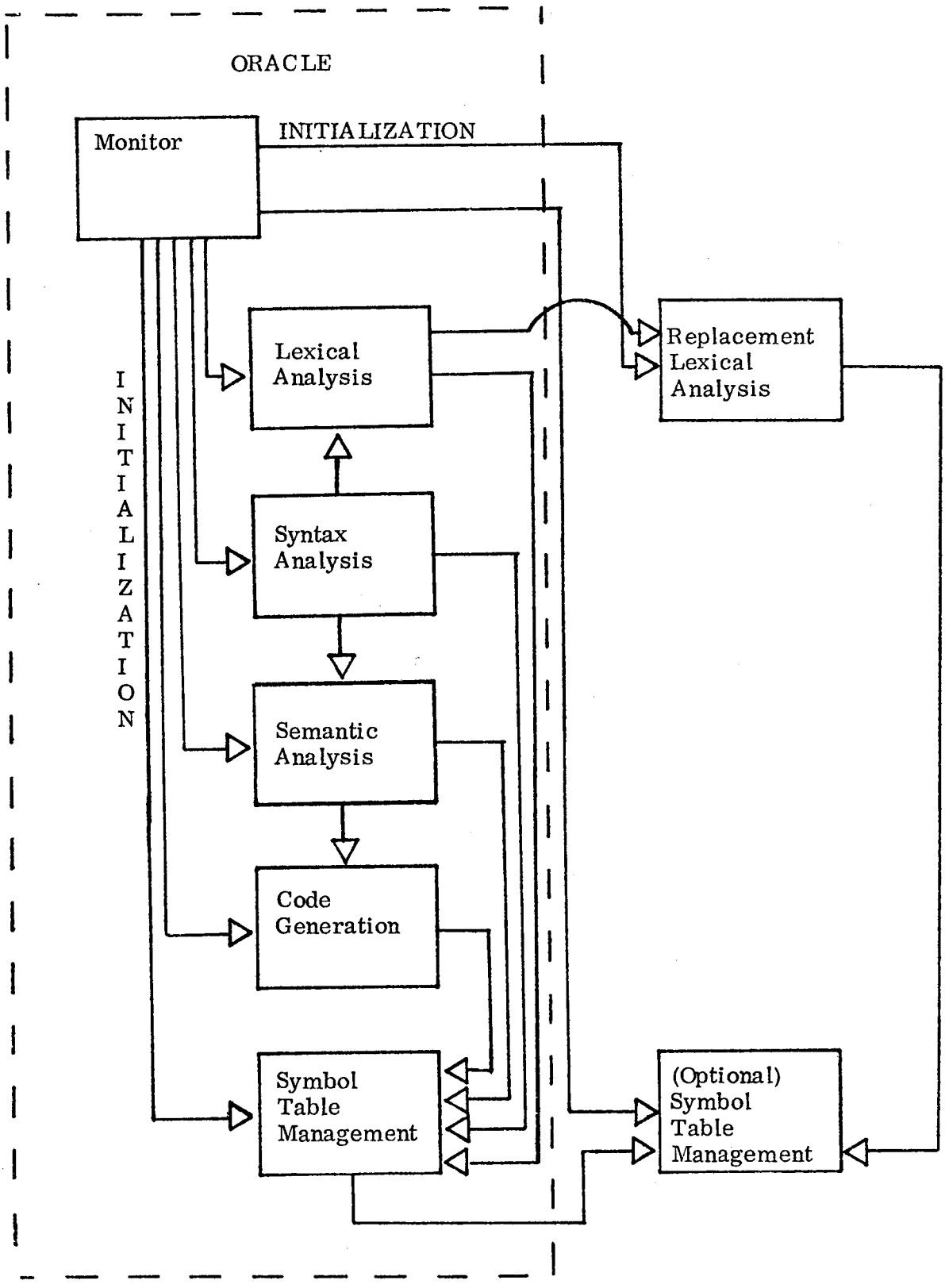


FIGURE 3

A GRAPHIC VIEW OF REPLACEMENT MODE 2

in the PARM field. Another keyword, C=0, can also be specified to request unconditional replacement. Otherwise, Replacement Mode 3 will be terminated if ORACLE recognizes a syntax error in the input string.

In addition to these, the student should supply a replacement module which can be processed by the Linkage Editor and a second copy of the source program. The student may also supply lexical analysis and symbol table management modules for simultaneous replacement. If these options are not exercised by the student, ORACLE will supply copies of its own modules for lexical analysis and symbol table management. However, ORACLE will not provide checking for these optional modules. For Replacement Mode 3, ORACLE also provides semantic analysis and code generation as a non-replacable module.

These provisions, a second copy of the source code and copies of the other compiler modules provide a simple solution to the possible resource synchronization problem which was previously discussed in Replacement Mode 2.

Communication

After Replacement Mode 3 has been activated, ORACLE will call the syntax analysis replacement module for initialization. Subsequent activations of the replacement module are performed by ORACLE's syntax analysis module. The general flow of these subsequent calls is as follows:

1. Before invoking lexical analysis, after a POP operation, ORACLE will invoke the syntax analysis replacement module.

2. Before invoking lexical analysis, after a POP operation, the replacement module should return to ORACLE.

A graphic representation of these actions is presented in Figure 4.

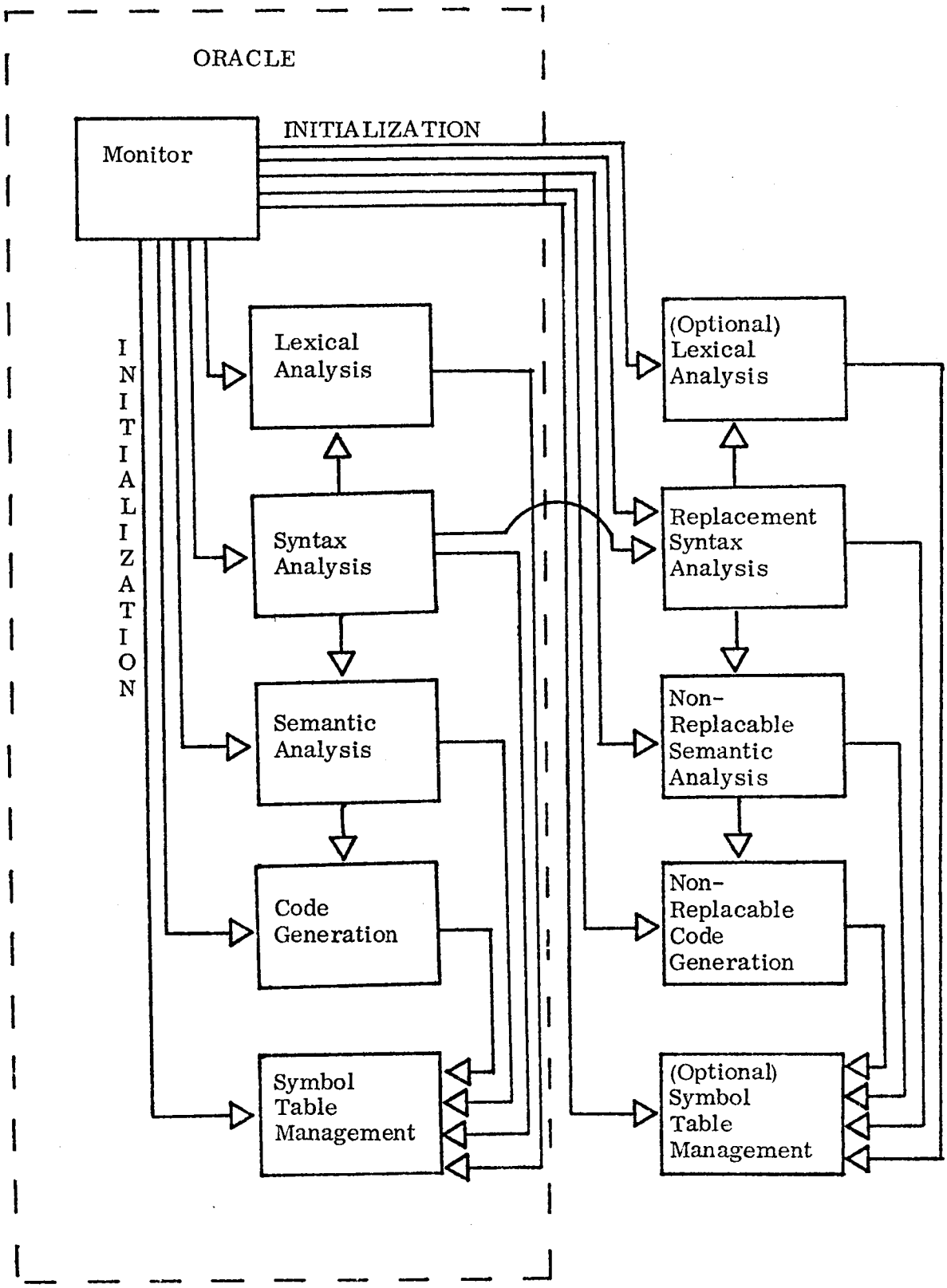


FIGURE 4

GRAPHIC VIEW OF REPLACEMENT MODE 3

Evaluation

The evaluation process for Replacement Mode 3 deviates from the general technique used in previous discussions. The procedure employed by Replacement Mode 3 is simply to provide TRACE output which can be evaluated by the student. This TRACE output is provided by the student's semantic analysis which is supplied by ORACLE. To aid the evaluation process, similar TRACE output will be provided by ORACLE's semantic analysis. This output consists of the last production applied by the parser and the token passed to semantic analysis.

Obviously, this procedure places more of the evaluation burden on the student than is desirable. However, the complexity of syntax analysis and the differences which could exist in implementation between a student's module and ORACLE's module did not permit a more sophisticated evaluation procedure to be developed within the time constraints of the project.

CHAPTER V

ORACLE IN RETROSPECTION AND THOUGHTS FOR THE FUTURE

In reviewing this project, there are several considerations provided by hindsight which could have aided in the development of ORACLE and which could provide direction for future modifications to ORACLE. The following discussion will first look at the compiler as a whole, and then highlight some of the considerations as they apply to each design module of ORACLE.

In the overall development of ORACLE, there were two major strategies applied in the design. One was in-line documentation. In addition to the obvious debugging aid, this strategy also provided a program logic manual simultaneous with the implementation of the program. The other was modular programming. The assets derived from this strategy were twofold. First, it permitted code to be written and debugged quickly. And second, it allowed the components of the compiler to be designed and implemented in a relatively independent manner. In other words, the interaction between modules required one to know "what" another was doing; however, one module did not dictate "how" another would perform its function. These strategies have proven their value time and again during the development of ORACLE, and the author feels that the result of these strategies was well worth the extra effort which they required.

In a review of the components of ORACLE, the design of lexical analysis module is one which the author would not alter. The simplicity of

the design made this module the easiest to implement. In the author's opinion, an alternate strategy such as a lexical grammar as discussed by DeRemer (1975) would provide no real advantage in speed or in flexibility over the current technique.

Like the design of lexical analysis, the basic structure of the symbol table management module is also one the author would not change. However, the usage of a more sophisticated search technique could warrant consideration. This would be reinforced if the size of the symbol table were enlarged or if the number of searches for a unique symbol table entry were increased. Several alternate search techniques were surveyed by McKeeman (1975), whose symbol table structure was adapted for ORACLE.

The next design module, syntax analysis, is the first module which could provide several areas for re-examination. The basic structure of this module is, in the author's opinion, well defined. However, several components of syntax analysis could provide a foundation for future investigation.

One particular area which impeded the development of ORACLE was the method used for generating the parse table. When done by hand, as was the case in the development of ORACLE, this process is a tedious task. Hence, an automated method for generating the parse table could provide a significant impact on the ease in which modifications to the parse table could be made. This would be particularly true if the entire table would require replacement.

Another area for future consideration is syntax error recovery. In particular, the set of "error productions" used by ORACLE could be expanded to handle more situations than are presently resolved in this manner. The examination of syntax error recovery in general could

encompass as much time as that devoted to the entire development of ORACLE.

Though not a proper component of syntax analysis, the grammar of ORACLE could also be modified to increase the semantic evaluation provided in parsing. More specifically, the structure associated with an identifier, such as scalar or array, could be incorporated into an expansion of the terminals of the grammar. Thus, the recognition of addressing errors as associated with an identifier could be recognized immediately by the parser. This would alleviate the after-the-fact diagnosis presently provided by semantic analysis. An example of an addressing error in this context would be an attempt to apply a subscript to an identifier which had not been declared as an array.

In addition to these, the parse technique could be reviewed to examine alternatives to LL(1) parsing. Current discussions on syntax error recovery (Graham and Rhodes, 1975) and mechanized parse table generation (Aho and Johnson, 1974) are primarily directed at LR parsing. A reconsideration of the parse technique would be reinforced if a desirable change in the language could not be recognized by an LL(1) parser.

As in development, the semantic analysis and code generation module is the last to be reviewed. Unlike the other components of ORACLE, this module is incomplete. It does not yet generate executable code. However, all of the foreseeable needs of an interpretive module have been provided in the current implementation. Thus, this last module could provide the basis for the first modification to ORACLE.

In conclusion, the features and capabilities of ORACLE which have been discussed in previous chapters have been implemented. However, the

author is certain that the usage of ORACLE will direct necessary modifications in the compiler and that these changes will require a minimal programming effort.

BIBLIOGRAPHY

BIBLIOGRAPHY

Aho, A. V., and S. C. Johnson (1974). "LR Parsing." Computing Surveys 6:2, 99-124.

