A STATEMENTS EVALUATION SYSTEM FOR FUNCTIONALLY EQUIVALENT RESPONSES

By

PETER YU YEE TSANG

Bachelor of Science

University of Wisconsin - Madison

Madison, Wisconsin

1984

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1987

A STATEMENTS EVALUATION SYSTEM FOR
FUNCTIONALLY EQUIVALENT RESPONSES

Thesis Approved:

Donald D Fisher
Thesis Adviser

Donald W. Grace

[illegible]

Norman N. Durham
Dean of the Graduate College

1275752

PREFACE

This study is concerned with the development of the statements evaluation system to evaluate responses to (rather special) questions associated with writing program segments.

The earliest motivation for this study was provided by Dr. Donald D. Fisher, who also is my major advisor. I would like to express my gratitude to him for his guidance and invaluable help during this study. I would also like to thank Dr. K. M. George for his advice and help, and Dr. Donald W. Grace for serving on my graduate committee. Finallly, I express my gratitude to my family, for their support, patience, and encouragement.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
II. LITERATURE REVIEW	7
CAI Overview	7
The Requirement	7
History	8
State of the Art Assessments	9
Parsing	11
Translation	12
III. FORMAL LANGUAGE THEORY	14
Formal Grammar	14
The Need	14
Different Classes of Grammars	15
Recognizers	18
Introduction	18
Different Classes of Recognizers	19
IV. AN OVERVIEW OF THE STATEMENTS EVALUATION SYSTEM	22
V. DESIGN AND IMPLEMENTATION	28
Lexical Analyzer	28
The Role of Lexical Analyzer	28
The Need for a Lexical Analyzer	30
Regular Grammar	30
Finite State Machine	31
Implementation	36
Parser	45
The Role of the Parser	45
Basic Design for Recursive-Descent Parser	46
Parsing Procedures for the Mini-Language	50
Translator	74
The Role of the Translator	74
Semantic Actions	75
Implementation of Systax-Directed Translator	75
Translation Scheme for Declaration	76

Chapter Page

Translation Scheme for Assignment Statement 80
Translation Scheme for Multiple Statements 91
Translation Scheme for Conditional Statement 93

VI. EXAMPLES OF DIFFERENT STATEMENTS AND RESPONSES 96

Introduction 96
Declarations 97
Assignment Statements 98
Addition 98
Subtraction 99
Multiplication 101
Compound Statement 102
If-then-else Statements 106

VII. SUMMARY, FUTURE STUDY AND DEVELOPMENT 108

Summary 108
Future Study 108

BIBLIOGRAPHY 110

APPENDIX A 113

APPENDIX B 115

APPENDIX C 116

APPENDIX D 122

APPENDIX E 126

LIST OF FIGURES

Figure | Page

1. Process of Translation 12
2. Basic Turing Machine 19
3. Stack 20
4. Finite State Machine 21
5. Structure of Statements Evaluation System . . . 24
6. Translation of "declare x,y; x := y + z*w" . . 25
7. Translation of "declare y; declare x; x := w*z + y" 26
8. General Description of Lexical Analyzer 28
9. A Tape Reading Description of Finite State Machine 32
10. A Finite State Diagram for Identifier 33
11. A Finite State Diagram for Integer number . . . 33
12. State Transition Diagram for the Mini-language. 35
13. Structure of the Lexical Analyzer 36
14. Procedure Read_input 37
15. Procedure Scanner 38
16. Internal Representation of Token 41
17. Procedure Get_token 42
18. Procedure Check_reserve 43
19. The Role of Parser in the Evaluation System . . 45
20. Procedure Match for Matching Input Token . . . 47
21. Function Next that Returns the Lookahead Token. 48

Figure Page

22. Parsing Procedure A 49

23. Hierarchial Structure of the Parser 51

24. Parsing Procedure for <start> 53

25. Parsing Procedure for <dec_seq> 55

26. Parsing Procedure for <declaration> 56

27. Parsing Procedure for <dec_tail> 56

28. Parsing Procedure for <id_list> 57

29. Parsing Procedure for <id_list_tail> 58

30. Parsing Procedure for <stmt_seq> 59

31. Parsing Procedure for <statement> 60

32. Parsing Procedure for <stmt_tail> 61

33. Parsing Procedure for <assgn_stmt> 61

34. Parsing Procedure for <if_stmt> 62

35. Parsing Procedure for <endif_else> 63

36. Parsing Procedure for <loop_stmt> 64

37. Parsing Procedures for <comparsion>,
<comp_tail> 65

38. Parse Trees Generated by Ambiguous Grammar . . 66

39. Procedures for Parsing an Expression 67

40. Structure of Statements Evaluation System . . . 74

41. Declaration Table 77

42. Parsing Procedure id_list 77

43. Insert Procedure 78

44. Declaration Table for "declare u,w;" 79

45. Declaration Table for "declare w; declare u;" . 80

46. Table Representation of x := a*3 + b + c . . . 82

Figure Page

47. Table Representation of x := 3*a + c + b . . . 83

48. Table Representation of x := a + b - c 84

49. Addition of Expression Tables 85

50. Multiplication of Expression Tables 85

51. Addition of Expression Tables [a*u+b+c], [d-e*w] 86

52. Multiplying the Expression Tables of [a+b-c], [u-w] 87

53. XOR Table 87

54. Generation of Expression Tables 89

55. Assignment Table for x := w; x := a; y := x +b. 92

56. If-block and Else-block 94

57. Cross Comparisons Between Complements 95

CHAPTER I

INTRODUCTION

This thesis project investigated the possibility of using a computer to evaluate responses to (rather special) questions associated with writing program segments. If a general statements evaluation system (SES) could be developed, then a computer could be used to conduct computer based learning exercise at a much higher intellectual level than is currently possible. A problem statement might be

"Construct a program segment to compute the payment of mortgage loan"

The loan payment could be computed in several different formats and still be correct; furthermore arbitrary intermediate substitutions, if correct, should be allowed and evaluated as correct by such SES. The idea is to supply the SES with a template, regarded correct, to be used to determine whether the user's response is functionally equivalent to the template. It is the ability to determine functionally equivalence or non-equivalence that provides the increased capability of this system over systems which can only determine whether a single response is an exact match of a given answer. The implemented model could be developed into a computer assisted instruction (CAI) system, which is a useful tool in helping students in programming, debugging, and for retraining technical people in industry. The CAI system is in tutorial format, asking students to compose statements for particular tasks, beginning with the simple single statement and proceeding to compound multiple statements. The CAI system generates responses to student's input, replies include error messages and the correct answer to the problem.

The statements evaluation system (SES) is implemented to respond to the mini-language by Ledgard and Marcotty (1981), and it can be modified to adopt an appropriate subset of any other similar high level language like C (Kernighan and Ritchie 1978), Pascal (Jensen and Wirth 1975) or Fortran (1966). In the mini-language there are basically four types of statements:

(1) declaration statement;

(2) assignment statement;

(3) conditional statement (if then else);

(4) loop statement (while loop).

The system evaluates on declaration statement, assignment statement and one level conditional statement. Because of different complexity and structure of each type of statement, each has a separate evaluation method. For example, some of the many possible ways to declare variables x, y and z are

(1) declare x,y,z;

(2) declare y,x,z;

```
(3) declare x,y;
    declare z;
(4) declare z,y;
    declare x;
(5) declare x;
    declare y;
    declare z;
```

Above are not all the variations, there are a total of 18 different formats just to declare the three variables x, y and z. The template answer provided to the system is one of the 18 formats, and the system must be able to recognize the other 17 formats are functionally equivalent to the template answer. There is only one variable type (integer) in the Ledgard mini-language, therefore the system does not do any type checking on the variable types.

The different priorities of operators (+, -, *), levels of parenthesis and substitution of variables increase the difficulties and complexities of evaluation of assignment statement. To be able to determine the template's assignment statements and the input's assignment statements are functionally equivalent, the system translated all assignment statements into standard format with all the parenthesis removed and all variables are substituted with their latest assigned value. For example the statement

```
x := 8;
y := x *( 2 + 3) - (8 + 9);
```

is translated into

```
x := 8;

y := 8 *2 + 8 * 3 - 8 - 9;
```

The system uses tables to represent assignment statements, which is more easy to implement the translation and keep track of the recent assigned value of every variable. Chapter five has a detailed description of the method and implementation of the table translation.

Because of its various formats and complexities of expression in assignment statement, therefore, this study emphasizes on the evaluation of the sequential assignment statements. Below is an example which shows that a simple assignment statement can be transformed into different formats with different complexities, which complicate the evaluation process.

```
(1) x := a - b*e + c*e + d*e;

(2) x := a - b*e + e*(c + d);

(3) x := a -(b - c -d) * e;

(4) x := a -(b - ( c + d) ) *e;

(5) u := c + d;

    t := (b - u) * e;

    x := a - t;

(6) u := c* ( e + d -b);

    x := u + a;
```

Above are only some of the possible formats, the variations are almost unlimited by using parenthesis and substitution with multiple assignment statements.

The conditional and loop statements are the most unpredictable, especially with the nested if-then-else and while loop statements. To restrict the problem, this study concentrates on one format of if-then-else statement and its variations, which is a one level if-then-else with the condition in this format,

(variable conditional operator variable)

Below is an example of if-then-else statement, and its variations.

```
(1) if (a >b) then
        a := a - b - c;
    else
        a := a + b;
    end if;
(2) if (a > b) then
        u := b  + c;
        a := a - u;
    else
        a := a + b;
    end if;


(3) if (a < b) then
        a := a + b;
    else
        a := a - b - c;
    end if;
```

```
(4) if (a < b) then
        a := a + b;
    else
        a := a - ( b + c);
    end if;
```

Chapter II is a discussion on computer assisted instruction (CAI), it's history and development. The evolution of parsing and translation are also given in this chapter. Chapter III gives an introduction of formal language theory. Chapter IV gives an overview of the design of the system. The structure of the system (lexical analyzer, parser and translator), the implementation methods, and program codes are discussed in Chapter V. Examples of different statements and responses are given in Chapter VI. Chapter VII is the summary of this project, and future study and development are suggested.

CHAPTER II

LITERATURE REVIEW

CAI Overview

The Requirement

There are three basic educational requirements that make CAI inevitable (Loughary 1967):

(1) the trend to individualized instruction;

(2) the growth in information to be acquired;

(3) the shortage of qualified teachers.

Since 1950's, computer assisted instruction (CAI) has been developed and applied to these three problems in education from elementary school to professional training (Suppes, 1978). In training environments such as industry and the military, students are also paid. For this reason, in training environments the relationship between time and costs is a direct one --- costs can be reduced to the extent that reductions in instructional time can be achieved. A major advantage of CAI systems is that they can reduce instruction time while maintaining equivalent levels of performance when compared to the traditional type of lecture - discussion techniques.

History

The first use of computers for educational purpose was started at the end of the 1950's. One such research application was the PLATO project at the University of Illinois (Alpert and Bitzer, 1970), which began in 1960 with the goal of designing a large computer-based system for instruction. Soon after, IBM introduced COURSEWRITER, a programming language designed for preparing instructional materials on IBM's mainframe computer. At Stanford University and Pennsylvania State University, there were projects by Atkinson and Hansen (1966),Suppes, Jerman and Brian (1968), and Suppes and Morningstar (1972).

In the early 1970's the PLATO project introduced PLATO IV, a large time-shared instructional system. Students studied on individual terminals, hundreds of which were connected to a large computer on which all lessons and student data were stored. PLATO IV now allows up to 600 students to use the computer simultaneously.

In the mid-1970's, a few small companies began to experiment with microcomputers, including Radio Shack, Commodore Business Machines, and the Apple computer. With the success of microcomputers, it became possible for the individual university researcher, and public schools to possess a microcomputer and use it for educational purposes. From 1977 to today we have seen phenomenal growth in the educational uses of computers, and computer instructional system became affordable to public school or family.

State of the Art Assessments

The state of the art assessments are an idealized computer assisted instructional system, including hardware-software, courseware, learning strategies, management and development.

Baker (1971) provides the background of idealized CAI systems. A system is documented in the form of a systems concept document. The document has three main goals:

(1) provide a conceptual frame work for the CAI system;

(2) serve as the guidance document for the design and implementation of the CAI system;

(3) act as a baseline document for evaluation purpose.

Bushnell (1964) describes, briefly, developments in computer based teaching machines and rapid information retrieval systems, and the advances in computer technology for aiding teachers in the diagnosis of student learning needs and selection of appropriate teaching strategies. The most common teaching strategies used in courseware are:

(1) drill and practice;

(2) tutorial instruction;

(3) simulation;

(4) games.

We all are familiar with drill and practice in one form or another: work-books, flash cards, spelling bees. In a drill-and-practice system, a selection of questions or problems is presented repeatedly until the student answers or solves them all at some predetermined level of proficiency. Computer programs can enhance the effectiveness and efficiency of drill-and-practice. One of the latest drill-and-practice programming tool is Drillshell (Alessi, S. M. and Schwaegher, D. G. 1984) which allows CAI developers to produce drills without programming all the details of queuing and data storage.

Tutorial instructions are computer programs that teach by carrying on a dialogue with the student. They present information ask the student questions and make decisions based on the student's comprehension whether to move on to the next instruction or to engage in review and remediation. Tutorial instruction is the most basic and common form of CAI. The SOPHIE system developed by Brown (1975) is an example of a CAI tutorial program.

Simulation systems provide the student with the illusion of experiencing a real life occurrence. They have the advantages of convenience, safety, and controllability over real experiments, and are useful for giving students experiences that would not otherwise be possible. AIR SIM is an air flow simulation program by Fortner (1979). Lagowski (1970) and Gelder (n.d.) also have written several good examples of laboratory simulation programs which are very helpful for chemistry students to experience a dangerous experiment in a simulated environment.

Parsing

The two most common forms of parsers are bottom-up, and top-down. Floyd (1963) was the first one to come up with the operator-precedence idea and the use of precedence functions. Since then there have been a variety of other bottom-up parsing strategies developed, such as the Wirth-Weber precedence by Wirth and Weber (1966), bounded-context parsing (Floyd 1964 and Graham 1964), LL parsers as defined by Lewis and Stearns (1968), and the LR parsers by Knuth (1965).

Bottom-up parsing traverses the tree from the leaves (bottom) to the root (top). Top-down parsing does the reverse, i.e., it starts from the root of the parse tree and works its way down to the leaves. There are basically two types of top-down parser, one involves backtracking and the other does not (recursive descent parsing). META (Schorre 1964) and TMG (McClure 1965) are some of the compiler writing systems which used top-down parsing with backtracking. The parser of the statements evaluation system (SES) in this project is implemented in recursive descent parsing. Conway (1963) and Lucas (1961) were the ones who introduced this recursive descent parsing technique. In Chapter 3, there is a basic background of formal language theory which is essential for defining the grammar of the programming languages. Chapter 4 has the detailed description of recursive descent prasing and an implementation of the parser for mini-language (Ledgard and Marcotty, 1981) is given.

Translation

Syntax directed translation was first used by Irons (1961) as a method in compiler design. Aho and Ullman (1977) gave a basic diagram for syntax directed translations in their book, to explain the process of the translation.

$$\begin{tikzcd}
\text{input string} \arrow[r] & \text{parse tree} \arrow[r] & \text{dependency graph} \arrow[r] & \text{evaluation for semantic rules}
\end{tikzcd}$$

Figure 1. Process of Translation

A parse tree is generated during the parsing process of the input string, and it is traversed to generate the semantic actions during the traslation process. The semantic actions may be the computations of values of variables, generation of intermediate codes, printing messages or storing some values into a particular table for future reference.

The idea of a parser calling for semantic actions was first discussed by Samelson and Bauer (1960), and later by Brooker and Morris (1962). In the mid 60's, Eickel, Paul, Bauer and Samelson (1963), Cheatham and Sattley (1964), Ingerman(1966) and Feldman (1966) contributed a great amount of work to syntax-directed translations, which led to the development of the early theory of syntax-directed translation by Lewis and Stearns (1968). A more detailed description of syntax directed translation is given in Chapter 5.

CHAPTER III

FORMAL LANGUAGE THEORY

Formal Grammar

The Need

When we say about grammar, we all know English grammar. An English grammar is a set of rules either for constructing English sentences or for determining whether an English sentence is syntactically correct. Thus the sentence " I am working very hard." obeys and follows the grammatical rules, whereas the sentence " I working am hard very ." fails miserably. The grammar is concerned with the form of the sentence but not the meaning, therefore the meaningless sentence like "Books are working very hard." is quite acceptable grammatically. The grammar of a programming language is very similar to the grammar of spoken language, but more constricted. It either provides a set of rules for writing a program in that programming language or it determines whether a program is syntactically correct (but not necessarily meaningful). A program can be syntactically correct with no error but does not do anything meaningful at all. Grammars for programming languages are exact and precise, and they can be described in a formal mathematical notation, i.e., a formal grammar.

Different Classes of Grammars

A phrase-structure grammar (PSG) is an ordered quadruple,

$G = (N, \Sigma, P, S)$ where

(1) N is a finite set of nonterminal symbols (sometimes called variables or syntactic categories);

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

(3) P is a finite subset of
$(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
where an element (a,b) in P is written a ---> b and is called a production;

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

Below are some examples of PSG's and non-PSG's :

(1) G =({S,A}, {0,1}, P, S) where P consists of

```
S   ----> 0A1
0A  ----> 00A1
0A1 ----> 01
```

(2) G =({A,B}, {0,1}, P, S) where P consists of

```
S     ----> 0A1
01    ----> 00A1
1     ----> ABC
```

Example (1) is a PSG, and it generates strings of the form 01, 0011, 000111, ... and so on indefinitely. Example (2) is not a PSG, for it violates rules 3 and 4 in the grammar. The terminal strings 01 and 1 are not in the set $(N \cup \Sigma)^* N (N \cup \Sigma)^*$, and S (starting symbol) is not an element in N. Example (1) is an unrestricted grammar which means the productions of the grammar with the form A ----> B, where A and B are in $(N \cup \Sigma)^*$ are allowed.

The definition of phrase-structure grammars describes much too large a class of grammars to deal with in the process of translation and evaluation. However, it is possible to add some more restrictions to form a restrictive grammar, which is less flexible but easier to translate because of the restricted properties of the grammar. The restrictions are often placed on the format of the productions. A context-free grammar is a restrictive type grammar.

A grammar G = (N, Σ, P, S) is a context-free grammar (CFG) if and only if it is a PSG and the roots of all productions in P are single nonterminal symbols. Single productions with this property are referred to as context-free productions. Below is an example of context-free grammar :

G = ({E}, {+,*,(,),id}, P, E) where P consists of

E ----> E + E

E ----> E * E

E ----> (E)

E ----> id

This context-free grammar defines the arithmetic expressions with operators "+" and "*" and operands represented by symbol id. Here E is the only variable which represents expression, and the terminals are "+", "*", "(", ")" and id. The first two productions say that an expression can be composed of two expressions connected by addition or multiplication sign. The third production says that an expression may be another expression surrounded by parenthesis. The last says a single operand is an expression. By applying productions repeatedly we can obtain more and more complicated expressions. For example,

```
E ----> E * E                        (2)
  ----> E * (E)                      (3)
  ----> E * (E + E)                  (1)
  ----> (E) * (E + E)                (3)
  ----> (E + E) * (E + E)            (1)
  ----> (id + E) * (E + E)           (4)
  ----> (id + id) * (E + E)          (4)
  ----> (id + id) * (id + E)         (4)
  ----> (id + id) * (id + id)        (4)
```

The symbol "---->" denotes the act of deriving, that is, replacing a variable by the right-hand side of a production for that variable. The numbers appearing on the right-hand side of the derivations are the production numbers used by the derivations.

A grammar $G = (N, \Sigma, P, S)$ is a context-sensitive grammar (CSG) if and only if it is a PSG and each production in P is of the following form

(1) a ----> b, where a and b are in $(N \cup \Sigma)^*$ and the length of a is less than or equal to the length of b ($|a| <= |b|$).

(2) S ----> e, where S is the start symbol and e is the empty string.

The following is an example of context-sensitive grammar,

$G = (\{S,A\}, \{0,1\}, P, S)$ where P consists of

S ----> A

S ----> 0A1

0A1 ----> 00A11

This grammar generates strings of form 01, 0011, 000111 as the PSG's example before. The PFG, CFG, and CSG are some of the most common formal grammars which are discussed in formal language theory. There are also some other types of restricted grammars with more restricted rules like the Chomsky normal form and Greibach normal form, but they will not be discussed in this study.

Recognizers

Introduction

The other way to specify a language is in a recognitive manner, that means defining a tool to recognize it. We define a recognizer which accepts all the possible output strings of the language.

Different Classes of Recognizers

A turing machine is the most general class of recognizer. It recognizes the class of languages definable by an unrestricted grammar. The basic model of a turing machine, illustrated in Fig. 2, has a finite control, an input tape that is divided into cells, and a tape head that scans one cell of the tape at a time.

```
        Input tape
  ------------------------------
 |a1|a2|a3| ...            |an|
  ------------------------------
        /\
       |  |
  ------------
 | Finite     |
 | control    |
  ------------
```

Figure 2. Basic Turing Machine

The input tape has a leftmost cell but is infinite to the right. Each cell may hold exactly one of a finite number of tape symbols (tokens). The current symbol is scanned by the tape head to determine what to do next, i.e., whether to change state or to reposition the tape head. The tape head can be repositioned to the left or right, one cell at a time.

A pushdown automaton is a recognizer with a read-only input tape, a finite state control, and a push-down stack or "first in - last out" list. That is, symbols may be entered or removed at the top of the list. Fig. 3 is an example of the stack, the number "1" is the first one input into the stack and then "2", "3" and "4", but the number "1" will be the last one to get out from the stack. A nondeterministic pushdown automaton recognizes the class of context-free languages.

```
5 ---
    :
   \ /

  : 4  :         : 5  :
  : 3  :         : 4  :
  : 2  :         : 3  :
  : 1  :         : 2  :
  :    :         : 1  :
  :    :         :    :
  :    :         :    :
  :    :         :    :
   ----           ----
```

Figure 3. Stack

A pushdown automaton uses the current input symbol on the tape, the contents of the top element of the stack, and the current state of the finite state control to determine an appropriate move. A language is said to be accepted by the pushdown automaton when some input symbol causes the push down automaton to enter a final state or when the pushdown automaton has emptied its stack after some sequence of moves.

The last recognizer to be discussed is the finite state automaton. It is equivalent to the pushdown automaton without the pushdown stack. For determining the next move, it uses only the current input symbol and the current state of the finite state control. A finite state machine is always described by the transition diagram. Fig. 4 is a transition diagram of a finite state machine, which accepts all the strings beginning with one or more a's and ending with one or more b's.

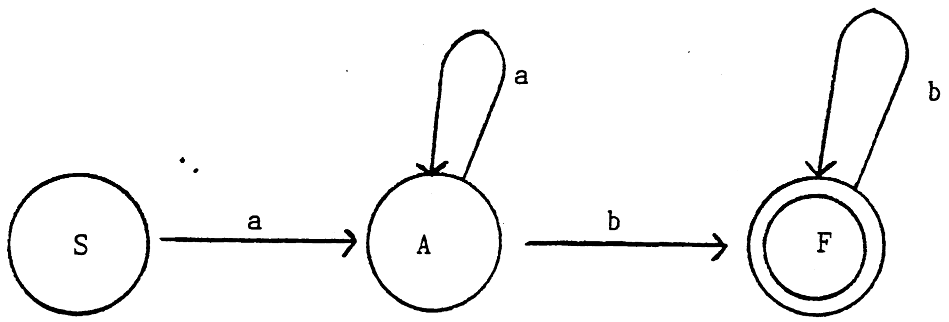

Figure 4. Finite State Machine

In Fig. 4, S is the starting state, and F is the final state . Each label arc defines a transition between the states caused by the symbol shown on the arc.

CHAPTER IV

AN OVERVIEW OF THE STATEMENTS EVALUATION SYSTEM

The purpose of the statements evaluation system (SES) is to evaluate the syntactic and semantic correctness of user's input program segments by comparing them with the template answer provided to the system. The program segment can include a combination of declaration statements, assignement statements and if-then-else statements. The system responses include a lexical analysis report, error messages and the correct answer to the problem. For example, with the template answer,

```
template :   declare x,y;
             x := y + z;
             if (a > b) then
                  a := c * ( b + 1);
                 else
                  a := c * ( b - 1);
             end if;
```

the system is able to determine that these two program segments,

```
        1.   declare x;
             declare y;
```

```
x := z + y;
if (a > b) then
    a := (b + 1) * c;
  else
    a := b*c - 1*c;
end if;
```

```
2.  declare y,x;
    x := y + z;
    if (a <=b) then
        a := c * (b - 1);
      else
        a := (b + 1) * c;
    end if;
```

are equivalent to the template answer. On the other hand, the system recognizes that the program segment,

```
3.  declare x,y.
    x := y + z;
    if (a > b) then
        a := c * ( b + 1);
      else
        a := c * ( b - 1);
    end if;
```

is not equivalent to the template answer, so error messages

```
output :    declare x, z. ** syntax error **
            syntax error "," or ";" expected
```

are printed as a response to the incorrect input.

Program segment number 4 has no syntactical error, but it is not performing the same function as the template answer, therefore, the system responds with an error message and the correct answer.

```
4.   declare x,y;
     x := y + z;
     if (a > b) then
          a := c * ( b - 1);
        else
          a := c * ( b + 1);
     end if;
output :  incorrect if-then-else statement
```

The SES bascially has 3 phases, namely, the lexical analyzer, the parser, and the translator (see Fig. 5).

```
                                                tables of
                                                declarations
input text    stream                            assignments
              of tokens      parse tree         expressions
                                                for evaluation
     --------        -------          ----------
--->|Lexical |---->|Parser |------->|Translator|------->
    |Analyzer|     |       |        |          |
     --------        -------          ----------
```

Figure 5. Structure of Statements Evaluation System

The lexical analyzer divides the input text into separate tokens (variables, keywords, labels, constants and operators). The purpose of the parser is to performs syntactic checking on the input token stream which is generated by the lexical analyzer. The translator translates the statements into standard table formats, so they can be easily evaluated. Consider the statements

```
1. declare x,y;
   x := y + z * w;

2. declare y;
   declare x;
   x := w * z + y;
```

The translator translates these statements into 3 different kinds of tables, namely, declaration table, assignment table, and expression table (see Fig. 6 and Fig. 7)

```
 Declaratin table
   ___________
  | x |integer|
  |-----------|
  | y |integer|
  |___________|

 Assignment table      Expression table

   -----------
  | x         |------> ---------------
  |-----------|       | y | z|     +/*|
  |           |       |---------------|
   -----------        |    | w|       |
                      |---------------|
                      |    |  |       |
                       ---------------
                        0   0          sign bit
```

Figure 6. Translation of "declare x,y; x := y+z*w"

```
Declaration table

 ____________
¦ y ¦integer¦
¦-----------¦
¦ x ¦integer¦
¦___________¦


Assignment table      Expression table

 -----------
¦ x         ¦------>  --------------
¦-----------¦        ¦ w ¦ y¦     +/*¦
¦           ¦        ¦---------------¦
 -----------         ¦ z ¦  ¦        ¦
                     ¦---------------¦
                     ¦   ¦  ¦        ¦
                      --------------
                       0   0        sign bit
```

Figure 7. Translation of "declare y;declare x; x := w*z + y"

The first row of the expression table represents addition and each column represents multiplication . The last row is the sign bit for each column, it is set to 0 if the column is positive and set to 1 if the column is negative.

How do the tables help in the evaluation? Two declaration tables are equivalent if they have the same variables in the table regardless of their order. Two expression tables are equivalent if they have the same elements regardless of the order of the columns and the order of the rows of each individual column. Therfore, we can determine that the tables in Fig. 6 and Fig. 7 are equivalent. That means the program segment "declare x,y; x := y + z*w;" is equivalent to program segment "declare y; declare x; x := w*z + y;".

This chapter only gives a brief description of the design of the statements evaluation system. A detail description of the design and implementation is in the follwing chapter.

CHAPTER V

DESIGN AND IMPLEMENTATION

Lexical Analyzer

The Role of Lexical Analyzer

The purpose of the lexical analyzer is to read the input, character by character, and to group individual characters into tokens (variable names, keywords, labels, constants, and operators).

```
                 -------------
input ------>| Lexical     |--------> stream of
stream       | Analyzer    |          tokens
                 -------------
```

Figure 8. General Description of Lexical Analyzer

To be able to return a token, the lexical analyzer must isolate the next sequence of characters in the input stream which designate a valid token. The lexical analyzer must be able to ignore blanks, and it is responsible for differentiating between different terminal symbols in a grammar. Appendix B contains a table of all the terminal and non-terminal symbols of the context-free grammar for the Ledgard mini-language. Each terminal and non-terminal has its own symbol number which is an internal representation number for that symbol. The lexical analyzer produces a token and the number associated with each token, Each (token,number) tuple is fed to the parser for syntactic analysis.

For example, with the input statements

```
a := b + c ;
if a > b then a := c; end if
```

the lexical analyzer returns the following items :

Token	symbol #	description
a	2	identifier
:=	21	assignment operator
b	2	identifier
+	15	addition operator
c	2	identifier
;	6	semicolon
if	3	reserved word
a	2	identifier
>	14	greater than
b	2	identifier
then	4	reserved word
a	2	identifier
:=	21	assignment operator
c	2	identifier
;	6	semicolon
end	7	reserved word
if	3	reserved word

The Need for a Lexical Analyzer

The statements evaluation system (SES) has 3 phases to analyze the source text, namely, the lexical analyzer, the parser, and the tanslator. The lexical analyzer performs lexical analysis and the parser performs syntactic analysis. By separating the lexical and the syntactic analysis processes, the system is easier to implement and we can construct a more specialized and effecienct recognizer for tokens. Furthermore, this separation simplifies the design of the parser.

Regular Grammar

As described in the previous section, the main purpose of the lexical analyzer is to return the next input token to the parser. To be able to return a token, the lexical analyzer must be able to isolate the next sequence of characters in the source text which designates a valid token. To do this, the lexical analyzer must recognize every valid token, while ignoring "noise" symbol strings such as comments, blanks, line boundaries, and whatever else is not important to the parsing process.

Tokens can be described in several ways. One way of describing tokens is by using a regular grammar. Using this method of specificiation, generative rules are given for producing the desired tokens. For example, the regular grammar,

```
<identifier> ---> a|b|c|....|z|0|1|....|9|
                  a<identifier>|b<identifier>|....
                  ....z<identifier>
```

contains the rules for generating the set of identifiers in the mini-Language.

The other way to describe tokens is in recognitive manner. Describing tokens by means of how they can be recognized (or accepted) is often done in terms of a mathematical model called a finite state machine (or finite automaton).

Finite State Machine

The output of the lexical analyzer is a function of the input, and there are only a finite number of actions which the lexical analyzer can take for any input. Thus, the lexical analyzer can be discribed by a finite state machine. A finite state machine can be thought of as a machine consisting of a read head and a finite state control box. The machine reads a tape one character at a time (from left to right), as shown in Fig. 9. At any instant a FSM can be in only one of a finite number of different states. A change in state occurs in the machine whenever the next character is read. Whenever an FSM begins reading a tape, it is always in a certain state designated as the starting state. Another type of state is a final state, and if the FSM attempts to read beyond the end of the tape while in a final state, the string which was on the tape is said to be accepted by the

FSM. In other words, the string belongs to the language which is accepted by the FSM.

```
tape    ________________________________
        ¦i¦d¦e¦n¦t¦i¦f¦i¦e¦r¦ ¦ ¦ ¦ ¦ ¦
        --------------------------------
              /\   Read
             ¦  ¦  Head
         -------------
        ¦Finite State ¦
        ¦Control Box  ¦
        ---------------
```

Figure 9. A Tape Reading Description of Finite State Machine

Finite state diagrams or transition diagrams are often used to represent an FSM pictorially. An example of such is illustrated in Fig. 10. The FSM represented in the diagram accepts identifier in the mini-language. The first character in the identifier must be a letter and follow by letters or digits. The nodes of the finite state diagram represent the states of the FSM, and in Fig. 10, the states are named S (starting state) and A (final state). The arcs leading from one state to another indicate the state transitions, with the characters immediately above or beside the arcs denoting the input characters which cause this state transition. The arrow and the word "START" signify which state of the FSM is the starting state. In Fig. 10, the starting state is S. The nodes that consist of a pair of concentric circles are final states. In Fig. 10, only state A is a final state. Fig. 11 is a transition diagram for an integer number.

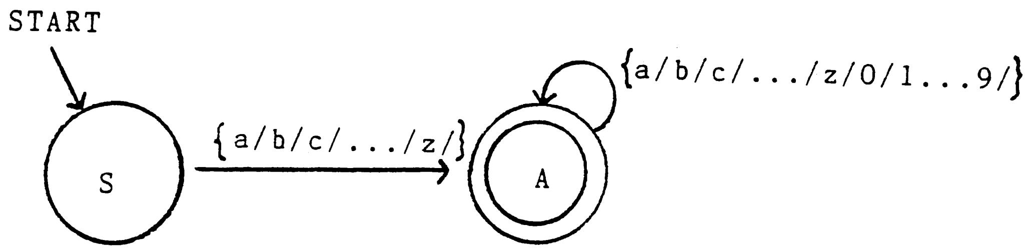

Figure 10. A Finite State Diagram for Identifier

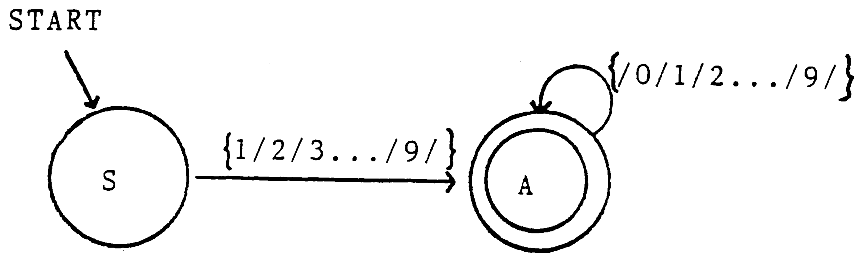

Figure 11. A Finite State Diagram for Integer Number.