Aho, A. V., and J. D. Ullman (1972). The Theory of Parsing, Translation, and Compiling, Vol. 1. Parsing, Prentice-Hall, New Jersey.

DeRemer, F. L. (1975). "Lexical Analysis," in Compiler Construction, An Advanced Course, ed. F. L. Bauer and J. Eickel, Springer-Verlag Lecture Notes in Computer Science - 21, pp. 109-120.

Feldman, J. A. (1966). "Translater Writing Systems." CACM 11 (Feb. 1968), 77-113.

Graham, S. L., and S. P. Rhodes (1973). "Practical Syntaxtic Error Recovery in Compilers." Conference record of ACM symposium on principles of programming languages, Boston, 52-58.

Graham, S. L., and S. P. Rhodes (1975). "Practical Syntaxtic Error Recovery," CACM 8:11, pp. 639-650.

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

Griffiths, M. (1975). "LL(1) Grammars and Analysers," in Compiler Construction, An Advanced Course, ed. F. L. Bauer and J. Eickerl, Springer-Verlag Lecture Notes in Computer Science, pp. 57-84.

Holt, R. C., and D. B. Wortman (1973). "Structured Subsets of PL/I." Computer Science Research Group Technical Report CSRG-27, University of Toronto.

Horning, J. J. (1975). "What the Compiler Should Tell the User," in Compiler Construction, An Advanced Course, ed. F. L. Bauer and J. Eicker, Springer-Verlag Lecture Notes in Computer Science - 21, 535-548.

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

Lenius, R. P. (1970). "Error Detection and Recovery for Syntax Directed Compiler Systems." Ph.D. Thesis, University of Wisconsin, Madison.

Lenius, R. P. (1976). Author's notes from personal conservation on syntax error recovery.

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

Lukasiewicz, J. (1929). 'Oznaczenui i potrzebach logiki matematiycznej." (On the importance and needs of mathematical logic.) Nauka Polska 10, 604-620.

McKeeman, W. M. (1975). "Symbol Table Access" in Compiler Construction, An Advanced Course, ed. F. L. Bauer and J. Eickel, Springer-Verlag Lecture Notes in Computer Science - 21, 253-301.

Morris, R. (1968). "Scatter Storage Techniques," CACM 11:1, pp. 38-44.

Wirth, N. (1968). 'PL360 - a programming language for the 360 computer." J. ACM 15:2, 37-74.

Wirth, N. (1971). 'The Programming Language PASCAL," ACTA Informatica 1, Springer-Verlag, pp. 35-63.

Wirth, N., and K. Jensen (1975). "PASCAL User Manual and Report," Springer-Verlag Lecture Notes in Computer Science - 18.

APPENDIXES

APPENDIX A

PROGRAM LOGIC MANUAL

```
ORACLE: PROGRAM OVERVIEW.
  ACTION:

    MODULE INITIALIZATION.

      COMPILATION.
        LEXICAL ANALYSIS  - INPUT AND RECOGNIZE NEXT TOKEN.
        SYNTAX ANALYSIS   - CONSTRUCT SYNTAX TREE.
        SEMANTIC ANALYSIS - EVALUATE SYNTAX TREE AND GENERATE
                            INTERMEDIATE CODE (POLISH SUFFIX).
        CODE GENERATION   - CONSTRUCT AND GENERATE
                            INTERMEDIATE CODE STREAM.
        SYMBOL TABLE      - CONSTRUCT AND MAINTAIN SYMBOL
                            TABLE.

      POST COMPILATION ACTIVITY.


  REPLACEMENT MODE.
    REPLACEMENT MODE 0 - SOURCE COMPILATION ONLY.
    REPLACEMENT MODE 1 - REPLACE AND MONITOR SYMBOL TABLE.
    REPLACEMENT MODE 2 - REPLACE AND MONITOR LEXICAL ANALYSIS.
    REPLACEMENT MODE 3 - REPLACE AND MONITOR SYNTAX ANALYSIS.


  INPUT:
    SOURCE1 - SOURCE INPUT TO ORACLE.
    RSOURCE - (MODE 2 & 3 ONLY) SOURCE INPUT TO REPLACEMENT
              MODULE.


  LISTINGS:
    PRINT1 - SOURCE LISTING AND INTERMEDIATE CODE
             GENERATED BY ORACLE.
    RPRINT - (MODE 1,2 & 3) SOURCE LISTING AND INTERMEDIATE
             CODE  GENERATED BY THE REPLACEMENT MODULE.
    XXMSG  - TRACE, DEBUG, AND DIAGNOSTIC MESSAGES GENERATED
             BY ORACLE.
    RXXMSG - (MODE 1,2 & 3) TRACE, DEBUG, AND DIAGNOSTIC
             MESSAGES GENERATED BY THE REPLACEMENT MODULE.
```

```
 ACCESS RIGHTS - SUPERVISOR MODULE (MONITOR):
   GLOBAL VARIABLES:

** TRACE FLAGS **
#T#LEX BIT (1) STATIC EXTERNAL      ** LEXICAL ANALYSIS   **
#T#SYN BIT (1) STATIC EXTERNAL      ** SYNTAX ANALYSIS    **
#T#SEM BIT (1) STATIC EXTERNAL      ** SEMANTIC ANALYSIS  **
#T#CODE BIT (1) STATIC EXTERNAL     ** CODE GENERATION    **
#T#STAB BIT (1) STATIC EXTERNAL     ** SYMBOL TABLE
                                       MANAGEMENT **
** DEBUG FLAGS **
#D#LEX BIT (1) STATIC EXTERNAL      ** LEXICAL ANALYSIS   **
#D#SYN BIT (1) STATIC EXTERNAL      ** SYNTAX ANALYSIS    **
#D#SEM BIT (1) STATIC EXTERNAL      ** SEMANTIC ANALYSIS  **
#D#CODE BIT (1) STATIC EXTERNAL     ** CODE GENERATION    **
#D#STAB BIT (1) STATIC EXTERNAL     ** SYMBOL TABLE
                                       MANAGEMENT **
** PRINT SOURCE **
#T#UPGM BIT (1) STATIC EXTERNAL
** ERROR PRINT **
#T#ERR BIT (1) STATIC EXTERNAL

** REPLACEMENT MODE FLAGS **
** MODE 1 - REPLACE SYMBOL TABLE  **
#R#STAB BIT (1) STATIC EXTERNAL
** MODE 2 - REPLACE LEXICAL ANALYSIS  **
#R#LEX BIT (1) STATIC EXTERNAL
** MODE 3 - REPLACE SYNTAX ANALYSIS  **
#R#SYN BIT (1) STATIC EXTERNAL
** REPLACEMENT MODE 3 TERMINATION CONDITION **
#R#COND BIT (1) STATIC EXTERNAL

** EXTERNAL ENTRY POINTS **
  ** ORACLE COMPILATION ROUTINES - INITIALIZATION ENTRY
                                   POINTS **
SCANNER ENTRY                      ** LEXICAL ANALYSIS   **
PARSER ENTRY                       ** SYNTAX ANALYSIS    **
SEMANT ENTRY                       ** SEMANTIC ANALYSIS  **
CODEGEN ENTRY                      ** CODE GENERATION    **
SYMTAB ENTRY                       ** SYMBOL TABLE
                                      MANAGEMENT **
  ** POST COMPILATION ROUTINES **
PCSTAT ENTRY
  ** REPLACEMENT MODULE INITIALIZATION ENTRY POINTS **
RLEX ENTRY                         ** LEXICAL ANALYSIS   **
RSYNTAX ENTRY                      ** SYNTAX ANALYSIS    **
RSMAN ENTRY                        ** SEMANTIC ANALYSIS  **
RCODE ENTRY                        ** CODE GENERATION    **
RTABLE ENTRY                       ** SYMBOL TABLE
                                      MANAGEMENT **
```

LEXICAL ANALYSIS: A REQUEST FOR THE NEXT TOKEN HAS BEEN INITIATED BY SYNTAX ANALYSIS.

ACTION:
SCANNER ENTRY - INITIALIZE FLAGS, COUNTERS, VALUES, AND VARIABLES USED IN LEXICAL ANALYSIS.
SCAN ENTRY - INPUT AND RECOGNIZE NEXT TOKEN.

INPUT ROUTINES:
GETNEXT - FETCH AND CLASSIFY NEXT CHARACTER.
GETCARD - FETCH NEXT EIGHTY BYTE SOURCE RECORD.

RECOGNITION ROUTINES:
COMPILER AID - PROCESS SOURCE PROGRAM REQUEST FOR COMPILER ASSISTANCE.
ERROR - CLASSIFY TOKEN AS UNKNOWN.
IDENTIFIER - BUILD LONGEST IDENTIFIER.
OBTAIN IDENTIFIER TYPE AND ADDRESS FROM SYMBOL TABLE.
RECOGNIZE LABEL DECLARATIONS.
INTEGER CONSTANT - BUILD LONGEST INTEGER CONSTANT.
CLASSIFY TOKEN AS INTEGER CONSTANT AND OBTAIN ADDRESS FROM SYMBOL TABLE.
DELIMITER - CLASSIFY TOKEN AS NON-BLANK DELIMITER.
STRING CONSTANT - BUILD LONGEST STRING CONSTANT.
CLASSIFY TOKEN AS STRING CONSTANT AND OBTAIN ADDRESS FROM SYMBOL TABLE.
COMMENT - IGNORE SOURCE PROGRAM COMMENTS.

DEFINITION OF THE ALPHABET OF THE LANGUAGE RECOGNIZED BY ORACLE.

LETTER A/B/C/D/E/G/H/I/J/K/L/M/N/O/P/Q/R/S/T/U/V/W/X/Y/Z
DIGIT 0/1/2/3/4/5/6/7/8/9
OPERATORS = (ASSIGNMENT)
+ AND - (ADDITION)
* AND / (MULTIPLICATION)
LT/LE/EQ/GE/GT/NE (RELATION)
DELIMITERS , (COMMA)/((LEFT PARENTHESIS)/
) (RIGHT PARENTHESIS)/; (SEMICOLON)
SPECIAL DELIMITERS (BLANK)/' (SINGLE QUOTE)/: (COLON)
SPECIAL CHARACTER $
COMMENT BRACKETS < >
KEYWORDS BEGIN/ELSE/END/GOTO/IF/INPUT/INTEGER
OUTPUT/REPEAT/STRING/THEN/UNTIL/
(AND THE RELATIONAL OPERATORS)

DEFINITION OF TOKEN VALUES:

VALUE	MNEMONIC
0	#UNKNOWN#
1	#COMMA#
2	#LEFT PARENTHESIS#
3	#RIGHT PARENTHESIS#
4	#SEMICOLON#
5	#ASSIGNMENT OPERATOR, = #
6-9	#UNDEFINED#
10	#ADDITION OPERATOR, + #
11	#ADDITION OPERATOR, - #
12-19	#ADDITION OPERATOR, UNDEFINED #
20	#MULTIPLICATION OPERATOR, * #
21	#MULTIPLICATION OPERATOR, / #
22-29	#MULTIPLICATION OPERATOR, UNDEFINED #
30	#RELATIONAL OPERATOR, GE #
31	#RELATIONAL OPERATOR, GT #
32	#RELATIONAL OPERATOR, EQ #
33	#RELATIONAL OPERATOR, LE #
34	#RELATIONAL OPERATOR, LT #
35	#RELATIONAL OPERATOR, NE #
36-39	#RELATIONAL OPERATOR, UNDEFINED #
40	#UNDEFINED CONSTANT#
41	#LABEL CONSTANT#
42	#INTEGER CONSTANT#
43	#STRING CONSTANT#
44-49	#UNDEFINED CONSTANT#
50	#NEW IDENTIFIER#
51	#LABEL IDENTIFER#
52	#INTEGER IDENTIFIER#
53	#STRING IDENTIFIER#
54-59	#UNDEFINED IDENTIFIER#
60	#BEGIN#
61	#ELSE#
62	#END#
63	#GOTO#
64	#IF#
65	#INPUT#
66	#INTEGER#
67	#OUTPUT#
68	#REPEAT#
69	#STRING#
70	#THEN#
71	#UNTIL#
80	@EMPTY STRING@

*** NOTE: @EMPTY STRING@ IS A SPECIAL TOKEN USED BY ORACLE TO SIGNAL THE RECOGNITION OF THE END OF THE INPUT STRING. ***

```
  ACCESS RIGHTS - LEXICAL ANALYSIS:
    GLOBAL VARIABLES:

** TRACE FLAGS **
#T#LEX BIT (1) STATIC EXTERNAL          ** LEXICAL ANALYSIS **
#T#SYN BIT (1) STATIC EXTERNAL          ** SYNTAX ANALYSIS **
#T#SEM BIT (1) STATIC EXTERNAL          ** SEMANTIC ANALYSIS **
#T#CODE BIT (1) STATIC EXTERNAL         ** CODE GENERATION **
#T#STAB BIT (1) STATIC EXTERNAL         ** SYMBOL TABLE
                                           MANAGEMENT **

** DEBUG FLAGS **
#D#LEX BIT (1) STATIC EXTERNAL          ** LEXICAL ANALYSIS **
#D#SYN BIT (1) STATIC EXTERNAL          ** SYNTAX ANALYSIS **
#D#SEM BIT (1) STATIC EXTERNAL          ** SEMANTIC ANALYSIS **
#D#CODE BIT (1) STATIC EXTERNAL         ** CODE GENERATION **
#D#STAB BIT (1) STATIC EXTERNAL         ** SYMBOL TABLE
                                           MANAGEMENT **

** PRINT SOURCE **
#T#UPGM BIT (1) STATIC EXTERNAL

** ERROR PRINT **
#T#ERR BIT (1) STATIC EXTERNAL

** REPLACEMENT MODE FLAG **
  ** MODE 2 - REPLACE LEXICAL ANALYSIS **
#R#LEX BIT (1) STATIC EXTERNAL


** EXTERNAL ENTRY POINTS AND PARAMETER LIST ATTRIBUTES **

  ** SYMBOL TABLE **
SEARCH ENTRY (FIXED BINARY (31),FIXED BINARY (31),
              CHARACTER (70) VARYING)
INSERT ENTRY (FIXED BINARY (31),FIXED BINARY (31),
              CHARACTER (70) VARYING)
QSCOPE ENTRY (FIXED BINARY (31))
G@LEVL ENTRY (FIXED BINARY (31),FIXED BINARY (31))

  ** REPLACEMENT MODULE - LEXICAL ENTRY **
RSCAN ENTRY (FIXED BINARY (31),FIXED BINARY (31),
             CHARACTER (70) VARYING)
```

*** INPUT ROUTINES ***

GETNEXT: A REQUEST FOR THE NEXT SOURCE CHARACTER HAS BEEN ISSUED BY LEXICAL ANALYSIS.
ACTION:
RECOGNIZE NEXT SOURCE CHARACTER.
CLASSIFY CHARACTER FOR USE IN LEXICAL ANALYSIS.