The operation of the lexical analyzer for the mini-language is shown in the state transition diagram in Fig. 12. The arcs of the diagram are labelled with the input symbol which causes the transition. If the input symbol is such that it corresponds to no arc leaving the state, the symbol is invalid and the scanner prints an error message. The actions are labelled on the arc when a transition is made. The action

```
RETURN(token,symbol#)
```

signifies that token with corresponding symbol number should be returned to the parser as the input token. With the finite-state machine description of the lexical analyzer, a procedure can be implemented which emulates the actions of the state diagram in Fig. 12.

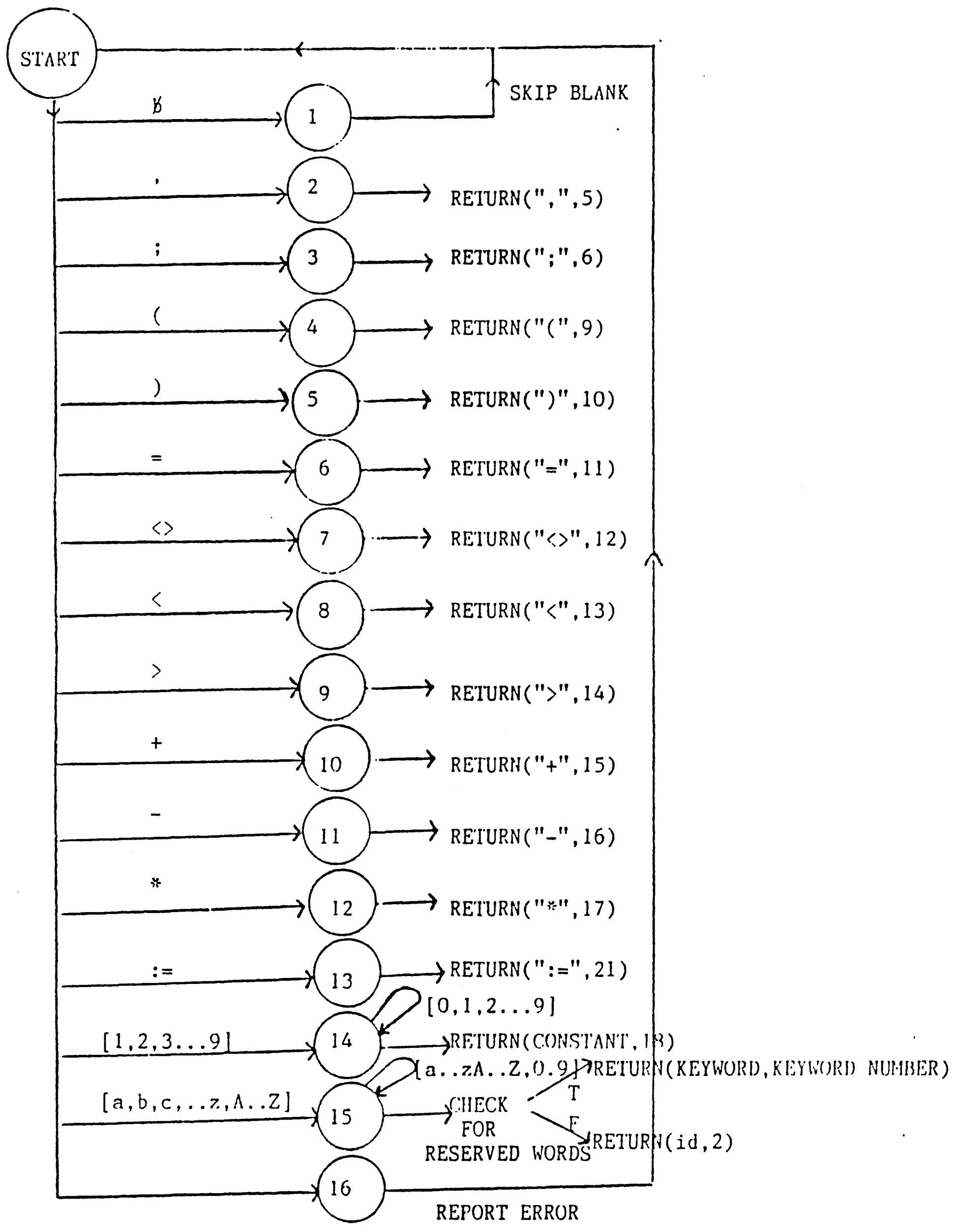

Figure 12. State Transition Diagram for the Mini-Language

Implementation

The algorithm for the lexical analyzer can be described in a top-down manner, with five different routines performing individual functions. Fig. 13 illustrates the structure of the lexical analyzer.

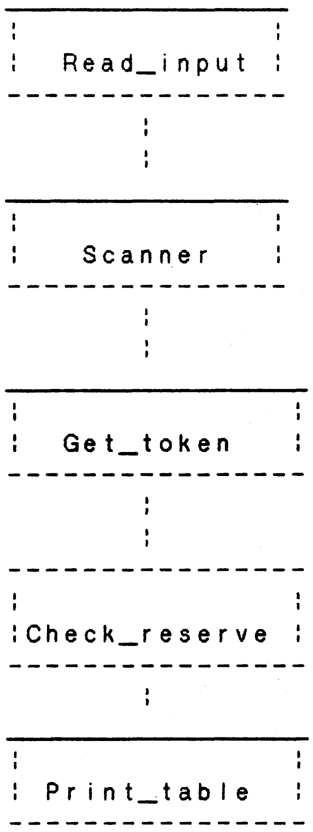

Figure 13. Structure of the Lexical Analyzer

The Read_input routine is used to read the source text, return characters, and store them in an array structure called buffer, see Fig. 14.

```
procedure Read_input;

   var
        buffer : stream;      {buffer is for storing input
                                characters}
        charnum : integer;
        i : integer;
        ch : char;

   begin
        for i := 1 to 100 do
             buffer[i] := ' ';

        ch := ' ';
        charnum := 0;
        while (not eoln(trm)) and (ch <> '@') do
        begin
             charnum := charnum + 1;
             read(trm,ch);
             if ch <> '@' then    {'@' is the  end marker of
                                     template}
                  buffer[charnum] := ch
             else
               tem := false;
        end;
   end;
```

Figure 14. Procedure Read_input

The heart of the lexical analyzer is the Scanner procedure which is implemented to emulate the actions of the finite state machine diagram in Fig. 12. Its function is to group individual characters into tokens and it must be able to isolate the next sequence of characters in the input buffer which designates a valid token. The Scanner marks the beginning and the end of the token in the input buffer, send the token and its symbol number to the Get_token routine for linking all the tokens together to form a token's stream. See Fig. 15 for the procedure Scanner.

```
procedure Scanner(buffer:stream; charnum : integer);


var
   i,j,k : integer;


begin

     i := 1;

     while i <= charnum do

     begin
          case buffer[i] of

          ' ' : i := i + 1;       {skip blanks}

          'A'..'Z','a'..'z' :     {identifier}
                            begin
                            j := i;
                            repeat
                            i := i + 1;
                            until not(buffer[i] in
                             ['A'..'Z','a'..'z','0'..'9']);
                            k := i - 1;
                            get_token(buffer,j,k,2);
                            end;

          '0'..'9' :                {integer}
                            begin
                            j := i;
                            repeat
                            i := i + 1;
                            until    not    (buffer[i]    in
                                     ['0'..'9']);
                            k := i - 1;
                            get_token(buffer,j,k,18);
                            end;
```

Figure 15. Procedure Scanner

```
'+' :
                    begin
                    get_token(buffer,i,i,15);
                    i := i + 1;
                    end;

'-' :               begin
                    get_token(buffer,i,i,16);
                    i := i + 1;
                    end;

'*' :               begin
                    get_token(buffer,i,i,17);
                    i := i + 1;
                    end;

'=' :
                    begin
                    get_token(buffer,i,i,11);
                    i := i + 1;
                    end;

',' :
                    begin
                    get_token(buffer,i,i,5);
                    i := i + 1;
                    end;

';' :
                    begin
                    get_token(buffer,i,i,6);
                    i := i + 1;
                    end;

'>' :
                    begin
                    get_token(buffer,i,i,14);
                    i := i + 1;
                    end;
```

Figure 15. (Continued)

```
        '<' :
                         begin
                         j := i + 1;
                         if s[j] <> '>' then
                         begin
                         get_token(buffer,i,i,13);  {<}
                         i := i + 1;
                         end
                         else
                         begin
                         get_token(buffer,i,j,12);{<>}
                            i := i + 2;
                         end;
                         end;
         ':' :
                         begin
                         j := i;
                         i := i + 1;
                         if s[i] = '=' then
                         begin
                            get_token(buffer,j,i,21);{:=}
                            i := i + 1;
                         end;
                         end;
        '(' :
                         begin
                         get_token(buffer,i,i,9);
                         i := i + 1;
                         end;

        ')' :
                         begin
                         get_token(buffer,i,i,10);
                         i := i + 1;
                         end;

        else begin
              get_token(buffer,i,i,0);  {invalid input}
              i := i + 1;
              end;
       end;
   end;

end;
```

Figure 15. (Continued)

Before describing the function of Get_token, we must understand how the token is represented and stored. The internal representation of the token is a record which contains the symbol, its symbol number, and a pointer to the next input token (see Fig. 16).

```
string = packed array[1..7] of char;
token_list = record
                 sym : string;
                 sym_num : integer;
                 link : tokenptr;
               end;
```

```
 ______________________
 | Symbol         | L |
 |                | i ---------->
 |----------------| n |
 | Symbol number  | k |
 |________________|___|
```

Figure 16. Internal Representation of Token

Fig. 17 is the procedure Get_token, the routine is used to receive tokens from the Scanner procedure, linking all tokens together to form a stream of tokens, and build symbol table for tokens.

```
procedure Get_token(buffer:stream; j,i,des : integer);

var

k,l : integer;
token : tokenptr;

begin
   new(token);
   initptr(token);  {initialize pointer}
   l := 0;
for k := j to i do
   begin
        l := l + 1;
        token^.sym[l] := buffer[k];
   end;
   token^.sym_num := des;
   if des = 2 then
        checkres(token,restable);  {check  for  reserve
                                    words}
   if tem = true then
        buildtable(token,temphead,templast)      {build
                              symbol table for templates}
   else
        buildtable(token,inputhead,inputlast);
             {build symbol table for input}
end;
```

Figure 17. Procedure Get_token

The Check_reserve routine is used to compare all identifiers with entries in the reserved word table (declare, if, then, end, else, while, loop). Fig. 18 is the procedure Check_reserve.

```
procedure    Check_reserve(var    token:    tokenptr;
                        reserve_table : table);
var

        i : integer;

begin

   for i := 1 to 7 do
   begin
        if reserve_table[i]^.sym = token^.sym then
                        begin
                            token^.sym_num :=
                            restable[i]^.sym_num;
                        end;
   end;

end;
```

Figure 18. Procedure Check_reserve

The last procedure Print_table is used to print all the input tokens recognized by the lexical analyzer. The token symbol, its symbol number and description are printed, An example listing generated by the lexical analyzer follows.

Input statements :

```
        while (a > b) loop

             x := x * 1;

             a := a + 1;

        end loop;
```

Output listing from the lexical analyzer :

```
     Token          Symbol #        Description
  ---------------------------------------------------
     while          19              reserved word
     (              9               left parenthesis
     a              2               identifier
     >              14              greater than
     b              2               identifier
     )              10              right parenthesis
     loop           20              reserved word
     x              2               identifier
     :=             21              assignment operator
     x              2               identifier
     #              0               invalid token
     1              18              constant
     ;              6               semicolon
     a              2               identifier
     :=             21              assignment operator
     a              2               identifier
     +              15              addition operator
     1              18              constant
     ;              6               semicolon
     end            7               reserved word
     loop           20              reserved word
     ;              6               semicolon
```

Parser

<u>The Role of the Parser</u>

The parser performs syntactic checking in the evaluation system, see Fig. 19. The parser input is a token stream generated by the lexical analyzer, and the output is a parse tree generated for the input statement.

```
                          check syntax
 --------                 ----------
¦Lexical ¦------->¦ Parser    ¦----->
¦Analyzer¦ stream ¦           ¦      generate error
 --------    of     ----------          messages
           tokens                    create parse trees
```

Figure 19. The Role of Parser in the Evaluation System

The parse tree produced by the parser is not created physically, the parse tree only exists abstractly as a sequence of actions made by stepping through the tree construction process. There are two common forms of parsers ---- operator precedence and recursive descent. The parsing algorithm used in the implementation of the parser in the statements evaluation system (SES) is the recursive descent. A recursive descent parser is constructed by a set of recursive procedures to recognize its input with no backtracking. This method of parsing is more effecient (though less general) than most top-down parsing method that allow backup. It should be noted, however, that this highly recursive technique does not work on all context-free grammars. That is, certain grammars require backup in order for sucessful parsing to occur.

In the recursive-descent method of parsing, a sequence of production applications is realized in a sequence of function or procedures calls. In particular, functions or procedures are written for each non-terminal symbol. Each procedure recognizes substrings which are expansions of the non-terminal. Error signals and error messages should result when an unexpected terminal is recognized.

Basic Design for Recursive-Descent Parser

Appendix A contains a context free grammar for the mini-language consisting of 20 non-terminals. Each non-terminal of the language has a parsing procedure associated with it that is used to determine if that nonterminal may generate an initial substring of the tokens remaining in the input. Within a parsing procedure, both nonterminals and terminals can be "matched". To match a non-terminal "A", we call the parsing procedure corresponding to "A" (there may be recursive calls). To match a terminal symbol "t", we call a procedure Match(ptr,x,y); ptr is the pointer which points to the current position in the input tokens stream, x is the symbol number associated with the token "t" to be matched, y is the message number associated with the error message to be printed if the token is not in the input stream. For example to match the token "declare", the symbol number associated with the terminal "declare" is 1, therefore the procedure call is

```
match(ptr,1,1);
```

Match calls the scanner to get the next token. If this token is "declare", everything is as expected, and the token is consumed. Otherwise, a syntax error is detected which results in an error message followed by termination of the parsing process. The procedure Match is in Fig. 20.

```
procedure Match(var ptr : tokenptr;num : integer; messcode :
                                                   integer);

begin

     if error = false then  {no syntax error occured before}
     begin
          if next(ptr) = num then    {matched next token}
          {function Next will returns the lookahead token}
                 ptr := ptr^.link
          else           {next input token is error}
              begin
                 error := true;
                 ptr :=  ptr^.link;          {skip the error
                                              token}
              end;
          {print error message}
          if error = true then syntaxerror(messcode);
     end;

end;
```

Figure 20. Procedure Match for Matching Input Token

To be able to look ahead and not to consume the next input token, a function called "Next" is defined. The function Next returns the symbol number of the lookahead token. It is different from Match in that Next just "peeks" at the next token, whereas Match tries to match and consume it. Fig. 21 is the function Next.

```
function Next(var ptr : tokenptr) : integer;

var
    temp : tokenptr;

begin
    temp := ptr^.link;
    next := temp^.descrip;

end;
```

Figure 21. Function Next that Returns the Lookahead Token

The parsing procedure for each non-terminal is very easy to implement: If the production for A is

A -----> X1 X2 ... Xm

then procedure "A" is simply X1; X2;... Xm, see Fig. 22; (if some Xi is a terminal, then we call match(ptr,a,b), where a is the symbol number associated with Xi, and b is the message number associated with the error message to be printed if Xi is not in the input stream).

```
procedure A;
begin
     X1;     {call procedure X1}
     X2;     {call procedure X2}
     .
     .
     match(ptr,a,b)    {match the terminal Xi with
                        symbol number equal a}
     .
     .
     Xm;     {call procedure Xm}
end;
```

Figure 22. Parsing Procedure A

Parsing Procedures for the Mini-Language

The parsing procedure A in the last section seems very easy to implement. But how are we going to define the parsing procedure if A has more than one production, for example,

```
A -----> X1 X2 X3 ... Xm
A -----> Y1 Y2 Y3 ... Yn
```

We must decide what production to try to match, therefore we need to lookahead and use the lookahead token to decide what production to choose. Appendix C has a brief description on LL(1) grammar and generation of the predict set of production.

The design of the parser is a hierarchial structure of parsing procedures, which call each other recursively. There are a total of twenty parsing procedures, each for every non-terminal in the context-free grammar. The basic structure of the parser follows the production rules of the grammar. Fig. 23 is the hierarchial structure of the parser, which also shows the execution flow of the parsing procedures. The alphabets on the arcs are the choices of execution flow and the numbers on the arcs are the sequence steps of the execution flow. For example, the parsing procedure "dec_seq", it has two choices of A and B determined by the input token. If the token is not an identifier, "if" or "while, it calls the parsing procedures "declaration" and "dec_tail" in that order, otherwise, it

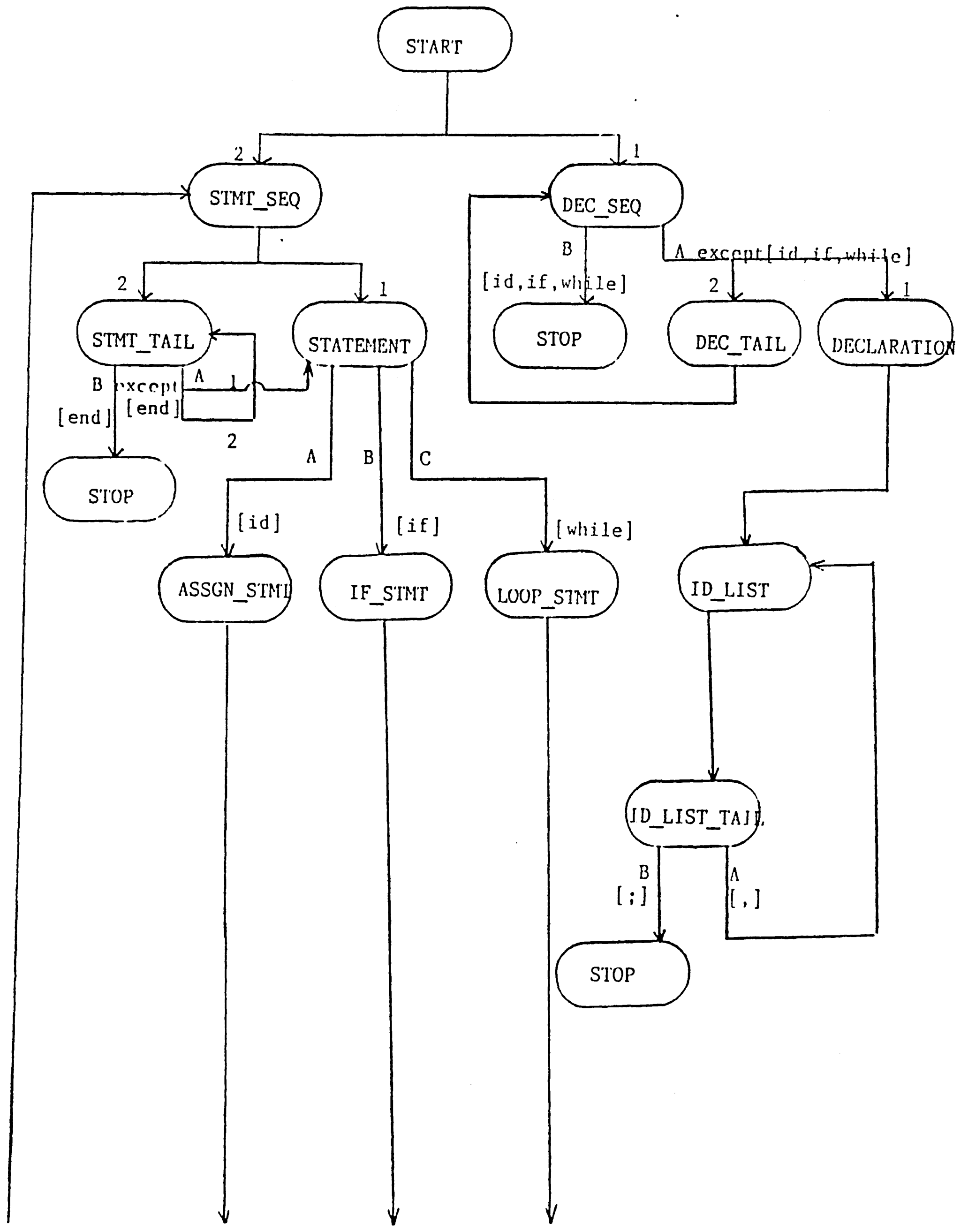

Figure 23. Hierarchial Structure of the Parser

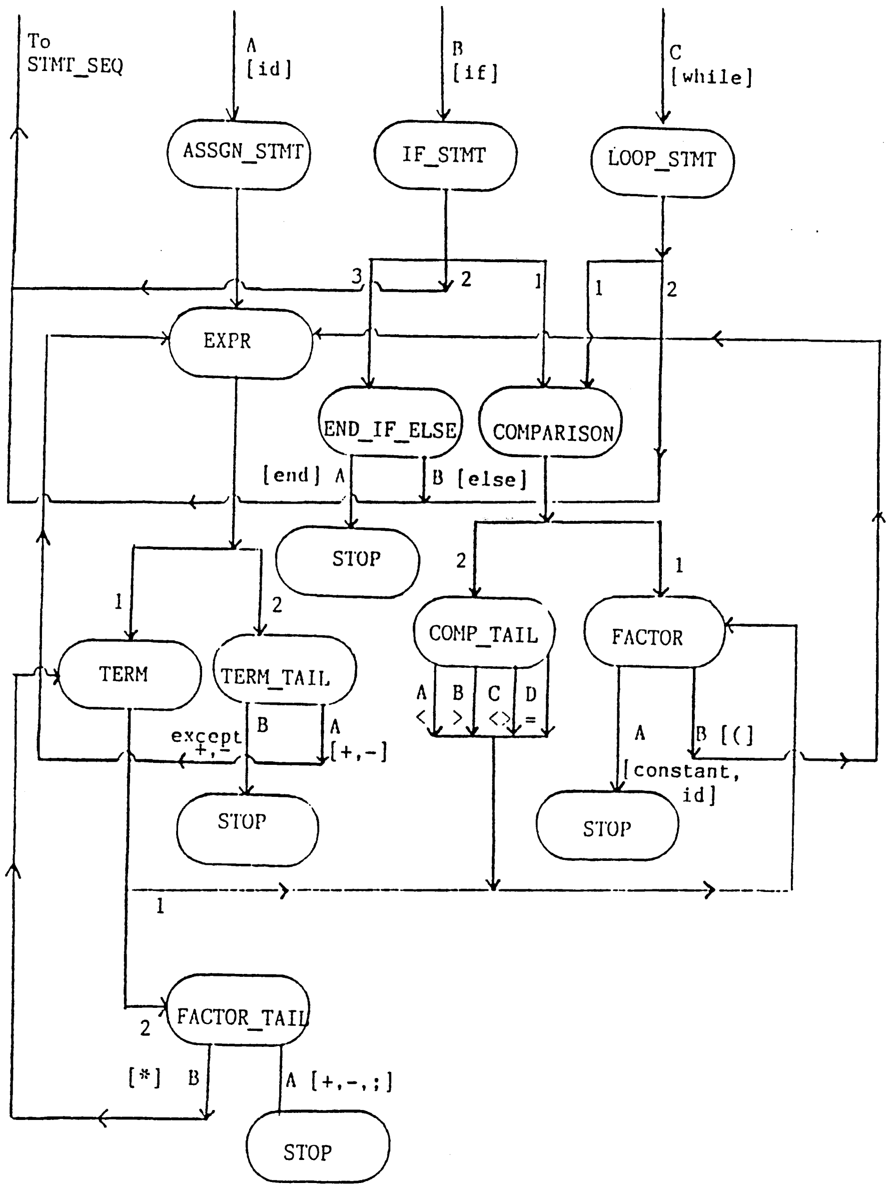

Figure 23. (Continued)

stops and the execution begins at stmt_seq".

For the mini-language we start with the non-terminal "start", the production for "start" :

```
<start> ----> <dec_seq> <stmt_seq>
```

The procedure "start" is very simple, calling two other parsing procedures "dec_seq" and "stmt_seq" (see Fig. 24).

```
procedure start(var ptr : tokenptr);

begin
     error :=  false;          {error is a flag which
                               sets to  true if syntax
                               error occurs}
     dec_seq(ptr);
     stmt_seq(ptr);

end;
```

Figure 24. Parsing Procedure for <start>

As we have seen in Appendix C, the parsing procedure of <dec_seq> is more complicated, for <dec_seq> has 2 productions in the grammar. To construct the parsing procedure for <dec_seq>, we need to have the predict sets to make the decision which production to choose. To obtain the predict sets of non-terminal <dec_seq>, we need to get first(<dec_seq>) and follow(<dec_seq>), since <dec_seq> can produce epsilon.

Production :

```
<dec_seq> ----> <declaration> <dec_tail>
<dec_seq> ----> epsilon
```

We can obtain the predict sets of <dec_seq> by the following steps :

```
predict(<dec_seq> ----> <declaration> <dec_tail>) =
                                    first(<declaration>)
first(<declaration>) = {declare}
predict(<dec_seq> ----> epsilon) = follow(dec_seq)
```

To obtain the follow(dec_seq), we need to search for all the productions in the grammar with <dec_seq> at the right hand side of the production. There is only one production,

```
<start> ----> <dec_seq> <stmt_seq>
```

with <dec_seq> at the right hand side of the production (see Appendix A).

```
follow(dec_seq) = first(<stmt_seq>)
first(<stmt_seq>) = first(<statement>)
first(<statement>) = {id, if, while}
```

Therefore, the predict sets for <dec_seq> are {declare} and {id, if, while}. That is, if the lookahead token is in one of the predict sets of the productions, in this case, we choose the predicted production according to whatever the lookahead token is; otherwise, if the lookahead token is not in any predict set, the lookahead token occurs in an illegal position and we have a syntax error. See Fig. 25 for the procedrue dec_seq.

```
procedure dec_seq;

begin

if error = false then
     case next(ptr) of        {next(ptr) returns
                               lookahead token}
     2,3,19 : ;     {id, if, while}
     else begin
              declaration(ptr);
              dec_tail(ptr);
          end;
     end;

end;
```

Figure 25. Parsing Procedure for <dec_seq>

The predict set of <declaration> is much simpler than <dec_seq>, for it has only one production.

```
predict(<declaration> ---->declare <id_list>) =
                                              {declare}
```

<declaration> has only one predict set and only one element in the set, that makes the parsing procedure fairly simple, see Fig. 26.

```
procedure declaration;

begin

   if error = false then
   begin
        match(ptr,1,1);             {declare}
        id_list(ptr);
   end;

end;
```

Figure 26. Parsing Procedure for <declaration>

```
predict(<dec_tail> ----> ; <dec_seq>) = {;}
```

```
procedure dec_tail;

begin

   if error = false then
   begin
        match(ptr,6,6);        {;}
        dec_seq(ptr);
   end;

end;
```

Figure 27. Parsing Procedure for <dec_tail>

```
predict(<id_list> ----> <id> <id_list_tail>) = {id}
```

```
procedure id_list;

begin
   if error = false then
   begin
        match(ptr,2,2);         {id}
         id_list_tail(ptr);
   end;

end;
```

Figure 28. Parsing Procedure for<id_list>

```
predict(<id_list_tail> ----> , <id_list>) = {,}

predict(<id_list_tail> ----> epsilon) =

                                         follow(<id_list_tail>)
```

To compute follow(<id_list_tail>), we check all occurences of <id_list_tail> on various right hand sides of all the productions. Since it appears only in

<id_list> ----> <id> <id_list_tail>

follow(<id_list_tail>) = follow(<id_list>)

Inspecting all occurences of <id_list> on the right hand sides of all productions, we conclude that

follow(<id_list>) = follow(<declaration>)

since <declaration> ----> declare <id_list> is the only production with <id_list> at the right hand side.

follow(<declaration>) = first(<dec_tail>) = {;} =

follow(<id_list_tail>) =

predict(<id_list_tail> ----> epsilon)

Therefore, the predict sets for <id_list_tail> are {,} and {;}, the parsing procedure for <id_list_tail> is in Fig. 29.

```
procedure id_list_tail;

begin
  if error = false then
  case next(ptr) of
  6: ;                             {;}
  else   begin
           match(ptr,5,26);        {,}
            id_list(ptr);

         end;
  end;

end;
```

Figure 29. Parsing Procedure for <id_list_tail>

We have finished all the parsing procedures for the declarations part of the mini-language, and are ready for the statements sequence procedure. The start symbol for statements sequence is <stmt_seq> with production

<stmt_seq> ----> <statement> ; <stmt_tail>

The procedure <stmt_seq> is very simple, call <statement>, match ';', and call <stmt_tail>, see Fig. 30.

```
procedure stmt_seq;

begin
   if error = false then
   begin
        statement(ptr);
        match(ptr,6,6);             {;}
        stmt_tail(ptr);

   end;
end;
```

Figure 30. Parsing Procedure for <stmt_seq>

The non-terminal <statement> has 3 productions,

```
<statement> ----> <assgn_stmt>

<statement> ----> <if_stmt>

<statement> ----> <loop_stmt>


predict(<statement> ----> <assgn_stmt>) =

                  first(<assgn_stmt>) = {id}

predict(<statement> ----> <if_stmt>) =

                  first(<if_stmt>) = {if}

predict(<statement> ----> <loop_stmt>) =

                  first(<loop_stmt> = {while}
procedure statement;

begin
   case next(ptr) of

        2     : assgn_stmt(ptr);                        {id}
        3     : if_stmt(ptr);                           {if}
        19    : loop_stmt(ptr);                      {while}
        else    syntaxerror(25);

   end;

end;
```

Figure 31. Parsing Procedure for <statement>

<stmt_tail> has 2 productions in the grammar,

<stmt_tail> ----> <statement> ; <stmt_tail>

<stmt_tail> ----> epsilon

predict(<stmt_tail> ----> <statement> ; <stmt_tail>) =

first(<statement>) = {id, if, while}

predict(<stmt_tail> ----> epsilon) =

follow(<stmt_tail>) =

follow(<stmt_seq>) = {end}, {else}, and [end of input].

The <stmt_tail> procedure is in Fig. 32.

```
procedure stmt_tail;

begin
     if error = false then
          if ptr^.link <> nil then       {not end of input}

               case next(ptr) of
               7 : ;                     {end}
               8 : ;                     {else}
               else begin
                         statement(ptr);
                         match(ptr,6,6);      {;}
                         stmt_tail(ptr);
                    end;
               end;

end;
```

Figure 32. Parsing Procedure for <stmt_tail>

Fig. 33 to Fig. 36 are the parsing procedures for <assgn_stmt>, <if_stmt>, <endif_else>, and <loop_stmt>.

```
<assgn_stmt> ----> <id> := <expr>


procedure assgn_stmt;

begin
   if error = false then
   begin
        match(ptr,2,2);              {id}
        match(ptr,21,21);            {:=}
        expr(ptr);
   end;

end;
```

Figure 33. Parsing Procedure for <assgn_stmt>

```
<if_stmt> ----> if <comparison> then
                      <stmt_seq>
            <endif_else>
```

```
procedure if_stmt;

begin

   if error = false then

   begin
        match(ptr,3,3);                  {if}
        comparison(ptr);
        match(ptr,4,4);                  {then}
        stmt_seq(ptr);
        endif_else(ptr);

   end;

end;
```

Figure 34. Parsing Procedure for <if_stmt>

```
<endif_else> ----> end if

<endif_else> ----> else
                       <stmt_seq>
                   end if




procedure endif_else;

begin

   if error = false then
   begin

        case next(ptr) of
          7 : begin                                   {end}
                   match(ptr,7,7);
                   match(ptr,3,3);                    {if}
              end;

          8 : begin                                   {else}
                   match(ptr,8,8);
                   stmt_seq(ptr);
                   match(ptr,7,7);                    {end}
                   match(ptr,3,3) ;                   {if}
              end;
        end;
   end;

end;
```

Figure 35. Parsing Procedure for <endif_else>

```
<loop_stmt> ----> while <comparison> loop
                           <stmt_seq>
                      end loop
```

```
procedure loop_stmt;


begin
   if error = false then
   begin
        match(ptr,19,19);     {while}
        comparison(ptr);
        match(ptr,20,20);     {loop}
        stmt_seq(ptr);
        match(ptr,7,7);       {end}
        match(ptr,20,20);     {loop}
   end;

end;
```

Figure 36. Parsing Procedure for <loop_stmt>

The "comparison" in the while statement is in the form of

```
( a > b)
(count <> 10)
```

There are four relational operators in the mini-language ("=", "<", ">", "<>"), the parser looksahead for the token and returns an error signal if the token is not in the set of relational operators. The following is the LL(1) grammar for <comparison>.

```
<comparison> ----> ( <factor> <comp_tail>)
<comp_tail>  ----> = <factor> )
<comp_tail>  ----> <> <factor> )
```

```
<comp_tail>  ----> < <factor> )

<comp_tail>  ----> > <factor> )



procedure comparison;

begin
   if error = false then
   begin
        match(ptr,9,9);        {(}
        factor(ptr);
        comp_tail(ptr);
   end;

end;


procedure comp_tail;

begin
   if error = false then

     case next(ptr) of
     11 : begin
              match(ptr,11,11);      {=}
              factor(ptr);
              match(ptr,10,10);      {)}
          end;
     12 : begin
              match(ptr,12,12);      {<>}
              factor(ptr);
              match(ptr,10,10);      {)}
          end;
     13 : begin
              match(ptr,13,13);      {<}
              factor(ptr);
              match(ptr,10,10);      {)}
          end;
     14 : begin
              match(ptr,14,14);      {>}
              factor(ptr);
              match(ptr,10,10);      {)}
          end;
     else syntaxerror(28);
     end;
end;
```

Figure 37. Parsing Procedures for <comparison>, <comp_tail>

Let's look at a simple arithematic expression,

x * a + b * c

this expression gives different results, depending on the grammar for the expression. For example if the grammar for the expression is

```
<expr> ----> <expr> + <expr>
<expr> ----> <expr> * <expr>
<expr> ----> <expr> - <expr>
<expr> ----> id | constant | (expr)
```

This grammar is ambiguous because there can be more than one parse tree generated by the grammar, see Fig. 38.