GETCARD: A REQUEST FOR THE NEXT SOURCE CARD - EIGHTY BYTE RECORD, HAS BEEN ISSUED BY LEXICAL ANALYSIS.
ACTION:
INPUT NEXT CARD.
RESET AND INCREMENT COUNTERS.

IMPLEMENTATION RESTRICTIONS:
CARD COLUMNS 73 THROUGH 80 ARE IGNORED.
A SOURCE PROGRAM MUST NOT EXCEED 1000 CARDS.

```
*** RECOGNITION ROUTINES ***

COMPILER AIDS:  A REQUEST FOR COMPILER ASSISTANCE HAS BEEN
                INITIATED BY THE USER PROGRAM.
  ACTION:
    PROCESS DIRECTIVES AND INITIATE REQUESTED AIDS.
    RETURN TO SEARCH FOR NEXT TOKEN.

   DEFINITION OF COMPILER AIDS:
     <AIDS> --> $<DIRECTIVE> <OPERAND> <2OPERAND>
       <DIRECTIVE> --> DEBUG/NODEBUG
                   --> TRACE/NOTRACE
         <OPERAND> --> ALL/CODE/LEXICAL/SEMANTIC
                   --> SYNTAX/TABLE
        <2OPERAND> --> <JUNK> <OPERAND> <2OPERAND>
                   --> %EMPTY%
            <JUNK> --> ANY PRINTABLE CHARACTER
                   --> %EMPTY%

NOAID: A LEGAL DIRECTIVE COULD NOT BE RECOGNIZED.
       ACTION:
         FLAG THE ERROR.
         IGNORE REQUEST FOR COMPILER AIDS.

NODEBUG: A REQUEST FOR THE TERMINATION OF DEBUGGING AIDS
         HAS BEEN ENCOUNTERED.
         ACTION:
           SET DEBUG SWITCH TO OFF.

DEBUG: A REQUEST FOR THE INITATION OF DEBUGGING
       AIDS HAS BEEN ENCOUNTERED.
       ACTION:
          SET DEBUG SWITCH FOR REQUESTED OPERANDS.

NOTRACE: A REQUEST FOR THE TERMINATION OF TRACING AIDS HAS
         BEEN ENCOUNTERED.
         ACTION:
           SET TRACE SWITCH TO OFF.

TRACE: A REQUEST FOR THE INITIATION OF TRACING AIDS
       HAS BEEN ENCOUNTERED.
       ACTION:
         SET TRACE SWITCH FOR REQUESTED OPERANDS.


ERROR: AN UNRECOGNIZABLE CHARACTER IN THE INPUT STREAM HAS
       BEEN DETECTED AS THE FIRST CHARACTER OF THE NEXT
       TOKEN.
       ACTION:
         FLAG THE ERROR.
         RETURN A TOKEN VALUE OF UNKNOWN.
```

```
IDENTIFIER: A LETTER HAS BEEN RECOGNIZED AS THE FIRST
            CHARACTER OF THE NEXT TOKEN.
  ACTION:
    BUILD THE LONGEST IDENTIFIER.
    OBTAIN IDENTIFIER TYPE AND ADDRESS FROM
    SYMBOL TABLE.
    RECOGNIZE AND CLASSIFY LABEL CONSTANTS.
    RESOLVE MULTIPLE DECLARATION OF LABEL.
    RETURN THE TOKEN VALUE, THE SYMBOL
    TABLE ADDRESS, AND THE INPUT STRING RECOGNIZED.

  DEFINITION OF IDENTIFIER TOKEN:
    <IDENTIFIER> --> <LETTER> <2LETTER>
      <LETTER>--> A/B/C/D/E/F/G/H/I/J/K/L/M
              --> N/O/P/Q/R/S/T/U/V/W/X/Y/Z
        <2LETTER> --> <LETTER> <2LETTER>
                  --> <DIGIT> <2LETTER>
                  --> %EMPTY%
          <DIGIT> --> 0/1/2/3/4/5/6/7/8/9

  DEFINITION OF LABEL CONSTANT TOKEN:
    <LABEL CONSTANT> --> <IDENTIFIER>:

  IMPLEMENTATION RESTRICTIONS:
    DISTINCT IDENTIFIERS MUST BE UNIQUE IN THE FIRST
    EIGHT CHARACTERS.
    IDENTIFIERS MAY NOT SPAN CARD BOUNDARIES.


INTEGER_CONSTANT: A DIGIT HAS BEEN RECOGNIZED AS THE FIRST
                  CHARACTER OF THE NEXT TOKEN.
    ACTION:
      BUILD INTEGER CONSTANT.
      CLASSIFY TOKEN AS INTEGER CONSTANT.
      OBTAIN ADDRESS FROM SYMBOL TABLE.
      RETURN TOKEN VALUE, SYMBOL TABLE ADDRESS,
      AND THE INPUT STRING RECOGNIZED.

    DEFINITION OF INTEGER CONSTANT TOKEN:
     <INTEGER CONSTANT> --> <DIGIT> <2DIGIT>
       <DIGIT> --> 0/1/2/3/4/5/6/7/8/9
       <2DIGIT> --> <DIGIT> <2DIGIT> / %EMPTY%

    IMPLEMENTATION RESTRICTIONS:
       INTEGER CONSTANTS MAY NOT EXCEED 6 DIGITS.
       INTEGER CONSTANTS MAY NOT SPAN CARD BOUNDARIES.
```

DELIMITER: A DELIMITER, NOT A BLANK, HAS BEEN RECOGNIZED AS THE FIRST CHARACTER OF THE NEXT TOKEN.

ACTION:
CLASSIFY TOKEN AS NON BLANK DELIMITER.
RETURN THE TOKEN VALUE, THE SYMBOL TABLE ADDRESS (UNDEFINED), AND THE INPUT STRING RECOGNIZED.

DEFINITION OF NON BLANK DELIMITER:
<DELIMITER> --> ,/(/)/;/=/<ADD OP>/<MULT OP>
<ADD OP> --> - OR +
<MULT OP> --> * OR /

STRING CONSTANT: A QUOTE MARK HAS BEEN RECOGNIZED AS THE FIRST CHARACTER OF THE NEXT TOKEN.

ACTION:
BUILD STRING CONSTANT.
CLASSIFY TOKEN AS STRING CONSTANT.
OBTAIN ADDRESS FROM SYMBOL TABLE.
RETURN TOKEN VALUE, SYMBOL TABLE ADDRESS, AND THE INPUT STRING RECOGNIZED.

DEFINITION OF STRING CONSTANT TOKEN:
<STRING CONSTANT> --> '<CHAR>'
<CHAR> --> ANY STRING OF PRINTABLE CHARACTERS EXCLUDING A QUOTE MARK BUT INCLUDING CHARACTERS NOT IN THE ALPHABET OF THE LANGUAGE.

IMPLEMENTATION RESTRICTIONS:
STRING CONSTANTS MAY NOT EXCEED SEVENTY CHARACTERS.
STRING CONSTANTS MAY NOT SPAN CARD BOUNDARIES.

COMMENT: A BEGIN-COMMENT ,<, HAS BEEN RECOGNIZED AS THE FIRST CHARACTER OF THE NEXT TOKEN.
ACTION:
PROCESS SOURCE COMMENT - IGNORE INPUT UNTIL AN END-COMMENT, >, HAS BEEN RECOGNIZED.
RETURN TO SEARCH FOR NEXT TOKEN.

DEFINITION OF SOURCE COMMENT:
<COMMENT> --> #BEGIN COMMENT# <TEXT> #END COMMENT#
#BEGIN COMMENT# --> >
<TEXT> --> ANY STRING OF PRINTABLE CHARACTERS EXCLUDING AN END-COMMENT BUT INCLUDING CHARACTERS NOT IN THE ALPHABET OF THE LANGUAGE.
#END COMMENT# --> <

IMPLEMENTATION RESTRICTIONS:
COMMENTS MAY NOT SPAN CARD BOUNDARIES.

SYNTAX ANALYSIS: CONSTRUCT THE SYNTAX TREE AND VERIFY THE GRAMMATICAL CORRECTNESS OF THE SOURCE PROGRAM.

ACTION:

PARSER ENTRY - INITIALIZE POINTERS, LIMITS, AND VARIABLES USED BY SYNTAX ANALYSIS.

OBTAIN THE NEXT TOKEN FROM LEXICAL ANALYSIS.
EXECUTE THE PARSING ACTION DETERMINED BY CONSULTING A PARSING ACTION TABLE WHICH IS CROSS-REFERENCED BY THE TOP OF THE PARSE STACK AND THE NEXT TOKEN.
INITIATE THE EVALUATION OF THE SYNTAX TREE BY SEMANTIC ANALYSIS.

PARSING ACTION OVERVIEW:

ERROR - ATTEMPT TO RECOVER FROM SYNTAX ERROR.
POP - POP THE TOP OF THE PARSE STACK.
DECLARE - RESOLVE MULTIPLE DECLARATION OF IDENTIFIER.
PUSH - REPLACE THE TOP OF THE PARSE STACK WITH A NEW PRODUCTION.
ACCEPT - ACCEPT THE SOURCE PROGRAM AND EXIT SYNTAX ANALYSIS.

DEFINITION OF THE GRAMMAR:

GRAMMAR = (NON-TERMINALS, TERMINALS, PRODUCTIONS, PROGRAM)
WHERE

NON-TERMINALS = (<PROGRAM> , <BODY> , <DECLARE BLOCK> ,
<DECLARE VARIABLE> , <SIZE> , <1SIZE> , <2SIZE> ,
<2DECLARE VARIABLE> , <EXECUTE BLOCK> , <LABEL> , <BLOCK> ,
<SIMPLE STATEMENT> , <IF STATEMENT> , <RELATION> ,
<INTEGER EXPRESSION> , <SIGNED INTEGER TERM> ,
<INTEGER TERM> , <INTEGER WORD> , <INTEGER VARIABLE> ,
<SUBSCRIPT> , <2SUBSCRIPT> , <2INTEGER WORD> ,
<2INTEGER TERM> , <2INTEGER RELATION> ,
<STRING EXPRESSION> , <STRING VARIABLE> ,
<2STRING RELATION> , <THEN CLAUSE> ,
<ELSE CLAUSE> , NOT IF STATEMENT> , <ASSIGNMENT> ,
<REPEAT BLOCK> , <LIST> , <2LIST> , <2LABEL>)