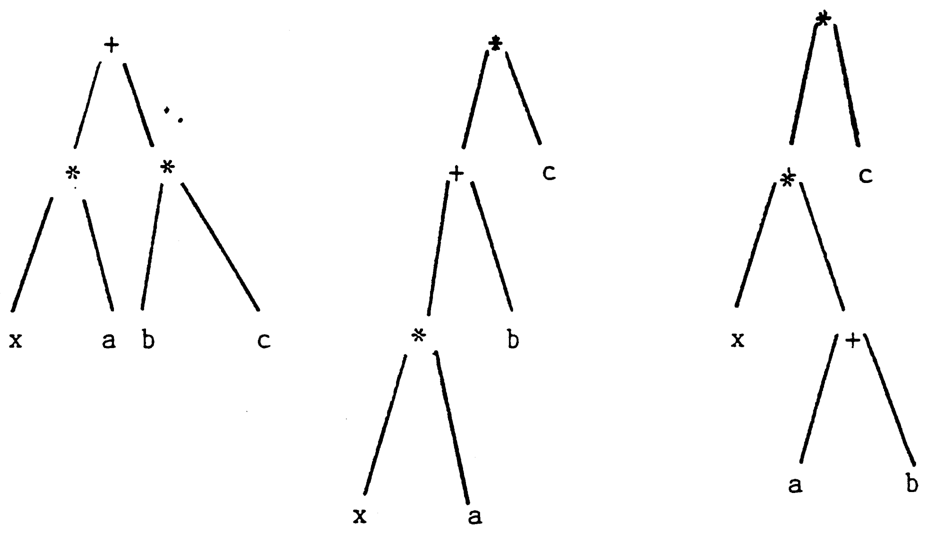

Figure 38. Parse Trees Generated by Ambiguous Grammar.

We cannot use an ambiguous grammar in the parser, for we cannot uniquely determine which parse tree to select for a sentence. To make the grammar unambiguous, we have to separate the multiplication part from the addition, subtraction part. The modified grammar in which multiplication has higher priority than addition and subtraction is as follows,

```
<expr>             ----> <term> <term_tail>
<term_tail>        ----> + <expr>
<term_tail>        ----> - <expr>
<term_tail>        ----> epsilon
<term>             ----> <factor> <factor_tail>
<factor_tail>      ----> * <term>
<factor_tail>      ----> epsilon
<factor>           ----> <constant>
<factor>           ----> <id>
<factor>           ----> ( <expr> )
```

Fig. 39 gives the parsing procedures needed to parse expression.

```
procedure expr;

begin
   if error = false then
   begin
     term(ptr);
     term_tail(ptr);
   end;
end;
```

Figure 39. Procedures for Parsing an Expression

```
predict(<term_tail> ----> + <expr>) = {+}

predict(<term_tail> ----> - <expr>) = {-}

predict(<term_tail> ----> epsilon) = follow(<term_tail>) =

follow(<expr>) = follow(<assgn_stmt>) = follow(<statement>)
= {;}




procedure term_tail;

begin

     if error = false then
     begin
          case next(ptr) of
          15 : begin
                    match(ptr,15,15);   {+}
                    expr(ptr);
               end;
          16 : begin
                    match(ptr,16,16);   {-}
                    expr(ptr);
               end;
          6  : ;                        {;}
          else ;

          end;
     end;

end;
```

Figure 39. (Continued)

```
procedure term;

begin
     if error = false then
     begin
          factor(ptr);
          factor_tail(ptr);
     end;
end;
```

```
predict(<factor> ----> <constant>) = {constant}

predict(<factor> ----> <id>) = {id}

predict(<factor> ----> ( <expr> ) ) = {(}
```

```
procedure factor;

begin

     if error = false then

     case next(ptr) of
     18 : match(ptr,18,18);      {constant}
     2  : match(ptr,2,2);        {id}
     9  : begin
               match(ptr,9,9);         {(}
               expr(ptr);
               match(ptr,10,10);       {)}
          end;
     else match(ptr,0,27);        {skip error token}
     end;

end;
```

Figure 39. (Continued)

```
procedure factor_tail;

begin

   if error = false then
   begin
        case next(ptr) of
        15,16,6 : ;            {+,-,;}

        17 : begin
                  match(ptr,17,17);    {*}
                  term(ptr);
             end;
        6 : ;
        end;
   end;

end;
```

Figure 39. (Continued)

We have defined all the parsing procedures for each non-terminal symbol in the Ledgard mini-language. Now let's look at a simple example to see how the praser works. For example, the input statement is,

```
id := id * constant + id ; [end of input]
```

```
Step    Procedure Calls          Remaining Input

1       start               id := id * constant + id ;[end]
```

```
2       dec_seq             id := id * constant + id ; [end]

3       stmt_seq            id := id * constant + id; [end]

4       statement           id := id * constant + id; [end]
        match(";")
        stmt_tail

5       assgn_stmt          id := id * constant + id; [end]
        match(";")
        stmt_tail

6       match(id)           id := id * constant + id; [end]
        match(":=")
        expr
        match(";")
        stmt_tail

7       match(":=")         := id * constant + id; [end]
        expr
        match(";")
        stmt_tail

8       expr                id * constant + id; [end]
        match(";")
        stmt_tail

9       term                id * constant + id; [end]
        term_tail
        match(";")
        stmt_tail

10      factor              id * constant + id; [end]
        factor_tail
        term_tail
        match(";")
        stmt_tail

11      match(id)           id * constant + id; [end]
        factor_tail
        term_tail
        match(";")
        stmt_tail
```

```
12      factor_tail         * constant + id; [end]
        term_tail
        match(";")
        stmt_tail


13      match("*")          * constant + id; [end]
        term
        term_tail
        match(";")
        stmt_tail


14      term                constant + id ; [end]
        term_tail
        match(";")
        stmt_tail


15      factor              constant + id ; [end]
        factor_tail
        term_tail
        match(";")
        stmt_tail


16      match(constant)     constant + id ; [end]
        factor_tail
        term_tail
        match(";")
        stmt_tail


17      factor_tail         + id ; [end]
        term_tail
        match(";")
        stmt_tail


18      term_tail           + id ; [end]
        match(";")
        stmt_tail


19      match("+")          + id ; [end]
        expr
        match(";")
        stmt_tail
```

```
20      expr                 id ; [end]
        match(";")
        stmt_tail


21      term                 id ; [end]
        term_tail
        match(";")
        stmt_tail


22      factor               id ; [end]
        factor_tail
        term_tail
        match(";")
        stmt_tail


23      match(id)            id ; [end]
        factor_tail
        term_tail
        match(";")
        stmt_tail


24      factor_tail          ; [end]
        term_tail            {factor_tail will match epsilon}
        match{";")
        stmt_tail


25      term_tail            ; [end]
        match(";");          {term_tail will match epsilon}
        stmt_tail


26      match(';")           ; [end]
        stmt_tail


27      stmt_tail            [end]


28      Done! {stmt_tail will match end of input}
```

Translator

The Role of the Translator

The best way to evaluate different statements is to translate the statements into a standard format, and then compare it to the template answer. The standard format can be a symbol table, 3-address code, quadruples or tree structure. The translator in this project translates different statements into different structures, depending on the statement structure and its complexity. Fig. 40 is the structure of the statements evaluation system (SES) including the translator.

```
                                                     tables of
                 stream                              declarations
input text    of tokens      parse tree              assignments
                                                     expressions
        _________      _______      __________      for evaluation
--->|Lexical |----->|parser|----->|translator|------>
    |Analyzer|      |______|      |__________|
     --------
```

Figure 40. Structure of Statements Evaluation System

The translation scheme used in this project is a syntax-directed translation scheme, which allows a semantic action (subroutine) to be attached to the production of the context-free grammar. The subroutine is attached to the parsing procedure of the recursive descent parser, which is called at the appropriate time by the parser. The advantages of the syntax-directed translation scheme are its directed translation in terms of the syntactic structure of the grammar and its easiness in modification without disturbing the existing translations, which simplifies the design of the translator and efficiently exploits the parser.

<u>Semantic Actions</u>

The semantic action is to generate output when a particular production is recognized from the input. For example

U ----> ABC {called subroutine w}

is a production with semantic action w associated with it. The semantic action (called subroutine w) is executed whenever the parser recognizes in its input a substring x which has a derivation of the form U ----> ABC --*--> x. The semantic action can be the generation of intermediate code (3-address code, quadruples), or the placement of data into a symbol table, or the computation of values for variables or the transfering of symbols into different (standard) formats.

<u>Implementation of Syntax-Directed Translator</u>

To design the syntax-directed tanslator for the mini-language, we need to define semantic actions for the parsing procedures in the recursive descent parser. After the semantic actions are defined, subroutine codes are generated corresponding to each semantic action. Subroutine calls are added to the parsing procedure wherever the semantic action is required.

Translation Scheme for Declaration

```
(1)  declare x,y,z;
(2)  declare x, y;
     declare z;
(3)  declare x;
     declare y;
     declare z;
```

The 3 sets of declaration statements above have the same effect (define the variables x,y,z). Since x,y and z can be in any order, there are 3C2 = 6 variations in the first set, 6 variations in the second set and 6 in the third set, which make up a total of 6 + 6 + 6 = 18 combinations of formats for declaring just 3 variables x,y,and z. The statements evaluation system should be able to recognize all these different formats of declarations and determine the equivalence of each statement.

The translation scheme is to input the variables into a symbol table (declaration table) when the variable is recognized by the parser. By adding semantic action to the declartion part of the production grammar,

```
<dec_seq>       ----> <declaration> <dec_tail>
<dec_seq>       ----> epsilon
<declaration>   ----> declare <id_list>
<dec_tail>      ----> ; <dec_seq>
```

```
<id_list>      ----> <id> {ACTION 1} <id_list_tail>

<id_list_tail> ----> , <id_list>

<id_list_tail> ----> epsilon
```

ACTION 1 : input id into declaration table

the recognized variable is placed into the declaration table, see Fig. 41.

variable name	type
x	integer
y	integer
z	integer

Figure 41. Declaration Table

The declaration table is implemented as a linked list, which stores the name and the type of the variables (only the single type integer occurs in the mini-language). Fig. 42 is the parsing procedure id_list for declaration with semantic action added.

```
procedure id_list;
begin
if error = false then
   begin
        match(ptr,2,2);         {id}
        insert(dechead,dectail,ptr); {semantic action}
        id_list_tail(ptr);
   end;
end;
```

Figure 42. Parsing Procedure id_list

The insert routine is the semantic action for inputing the declared variables into the declaration table. The procedure insert is in Fig. 43 .

```
procedure  insert(var  headptr,  tailptr  :  varptr;  ptr  :
                                                   tokenptr);

{insert  element  into  the  linked  list  with  headptr and
tailptr point to the head and the tail of the list}


var
    idrec : varptr;

begin
     new(idrec);
     initvar(idrec);
     idrec^.id[1] := ptr^.sym;
     idrec^.len := 1;
     if headptr^.link <> nil then    {insert at the end}
     begin
        tailptr^.link := idrec;
        tailptr := idrec;
     end
     else
        begin                         {first element}
        headptr^.link := idrec;
        tailptr := idrec;
        end;

end;
```

Figure 43. Insert Procedure

The following is an example of how the system evaluates declaration statements,

```
        Template answer : declare u, w;

        Input answer    : declare w;

                          declare u;
```

The template answer is first fed into the evaluation system. When parsing the template answer, a template declaration table is created for storing all the declared variables in the template answer, which is used to compare with the input statements later on in the evaluation process. Fig. 44 is the declaration table for "declare u,w;".

variable name	type
u	integer
w	integer

Figure 44. Declaration Table for "declare u,w;"

Fig. 45 is the declaration table generated for the input statement when it is translated by the translator. The evaluation system compares both declarations, they are considered functionally identical if they all have the same variables regardless of their order in the table. For example, Fig. 44 and Fig. 45 have the same variables although they are not located at the same locations inside the tables. Therfore, we conclude that the input answer is correct, which declares the variables "u" and "w" as in the template answer.

variable name	type
w	integer
u	integer

Figure 45. Declaration Table for "declare w;" "declare u"

Translation Scheme for Assignment Statement

The translation process for an assignment statement is a more complicated process than the translation of a declaration statement. Generally assignment statements can be written in many different forms, which when combined with different priorities and characteristics of operations like multiplication, addition, subtraction and parenthesis, lead to translation difficulties. For example, a simple assignment statement like

```
x := a + b * ( 2 + 3) - c;
```

can be written in these different forms, which are all funtionally equivalent,

```
1. x := b * ( 2 + 3) -c + a;
2. x := a - c + b *(2 + 3);
3. x := a + 2 *b - c + 3*c;
4. x := b*2 + b*3 -c + a;
5. x := a -c + 2*b + b * 3;
6. x := (a - c) + b* (2 + 3);
```

or in the following similar forms, which are not functionally equivalent and do not produce the same result when the statement is executed.

```
1. x :=  b + a *(2 + 3) -c;
2. x := a + b*2 + 3 - c;
3. x := (a + b) * (2 + 3) -c;
```

Let us begin by looking at the production grammar for the assignment statement.

```
<assgn_stmt>   ----> <id> := <expr>
<expr>         ----> <term> <term_tail>
<term_tail>    ----> + <expr>
<term_tail>    ----> - <expr>
<term_tail>    ----> epsilon
<term>         ----> <factor> <factor_tail>
<factor_tail>  ----> * <term>
<factor_tail>  ----> epsilon
<factor>       ----> <constant> ¦ <id> ¦ <expr>
```

The context-free grammar above can produce the following statements :

```
1. x := a * 3 + b + c;
2. x := 3 * a + c + b;
```

Because of the commutative characteristic of the "+" and "*" operators, the two statements generate the same result upon execution. When an operator is commutative like "*" and "+", the order of the operands does not affect the function of the statement. The evaluation system should be able to recognize that the two statements are functionally equivalent. To simplify the evaluation process, the translation scheme translates the assignment statements into table formats which eliminated the parenthesis in the assignment statement. Using the table approach made the internal represenation of the assignment statement easy to implement and it also simplifies the task of keeping track the latest assigned value of each variable for substitution. The assigned identifier is placed into the assignment table; the expression table which holds all the variables in the expression is linked to the assigned identifier, see Fig. 46 and Fig. 47.

```
 Assignment table
 ------------------
¦                  ¦
¦------------------¦              Expression table
¦        x         ¦  ----> ------------------------------
¦------------------¦       ¦ a¦ b¦ c¦  ¦  ¦            ¦+/*¦
¦                  ¦       ¦--¦--¦--¦--¦--¦----------------¦
¦------------------¦       ¦ 3¦  ¦  ¦  ¦  ¦                ¦
                            ------------------------------
                             0  0  0             sign bit
```

Figure 46. Table Representation of x := a*3 + b + c

The first row of the expression table represents addition, and each column represents multiplication. The last row is the sign bit for each column, it is set to 0 if the column is positive and set to 1 if the column is negative.

```
Assignment table
-----------------
|               |
|---------------|              Expression table
|       x       |  ---->  -----------------------------
|---------------|         | 3| c| b|  |  |        |+/*|
|               |         |--|--|--|--|--|-------------|
|---------------|         | a|  |  |  |  |             |
                          -----------------------------
                           0  0  0
```

Figure 47. Table Representation of x := 3*a + c + b

Because of the commutative characteristic of addition and multiplication, the order of the rows of each individual column and the order of the columns of the expression table do not affect the result of the assignment statement, therefore the two representations in Fig. 46 and Fig. 47 are functionally equivalent. Using the table representation of the assignment statement can simplify and speed up the evaluation process, and it can be implemented easily by arrays or linked lists.

The evaluation system generates separate assignment tables and expression tables for the template and the input statements. The input assignment statements are evaluated by comparing the tables with the template answer. The input answer is correct if the assignment table is matched with the assignment table of the template.

For subtraction, the sign bit of the expression table is set to 1 with the subtracted variable placed into the expression table. See Fig. 48 for the table representation for x := a + b - c.

```
 Assignment table
 -----------------
¦                 ¦
¦-----------------¦                 Expression table
¦        x        ¦   ---->  ------------------------------
¦-----------------¦         ¦ a¦ b¦ c¦  ¦  ¦              ¦+/*¦
¦                 ¦         ¦--¦--¦--¦--¦--¦-----------------¦
¦-----------------¦         ¦  ¦  ¦  ¦  ¦  ¦
                             ------------------------------
                              0  0  1
```

Figure 48. Table Representation of x := a + b - c

There are two operations on the expression tables, addition and multiplication. The best way to understand these two operations is to look at the examples in Fig. 49 and Fig. 50.

```
  Expression table                      Expression table
 ------------------                 ------------------------
¦ a¦ b¦ c¦  ¦  ¦    ¦              ¦ d¦ e¦  ¦  ¦            ¦
¦--¦--¦--¦--¦--¦----¦      +       ¦--¦--¦--¦--¦------------¦
¦  ¦  ¦  ¦  ¦  ¦    ¦              ¦  ¦  ¦  ¦  ¦            ¦
 ------------------                 ------------------------
  0  0  0                            0  0

                     Expression table
                    -----------------
     =             ¦ a¦ b¦ c¦ d¦ e¦   ¦
                   ¦--¦--¦--¦--¦--¦---¦
                   ¦  ¦  ¦  ¦  ¦  ¦   ¦
                    -----------------
                     0  0  0  0  0
```

Figure 49. Addition of Expression Tables

The example above shows the addition of the expression tables [a+b+c] and [d+e], the result is an expression table with expression [a+b+c+d+e].

```
  Expression table                      Expression table
 ------------------                 ------------------------
¦ a¦ b¦ c¦  ¦  ¦    ¦              ¦ d¦ e¦  ¦  ¦            ¦
¦--¦--¦--¦--¦--¦----¦      *       ¦--¦--¦--¦--¦------------¦
¦ u¦  ¦  ¦  ¦  ¦    ¦              ¦  ¦  ¦  ¦  ¦            ¦
 ------------------                 ------------------------
  0  0  0                            0  0

                     Expression table
                    -----------------
                   ¦ a¦ a¦ b¦ b¦ c¦ c ¦
                   ¦--¦--¦--¦--¦--¦---¦
     =             ¦ u¦ u¦ d¦ e¦ d¦ e ¦
                   ¦--¦--¦--¦--¦--¦---¦
                   ¦ d¦ e¦  ¦  ¦  ¦   ¦
                    -----------------
                     0  0  0  0  0  0
```

Figure 50. Multiplication of Expression Tables

The multiplication of expression tables is more complicated than addition. An example of multiplying two expression tables [a*u+b+c] and [d+e] and returns the expression table [a*u*d+a*u*e+b*d+b*e+c*d+c*e] as a result appears in Fig. 50.

An example of expression table addition with negative sign is given in Fig. 51, and Fig. 52 is an example of multiplying two expression tables [a-b+c] and [u-w]. The sign bit is determined by the XOR of the sign bits of the two colunms which are being multiplied. Fig. 53 is the XOR table.

```
 Expression table                          Expression table
 ------------------                       ----------------------
¦ a¦ b¦ c¦  ¦  ¦    ¦                    ¦ d¦ e¦  ¦  ¦           ¦
¦--¦--¦--¦--¦--¦----¦         +          ¦--¦--¦--¦--¦-----------¦
¦ u¦  ¦  ¦  ¦  ¦    ¦                    ¦  ¦ w¦  ¦  ¦           ¦
 ------------------                       ----------------------
 0  0  0                                  0  1

                      Expression table
                     ----------------
                    ¦ a¦ b¦ c¦ d¦ e¦   ¦
      =             ¦--¦--¦--¦--¦--¦---¦
                    ¦ u¦  ¦  ¦  ¦ w¦   ¦
                     ----------------
                     0  0  0  0  1
```

Figure 51. Addition of Expression Tables
[a*u+b+c], [d-e*w]

```
   Expression table                               Expression table
 ------------------                            ----------------------
¦ a¦ b¦ c¦  ¦  ¦    ¦                         ¦ u¦ w¦  ¦  ¦            ¦
¦--¦--¦--¦--¦--¦----¦          *              ¦--¦--¦--¦--¦------------¦
¦  ¦  ¦  ¦  ¦  ¦    ¦                         ¦  ¦  ¦  ¦  ¦            ¦
 ------------------                            ----------------------
  0  0  1                                       0  1

                             Expression table
                            ------------------
                           ¦ a¦ a¦ b¦ b¦ c¦ c¦  ¦
     =                     ¦--¦--¦--¦--¦--¦--¦--¦
                           ¦ u¦ w¦ u¦ w¦ u¦ w¦  ¦
                            ------------------
                             0  1  0  1  1  0
```

Figure 52. Multiplying the Expression Tables of [a+b-c], [u-w]

XOR	1	0
0 ¦	1	0
1 ¦	0	1

Figure 53. XOR Table

After understanding the table translation of the assignment statements, we are ready to add semantic actions to the productions. The following is a revised grammar for assignment statement with semantic action added.

```
<assgn_stmt>    ----> <id> {input id into assignment
                           table} := <expr> {linked
                           expression table to assigned
                           id}
```

```
<expr>          ----> {create  expression table}
                      <term> <term_tail>
<term_tail>     ----> + <expr> { + expression table
                                 derived from <expr>}
<term_tail>     ----> - <expr> {set negative flag to true}
                               { + expression table}


<term_tail>     ----> epsilon
<term>          ----> <factor> <factor_tail>
<factor_tail>   ----> *  <term> { * expression table
                                 derived from <term>
                                 to the last <factor>}
<factor_tail>   ----> epsilon
<factor>        ----> <constant> {input constant into
                                  expression table}
                                 {if negative is true
                                  set  negative  sign to 1
                                  in expression table, set
                                  negative flag to false}
<factor>        ---->  <id>   {input  id  into  expression
                           table}
                          {if negative is true
                           set negative sign to 1 in
                           expression table, set
                           negative flag to false}
```

```
<factor>      ----> (<expr>)  {return expression table
                               derived from <expr>}
                              {if negative is true then
                               return expression table
                               with all the signs
                               changed; set negative flag
                               to false}
```

The semantic actions are inside the "{}", and they are added into the parsing procedures at the same location they are in the grammar. A semantic action can be implemented in one or more subroutines, depending on its complexity and function.

The following Figure is an example showing the generation of the expression tables from the semantic actions for the expression,

a - (b + c) * e

```
Remaining input                   Expression tables created

a - (b + c) * e                    ------------
                                  :            :
                                   ------------


- (b + c) * e                      -------------
                                  : a:          :
                                   -------------
                                    0
```

Figure 54. Generation of Expression Tables

```
(b + c) * e                         -------------
                                   | a|          |  negative
                                    -------------   flag set to
                                    0               ture

b + c) * e                          -----------   -----------
                                   | a|        |+| |         |
                                    -----------   -----------
                                    0             0

+ c) * e                            -----------   -----------
                                   | a|        |+| b|        |
                                    -----------   -----------
                                    0             0

c) * e                    -----------   -----------   --------
                         | a|        |+| b|        |+| |      |
                          -----------   -----------   --------
                          0             0

) * e                     -----------   -----------   --------
                         | a|        |+| b|        |+| c|     |
                          -----------   -----------   --------
                          0             0             0

* e                                 -----------   -----------
                                   | a|        |+| b| c|     |
                                    -----------   -----------
                                    0             0  0

* e                                 -----------   -----------
                                   | a|        |+| b| c|     |
                                    -----------   -----------
                                    0             1  1
                                    negative flag set to false

e                         -----------   -----------   --------
                         | a|        |+| b| c|     |*|        |
                          -----------   -----------   --------
                          0             1  1
```

Fig 54. (Continued)

```
 -----------     -----------     --------
 | a|            |+| b| c|          |*| e|         |
 -----------     -----------     --------
   0                1  1              0


                 -----------     -----------
                 | a|            |+| b| c|     |
                 -----------     |--|--|-----|
                   0             | e| e|      |
                                  -----------
                                    1  1


                 -----------
                 | a| b| c|  |
                 |--|--|--|--|
                 |  | e| e|  |
                 ----------
                   0  1  1
```

Figure 54. (Continued)

Translation Scheme for Multiple Statements

The difficulty of evaluating multiple statements is to keep track of the same variable in different statements. The value of a variable is defined by the latest executed statement in which the variable is assigned. Consider the statements,

```
x := w;
x := a;
y := x + b;
```

After the execution of the first statement, the value of 'w' is assigned to 'x', and then the value of 'x' is replaced by the value of 'a' in the second statement. What is the value of 'y' in the third statement ?

If all the above statements are executed sequentially, then the latest value which is assigned to 'x' is the value in variable 'a', therefore, the value of y is equal to the value of 'a' plus the value of 'b'. To be able to evaluate such multiple statements, the statements evaluation system must be able to keep track of the order of all assigned variables, that means the system must know the latest value which is assigned to the variable. The best way to keep track of the variables is to place the assigned variable into an assignment table. Let's look at the assignment table in Fig. 55 for the above example.

```
   Assignment table
   ----------------
  |                |
  |----------------|                Expression table
  |       x        |  ---->  ------------------------------
  |----------------|        | w|  |  |  |  |          |+/*|
  |       x        |         ------------------------------
  |----------------|  ---->  ------------------------------
  |                |        | a|  |  |                |+/*|
  |                |         ------------------------------
  |----------------|
  |       y        |  ---->  ------------------------------
  |----------------|        | a| b|  |                |+/*|
  |                |         ------------------------------
   ----------------
```

Figure 55. Assignment Table for x := w; x := a; y := x + b;

Every variable appearing in the right hand side of the assignment statement is replaced by its latest assigned value. Since 'w' and 'a' in "x := w" and "x := a" are not previously defined, 'w' and 'a' are placed into the expression table. For the assignment statement "y := x + b", by searching the assignment table sequentially for 'x' and 'b' we found that 'x' was previously defined twice, therefore the latest value 'a' is substituted for 'x' in the expression, and the expression for 'y' becomes "a + b". The substitution takes place before the variable is input into the expression table. The semantic action added to the grammar is as follows :

```
        <factor> ----> id {checks assignment table for id
                           ,if found in assignment table,
                           substituted the latest assigned
                           value for id}
                          {input id or assigned value
                           into expression table, if
                           negative is true then set
                           negative sign in expression
                           table; set negative flag to
                           false}
```

<u>Translation Scheme for Conditional Statement</u>

Because of the complexity and different variations of nested if-then-else statements, therefore, this study is restricted to the one level if-then-else statement with the condition in this format,

(variable conditional operator variable)

The following are the four conditional operators in the mini-language,

> < <> =

The translation process for if-then-else uses a similar process to translate sequential statements, except the statements are separated into two different assignment blocks, the then-block and the else-block. The then-block is an assignment table with all the assignment statements between the 'then' and the 'end' or between the 'then' and the 'else', see Fig. 56. On the other hand, the else-block is an assignment table with all the assignment statements between the 'else' and the 'end', see Fig 56. The execution flow of the statements is determined by the condition of the if-then-else, therefore, distinct comparisons will take place between the then-block of the template and the input, as well as between the else-block of the template and the input.

```
if (a >b) then                                    if ( a >b) then

   c := t;          }Then-block{                     c := t;
                    }          {
   u := a *b;       }          {                     u := a * b;

else                                              end if;

   a := b + c;      } Else-block

end if;
```

Figure 56. Then-block and Else-block

The system also recognizes the complement of the if-then-else statement, that means the statements in Fig. 57 are considered functionally equaivalent by the system. To determine the equivalence of complements of an if-then-else statement, the system performs cross comparisons between the then-block and else-block of the template and the input, and vice versa.

```
if ( a > b) then                if ( a <=b)
   a := a + 1;                     a := a - 1;
else                            else
   a := a - 1;                     a := a + 1;
end if;                         end if;
```

Figure 57. Cross Comparisons Between Complements

The source listing of the translator is in Appendix E.

CHAPTER VI

EXAMPLES OF DIFFERENT STATEMENTS AND RESPONSES

Introduction

This chapter demonstrates how the system responds to different types of statements. The questions and the template answers are provided to the system, and the possible valid inputs are shown in each example. When a student logs on to the system, he is asked to compose a program segment for a specific programming task. The answer from the student is then evaluated by the system. First the system separates statements into a stream of tokens by its lexical analyzer, then the parser checks the syntax of the statements. If the statements contain no syntactical errors, they are translated into table formats by the translator. The final procedure of the system is to compare all the tables from the input program segment with the template answer provided to the system. The answer is correct if the input and the template are matched, otherwise the input is incorrect, and error messages and the correct template answer are printed.

Declarations

Example 1

Question : Write a program segment to declare the variables a, b and c.

Answer template :

```
declare a, b, c.
```

Valid inputs :

```
1) declare a, b, c;
2) declare a, b;
   declare c;
3) declare a;
   declare b;
   declare c;
```

(a, b and c can be in any order of the three sets of inputs)

If the input is valid, the system notifies the user that the input statements are correct. On the other hand, if the input is invalid, the system notifies the user that the answer is incorrect and prints the correct answer. For example, if the input is

```
declare a. b, c;
```

the output from the system is

```
input          :
                 declare a. **syntax error** b, c;
```

```
error          : syntax error "," or ";" expected
correct answer :
                 declare a, b, c;
```

Example 2

With the same question in example 1, let's look at another input,

```
declare a;
declare x,c;
```

These input statements are syntactically correct, but they have not fulfilled the answer of the question, which is to declare variables a, b and c. The following is the output from the system,

```
input          :
                 declare a;
                 declare x,c;
error          : incorrect answer
correct answer :
                 declare a, b, c;
```

Assignment Statements

<u>Addition</u>

```
Question         : Write a program segment to
                   calculate the 'sum' of a, b, c.
Answer template  :
                   sum := a + b + c;
```

Valid inputs :

```
1) sum := a + b + c;
2) u := a + b;
   sum := u + c;
3) u := a;
   v := u + b;
   sum := v + c;
4) sum := a + (b + c);
5) sum := (a + b) + c;
```

(a, b and c can be in any order in the 5 sets of inputs)

Assignment expression tables are generated by the translator when the statements are parsed, and they are used to compare with the answer template during the evaluation phase. Above are the valid inputs for the question, the variables a, b and c in the five valid inputs can be in any order, and the variables u and v are arbitrary.

Subtraction

Question : Write a program segment to calculate the net profit 'n' from the sales 's', tax 't' and cost 'c'.

Answer template :

```
n := s - t - c;
```

Valid inputs :

```
1) n := s - t - c;
```

```
2) n := s - c - t;
3) n := (s - t) - c;
4) n := (s - c) -t;
5) u := s - t;
   n := u - c;
6) u := s - c;
   n := u - t;
7) n := s - (c + t);
8) u := c + t;
   n := s - u;
```

The above question is very easy, we just need to subtract the cost and the tax from the sales to get the net profit as in number 1 in the valid inputs. As you can see a simple task like this can have eight different valid answers. Like the valid answer in number 6, we can first calculate the profit from sales minus cost, and then come up with the net profit by subtracting the tax from the profit.

From the example above, you can also see that subtraction is more restricted than addition (addition is commutative but subtraction is not) . For example the statement 'a - b' is functionally different from the statement 'b - a'. But in addition, the statement 'a + b' is functinally equivalent to the statement 'b + a'. Therefore, unlike addition, the order of the operands makes a difference in the function of a subtraction statement.

<u>Multiplication</u>

Question : Write a program segement to calculate the simple interest 'i' from capital 'c', interest rate 'r' and number of years 'y'.

Answer template :

```
i := c * r * y;
```

Valid inputs :

```
1) i := c * r * y;
2) u := c * r;
   x := u * y;
3) u := c;
   v := u * r;
   i := v * y;
4) i := c * (r * y);
5) i := (c * r) * y;
```

(c, r and y can be in any order in the 5 sets of inputs)

The characteristics of addition and multiplication are very similar, they are both commutative, that means the changing the order of the operands of the statement does not affect the result of the statement.

We can calculate the interest by first getting the interest for one year, then multiply it by the number of years, see number 2. Or we can do the whole calculation in one program statement as in number 1.

<u>Compound Statement</u>

All the examples above are statements with one kind of operator. In this section, we are going to encounter statements with more than one kind of operator. Such statements are called compound statements. Because of different characteristics and priorities of different operators, a compound statement can have more variations of statement formats and are more complicated to evaluate. Examples of simple compound statements follow,

Example 1

Question : Write a program segment to calculate the area 'a' of the following figure.

```
 --------------------
:                    :
:                    : w
:                    :
 --------------------
   x    :   y   :   z
```

It is a rectangle with width 'w', and the length is divided into 3 sections, 'x', 'y' and 'z'.