TERMINALS = (#UNKNOWN# , #,# , #(# , #)# , #;# , #=# ,
#ADDITION OPERATOR# , #MULTIPLICATION OPERATOR# ,
#RELATION OPERATOR# , #LABEL CONSTANT# ,
#INTEGER CONSTANT# , #STRING CONSTANT# ,
#NEW IDENTIFIER# , #LABEL IDENTIFIER# ,
#INTEGER IDENTIFIER# , #STRING IDENTIFIER# ,
#BEGIN# , #ELSE# , #END# , #GOTO# , #ELSE# , #END# ,
#INTEGER# , #OUTPUT# , #REPEAT# , #STRING# , #THEN# ,
#UNTIL#)

```
PRODUCTIONS:
 1 <PROGRAM> ==> #LABEL CONSTANT# <BODY> #;#
 2 <BODY> ==> #BEGIN# <DECLARE BLOCK> <EXECUTE BLOCK> #END#
 3 <DECLARE BLOCK> ==> #INTEGER# <DECLARE VARIABLE> #;#
                       <DECLARE BLOCK>
 4 <DECLARE BLOCK> ==> #STRING# <DECLARE VARIABLE> #;#
                       <DECLARE BLOCK>
 5 <DECLARE BLOCK> ==> %EMPTY%
 6 <DECLARE VARIABLE> ==> #NEW IDENTIFIER# <SIZE>
                          <2DECLARE VARIABLE>
 7 <SIZE> ==> #(# <1SIZE> <2SIZE> #)#
 8 <SIZE> ==> %EMPTY%
 9 <1SIZE> ==> #INTEGER CONSTANT#
10 <2SIZE> ==> #,# <1SIZE>
11 <2SIZE> ==> %EMPTY%
12 <2DECLARE VARIABLE> ==> #,# <DECLARE VARIABLE>
13 <2DECLARE VARIABLE> ==> %EMPTY%
14 <EXECUTE BLOCK> ==> <LABEL> <BLOCK> <EXECUTE BLOCK>
15 <EXECUTE BLOCK> ==> %EMPTY%
16 <LABEL> ==> #LABEL CONSTANT#
17 <LABEL> ==> %EMPTY%
18 <BLOCK> ==> <BODY> #;#
19 <BLOCK> ==> <SIMPLE STATEMENT>
20 <SIMPLE STATEMENT> ==> <IF STATEMENT>
21 <SIMPLE STATEMENT>  ==> <NOT IF STATEMENT> #;#
22 <IF STATEMENT> ==> #IF# <RELATION> #THEN# <THEN CLAUSE>
23 <RELATION> ==> <INTEGER EXPRESSION> <2INTEGER RELATION>
24 <RELATION> ==> <STRING EXPRESSION> <2STRING RELATION>
25 <INTEGER EXPRESSION> ==> <SIGNED INTEGER TERM>
                            <2INTEGER TERM>
26 <SIGNED INTEGER TERM> ==> #ADD OP# <INTEGER TERM>
27 <SIGNED INTEGER TERM> ==> <INTEGER TERM>
28 <INTEGER TERM> ==> <INTEGER WORD> <2INTEGER WORD>
29 <INTEGER WORD> ==> #(# <INTEGER EXPRESSION> #)#
30 <INTEGER WORD> ==> <INTEGER VARIABLE>
31 <INTEGER WORD> ==> #INTEGER CONSTANT#
32 <INTEGER VARIABLE> ==> #INTEGER IDENTIFIER# <SUBSCRIPT>
33 <SUBSCRIPT> ==> #(# <INTEGER EXPRESSION> <2SUBSCRIPT> #)#
34 <SUBSCRIPT> ==> %EMPTY%
35 <2SUBSCRIPT> ==> #,# <INTEGER EXPESSION>
36 <2SUBSCRIPT> ==> %EMPTY%
37 <2INTEGER WORD> ==> #MULT OP# <INTEGER WORD>
                       <2INTEGER WORD>
38 <2INTEGER WORD> ==> %EMPTY%
39 <2INTEGER TERM> ==> #ADD OP# <INTEGER TERM>
                       <2INTEGER TERM>
40 <2INTEGER TERM> ==> %EMPTY%
41 <2INTEGER RELATION> ==> #REL OP# <INTEGER EXPRESSION>
42 <STRING EXPRESSION> ==> <STRING VARIABLE>
43 <STRING EXPRESSION> ==> #STRING CONSTANT#
44 <STRING VARIABLE> ==> #STRING IDENTIFIER# <SUBSCRIPT>
45 <2STRING RELATION> ==> #REL OP# <STRING EXPRESSION>
46 <THEN CLAUSE> ==> <BODY> #;# <ELSE CLAUSE>
47 <THEN CLAUSE> ==> <NOT IF STATEMENT> #;# <ELSE CLAUSE>
48 <THEN CLAUSE> ==> <IF STATEMENT>
```

```
49 <ELSE CLAUSE> ==> #ELSE# <BLOCK>
50 <ELSE CLAUSE> ==> %EMPTY%
51 <NOT IF STATEMENT> ==> <ASSIGNMENT>
52 <NOT IF STATEMENT> ==> #UNTIL# <REPEAT BLOCK> #REPEAT#
53 <NOT IF STATEMENT> ==> #INPUT# <LIST>
54 <NOT IF STATEMENT> ==> #OUTPUT# <LIST>
55 <NOT IF STATEMENT> ==> #GOTO# <2LABEL>
56 <ASSIGNMENT> ==> <INTEGER VARIABLE> #=#
                     <INTEGER EXPRESSION>
57 <ASSIGNMENT> ==> <STRING VARIABLE> #=# <STRING EXPRESSION>
58 <REPEAT BLOCK> ==> <RELATION> #;# <EXECUTE BLOCK>
59 <LIST> ==> <INTEGER VARIABLE> <2LIST>
60 <LIST> ==> <STRING VARIABLE> <2LIST>
61 <2LIST> ==> #,# <LIST>
62 <2LIST> ==> %EMPTY%
63 <2LABEL> ==> #NEW IDENTIFIER#
64 <2LABEL> ==> #LABEL IDENTIFIER#


  ACCESS RIGHTS - SYNTAX ANALYSIS:
    GLOBAL VARIABLES:

** TRACE FLAGS **
#T#LEX BIT (1) STATIC EXTERNAL
#T#SYN BIT (1) STATIC EXTERNAL

** DEBUG FLAG **
#D#SYN BIT (1) STATIC EXTERNAL

** REPLACEMENT MODE FLAGS **
  ** MODE 3 - REPLACE SYNTAX ANALYSIS **
#R#SYN BIT (1) STATIC EXTERNAL
 ** MODE 3 - SYNTAX REPLACEMENT TERMINATION CODE **
#R#COND BIT (1) STATIC EXTERNAL

  ** EXTERNAL ENTRY POINTS AND PARAMETER LIST ATTRIBUTES **
  ** LEXICAL ANALYSIS **
SCAN ENTRY (FIXED BINARY (31),FIXED BINARY (31),
            CHARACTER (70) VARYING)

  ** SEMANTIC ANALYSIS **
EVAL ENTRY (FIXED BINARY (31),FIXED BINARY (31),
            FIXED BINARY (31))
SFLUSH ENTRY

  ** CODE GENERATION **
BUILD ENTRY (FIXED BINARY (31))
```

```
  ** SYMBOL TABLE **
INSERT ENTRY (FIXED BINARY (31),FIXED BINARY (31),
              CHARACTER (70) VARYING)
QSCOPE ENTRY (FIXED BINARY (31))
S@TYPE ENTRY (FIXED BINARY (31),FIXED BINARY (31))
G@LEVL ENTRY (FIXED BINARY (31),FIXED BINARY (31))

  ** REPLACEMENT MODE 3 - SYNTAX ANALYSIS **
RPARSER ENTRY


*** PARSING ROUTINES ***

SYNTAX ERROR: THE PARSING TABLE WAS UNABLE TO DETERMINE A
              VALID ACTION BASED ON THE CURRENT TOP OF THE
              PARSE STACK AND THE CURRENT TOKEN.
   ACTION:
     ISSUE ERROR MESSAGE.
     CATEGORIZE RECOVERY ACTION.
     ATTEMPT TO RECOVER FROM SYNTAX ERROR.
     TERMINATE SYNTAX ANALYSIS IF UNABLE TO RECOVER.
 ERROR RECOVERY OVERVIEW:
 ERROR CATEGORY        ACTION TAKEN

  TOKEN
    MISSING            SIMULATE GENERATION OF MISSING TOKEN
                       USING ERROR PRODUCTIONS.
    SEMANTICS          RESET TOKEN VALUE AND SYMBOL TABLE
                       ADDRESS TO CONSTANT OF APPROPRIATE TYPE
                       DEPENDING ON STATE OF THE PARSE STACK.
    UNDECLARED         ASSUME TYPE INTEGER OR TYPE STRING
                       DEPENDING ON THE STATE OF THE PARSE
                       STACK.
    INVALID            REJECT THE TOKEN.

  STATEMENT            REJECT INPUT UNTIL STATEMENT SEPARATOR
                       IS DETECTED.
                       FLUSH STACK UNTIL STATEMENT SEPARATOR
                       IS DETECTED.

  BLOCK                REJECT INPUT UNTIL STATEMENT SEPARATOR
                       IS DETECTED.

  UNRECOVERABLE        UNRECOVERABLE ERROR.
                       DUMP PARSE STACK AND TERMINATE SYNTAX
                       ANALYSIS.
```

```
SYNTAX ERROR DIAGNOSIS - TOKEN MISSING.
  ACTION:
    ISSUE ERROR MESSAGE.
    ATTEMPT TO RECOVER BY SIMULATING THE GENERATION OF
    A MISSING TOKEN USING AN ERROR PRODUCTION.
    RESUME SYNTAX ANALYSIS.

  DEFINITION OF ERROR PRODUCTIONS:
     LOGICAL END - RESET SYNTAX STACK.
       -1  <PROGRAM> ==> <EXECUTE BLOCK> #END# #;#
     GENERATE MISSING PROGRAM NAME.
       -2  <PROGRAM> ==> <BODY> #;#
     GENERATE MISSING PROGRAM BEGIN.
       -3  <BODY> ==> <DECLARE BLOCK> <EXECUTE BLOCK> #END#
     GENERATE MISSING TOKEN.
       -4  #TERMINAL# ==> %EMPTY%  (SIMULATE POP ACTION)
     TERMINATE REPEAT BLOCK.
       -5  <REPEAT BLOCK> ==> %EMPTY%


SYNTAX ERROR DIAGNOSIS - TOKEN SEMANTICS, INCONSISTENT
                         IDENTIFIER TYPE ENCOUNTERED.
  ACTION:
    ISSUE ERROR MESSAGE.
    DEPENDING UPON THE STATE OF THE PARSE STACK, RESET
    THE TOKEN VALUE TO PERMIT CONTINUATION OF SYNTAX
    ANALYSIS.
    FOR INTEGER TYPE - RESET TOKEN TO INTEGER CONSTANT.
                       RESET SYMBOL TABLE ADDRESS TO
                       INTEGER ONE.
    FOR STRING TYPE - RESET TOKEN TO STRING CONSTANT.
                      RESET SYMBOL TABLE ADDRESS TO STRING
                      BLANK.
    FOR LABEL TYPE - RESET TOKEN TO LABEL CONSTANT.
    NOTE: THE ONLY RECOVERY OF LABEL DECLARATION IS
          PERFORMED WHEN THE TOKEN VALUE IS #NEW IDENTIFIER#.
          THIS ALLEVIATES THE PROBLEM OF CHECKING FOR
          MULTIPLE LABEL DECLARATIONS.
    RESUME SYNTAX ANALYSIS WITH NEW TOKEN.


SYNTAX ERROR DIAGNOSIS - TOKEN UNDECLARED, INTEGER
                         IDENTIFIER EXPECTED.
  ACTION:
    ISSUE ERROR MESSAGE.
    ASSUME IDENTIFIER IS TYPE INTEGER.
    INSERT INTO SYMBOL TABLE AT THE CURRENT LEVEL.
    ASSIGN TYPE INTEGER IDENTIFIER TO TYPE ATTRIBUTE.
    RESUME PARSING WITH NEW TOKEN.
```

```
SYNTAX ERROR DIAGNOSIS - TOKEN UNDECLARED, STRING IDENTIFIER
                         EXPECTED.
  ACTION:
    ISSUE ERROR MESSAGE.
    ASSUME IDENTIFIER IS TYPE STRING.
    INSERT INTO SYMBOL TABLE AT THE CURRENT LEVEL.
    ASSIGN TYPE STRING IDENTIFIER TO TYPE ATTRIBUTE.
    RESUME PARSING WITH NEW TOKEN.


SYNTAX ERROR DIAGNOSIS - TOKEN INVALID.
   ACTION:
     ISSUE ERROR MESSAGE.
     IGNORE TOKEN.
     CONTINUE SYNTAX ANALYSIS WITH NEXT TOKEN.


SYNTAX ERROR DIAGNOSIS -  STATEMENT ERROR.
  ACTION:
    ISSUE ERROR MESSAGE.
    REJECT TOKENS UNTIL A STATEMENT SEPARATOR, #;# OR
    #THEN#, IS DETECTED.
    FLUSH PARSE STACK UNTIL A STATEMENT SEPARATOR, #;# OR
    #THEN, IS DETECTED.
    INITIATE SEMANTIC ERROR RECOVERY.
    RESUME PARSING WITH NEXT TOKEN.


SYNTAX ERROR DIAGNOSIS -  BLOCK ERROR.
  ACTION:
    ISSUE ERROR MESSAGE.
    REJECT TOKENS UNTIL A STATEMENT SEPARATOR, #;#, IS
    DETECTED.
    RESUME PARSING WITH THE NEXT TOKEN.


SYNTAX ERROR DIAGNOSIS - UNRECOVERABLE ERROR.
  ACTION:
    ISSUE TERMINATION MESSAGE.
    DUMP PARSE STACK.
    DISPLAY LAST TOKEN INPUT.
    TERMINATE SYNTAX ANALYSIS.
```

POP: THE TOP OF THE PARSE STACK AND THE NEXT TOKEN ARE THE SAME TERMINAL.
ACTION:
POP THE TOP OF THE PARSE STACK.
RESUME PARSING WITH NEXT TOKEN.

RESOLVE MULTIPLE DECLARATION - A DECLARATION IS IN PROGRESS AND THE IDENTIFIER RETURNED BY LEXICAL ANALYSIS HAS PREVIOUSLY BEEN ENTERED INTO THE SYMBOL TABLE.

ACTION:
OBTAIN AND COMPARE THE CURRENT LEVEL AND THE LEVEL AT WHICH THE PREVIOUS ENTRY WAS MADE IN THE SYMBOL TABLE.
IF EQUAL, MULTIPLE DECLARATION; RESET THE TOKEN VALUE TO #UNKNOWN# THEREBY TERMINATING THE DECLARATION.
OTHERWISE, INSERT A NEW ENTRY AT THE CURRENT LEVEL IN THE SYMBOL TABLE AND CONTINUE THE DECLARATION.

PUSH: THE TOP OF THE PARSE STACK IS A NON-TERMINAL, AND THE PRODUCTION TO REPLACE THE NON-TERMINAL ON THE PARSE STACK HAS BEEN DETERMINED BY THE NEXT TOKEN.
ACTION:
POP NON-TERMINAL FROM TOP OF PARSE STACK.
PUSH NEW PRODUCTION ONTO PARSE STACK.
CONSULT SEMANTIC ROUTINES FOR EVALUATION.
RESUME PARSING WITH THE CURRENT TOKEN.

ACCEPT: THE PARSE STACK IS EMPTY AND THE INPUT STRING IS EMPTY.
ACTION:
ACCEPT THE PROGRAM.
CONSULT SEMANTIC ROUTINES FOR TERMINATION OF SEMANTIC ANALYSIS.
TERMINATE SYNTAX ANALYSIS.

```
SEMANTIC ANALYSIS:  PROGRAM EVALUATION.
  ACTION:
    SEMANT ENTRY - INITIALIZE LIMITS, POINTERS, VARIABLES,
                   AND VALUES USED IN SEMANTIC ANALYSIS.
    EVAL ENTRY   - EVALUATE NODE OF SYNTAX TREE USING SYNTAX
                   DIRECTED TRANSLATION.
    SFLUSH ENTRY - SYNCHRONIZE SEMANTIC STACK WITH SYNTAX
                   STACK.


DEFINITION OF SYNTAX DIRECTED TRANSLATION ELEMENTS:
 1 #DCL LABEL# <BODY>
 2 #GEN BEGIN OP# <DECLARE BLOCK> <EXECUTE BLOCK>
   #GEN END OP#
 3 #DCL INTEGER# <DECLARE VARIABLE> #;# <DECLARE BLOCK>
 4 #DCL STRING# <DECLARE VARIABLE> #;# <DECLARE BLOCK>
 5 %EMPTY%
 6 #DCL ADDRESS# <SIZE> #DCL ID# <2DECLARE VARIABLE>
 7 <1SIZE> #DCL COLUMN# <2SIZE> #DCL ROW#
 8 #DCL SCALAR#
 9 #DCL DIMENSION VALUE#
10 <1SIZE>
11 %EMPTY%
12 <DECLARE VARIABLE>
13 %EMPTY%
14 <LABEL> <BLOCK> <EXECUTE BLOCK>
15 %EMPTY%
16 #DCL LABEL#
17 %EMPTY%
18 <BODY> #;#
19 <SIMPLE STATEMENT>
20 <IF STATEMENT>
21 <NOT IF STATEMENT> #;#
22 <RELATION> #THEN# <THEN CLAUSE>
23 <INTEGER EXPRESSION> <2INTEGER RELATION>
24 <STRING EXPRESSION> <2STRING RELATION>
25 <SIGNED INTEGER TERM> <2INTEGER TERM>
26 #DCL SIGN# <INTEGER TERM> #GEN OP#
27 <INTEGER TERM>
28 <INTEGER WORD> <2INTEGER WORD>
29 <INTEGER EXPRESSION>
30 <INTEGER VARIABLE> #GEN VALUE OF INTEGER OP#
31 #GEN OPERAND# #GEN VALUE OF INTEGER OP#
32 #GEN OPERAND# <SUBSCRIPT>
33 <INTEGER EXPRESSION> <2SUBSCRIPT>
34 #GENERATE SCALAR#
35 <INTEGER EXPRESSION> #GEN ROW MAXIMUM#
   #GEN SUBSCRIPT2 OF OP#
36 #GEN SUBSCRIPT1 OF OP#
37 #DCL OP TYPE = INTEGER# <INTEGER WORD> #GEN OP#
   <2INTEGER WORD>
38 %EMPTY%
```

```
39 #DCL OP TYPE = INTEGER# <INTEGER TERM> #GEN OP#
   <2INTEGER TERM>
40 %EMPTY%
41 #DCL OP TYPE = INTEGER# <INTEGER EXPRESSION> #GEN OP#
42 <STRING VARIABLE> #GEN VALUE OF STRING OP#
43 #GEN OPERAND# #GEN VALUE OF STRING OP#
44 #GEN OPERAND# <SUBSCRIPT>
45 #DCL OP TYPE = STRING# <STRING EXPRESSION> #GEN OP#
46 <BODY> #;# <ELSE CLAUSE>
47 <NOT IF STATEMENT> #;# <ELSE CLAUSE>
48 <IF STATEMENT>
49 <BLOCK> #GEN IF_THEN_ELSE OP#
50 #GEN IF_THEN OP#
51 <ASSIGNMENT>
52 #GEN UNTIL OP# <REPEAT BLOCK> #GEN REPEAT OP#
53 #DCL IO OP = INPUT# <LIST>
54 #DCL IO OP = OUTPUT# <LIST>
55 <2LABEL> #GEN GOTO OP#
56 <INTEGER VARIABLE> <INTEGER EXPRESSION>
   #GEN INTEGER ASSIGN OP#
57 <STRING VARIABLE> <STRING EXPRESSION>
   #GEN STRING ASSIGN OP#
58 <RELATION> #;# <EXECUTE BLOCK>
59 #DCL IO TYPE = INTEGER# <INTEGER VARIABLE> #GEN OP#
   <2LIST>
60 #DCL IO TYPE = STRING# <STRING VARIABLE> #GEN OP# <2LIST>
61 <LIST>
62 %EMPTY%
63 #GEN LABEL#
64 #GEN LABEL#
```

```
 ACCESS RIGHTS - SEMANTIC ANALYSIS:
   GLOBAL VARIABLES:

** TRACE FLAGS **
#T#SYN BIT(1) STATIC EXTERNAL          ** SYNTAX ANALYSIS **
#T#SEM BIT (1) STATIC EXTERNAL         ** SEMANTIC ANALYSIS **

** DEBUG FLAGS **
#D#SEM BIT (1) STATIC EXTERNAL

** REPLACEMENT MODE FLAG **
  ** MODE 3 - REPLACE SYNTAX ANALYSIS  **
#R#SYN BIT (1) STATIC EXTERNAL


** EXTERNAL ENTRY POINTS AND PARAMETER LIST ATTRIBUTES **

  ** CODE GENERATION **
BUILD ENTRY (FIXED BINARY (31))
ASTOR ENTRY (FIXED BINARY (31))
ACODE ENTRY (FIXED BINARY (31))
GEN ENTRY

  ** SYMBOL TABLE **
ISCOPE ENTRY (FIXED BINARY (31))
DSCOPE ENTRY (FIXED BINARY (31))
QSCOPE ENTRY (FIXED BINARY (31))
S@ADDR ENTRY (FIXED BINARY (31),FIXED BINARY (31))
S@TYPE ENTRY (FIXED BINARY (31),FIXED BINARY (31))
S@ROW ENTRY (FIXED BINARY (31),FIXED BINARY (31))
S@COL ENTRY (FIXED BINARY (31),FIXED BINARY (31))
G@ADDR ENTRY (FIXED BINARY (31),FIXED BINARY (31))
G@TYPE ENTRY (FIXED BINARY (31),FIXED BINARY (31))
G@LEVL ENTRY (FIXED BINARY (31),FIXED BINARY (31))
G@ROW ENTRY (FIXED BINARY (31),FIXED BINARY (31))
G@COL ENTRY (FIXED BINARY (31),FIXED BINARY (31))
G@IVAL ENTRY (FIXED BINARY (31),FIXED BINARY (31))
```

SEMANTIC ROUTINE: EVALUATE NODE OF SYNTAX TREE.
ACTION:
PUSH SEMANTIC PRODUCTION CORRESPONDING TO
SYNTAX PRODUCTION ONTO SEMANTIC STACK.
EXAMINE TOP OF SEMANTIC STACK AND PERFORM
SEMANTIC ACTION INDICATED.
UPON COMPLETION OF A SEMANTIC ACTION FOR
A TERMINAL, POP THE TERMINAL FROM THE STACK
AND REPEAT THE INTERROGATION ACTION.

SEMANTIC ACTION - OVERVIEW:

NON TERMINALS	RETURN.
TERMINALS:	
DECLARATION - NON GENERATION ACTIONS.	
DECLARE LABEL	ESTABLISH RUN-TIME-ADDRESS AND TYPE = #LABEL IDENTIFIER# FOR LABEL CONSTANT.
DECLARE INTEGER ID	ESTABLISH IDENTIFIER TYPE = INTEGER.
DECLARE STRING ID	ESTABLISH IDENTIFIER TYPE = STRING.
DECLARE ID	COMPLETE DECLARATION OF IDENTIFIER; I.E. ASSIGN VALUE TO COMPILE-TIME ATTRIBUTES: RUN-TIME-ADDRESS, IDENTIFIER TYPE, AND ROW AND COLUMN OF ARRAY.
DECLARE ADDRESS	SAVE SYMBOL TABLE ADDRESS OF NEW IDENTIFIER UNTIL COMPLETION OF DECLARATION.
DECLARE SCALAR	ESTABLISH IDENTIFIER STRUCTURE AS SCALAR.
DECLARE ROW	ESTABLISH ROW VALUE OF 2-DIMENSIONAL ARRAY.
DECLARE COLUMN	ESTABLISH COLUMN VALUE OF ARRAY.
DECLARE DIMENSION	ESTABLISH VALUE OF DIMENSION.
DECLARE INTEGER OP	ESTABLISH VALUE OF BINARY INTEGER OPERATOR.
DECLARE STRING OP	ESTABLISH VALUE OF BINARY STRING OPERATOR.
DECLARE SIGN OP	ESTABLISH VALUE OF UNARY INTEGER OPERATOR.
DECLARE IO OP	ESTABLISH VALUE OF INPUT/OUTPUT OPERATOR.
DECLARE INTEGER IO	ESTABLISH INPUT/OUTPUT TYPE AS INTEGER.
DECLARE STRING IO	ESTABLISH INPUT/OUTPUT TYPE AS STRING.
END STATEMENT MARKER	NO OP.

GENERATION - SYNTAX DIRECTED TRANSLATION ACTIONS.
GENERATE POLISH SUFFIX INTERMEDIATE CODE.

GENERATE BEGIN	INCREMENT SCOPE OF SYMBOL TABLE. GENERATE BEGIN-BLOCK OPERATOR.
GENERATE END	DECREMENT SCOPE OF SYMBOL TABLE. GENERATE END-BLOCK OPERATOR.
GENERATE UNTIL	INCREMENT SCOPE OF SYMBOL TABLE. GENERATE BEGIN-REPEAT-BLOCK OPERATOR.
GENERATE REPEAT	DECREMENT SCOPE OF SYMBOL TABLE. GENERATE END-REPEAT-BLOCK OPERATOR.
GENERATE END PROGRAM	THE ACCEPT ACTION HAS BEEN INITIATED BY SYNTAX ANALYSIS. GENERATE END-PROGRAM OPERATOR. ESTABLISH LOGICAL END STATUS. INITIATE GENERATION OF INTERMEDIATE CODE STREAM.
GENERATE OPERAND	GENERATE RUN-TIME ADDRESS OF IDENTIFIER/CONSTANT AS OPERAND.
GENERATE LABEL	RESOLVE BACKWARD LABEL REFERENCE WITHIN ACTIVE BLOCK AND GENERATE OPERAND. GENERATE TEMPORARY OPERAND FOR UNRESOLVED LABEL REFERENCE.
GENERATE SCALAR	VERIFY SCALAR STRUCTURE OF LAST OPERAND.
GENERATE SUBSCRIPT1	GENERATE OPERATOR FOR ONE-DIMENSIONAL ARRAY.
GENERATE ROW MAXIMUM	GENERATE ROW REFERENCE FOR TWO-DIMENSIONAL ARRAY.
GENERATE SUBSCRIPT2	GENERATE OPERATOR FOR TWO-DIMENSIONAL ARRAY.
GENERATE OPERATOR	GENERATE OP CODE ON THE TOP OF SEMANTIC STACK AS OPERATOR.