Answer template:

```
a := w*x + w*y + w*z;
```

Valid inputs :

```
1) a := w*x + w*y + w*z;
2) a := w*(x + y + z);
```

```
3) a := w*x + w* (y + z);
4) u := x + y + z;
   a := u *w;
5) u := x + y;
   t := u + z;
   a := w * t;
6) u := w*x;
   t := w*y;
   v := w*z;
   a := u + t + v;
(x, y and z can be in any order)
(u, t, v are arbitrary variables)
```

We can solve the question by adding up the areas of the 3 smaller size rectangles, which combine together to form the big rectangle (see numbers 1 and 6). On the other hand, we can calculate the length of the rectangle by adding up all the section lengths together, x + y + z. Then we can come up the area by multiplying the length by the width 'w' (see numbers 2 and 4).

Only a few of the valid inputs are listed above.

Example 2

Question : Write a program segment to calculate the area 'Area' of the following figure.

```
 --------------------
|                    | a
|                    | -
|                    | b
 --------------------
   x      |      y
```

It is a rectangle with width a+b, and the length is divided into 2 sections, 'x' and 'y'.

Answer template:

```
Area := a*x + a*y + b*x + b*y;
```

Valid inputs :

```
1) Area := a*x + a*y + b*x + b*y;

2) Area := (a+b) * (x+y);

3) Area := a*(x + y) + b*(x + y);

4) Area := x*(a + b) + y*(a + b);

5) u := a + b;

   v := x + y;

   Area := u * v;

6) Area := a*(x + y) + b*x +b*y;

7) Area := x*(a +b) + y*a + y*b;
```

Example 3

Question : Write a program segment to calculate the shaded area 'Area' of the following figure.

```
               u
 ---------------------
¦////////////¦        ¦
¦////////////¦        ¦ c
¦////////////¦        ¦
 ---------------------
             ¦ a  ¦ b ¦
```

It is a rectangle with width c, and the length is u.

Answer template:

```
Area := u*c - a*c - b*c;
```

Valid inputs :

```
1) Area := u*c - a*c - b*c;

2) Area := c * (u - a - b);

3) Area := (u - (a + b)) * c;

4) Area := c * ( u - a -b);

5) x := a + b;
   y := u - x;
   Area := y * c;

6) x := u - a;
   y := x - b;
   Area := c * y;
```

Again only a few of the many variations are given above.

If-then-else Statements

Example 1

Question : Write a program segment to add 1 to a if a is negative, and subtract 1 from a if a is positive.

Answer template :

```
if (a > 0) then
        a := a - 1;
else
        a := a + 1;
end if;
```

Valid inputs :

```
1) if ( a > 0) then
        a := a - 1;
   else
        a := a + 1;
   end if;
2) if ( a <=0) then
        a := a + 1;
   else
        a := a - 1;
   end if;
```

From the above examples, you can see the number of variations of formats that a simple statement can have. This chapter only gives some of the simple examples for demonstrations. The more complex the statements are, the more variations they can have, and the more difficult to evaluate them.

CHAPTER VII

SUMMARY, FUTURE STUDY AND DEVELOPMENT

Summary

Purpose of this study was to create a statements evaluation system, which can be developed into an interactive tutorial system in evaluating input program segments and responding with evaluation messages and correct answers. The system served as a computer assisted instruction system in helping users in improving programming skills and techniques. Through the system, a student can learn from his past mistakes; he will be able to improve his logic and his skills in developing algorithm.

The implemented system is written in Pascal running on an IBM PC environment, and it is implemented to respond to the mini-language by Ledgard and Marcotty. The system is built from the ground floor; from construction of the LL(1) grammar for the mini-language to the code generation of the lexical analyzer, parser and translator. All the components of the system are described in detail including the design and implementation methods.

Future Study

Availability of future development surrounding the area of tutorial system in program improvement is unlimited. Technology is changing so fast that programming languages are constantly developing in order to become more powerful, easy to read/write, and faster in terms of compilation and execution time. Program improvement systems will become very helpful, both in formal classroom teaching and technical training. Proposed area of further research associates with the area :

1) creation of a fully automatic system by utilizing the compiler optimization technique;
2) development of an interactive system which is capable of comparing separate inputs from different users, so that, students will be able to learn from other students' programming techniques or mistakes;
3) research in the area of automated algorithm improvement system.

BIBLIOGRAPHY

Aho, A. V., and Ullman.,J. D. Principles of Compiler Design. Addison-Wesley Publishing Company, Massachusetts, 1977.

Alpert, D., and Bitzer, D. L. "Advances in Computer-Based Education." Science, Vol 167, 1970, 1582-90.

Atkinson, R. C., and Hansen, D. N. "Computer-Based Instruction in Initial Reading: The Stanford Project." Reading Research Quarterly, Vol 2, 1966, 5-25.

Baker, F. B. "Computer - based Instructional System: A first look." Review of Educational Research, Vol 41:1, 1971, 51-70.

Brooker, R. A., and Morris, D. "A general translation program for phrase structure languages." J. ACM, Vol 9:1, 1962, 1-10.

Brown, J. S., Burton, R. R., and Bell, A. G. "SOPHIE - A step toward creating a reactive learning environment." International Journal of Man-Machine, studies 7, 1975, 675 -696.

Bushnell, D. D. "The Computer as a Instructional Tool, A summary & SDC, Learning needs, Teaching Strategies." Santa Monica California, System Dev. Corp. ,Report No. SP-1554, 1964.

Cheatham, T. E. jr. and Sattley, K. "Syntax directed compiling." Proceedings AFIPS 1964, Spring Joint Computer Conference., Spartan Books, Baltimore, Md., 1964, 31-57.

Conway, R. W. "Design of a separable transition diagram compiler." Communication ACM, Vol 6:7, 1963, 396-408.

Eickel, J., Paul, M., Bauer, F. L., and Samelson, K. "A syntax controlled generator of formal language processors." Communication ACM, Vol 6:8, 1963, 451-455.

Feldman, J. A. "A formal semantics for computer languages and its application in a compiler-compiler." Communication ACM, Vol 9:1, 1966, 3-9.

Feldman, J. A., and Gries, D. "Translator writing systems." Communication ACM, Vol 11:2, 1968, 77-113.
Floyd, R. W. "Syntactic analysis and operator precedence." J. ACM, Vol 10:3, 1963, 316-333.

Floyd, R. W. "Bounded context styntactic analysis." Communication ACM, Vol 7:2, 1964, 62-67.

Fortran. Ansi Standard Fortran. American National Standards Institute, New York, 1966.

Gelder, J. I. Unpublished Chemistry simulation programs. Oklahoma State University, Department of Chemistry, (n.d.).

Graham, R. M. "Bounded context translation." Proceedings AFIPS Spring JCC , Vol 40, 1964, 205-217.

Ingerman, P. Z. A Syntax Oriented Translator. Academic Press, New York, 1966.

Irons, E. T. "A syntax directed compiler for ALGOL 60." Communication ACM, Vol 4:1, 1961, 51-55.

Jensen, K., and Wirth, N. Pascal User Manual and Report. Springer-Verlag, New York, 1975.

Kernighan, B. W. and Ritchie, D. M. The C Programming Language. Prentice-Hall, Inc., New Jersey, 1978.

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

Lagowski, J. J. "Computer-Assisted Instruction in Chemistry." In W. H. Holtzman (ed.), Computer-Assisted Instruction, Testing, and Guidance. Harper and Ron, New York, 1970, 283 - 298.

Ledgard, H. and Marcotty, M. The Programming Language Landscape. Science Research Associates, Inc., 1981.

Lewis, P. M. II, Rosenkrantz, D. J., and Stearns, R. E. Compiler Design Theory. Addison-Wesley, Massachuetts, 1976.

Lewis, P. M. II, and Stearns, R. E. "Syntax-directed transduction." J. ACM, Vol 15:3, 1968, 465-488.

Loughary, J. W. "Educational system requirements and society." address at conference on Educational information system requirements during the next decade. University of Oregon, August 6-10, 1967.

McClure, R. M. "TMG - a syntax directed compiler." Proceedings 20th ACM National Conference, 1965, 262-274.

Samelson K., and Bauer, F.L. "Sequential formula translation." Communication ACM, Vol 3:2, 1960, 76-83.

Schorre, D. V. "META-II : a syntax-oriented compiler writing language." Proceedings 19th acm National Conference, 1964 D1.3-1-D1.3-11.

Suppes, P. "Current Trends in Computer-Assisted Instruction." In M.C. Yovits(ed.), Advances in Computers, Vol. 18, Academic Press, New York, 1979, 173-229.

Suppes, P., Jerman, M., and Brain,D. Computer-Aided Instruction: Standford's 1965-1966 Arithmetic Program. Academic Press, New York, 1968.

Suppes, P., and Mroningstar, M. Computer-Assisted Instruction at Stanford, 1966-1968: Data, Models, and Evaluation of the Arithematic Programs. New York, Academic Press, 1972.

Wirth, N. and Weber, H. "EULER : a generalization of ALGOL and its formal definition : Part I." Communication ACM, Vol 9:1, 1966, 13-23.

APPENDIX A

CONTEXT-FREE GRAMMAR (MINI-LANGUAGE ---LEDGARD)

```
1.  <start>           ---->    <dec_seq> <stmt_seq>

2.  <dec_seq>         ---->    <declaration> <dec_tail>

3.  <dec_seq>         ---->    epsilon

4.  <declaration>     ---->    declare <id_list>

5.  <dec_tail>        ---->    ; <dec_seq>

6.  <id_list>         ---->    <id> <id_list_tail>

7.  <id_list_tail>    ---->    , <id_list>

8.  <id_list_tail>    ---->    epsilon

9.  <stmt_seq>        ---->    <statement> ; <stmt_tail>

10. <stmt_tail>       ---->    <statement> ; <stmt_tail>

11. <stmt_tail>       ---->    epsilon

12. <statement>       ---->    <assgn_stmt>

13. <statement>       ---->    <if_stmt>

14. <statement>       ---->    <loop_stmt>

15. <assgn_stmt>      ---->    <id> := <expr>

16. <if_stmt>         ---->    if <comparison> then

                                     <stmt_seq>

                               <endif_else>
```

```
17. <endif_else>        ---->    end if

18. <endif_else>        ---->    else

                                     <stmt_seq>

                                 end if

19. <loop stmt>         ---->    while <comparison> loop

                                      <stmt_seq>

                                 end loop


20. <comparison>        ---->    ( <factor> <comp_tail>

21. <comp_tail>         ---->    = <factor> )

22. <comp_tail>         ---->    <> <factor> )

23. <comp_tail>         ---->    < <facotr>  )

24. <comp_tail>         ---->    > <facotr>  )


25. <expr>              ---->    <term> <term_tail>

26. <term_tail>         ---->    + <expr>

27. <term_tail>         ---->    - <expr>

28. <term_tail>         ---->    epsilon

29. <term>              ---->    <factor> <factor_tail>

30. <factor_tail>       ---->    * <term>

31. <factor_tail>       ---->    epsilon

32. <factor>            ---->    <constant>

33. <factor>            ---->    <id>

34. <factor>            ---->    ( <expr> )
```

APPENDIX B

TERMINAL AND NON-TERMINAL SYMBOLS OF MINI-LANGUAGE (LEGARD)

The numbers on the left-hand side of the symbols are the internal representation numbers of the symbols.

TERMINAL SYMBOLS		NON-TERMINAL SYMBOL	
1	declare		
2	id	22	<dec_seq>
3	if	23	<dec_tail>
4	then	24	<id_list>
5	,	25	<id_list_tail>
6	;	26	<stmt_seq>
7	end	27	<stmt_tail>
8	else	28	<statement>
9	(	29	<assgn_stmt>
10	)	30	<if_stmt>
11	=	31	<endif_else>
12	<>	32	<loop_stmt>
13	<	33	<comparison>
14	>	34	<comp_tail>
15	+	35	<expr>
16	-	36	<term_tail>
17	*	37	<term>
18	constant	38	<factor>
19	while	39	<factor_tail>
20	loop	40	<operand>
21	:=	41	<start>

APPENDIX C

Basic Background for LL(1) grammar and predict set

Let's take a look at a LL(1) grammar for the mini-language in Appendix A, the production for "dec_seq" :

```
<dec_seq>      -----> <declaration> <dec_tail>
<dec_seq>      -----> epsilon
```

In defining the parsing procedure corresponding to <dec_seq> we run into a problem: More than one production has <dec_seq> as a left hand side in the Grammar. We must decide what production to try to match. If we try to match the first production and fail, it is too late to try the second now since we have already consumed the input tokens. We therfore peek ahead one token (without deleting it) and use this lookahead symbol to decide what production to choose. Consider the production :

```
A ----> X1 X2 ... Xm
```

For what lookahead tokens should we decide to try this production? We need the set of all possible lookahead tokens that might indicate that this "A" production is to be matched, and none other. Sine a lookahead is only a single token, we want the set of first (leftmost) tokens that could be produced from the string X1 X2 ... Xm. We call this set first(X1...Xm). If the leftmost symbol X1 is a terminal, then clearly, first(X1...Xm) = X1. However, if X1 is a nonterminal, then first(X1...Xm) will depend on what terminals X1 can generate. So we begin by computing "first" for each right hand side corresponding to X1.

For example, the production of X1 is,

```
X1 ----> Y1 Y2 ... Yn
X1 ----> Z1 Z2 ... Zm
```

Since X1 has 2 productions, therefore the set of first(Y1) and first(Z1) will be included in first(X1...Xm).
What if X1 can generate epsilon?

```
A  ----> X1 X2 .... Xm
X1 ----> Y1 Y2 ... Yn
X1 ----> Z1 Z2 ... Zm
X1 ----> epsilon
```

Then first (X1...Xm) depends on X2 as well. In particular, if X2 is a terminal, it is then included in first(X1...Xm). If it is a non-terminal, we compute "first" for each of its corresponding right hand sides. Similarly, if both X1 and X2 can produce epsilon, we consider X3, and so on. What if the entire right hand side can produce epsilon?

```
A ----> epsilon
   or
X1 ... Xm -----> epsilon
```

The look ahead will then be determined by those terminals that can follow the left hand side ("A" in our example). We define a set of tokens follow(A) equal to those tokens that can follow "A" in some legal derivation. As an example, if the grammar has

```
Z ----> Y1 c Y2 ... A t ... Ym       Y1, Y2,...Ym, A
                                     are non-terminals
                                     c,t are terminals

X ----> V1 ... A B ... Vn            V1...Vn, A, B are
                                     non-terminals
B ----> a

B ----> b
```

as productions, then "t" will be in follow(A). Further, the terminals a, b in the "first" sets of all the right hand side of the B-productions will be in follow(A). The set follow(A) will have t, a and b. We now define the set of lookahead tokens that will cause the prediction of the production

```
A ----> X1 .... Xm
```

Call this set Predict. As we have seen,

```
        predict(A ----> X1 ... Xm) =
                first(X1 ... Xm) + (if X1 ... Xm ----> epsilon
                                     then follow(A))
```

That is, any token that can be the first symbol produced by the right hand side of a production will predict that production. Further, if the entire right hand side can produce epsilon, then tokens that can immediately follow the left hand side of a production will also predict that production.

We use predict to figure out which production to use if there is a choice. We may now have three cases :

1. The lookahead token is in the predict set of exactly one production. In this case, we choose the predicted production.
2. The lookahead is in the predict set of no production. In this case, clearly, the lookahead token occurs in an illegal position, so we have a syntax error.
3. The lookahead token is in the predict set of more than one production. This is not indicative of any error in the input string; it is, rather, a property of the grammar. We can analyze the grammar even before we start parsing to determine if some token can be in the predict set of more than one production. Such a CFG cannot be parsed by recursive descent, and some other, more powerful technique may have to be used.

Therfore, we will parse only those context-free grammar that have disjoint predict sets for productions that share a common left hand side. context-free grammar that obey this restriction are called LL(1) grammar. Appendix A is the LL(1) grammar for the mini-language. Since a language may be generated by more than one grammar, it may still be possible to write another grammar for the same language that has the LL(1) property. The following is the formal definition of an LL(1) grammar.

A grammar G is LL(1) if and only if

for all rules $A \longrightarrow \alpha 1 \mid \alpha 2 \mid \ldots \alpha n$,

1. $first(\alpha i) \cap first(\alpha j) = \emptyset$ for all $i <> j$

   and, furthermore, if $\alpha i \xrightarrow{*} epsilon$, then

2. $first(\alpha j) \cap follow(A) = \emptyset$ for all j.

The first and follow sets used in this definition are the same sets we defined before, and they can be defined in mathematical terms as follows. Given some string $\alpha \in V^*$, the set of terminal symbols given by $first(\alpha)$ represent the leftmost derivable symbols of a and this set is given by the equation

$$first(\alpha) = \{w \mid \alpha \xrightarrow{*} w \ldots \text{ and } w \in V\}$$

The follow sets are defined for a nullable nonterminal A (one which can produce the empty string). The definition for the follow sets is given by follow(A) = $\{w \in V : S' \xrightarrow{*} \alpha A \beta\}$

where $w \in \text{first}(\beta)$ and S is the start symbol of the grammar.