DEFINITION OF INTERMEDIATE OPERATOR CODES:
CP CODE = TOKEN VALUE (2 DIGITS) + OPERAND TYPE (1 DIGIT)
+ NUMBER OF OPERANDS (1 DIGIT)

CP CODE	ACTION	OPERAND TYPE	#OPERANDS
0521	ASSIGN	INTEGER	1
0531	ASSIGN	STRING	1
1021	PLUS	INTEGER	1
1022	ADDITION	INTEGER	2
1131	MINUS	INTEGER	1
1122	SUBTRACTION	INTEGER	2
2022	MULTIPLICATION	INTEGER	2
2122	DIVISION	INTEGER	2
3022	COMPARE LESS THAN OR EQUAL	INTEGER	2
3032	COMPARE LESS THAN OR EQUAL	STRING	2
3122	COMPARE LESS THAN	INTEGER	2
3132	COMPARE LESS THAN	STRING	2
3222	COMPARE EQUAL	INTEGER	2
3232	COMPARE EQUAL	STRING	2
3322	COMPARE GREATER THAN OR EQUAL	INTEGER	2
3332	COMPARE GREATER THAN OR EQUAL	STRING	2
3422	COMPARE GREATER THAN	INTEGER	2
3432	COMPARE GREATER THAN	STRING	2
3522	COMPARE NOT EQUAL	INTEGER	2
3532	COMPARE NOT EQUAL	STRING	2
6000	BEGIN_BLOCK		0
6100	IF_THEN_ELSE		0
6200	END_BLOCK		0
6311	GOTO	LABEL	1
6400	IF_THEN		0
6521	INPUT	INTEGER	1
6531	INPUT	STRING	1
6721	OUTPUT	INTEGER	1
6731	OUTPUT	STRING	1
6800	END_REPEAT_BLOCK		0
7100	BEGIN_REPEAT_BLOCK		0
8021	VALUE_OF	INTEGER	1
8031	VALUE_OF	STRING	1
8122	SUBSCRIPT_OF	INTEGER	2
8132	SUBSCRIPT_OF	STRING	2
8124	SUBSCRIPT_OF	INTEGER	4
8134	SUBSCRIPT_OF	STRING	4

*** STOP CODES ***

9200-9299	STOP SYNTAX ERROR	0
9300-9399	STOP SEMANTIC ERROR	0
9999	STOP END OF PROGRAM	0

SEMANTIC ROUTINE: SEMANTIC ERROR RECOVERY.
  ACTION:
    SYNCHRONIZE SEMANTIC STACK WITH THE SYNTAX STACK IN AN
    ERROR RECOVERY ATTEMPT BY FLUSHING THE SEMANTIC STACK
    UNTIL AN END-STATEMENT-MARKER IS RECOGNIZED.

```
CODE GENERATION - GENERATE INTERMEDIATE POLISH SUFFIX CODE
                  STREAM.
  ACTION:
    CODEGEN ENTRY - INITIALIZE LIMITS, POINTERS, AND
                    VARIABLES USED IN CODE GENERATION.
    BUILD ENTRY   - BUILD POLISH SUFFIX CODE STREAM.
    ACODE ENTRY   - ALLOCATE ADDRESS FOR LABEL CONSTANT.
    ASTOR ENTRY   - ALLOCATE STORAGE FOR CONSTANTS AND
                    IDENTIFIERS.
    GEN ENTRY     - RESOLVE ALL UNRESOLVED LABEL REFERENCES.
                    GENERATE POLISH SUFFIX CODE STREAM.
                    INITIATE GENERATION OF STORAGE
                    INITIALIZATION TABLES FOR INTEGER
                    CONSTANTS AND FOR STRING CONSTANTS.


  ACCESS RIGHTS - CODE GENERATION:
     GLOBAL VARIABLES - CODE GENERATION:

 ** TRACE FLAG **
  #T#CODE BIT(1) STATIC EXTERNAL,

 ** DEBUG FLAG **
  #D#CODE BIT(1) STATIC EXTERNAL,


** EXTERNAL ENTRY POINTS AND PARAMETER LIST ATTRIBUTES **

  ** SYMBOL TABLE **
ISCOPE ENTRY (FIXED BINARY (31)),
DSCOPE ENTRY (FIXED BINARY (31)),
G@ADDR ENTRY (FIXED BINARY (31),FIXED BINARY (31)),
G@NAME ENTRY (FIXED BINARY (31),CHARACTER (70) VARYING),
ISTAB ENTRY,


CODE GENERATION ROUTINE - BUILD INTERMEDIATE CODE STREAM.
  ACTION:
    PLACE CODE WORD IN CODE STREAM.


CODE GENERATION ROUTINE - ALLOCATE RUN-TIME-ADDRESS FOR
                          LABEL CONSTANT.
  ACTION:
    RETURN THE ADDRESS OF THE NEXT CODE WORD TO BE
    ALLOCATED.
```

CODE GENERATION ROUTINE - ALLOCATE STORAGE FOR CONSTANTS AND IDENTIFIERS.

ACTION:
ATTEMPT TO ALLOCATE STORAGE BASED UPON THE AMOUNT REQUESTED.
IF SUCCESSFUL, RETURN THE STARTING HIGH-CORE ADDRESS OF THE STORAGE ALLOCATED.
IF UNSUCCESSFUL, RETURN THE COMPLIMENT OF THE AMOUNT REQUESTED AS AN INDICATION OF FAILURE TO ALLOCATE.

CODE GENERATION ROUTINE - GENERATE INTERMEDIATE CODE STREAM.

ACTION:
RESOLVE ALL UNRESOLVED LABEL REFERENCES.
GENERATE POLISH SUFFIX CODE STREAM.
INITIATE GENERATION OF STORAGE INITIALIZATION TABLES FOR INTEGER CONSTANTS AND STRING CONSTANTS.

SYMBOL TABLE MANAGEMENT - CONSTRUCT, UPDATE, AND RESPOND TO QUERIES.

THERE ARE FOUR SEPARATE LIST MAINTAINED BY THE SYMBOL TABLE MANAGEMENT:

1) IDENTIFIER - KEYWORDS AND IDENTIFIERS.
2) INTEGER - INTEGER CONSTANTS.
3) STRING - STRING CONSTANTS.
4) USER - (MODE 1 ONLY) CONDENSED USER SYMBOL TABLE.

ROUTINES:

SYMTAB ENTRY - INITILIZE LISTS, POINTERS AND VARIABLES USED BY SYMBOL TABLE MANAGEMENT.

SEARCH ENTRY - SCAN THE APPROPRIATE LIST FOR PREVIOUS ENTRY. IF FOUND, RETURN THE ADDRESS. OTHERWISE INSERT AND RETURN THE ADDRESS ASSIGNED.

INSERT ENTRY - INSERT IDENTIFIER AT CURRENT LEVEL.

*** SCOPE QUERY ***
ISCOPE ENTRY - INCREMENT SCOPE OF IDENTIFIER LIST.
DSCOPE ENTRY - DECREMENT SCOPE OF IDENTIFIER LIST.
QSCOPE ENTRY - INTERROGATE SCOPE OF IDENTIFIER LIST.

*** SET ATTRIBUTE QUERY ***
S@ADDR ENTRY - ASSIGN VALUE TO RUN ADDRESS ATTRIBUTE.
S@TYPE ENTRY - ASSIGN VALUE TO TYPE ATTRIBUTE.
S@ROW ENTRY - ASSIGN VALUE TO ROW ATTRIBUTE.
S@COL ENTRY - ASSIGN VALUE TO COLUMN ATTRIBUTE.

*** GET ATTRIBUTE QUERY ***
G@ADDR ENTRY - FETCH VALUE OF RUN ADDRESS ATTRIBUTE.
G@TYPE ENTRY - FETCH VALUE OF TYPE ATTRIBUTE.
G@LEVL ENTRY - FETCH VALUE OF LEVEL ATTRIBUTE.
G@ROW ENTRY - FETCH VALUE OF ROW ATTRIBUTE.
G@COL ENTRY - FETCH VALUE OF COLUMN ATTRIBUTE.
G@NAME ENTRY - FETCH VALUE OF NAME ATTRIBUTE.
G@IVAL ENTRY - FETCH VALUE OF INTEGER_VALUE ATTRIBUTE.

*** CODE GENERATION OUTPUT ***
ISTAB ENTRY - GENERATE STORAGE TABLES.

*** POST COMPILATION ACTIVITIES ***
PCSTAT ENTRY - (MODE 1 ONLY) EVALUATE CONSISTENCY OF ADDRESS ALLOCATION BY REPLACEMENT MODULE.

```
  ACCESS RIGHTS - SYMBOL TABLE MANAGEMENT:
    GLOBAL VARIABLES:

** TRACE FLAG **
#T#STAB BIT (1) STATIC EXTERNAL

** DEBUG FLAG **
#D#STAB BIT (1) STATIC EXTERNAL

** REPLACEMENT MODE FLAGS **
  ** MODE 1 - REPLACE SYMBOL TABLE **
#R#STAB BIT (1) STATIC EXTERNAL
 ** MODE 2 - REPLACE LEXICAL ANALYSIS **
#R#LEX BIT (1) STATIC EXTERNAL


** EXTERNAL ENTRY POINTS AND PARAMETER LIST ATTRIBUTES **

  ** REPLACEMENT MODULE ENTRY POINTS **
RSEARCH ENTRY (FIXED BINARY (31),FIXED BINARY (31),
               CHARACTER (70) VARYING)
RINSERT ENTRY (FIXED BINARY (31),FIXED BINARY (31),
               CHARACTER (70) VARYING)
RISCOPE ENTRY (FIXED BINARY (31))
RDSCOPE ENTRY (FIXED BINARY (31))
RQSCOPE ENTRY (FIXED BINARY (31))
RS@ADDR ENTRY (FIXED BINARY (31),FIXED BINARY (31))
RS@TYPE ENTRY (FIXED BINARY (31),FIXED BINARY (31))
RS@ROW ENTRY (FIXED BINARY (31),FIXED BINARY (31))
RS@COL ENTRY (FIXED BINARY (31),FIXED BINARY (31))
RG@ADDR ENTRY (FIXED BINARY (31),FIXED BINARY (31))
RG@TYPE ENTRY (FIXED BINARY (31),FIXED BINARY (31))
RG@LEVL ENTRY (FIXED BINARY (31),FIXED BINARY (31))
RG@ROW ENTRY (FIXED BINARY (31),FIXED BINARY (31))
RG@COL ENTRY (FIXED BINARY (31),FIXED BINARY (31))
RG@IVAL ENTRY (FIXED BINARY (31),FIXED BINARY (31))
RG@NAME ENTRY (FIXED BINARY (31),CHARACTER (70) VARYING)
```

SYMBOL TABLE ROUTINE - SEARCH SYMBOL TABLE FOR FIRST OCCURANCE OF SPECIFIED ENTRY.

ACTION:
SEQUENTIAL SEARCH OF APPROPRIATE LIST FOR FIRST OCCURANCE.
IF ENTRY IS FOUND, RETURN SYMBOL TABLE ADDRESS.
IF ENTRY IS NOT FOUND - INSERT A NEW ENTRY, AND RETURN THE SYMBOL TABLE ADDRESS ASSIGNED.

IDENTIFIER LIST - THE ADDRESS OF A KEYWORD OR AN IDENTIFIER HAS BEEN REQUESTED BY LEXICAL ANALYSIS.

ACTION:
SCAN THE KEYWORD LIST - LEVEL ZERO OF THE IDENTIFIER LIST.
IF FOUND, RETURN THE KEYWORD TYPE VALUE AND THE SYMBOL TABLE ADDRESS.
OTHERWISE - SCAN THE IDENTIFIER LIST, BEGINNING WITH THE INNERMOST LEVEL.
IF FOUND, RETURN THE IDENTIFIER TYPE AND AND THE SYMBOL TABLE ADDRESS.
OTHERWISE INSERT THE NEW IDENTIFIER AT THE CURRENT LEVEL.
RETURN THE TYPE VALUE OF NEW IDENTIFIER AND THE SYMBOL TABLE ADDRESS ASSIGNED.

IDENTIFIER LIST STRUCTURE - OVERVIEW:
THE IDENTIFIER LIST IS A SEQUENTIAL LAST-IN-FIRST-OUT LIST WITH BLOCK STRUCTURE SUPERIMPOSED.

ELEMENT DEFINITION:

```
01 IDENTIFIER_LIST
   05 @RUN_ADDRESS FIXED BINARY (31)
   05 @TYPE FIXED BINARY (31)
   05 @LEVEL FIXED BINARY (31)
   05 @COLUMN FIXED BINARY (31)
   05 @ROW FIXED BINARY (31)
   05 @NAME CHARACTER (8)
```

IMPLEMENTATIONS RESTRICTIONS:
THE SEARCH KEY FOR THE IDENTIFIER LIST IS THE FIRST EIGHT CHARACTERS OF THE IDENTIFIER NAME.

INTEGER CONSTANT LIST - THE ADDRESS OF AN INTEGER CONSTANT HAS BEEN REQUESTED BY LEXICAL ANALYSIS.

ACTION:
SEQUENTIALLY SEARCH THE INTEGER CONSTANT LIST BEGINNING WITH THE MOST RECENT ENTRY.
IF FOUND, RETURN THE SYMBOL TABLE ADDRESS.
OTHERWISE, INSERT A NEW INTEGER CONSTANT AND RETURN THE SYMBOL TABLE ADDRESS ASSIGNED.

INTEGER LIST STRUCTURE - OVERVIEW:
THE INTEGER CONSTANT LIST IS A LAST-IN-FIRST-OUT SEQUENTIAL LIST.

ELEMENT DEFINITION:

```
01 INTEGER_LIST
   05 @RUN_ADDRESS FIXED BINARY (31)
   05 @INTEGER_VALUE FIXED BINARY (31)
```

IMPLEMENTATION RESTRICTIONS:
A VALUE OF ONE WILL BE SUBSTITUTED FOR INTEGER CONSTANTS WHICH EXCEED EIGHT DIGITS.
ONLY UNSIGNED INTEGER CONSTANTS ARE MAINTAINED IN THIS LIST.

STRING CONSTANT LIST - THE ADDRESS OF A STRING CONSTANT
HAS BEEN REQUESTED BY LEXICAL ANALYSIS.

ACTION:
SEQUENTIALLY SEARCH THE STRING
CONSTANT LIST BEGINNING WITH THE
MOST RECENT ENTRY.
IF FOUND, RETURN THE SYMBOL TABLE
ADDRESS.
OTHERWISE, INSERT A NEW STRING
CONSTANT AND RETURN THE SYMBOL TABLE
ADDRESS ASSIGNED.

STRING LIST STRUCTURE - OVERVIEW:
THE STRING CONSTANT LIST IS A
LAST-IN-FIRST-OUT SEQUENTIAL LIST.

ELEMENT DEFINITION:

```
01 STRING_LIST
   05 @RUN_ADDRESS FIXED BINARY (31)
   05 @SIZE FIXED BINARY (31)
   05 @STRING_VALUE CHARACTER (70)
```

IMPLEMENTATION RESTRICTIONS:
STRING CONSTANTS WHICH EXCEED SEVENTY
CHARACTERS WILL BE TRUNCATED TO
SEVENTY CHARACTERS.

SYMBOL TABLE ROUTINE - INSERT A NEW ENTRY AT THE CURRENT LEVEL.

ACTION:

INSERT A NEW IDENTIFIER AT THE CURRENT LEVEL.
ASSIGN TYPE = #NEW IDENTIFIER#.
RETURN TYPE AND SYMBOL TABLE ADDRESS ASSIGNED.

SYMBOL TABLE ROUTINE - INCREMENT SCOPE.

ACTION:

INCREMENT SCOPE OF IDENTIFIER LIST.
ADJUST BLOCK POINTERS FOR NEW LEVEL.
RETURN NEW ACTIVE LEVEL VALUE.

SYMBOL TABLE ROUTINE - DECREMENT SCOPE.

ACTION:

DECREMENT SCOPE OF IDENTIFIER LIST.
RETURN NEW ACTIVE LEVEL VALUE.

SYMBOL TABLE ROUTINE - QUERY SCOPE.

ACTION:

INTERROGATE THE SCOPE OF THE INNER-MOST ACTIVE LEVEL OF THE IDENTIFIER LIST.
RETURN THE ACTIVE LEVEL VALUE.

SYMBOL TABLE ACTION - ATTRIBUTE QUERY ADDRESS VERIFICATION.
ACTION:
DETERMINE THE ACCESSIBILITY OR INACCESSIBILITY OF THE SYMBOL TABLE ADDRESS REQUESTED BY AN ATTRIBUTE QUERY.
IF ACCESSIBLE, ACTIVATE THE APPROPRIATE LIST.
IF INACCESSIBLE, FLAG THE ERROR, SET ERROR RETURN VALUES AND EXIT; I.E., IGNORE THE QUERY.
(MODE 1 ONLY) ESTABLISH ADDRESS FOR CALL TO REPLACEMENT MODULE.
NOTE: BY DESIGN ALL SYMBOL TABLE ADDRESSES WITHIN ORACLE ARE UNIQUE; THEREFORE ONLY ONE LIST WILL BE ACTIVE ON ANY ONE ENTRY.
THE KEYWORDS ARE NEVER ACCESSIBLE FOR AN ATTRIBUTE QUERY.

SYMBOL TABLE ROUTINE - ASSIGN VALUE TO RUN ADDRESS ATTRIBUTE.
ACTION:
ASSIGN VALUE TO RUN ADDRESS ATTRIBUTE OF ENTRY, IDENTIFIER OR CONSTANT, SPECIFIED.

SYMBOL TABLE ROUTINE - ASSIGN VALUE TO TYPE ATTRIBUTE.
ACTION:
ASSIGN VALUE TO TYPE ATTRIBUTE OF IDENTIFIER SPECIFIED.

SYMBOL TABLE ROUTINE - ASSIGN VALUE TO ROW ATTRIBUTE.
ACTION:
ASSIGN VALUE TO ROW ATTRIBUTE OF IDENTIFIER SPECIFIED.

SYMBOL TABLE ROUTINE - ASSIGN VALUE TO COLUMN ATTRIBUTE.
ACTION:
ASSIGN VALUE TO COLUMN ATTRIBUTE OF IDENTIFIER SPECIFIED.

SYMBOL TABLE ROUTINE - FETCH VALUE OF RUN ADDRESS ATTRIBUTE.
ACTION:
FETCH VALUE OF RUN ADDRESS ATTRIBUTE OF ENTRY, IDENTIFIER OR CONSTANT, SPECIFIED.

SYMBOL TABLE ROUTINE - FETCH VALUE OF TYPE ATTRIBUTE.
ACTION:
FETCH VALUE OF TYPE ATTRIBUTE OF IDENTIFIER SPECIFIED.

SYMBOL TABLE ROUTINE - FETCH VALUE OF LEVEL ATTRIBUTE.
ACTION:
FETCH VALUE OF LEVEL ATTRIBUTE OF IDENTIFIER SPECIFIED.

SYMBOL TABLE ROUTINE - FETCH VALUE OF ROW ATTRIBUTE.
ACTION:
FETCH VALUE OF ROW ATTRIBUTE OF IDENTIFIER SPECIFIED.

SYMBOL TABLE ROUTINE - FETCH VALUE OF COLUMN ATTRIBUTE.
ACTION:
FETCH VALUE OF COLUMN ATTRIBUTE OF IDENTIFIER SPECIFIED.

SYMBOL TABLE ROUTINE - FETCH VALUE OF NAME ATTRIBUTE.
ACTION:
FETCH VALUE OF NAME ATTRIBUTE OF IDENTIFIER SPECIFIED.

SYMBOL TABLE ROUTINE - FETCH VALUE OF INTEGER VALUE ATTRIBUTE.
ACTION:
FETCH VALUE OF INTEGER CONSTANT SPECIFIED.

SYMBOL TABLE ROUTINE - GENERATE STORAGE TABLES.
ACTION:
GENERATE STORAGE INITIALIZATION TABLES FOR INTEGER CONSTANTS AND STRING CONSTANTS.

SYMBOL TABLE ROUTINE - POST COMPILATION STATISTICS.
ACTION:
(MODE 1 ONLY) EVALUATE THE CONSISTENCY OF ADDRESS ALLOCATION BY CHECKING THE CONDENSED USER SYMBOL TABLE WHICH IS MAINTAINED BY ORACLE FOR MULTIPLE ALLOCATIONS OF THE SAME ENTRY ADDRESS.

REPLACEMENT MODE ACTIVATION - PARM FIELD VALUES.

THE REPLACEMENT MODE OPERATIONS ARE DETERMINED BY INPUT TO THE MAIN ROUTINE AT RUN TIME USING THE PARM FIELD. THE ORDER OF THE KEYWORD PARAMETERS IS NOT RESTRICTIVE; HOWEVER, THERE SHOULD BE 1 VALUE FOR THE REPLACEMENT MODE TO BE ACTIVATED:

M=0 ==> NO REPLACEMENT (DEFAULT IF REPLACEMENT IS NOT SPECIFIED).
M=1 ==> REPLACEMENT MODE 1 - SYMBOL TABLE MANAGEMENT.
M=2 ==> REPLACEMENT MODE 2 - LEXICAL ANALYSIS.
M=3 ==> REPLACEMENT MODE 3 - SYNTAX ANALYSIS.

AND THE CONDITION WHICH WILL ALLOW TERMINATION OF REPLACEMENT:

C=0 ==> (MODE 1 MODE 2) UNCONDITIONAL REPLACEMENT.
C=1 ==> (MODE 3) TERMINATE REPLACEMENT ON SYNTAX ERROR IN THE INPUT STRING.

DEFAULT VALUE:
PARM=(M=0,C=0) (NO REPLACEMENT, NO TERMINATION CONDITION)

```
** UNIVERSAL ENTRY NAMES AND PARAMETER LIST ATTRIBUTES FOR
   REPLACEMENT MODE **

 *** LEXICAL ANALYSIS ***
 RLEX: PROCEDURE RECURSIVE   ** INITIALIZATION ENTRY **
 RSCAN: ENTRY (ADDRESS,TOKEN,NAME)  ** LEXICAL ENTRY **

** DEFINITION OF PARAMETERS **
  ADDRESS - FIXED BINARY (31)
    ENTRY VALUE = NONE ASSUMED
    EXIT VALUE = SYMBOL TABLE ADDRESS OF CURRENT TOKEN
  TOKEN - FIXED BINARY (31)
    ENTRY VALUE = NONE ASSUMED
    EXIT VALUE = TOKEN VALUE RECOGNIZED
  NAME - CHARACTER (70) VARYING
    ENTRY VALUE = NONE ASSUMED
    EXIT VALUE = INPUT STRING RECOGNIZED


*** SYNTAX ANALYSIS ***
 RSYNTAX: PROCEDURE RECURSIVE  ** INITIALIZATION ENTRY **
 RPARSER: ENTRY     ** SYNTAX ANALYSIS ENTRY **


*** SEMANTIC ANALYSIS ***
REVAL: ENTRY (ADDRESS,TOKEN,PRODUCTION#)  ** SEMANTIC
                                          EVALUATION ENTRY **
RSFLUSH: ENTRY ** SEMANTIC STACK SYNCHRONIZATION ENTRY
                  FOR SYNTAX ERROR RECOVERY **

** DEFINITION OF PARAMETERS **
   ADDRESS - FIXED BINARY (31)
     ENTRY VALUE = SYMBOL TABLE ADDRESS OF CURRENT TOKEN
     EXIT VALUE = NONE ASSUMED
   TOKEN - FIXED BINARY (31)
     ENTRY VALUE = CURRENT TOKEN VALUE
     EXIT VALUE = #UNKNOWN# (ZERO) IF ERROR.
                  OTHERWISE SAME AS ENTRY VALUE.
   PRODUCTION# - FIXED BINARY (31)
     ENTRY VALUE = PRODUCTION NUMBER APPLIED BY SYNTAX
                   ANALYSIS
     EXIT VALUE = UNDEFINED (ZERO) IF ERROR.
                  OTHERWISE SAME AS ENTRY VALUE.
```

```
*** CODE GENERATION ***
 RBUILD: ENTRY (WORD)  ** ERROR CODE GENERATION ENTRY **

** DEFINITION OF PARAMETERS **
   WORD - FIXED BINARY (31)
     ENTRY VALUE = ERROR CODE TO BE PLACED IN CODE STREAM
     EXIT VALUE = 9400 IF CODE GENERATION ABNORMALLY
                  TERMINATES.
                  OTHERWISE SAME AS ENTRY VALUE.

  ERROR CODES:
    9100-9199   LEXICAL ANALYSIS
    9200-9299   SYNTAX ANALYSIS
    9300-9399   SEMANTIC ANALYSIS
    9400-9499   CODE GENERATION
    9500-9599   SYMBOL TABLE MANAGEMENT
    9999        END OF PROGRAM


*** SYMBOL TABLE ***
  RTABLE: PROCEDURE RECURSIVE  ** INITIALIZATION ENTRY **
  RSEARCH: ENTRY (ADDRESS,TOKEN,NAME)  ** SEARCH ENTRY **
  RINSERT: ENTRY (ADDRESS,TOKEN,NAME) ** INSERT ENTRY **
  RISCOPE: ENTRY (ACTIVE_LEVEL)  ** INCREMENT SCOPE ENTRY **
  RDSCOPE: ENTRY (ACTIVE_LEVEL) ** DECREMENT SCOPE ENTRY **
  RQSCOPE: ENTRY (ACTIVE_LEVEL)  ** QUERY SCOPE ENTRY **
  RS@ADDR: ENTRY (ADDRESS,VALUE)  ** SET ATTRIBUTE RUN
                                     ADDRESS **
  RS@TYPE: ENTRY (ADDRESS,VALUE)  ** SET ATTRIBUTE TYPE **
  RS@ROW: ENTRY (ADDRESS,VALUE)  ** SET ATTRIBUTE ROW **
  RS@COL: ENTRY (ADDRESS,VALUE)  ** SET ATTRIBUTE COLUMN **
  RG@ADDR: ENTRY (ADDRESS,VALUE)  ** FETCH ATTRIBUTE RUN
                                     ADDRESS **
  RG@TYPE: ENTRY (ADDRESS,VALUE)  ** FETCH ATTRIBUTE TYPE **
  RG@LEVL: ENTRY (ADDRESS,VALUE)  ** FETCH ATTRIBUTE LEVEL **
  RG@ROW: ENTRY (ADDRESS,VALUE)  ** FETCH ATTRIBUTE ROW **
  RG@COL: ENTRY (ADDRESS,VALUE)  ** FETCH ATTRIBUTE COLUMN **
  RG@NAME: ENTRY (ADDRESS,NAME)  ** FETCH ATTRIBUTE NAME **
  RG@IVAL: ENTRY (ADDRESS,VALUE)  ** FETCH ATTRIBUTE INTEGER
                                     VALUE **
```

```
** DEFINITION OF PARAMETERS **
  RSEARCH AND RINSERT ENTRY:
   ADDRESS - FIXED BINARY (31)
     ENTRY VALUE = NONE ASSUMED
     EXIT VALUE = VALUE ASSIGNED TO SYMBOL TABLE ENTRY.
   TOKEN - FIXED BINARY (31)
     ENTRY VALUE = LIST CLASS (IDENTIFIER,INTEGER CONSTANT,
                               OR STRING CONSTANT) FOR SEARCH
                 - LIST CLASS IS IDENTIFIER FOR INSERT
     EXIT VALUE = VALUE OF TYPE ATTRIBUTE
   NAME - CHARACTER (70) VARYING
     ENTRY VALUE = SEARCH KEY FOR SEARCH
                   VALUE OF NAME ATTRIBUTE FOR INSERT
     EXIT VALUE = NONE ASSUMED

  RISCOPE, RDSCOPE, AND RQSCOPE ENTRY:
   ACTIVE_LEVEL - FIXED BINARY (31)
     ENTRY VALUE = NONE ASSUMED
     EXIT VALUE = VALUE OF CURRENT LEVEL OF NESTING

  RS@ADDR, RS@TYPE, RS@ROW, AND RS@COL ENTRY:
    ADDRESS - FIXED BINARY (31)
      ENTRY VALUE = SYMBOL TABLE ADDRESS OF IDENTIFIER
      EXIT VALUE = ZERO IF ENTRY NOT FOUND IN ACTIVE SYMBOL
                   TABLE.
                   OTHERWISE SAME AS ENTRY VALUE.
    VALUE - FIXED BINARY (31)
      ENTRY VALUE = VALUE TO BE ASSIGNED SYMBOL TABLE
                    ATTRIBUTE.
      EXIT VALUE = ZERO IF ENTRY NOT FOUND IN ACTIVE SYMBOL
                   TABLE.
                   OTHERWISE SAME AS ENTRY VALUE.

  RG@ADDR, RG@TYPE, RG@LEVL, RG@ROW, RG@COL, RG@NAME ENTRY:
    ADDRESS - FIXED BINARY (31)
      ENTRY VALUE = SYMBOL TABLE ADDRESS OF IDENTIFIER
      EXIT VALUE = ZERO IF ENTRY NOT FOUND IN ACTIVE SYMBOL
                   TABLE.
                   OTHERWISE SAME AS ENTRY VALUE.
    VALUE - FIXED BINARY (31)
      ENTRY VALUE = NONE ASSUMED
      EXIT VALUE = ZERO IF ENTRY NOT FOUND IN SYMBOL TABLE
                   OTHERWISE VALUE OF ATIRIBUTE FETCHED

  RG@IVAL ENTRY:
    ADDRESS - FIXED BINARY (31)
      ENTRY VALUE = SYMBOL TABLE ADDRESS OF INTEGER CONSTANT.
      EXIT VALUE = ZERO IF ENTRY NOT FOUND IN SYMBOL TABLE
                   OTHERWISE SAME AS ENTRY VALUE.
    VALUE - FIXED BINARY (31)
      ENTRY VALUE = NONE ASSUMED
      EXIT VALUE = ZERO IF ENTRY NOT FOUND IN SYMBOL TABLE.
                   OTHERWISE VALUE OF INTEGER CONSTANT.
```