The predict sets of all the non-terminals in the mini-language are in the LL(1) parse table in Appendix D.

APPENDIX D

LL(1) PARSE TABLE FOR MINI-LANGUAGE (LEDGARD)

The following is the LL(1) Parse Table for Mini-Language (Ledgard Henry). For each non-terminal symbol, a list of terminals and the productions they predict are listed. Terminals not listed predict no production and thus are erroneous.

<start>	symbol #	production #
declare	1	1
id	2	1
if	3	1
while	19	1

<dec_seq>	symbol #	production #
declare	1	2
id	2	3
if	3	3
while	19	3

<declaration>	symbol #	production #
declare	1	4

<dec_tail>	symbol #	production #
;	6	5

<id_list>	symbol #	production #
id	2	6

<id_list_tail>	symbol #	production #
,	5	7
;	6	8

<stmt_seq>	symbol #	production #
id	2	9
if	3	9
while	19	9

<stmt_tail>	symbol #	production #
id	2	10
if	3	10
while	19	10
else	8	11
end	7	11
end of input		11

<statement>	symbol #	production #
id	2	12
if	3	13
while	19	14

<assgn_stmt>	symbol #	production #
id	2	15

<if_stmt>	symbol #	production #
if	3	16

<endif_else>	symbol #	production #
end	7	17
else	8	18

<loop_stmt>	symbol #	production #
while	19	19

<comparison>	symbol #	production #
(	9	20

<comp_tail>	symbol #	production #
=	11	21
<>	12	22
<	13	23
>	14	24

<expr>	symbol #	production #
constant	18	25
id	2	25
(	9	25

<term_tail>	symbol #	production #
+	15	26
-	16	27
;	6	28

<term>	symbol #	production #
constant	18	29
id	2	29
(	9	29

<factor>	symbol #	production #
constant	18	32
id	2	33
(	9	34

<factor_tail>	symbol #	production #
*	17	30
+	15	31
-	16	31
;	6	31

APPENDIX E

Source Listing of translator

```
procedure syntaxerror(messcode : integer); forward;
procedure dec_seq(var ptr: tokenptr); forward;
procedure declaration(var ptr : tokenptr); forward;
procedure dec_tail(var ptr : tokenptr); forward;
procedure id_list(var ptr : tokenptr); forward;
procedure id_list_tail(var ptr : tokenptr) ; forward;
procedure stmt_seq(var ptr : tokenptr); forward;
procedure stmt_tail(var ptr : tokenptr); forward;
procedure statement(var ptr : tokenptr); forward;
procedure assgn_stmt(var ptr : tokenptr); forward;
procedure if_stmt(var ptr : tokenptr); forward;
procedure endif_else(var ptr : tokenptr); forward;
procedure  expr(var  ptr  :  tokenptr;var  idhead,idtail
:varptr); forward;
procedure  term_tail(var  ptr  :  tokenptr;var idhead,idtail
:varptr;var multi:boolean;var old :varptr); forward;
procedure  term(var  ptr  :  tokenptr;var  idhead,idtail
:varptr;var multi:boolean;var old : varptr); forward;
procedure factor_tail(var ptr :  tokenptr;var idhead,idtail:
varptr;var multi:boolean;var old : varptr); forward;
procedure  factor(var  ptr  :  tokenptr;var  idhead,idtail :
varptr;var multi:boolean;var old : varptr); forward;
procedure comparison(var ptr : tokenptr); forward;
procedure comp_tail(var ptr : tokenptr); forward;
procedure loop_stmt(var ptr : tokenptr); forward;
procedure  merge(var   heada,taila,headb,tailb:   varptr);
forward;


procedure match(var ptr : tokenptr;num : integer; messcode :
integer);
var
   i : integer;
begin
     if error = false then
     begin
          if next(ptr) = num then
                 ptr := ptr^.link
          else
              begin
                 error := true;
```

```
                error := true;

                ptr :=  ptr^.link;          {skip the error
                                          token}
            end;
        i := 1;
        while (ptr^.sym[i] <>' ') and(i <= 7) do
        begin
            write(trm,ptr^.sym[i]);
            i := i + 1;
        end;
        write(trm,' ');


        if error = true then syntaxerror(messcode);
        if ptr^.sym[1] = ';' then writeln(trm);
    end;
end;




procedure syntaxerror;

begin
    case messcode of
    1    : writeln(trm,'**declarations or statements
 expected** ');
    2    : writeln(trm,'**id expected**');
    3    : writeln(trm,'**if expected**');
    4    : writeln(trm,'**then expected**');
    5    : writeln(trm,'**"," expected**');
    6    : writeln(trm,'**";" expected**');
    7    : writeln(trm,'**end expected**');
    8    : writeln(trm,'**else expected**');
    9    : writeln(trm,'**"(" expected**');
    10   : writeln(trm,'**")" expected**');
    11   : writeln(trm,'**"=" expected**');
    17   : writeln(trm,'**"*" expected**');
    18   : writeln(trm,'constant expected');
    19   : writeln(trm,'**while expected**');
    20   : writeln(trm,'**loop expected**');
    21   : writeln(trm,'**":=" expected**');
    25   : writeln(trm,'**assgn,if_then_else,while_loop
           statements expected');
    26   : writeln(trm,'**"," or ";" expected');
    27   : writeln(trm,'**constant,id,or "(" expected**');
    28   : writeln(trm,'**relational operator expected**');

    end;
    writeln('***Execution terminated***');
    error := true;
```

```
end;


procedure initvar(var idrec : varptr);

var
   i : integer;

begin
     idrec^.link := nil;
     for i := 1 to 10 do
         idrec^.id[i] := '          ';
     idrec^.len := 0;
end;



procedure subsit(var head, tail : varptr; var ptr :
tokenptr);
var
   temphead,temptail : varptr;
   i,j : integer;
   tempptr,loc : varptr;
begin
     new(temphead);
     initvar(temphead);
     temptail := nil;
     for i := 1 to expnum - 1 do
     begin
          if exphead[i]^.link^.id [1] = ptr^.sym then
          begin
               loc := exphead[i]^.link^.link;
               while loc <> nil do
               begin
                    new(tempptr);
                    initvar(tempptr);
                    for j := 1 to loc^.len do
                        tempptr^.id[j] := loc^.id[j];
                    tempptr^.len := loc^.len;
                    if temphead^.link <> nil then
                    begin
                       temptail^.link := tempptr;
                       temptail := tempptr;
                    end
                    else
                    begin
                       temphead^.link := tempptr;
                       temptail := tempptr;
                    end;
                    loc := loc^.link;
               end;
          end;
     end;
     if temptail = nil then
          insert(head,tail,ptr)
     else
          merge(head,tail,temphead,temptail);
end;




procedure merge;
begin
     if heada^.link <> nil then
     begin
      taila^.link := headb^.link;
      taila := tailb;
     end
     else
     begin
      heada^.link := headb^.link;
      taila := tailb;
     end;

end;

procedure  concat(var   heada,taila,old,   headb,   tailb  :
varptr);
var
     ptr1, ptr2 : varptr;
     temphead,temptail : varptr;
     ptr,save : varptr;
     i : integer;

begin
     new(temphead);
     initvar(temphead);
     new(temptail);
     initvar(temptail);
     ptr1 := old^.link;

     while ptr1 <> nil do
     begin
          ptr2 := headb^.link;
          while ptr2 <> nil do
          begin
               new(ptr);
               initvar(ptr);
               if temphead^.link = nil then
               begin
```

```
                  temphead^.link := ptr;
                  temptail := ptr;
               end
               else
               begin
                    temptail^.link := ptr;
                    temptail := ptr;
               end;
               for i := 1 to ptr1^.len do
               begin
                    ptr^.len := ptr^.len + 1;
                    ptr^.id[ptr^.len] := ptr1^.id[i];
               end;

               for i := 1 to ptr2^.len do
               begin
                    ptr^.len := ptr^.len + 1;
                    ptr^.id[ptr^.len] := ptr2^.id[i];
               end;
               ptr2 := ptr2^.link;
          end;
          ptr1 := ptr1^.link;
     end;
     save := old;
     merge(heada,old,temphead,temptail);
     taila := temptail;
     old := save;
 end;


procedure printdec(dechead : varptr);

var
   ptr : varptr;

begin
     writeln('********************');
     ptr := dechead^.link;
     while ptr <> nil do
     begin
          writeln(ptr^.id[1]);
          ptr := ptr^.link;
     end;
end;


procedure printid(var exphead : vartable);

var
   ptr : varptr;
   i : integer;          j : integer;

begin
```

```
          writeln;

          writeln('***The identifier***');
          for i := 1 to expnum do
          begin
               ptr := exphead[i]^.link;
               while ptr <> nil do
               begin
                    if ptr^.len > 0 then
                    begin
                         for j := 1 to ptr^.len do
                         write(ptr^.id[j]);
                         writeln;

                  end;
                    ptr := ptr^.link;
               end;
          end;
     end;



     procedure id_list_tail;

     begin
          if error = false then

          case next(ptr) of
          6: ;                                {;}
          else   begin
                       match(ptr,5,26);       {,}
                       id_list(ptr);

                 end;

          end;

     end;

     procedure dec_tail;

     begin
          if error = false then
          begin
               match(ptr,6,6);          {;}
               dec_seq(ptr);

          end;

     end;
```

```
procedure declaration;

begin
     if error = false then
     begin
          match(ptr,1,1);              {declare}
          id_list(ptr);
     end;
end;



procedure dec_seq;

begin
     if error = false then

          case next(ptr) of
          2,3,19 : ;     {id, if, while}
          else begin
                   declaration(ptr);
                   dec_tail(ptr);
              end;
          end;

end;


procedure stmt_seq;

begin
     if error = false then
     begin
          statement(ptr);
          match(ptr,6,6);                      {;}
          stmt_tail(ptr);

     end;
end;


procedure stmt_tail;

begin
     if error = false then
          if ptr^.link <> nil then       {not end of input}

               case next(ptr) of
               7 : ;                     {end}
```

```
                  else begin
                        statement(ptr);
                        match(ptr,6,6);      {;}
                        stmt_tail(ptr);
                     end;
                  end;
end;



procedure statement;

begin
   case next(ptr) of
      2    : assgn_stmt(ptr);                   {id}
      3    : if_stmt(ptr);                      {if}
      19   : loop_stmt(ptr);                    {while}
      else   syntaxerror(25);
   end;

end;



procedure assgn_stmt;

var
  idhead,idtail : varptr;
  printptr : varptr;

begin
   expnum := expnum + 1;
   new(idhead);
   initvar(idhead);
   idtail := nil;
   if error = false then
   begin
      match(ptr,2,2);      {id}

insert(exphead[expnum],exptail[expnum],ptr);
      match(ptr,21,21);                         {:=}
      expr(ptr,idhead,idtail);

merge(exphead[expnum],exptail[expnum],idhead,idtail);
   end;

end;
procedure if_stmt;

begin
   if error = false then
   begin
```

```
          match(ptr,3,3);                                          {if}

          comparison(ptr);
          match(ptr,4,4);                                        {then}

          stmt_seq(ptr);
          endif_else(ptr);
     end;

end;


procedure endif_else;

begin
     if error = false then
     begin
          case next(ptr) of
               7 : begin                                          {end}

                        match(ptr,7,7);
                        match(ptr,3,3);                            {if}
                   end;
               8 : begin
{else}
                        match(ptr,8,8);
                        stmt_seq(ptr);
                        match(ptr,7,7);
{end}
                        match(ptr,3,3) ;
{if}
                   end;
          end;
      end;

end;



procedure expr;
var
   multi : boolean;
   old : varptr;
begin             multi := false;          old := nil;
     if error = false then
     begin
          term(ptr,idhead,idtail,multi,old);
          term_tail(ptr,idhead,idtail,multi,old);
     end;
end;
```

```
procedure term_tail;

begin
     if error = false then
     begin
          case next(ptr) of
          15 : begin
                    match(ptr,15,15);   {+}
                    expr(ptr,idhead,idtail);
               end;

          16 : begin
                    match(ptr,16,16);   {-}
                    expr(ptr,idhead,idtail);
               end;
          6  : ;                   {;}
          else ;
          end;
     end;
end;


procedure term;
begin
     if error = false then
     begin
          factor(ptr,idhead,idtail,multi,old);
          factor_tail(ptr,idhead,idtail,multi,old);
     end;
end;



procedure factor;

var
   newhead : varptr;
   newtail : varptr;
   temphead, temptail : varptr;
   yes : boolean;

begin
     if error = false then
     begin

     case next(ptr) of
     18 : begin
               match(ptr,18,18);       {constant}
               if multi = true then
```

```
                    begin
                         new(temphead);
                         initvar(temphead);
                         new(temptail);
                         initvar(temptail);
                         insert(temphead,temptail,ptr);

concat(idhead,idtail,old,temphead,temptail);
                         multi := false;
                    end
                    else



                   begin
                         if idtail = nil then
                            old := idhead
                         else
                            old := idtail;
                         insert(idhead,idtail,ptr);
                    end;


          end;
     2    : begin
               match(ptr,2,2);          {id}
               if multi = true then
                    begin
                         new(temphead);
                         initvar(temphead);
                         new(temptail);
                         initvar(temptail);
                         subsit(temphead,temptail,ptr);
              concat(idhead,idtail,old,temphead,temptail);
                         multi := false;
                    end
                    else
                    begin
                         if idtail = nil then
                            old := idhead
                         else
                            old := idtail;
                         subsit(idhead,idtail,ptr);
                    end;
          end;
     9    : begin
               match(ptr,9,9);          {(}
                    new(newhead);
                    initvar(newhead);
                    newtail := nil;
               expr(ptr,newhead,newtail);
                    if multi = true then
                    begin
concat(idhead,idtail,old,newhead,newtail);
                         multi := false;
                    end
                    else
                    begin
                         if idtail = nil then
                            old := idhead
                         else
                            old := idtail;



                    merge(idhead,idtail,newhead,newtail);
                    end;
               match(ptr,10,10);        {)}
          end;
     else match(ptr,0,27);          {skip error token}
     end;
     end;
end;


procedure factor_tail;

begin
     if error = false then
     begin
     case next(ptr) of
     15,16,6 : ;             {+,-,;}
     17 : begin              {*}
               match(ptr,17,17);       multi := true;
               term(ptr,idhead,idtail,multi,old);
          end;
     else ;
     end;
     end;

end;

procedure loop_stmt;

begin
     if error = false then
     begin
          match(ptr,19,19);     {while}
          comparison(ptr);
          match(ptr,20,20);     {loop}
          stmt_seq(ptr);
          match(ptr,7,7);       {end}
          match(ptr,20,20);     {loop}
     end;

end;


procedure comparison;

var
   idhead, idtail,old : varptr;multi :boolean;
begin
     if error = false then
     begin
          match(ptr,9,9);        {(}
          factor(ptr,idhead,idtail,multi,old);
          comp_tail(ptr);
     end;
end;




procedure comp_tail;
var
idhead,idtail,old : varptr;
            multi : boolean;

begin
     if error = false then

          case next(ptr) of
          11 : begin
                    match(ptr,11,11);     {=}
                    factor(ptr,idhead,idtail,multi,old);
                    match(ptr,10,10);     {)}
               end;
          12 : begin
                    match(ptr,12,12);     {<>}
                    factor(ptr,idhead,idtail,multi,old);
                    match(ptr,10,10);     {)}
               end;
          13 : begin
                    match(ptr,13,13);     {<}
                    factor(ptr,idhead,idtail,multi,old);
                    match(ptr,10,10);     {)}
               end;
          14 : begin
                    match(ptr,14,14);     {>}
                    factor(ptr,idhead,idtail,multi,old);
                    match(ptr,10,10);     {)}
               end;
          else syntaxerror(28);
          end;
```

```
end;



procedure start(var ptr : tokenptr);

begin
     error := false;
     expnum := 0;
     dec_seq(ptr);
                  printdec(dechead);
     stmt_seq(ptr);
                  printid(exphead);
end;
```

2

VITA

Peter Yu Yee Tsang

Candidate for the Degree of

Master of Science

Thesis: A STATEMENTS EVALUATION SYSTEM FOR FUNCTIONALLY EQUIVALENT RESPONSES

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Hong Kong, October 8, 1960, the son of Yue Lap and Kwai Chi Tsang.

Education: Graduated from Chan Sui Ki (La Salle) College, Hong Kong, in May, 1980. Attended University of Wisconsin - Milwaukee from June, 1981 to May, 1982. Received a Bachelor of Science degree in Computer Science and Mathematics from University of Wisconsin - Madison, May, 1984. Completed the requirements for a Master of Science degree in Computing and Information Science at Oklahoma State University, May, 1987.

Professional Experience: Reasearch Assistant, Department of Computer Science, University of Wisconsin - Madison, December 1984 to May 1985; Computer Tutor, Oklahoma State University, December 1984 to December 1985; Computer Programmer, Bursar Office, Oklahoma State University, December 1985 to May 1987.