** UNIVERSAL FILE DESCRIPTIONS FOR REPLACEMENT MODE **

RSOURCE FILE RECORD ** (MODE 2 & 3) SOURCE INPUT TO REPLACEMENT MODULE **

RPRINT FILE PRINT ** SOURCE LISTING AND INTERMEDIATE CODE GENERATED BY THE REPLACEMENT MODULE **

RXXMSG FILE PRINT **TRACE, DEBUG, AND DIAGNOSTIC MESSAGES GENERATED BY THE REPLACEMENT MODULE **

REPLACEMENT MODE 1 - REPLACE SYMBOL TABLE.
ACTION:
INVOKE APPROPRIATE REPLACEMENT ROUTINE.
EVALUATE CONSISTENCY OF REPLACEMENT ROUTINE.

OVERVIEW OF EVALUATION FOR REPLACEMENT MODE 1.

ENTRY POINT	EVALUATION
RSEARCH	FOR PREVIOUSLY DEFINED ENTRIES COMPARE THE INITIAL ENTRY ADDRESS AND THE CURRENT ENTRY ADDRESS OBTAINED BY THE REPLACEMENT MODULE.
RINSERT	NO CONSISTENCY EVALUATION. ESTABLISH INITIAL ENTRY ADDRESS.
SCOPE QUERY	
RISCOPE	COMPARE THE ACTIVE LEVEL AS RECOGNIZED BY
RDSCOPE	ORACLE AND THE ACTIVE LEVEL AS RECOGNIZED BY
RQSCOPE	REPLACEMENT MODULE.
SET ATR QUERY	NO EVALUATION.
GET ATR QUERY	COMPARE THE VALUE OBTAINED FOR AN ATTRIBUTE BY ORACLE AND THE VALUE OBTAINED FOR THE SAME ATTRIBUTE BY THE REPLACEMENT MODULE.

NOTE: THESE TECHNIQUES ARE NOT CAPABLE OF EVALUATING THE ACTIONS OF A SYMBOL TABLE ROUTINE WHICH RE-LOCATES ENTRIES AFTER THE INITIAL ENTRY ADDRESS HAS BEEN ESTABLISHED.

REPLACEMENT MODE 2 - REPLACE LEXICAL ANALYSIS.
ACTION:
INVOKE REPLACE MODULE RSCAN.
EVALUATE ACTION OF REPLACEMENT MODULE BY COMPARING THE TOKEN VALUE RECOGNIZED ORACLE AND THE TOKEN VALUE RECOGNIZED BY THE REPLACEMENT MODULE.

REPLACEMENT MODE 2 - UPDATE SYMBOL TABLE .
ACTION:
FOR ALL NON-SEARCH ACTIVITIES, INVOKE APPROPRIATE REPLACEMENT ROUTINE.
NO EVALUATION OF CONSISTENCY PERFORMED.

REPLACEMENT MODE 3 - REPLACE SYNTAX ANALYSIS.
ACTION:
INVOKE REPLACEMENT MODULE.

EVALUATION OF SYNTAX ANALYSIS REPLACEMENT MODULE:
DUE TO THE NUMBER OF ACTIONS PERFORMED AND THE SEQUENCE IN WHICH THESE ACTIONS COULD OCCUR WITHIN VARYING IMPLEMENTATIONS OF A PARSER, NO VIABLE EVALUATION TECHNIQUE IS PROVIDED WITHIN THE SYNTAX ANALYSIS ROUTINE. HOWEVER, THE NON-REPLACABLE SEMANTIC ANALYSIS MODULE PROVIDED THE USER IS CAPABLE OF TRACING THE PRODUCTIONS APPLIED BY SYNTAX ANALYSIS REPLACMENT MCDULE. THIS FACILITY IS OFFERED TO PROVIDE A REASONABLE SET OF DATA FOR EVALUATING THE SYNTAX ANALYSIS REPLACEMENT MODULE.
ANALYSIS MODULE PROVIDED THE USER IS CAPABLE OF TRACING THE PRODUCTIONS APPLIED BY THE SYNTAX ANALYSIS REPLACEMENT MODULE. THIS FACILITY IS OFFERED TO PROVIDE A REASONABLE SET OF DATA FOR EVALUATING THE SYNTAX ANALYSIS REPLACEMENT MODULE.

REPLACEMANT MODE 3 - SEMENTIC ANALYSIS.
THE SEMANTIC ROUTINES ARE PROVIDED BY ORACLE AS A NON-REPLACABLE MODULE.

REPLACEMENT MODE 3 - CODE GENERATION.
THE CODE GENERATION ROUTINES ARE PROVIDE BY ORACLE AS A NON-REPLACABLE MODULE.

ADDITIONAL INFORMATION CONCERNING ORACLE MAY BE OBTAINED FROM THE AUTHOR OR FROM THE COMPUTER SCIENCE DEPARTMENT AT THE UNIVERSITY OF TESSESSEE.

APPENDIX B

MESSAGES AND CODES

***** ERROR MESSAGES GENERATED BY LEXICAL ANALYSIS *****

9101	LEXICAL	END OF TEXT
9102	LEXICAL	TOKEN RECOGNITION TERMINATED AT CARD BOUNDARY
9103	LEXICAL	UNRECOGNIZABLE SYMBOL ENCOUNTERED, TOKEN VALUE IS #UNKNOWN#
9104	LEXICAL	ILLEGAL LABEL DECLARATION, DECLARATION IGNORED
9105	LEXICAL	ILL FORMED COMPILER AIDS DIRECTIVE, REQUEST IGNORED
9106	LEXICAL	PROGRAM EXCEEDS 1000 SOURCE CARDS - END OF TEXT GENERATED
9107	LEXICAL	TOKEN VALUE IS INVALID - VALUE #UNKNOWN# SUBSTITUTED

***** ERROR MESSAGES GENERATED BY SYNTAX ANALYSIS *****

9201	SYNTAX	LOGICAL END OF PROGRAM ENCOUNTERED - STACK RE-ESTABLISHED
9202	SYNTAX	PROGRAM NAME GENERATED
9203	SYNTAX	PROGRAM #BEGIN# GENERATED
9204	SYNTAX	MISSING TOKEN GENERATED
9205	SYNTAX	REPEAT BLOCK TERMINATED
9206	SYNTAX	UNKNOWN TOKEN IGNORED
9207	SYNTAX	STATEMENT DELETED
9208	SYNTAX	STATEMENT DELETED - SYNTAX STACK FLUSHED
9209	SYNTAX	UNRECOVERABLE ERROR - UNEXPECTED END
9210	SYNTAX	UNRECOVERABLE ERROR - PARSE STACK ERROR
9211	SYNTAX	UNRECOVERABLE ERROR - PARSE STACK OVERFLOW
9212	SYNTAX	MULTIPLE DECLARATION IGNORED
9213	SYNTAX	STRING TYPE EXPECTED - STRING CONSTANT BLANK SUBSTITUTED
9214	SYNTAX	INTEGER TYPE EXPECTED - INTEGER CONSTANT 1 SUBSTITUTED
9215	SYNTAX	COLON GENERATED FOR LABEL CONSTANT
9216	SYNTAX	MULTIPLICATION OPERATOR MISSING - STATEMENT DELETED
9217	SYNTAX	ADDITION OPERATOR MISSING - STATEMENT DELETED
9218	SYNTAX	UNDECLARED IDENTIFIER - TYPE INTEGER ASSUMED
9219	SYNTAX	UNDECLARED IDENTIFIER - TYPE STRING ASSUMED
9220	SYNTAX	SEMANTIC ABNORMAL TERMINATION RECOGNIZED
9221	SYNTAX	UNRECOVERABLE ERROR - PARSE STACK UNDERFLOW
9222	SYNTAX	CODE GENERATION ABNORMAL TERMINATION RECOGNIZED

***** ERROR MESSAGES GENERATED BY SEMANTIC ANALYSIS ****

9301	SEMANTIC	SEMANTIC STACK OVERFLOW - SEMANTIC ANALYSIS TERMINATED
9302	SEMANTIC	STACK UNDERFLOW - LOGICAL END - SEMANTIC ANALYSIS TERMINATED
9303	SEMANTIC	CODE GENERATION OVERFLOW - PROGRAM TOO LARGE
9304	SEMANTIC	STORAGE ALLOCATION OVERFLOW - PROGRAM TOO LARGE
9305	SEMANTIC	UNDEFINED IDENTIFIER ENCOUNTERED - STOP CODE GENERATED
9306	SEMANTIC	CODE GENERATION ABNORMAL TERMINATION RECOGNIZED
9307	SEMANTIC	ILLEGAL TOKEN RECEIVED - SEMANTIC ACTION SUSPENDED
9308	SEMANTIC	ILLEGAL PRODUCTION RECEIVED - SEMANTIC ACTION SUSPENDED
9309	SEMANTIC	LOGICAL END RECOGNIZED - SEMANTIC ACTION TERMINATED
9310	SEMANTIC	ADDRESSING ERROR - SUBSCRIPT MISSING FOR ARRAY
9311	SEMANTIC	ADDRESSING ERROR - IDENTIFIER IS NOT 1-DIMENSIONAL
9312	SEMANTIC	ADDRESSING ERROR - IDENTIFIER IS NOT 2-DIMENSIONAL

***** ERROR MESSAGES GENERATED BY CODE GENERATION *****

9401	CODE	PROGRAM TOO LARGE - CODE STREAM EXCEEDS 16-K WORDS
9402	CODE	WARNING - MAXIMUM LEVEL EXCEEDED IN CODE GENERATION
9403	CODE	WARNING - LOGICAL END ENCOUNTERED IN CODE GENERATION
9404	CODE	LABEL REFERENCE UNRESOLVED WITHIN ACTIVE BLOCK
9405	CODE	REQUESTED STORAGE UNAVAILABLE - PROGRAM EXCEEDS 16-K WORDS

***** ERROR MESSAGES GENERATED BY SYMBOL TABLE MANAGEMENT *****

9501 TABLE SYMBOL TABLE OVERFLOW - IDENTIFIER LIST EXCEEDS 100 ENTRIES
9502 TABLE SYMBOL TABLE OVERFLOW - INTEGER LIST EXCEEDS 50 ENTRIES
9503 TABLE SYMBOL TABLE OVERFLOW - STRING LIST EXCEEDS 50 ENTRIES
9504 TABLE SYMBOL TABLE OVERFLOW - ACTIVE LEVELS EXCEEDS 20
9505 TABLE IDENTIFIER TRUNCATED TO 8 CHARACTERS
9506 TABLE INTEGER TOO LARGE - VALUE 1 SUBSTITUTED
9507 TABLE STRING CONSTANT TRUNCATED TO 70 CHARACTERS
9508 TABLE INVALID SEARCH REQUEST BY LEXICAL ANALYSIS - REQUEST IGNORED
9509 TABLE SYMBOL TABLE UNDERFLOW - REQUESTED BLOCK EXIT IGNORED
9510 TABLE ATTRIBUTE NOT ACCESIBLE AT REQUESTED ADDRESS - REQUEST IGNORED
9511 TABLE ADDRESS NOT ACCESSIBLE - REQUEST IGNORED

VITA

William Roy Haynes, Jr., was born in Birmingham, Alabama, on July 9, 1947. He graduated from Lafayette, Louisiana, High School in June 1965 and began college at Southwestern Louisiana. Later he transferred to the University of Tennessee, Knoxville, where his college education was interrupted briefly by a tour of duty in the U. S. Navy. In June 1971 he returned to the University of Tennessee and received a B. A. degree in Mathematics in August 1972.

He entered the Graduate School at the University of Tennessee in March 1973 and received the Master of Science degree with a major in Computer Science in August 1976. His most recent employment is with the Tennessee Valley Authority, Knoxville, Tennessee.