

CORRECTNESS AND LEVEL OF INCORRECTNESS
DETERMINATIONS OF PROGRAM SEGMENTS;
FUNCTIONAL EQUIVALENCE OF
WHILE-DO STATEMENTS

by

KAY ELLEN SLACK

Bachelor of Science

East Central Oklahoma State University

Ada, Oklahoma

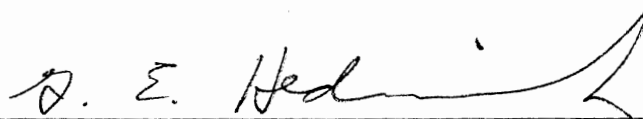
1984

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
December, 1988

thesis
1988
S631c
cop. 2

CORRECTNESS AND LEVEL OF INCORRECTNESS
DETERMINATIONS OF PROGRAM SEGMENTS;
FUNCTIONAL EQUIVALENCE OF
WHILE-DO STATEMENTS

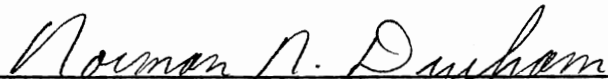
Thesis Approved:



Thesis Advisor







Dean of the Graduate College

PREFACE

This study is concerned with the expansion of the Statements Evaluation System which evaluates the functional equivalence of program segments. Program segments are input by the student user and compared to a set of expected responses (templates) developed by the instructor. It is believed that such a system will aid the student user in gaining knowledge through hands-on experience, and trial and error situations.

The motivation for this study was provided by my committee members: Dr. Donald D. Fisher, Dr. K. M. George, Dr. Michael Folk. The idea originated with Dr. Donald D. Fisher, who is my major advisor. I would like to express my appreciation to each member for the guidance he gave throughout this study. Finally, I would like to express my gratitude to my family members and my current employers for their support and encouragement.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
II. LITERATURE REVIEW	4
Functional Equivalence	4
Review of the Original SES	4
Computer-Based Instruction	7
III. STATEMENT OF THE PROBLEM	10
IV. DESIGN AND IMPLEMENTATION	11
The CORE Program Segment	11
The CORE Compiler	12
The Lexical Analyzer	12
YACC - Input Parsing	16
Equivalence Determination	20
Preliminary Matching	28
Basic Matching	31
Functional Equivalence	35
V. RESULTS OF THE STUDY	39
Equivalent and Nonequivalent Programs	39
Suggestions for Future Study	42
Summary	43
BIBLIOGRAPHY	44
APPENDIX A - SES INSTRUCTOR'S OPERATION MANUAL	46
APPENDIX B - SES STUDENT USER'S MANUAL	59
APPENDIX C - MINI-LANGUAGE CORE SYNTACTIC CATEGORIES	68

LIST OF TABLES

Table	Page
1. Numerical Representation of Token Sequence	14
2. Quadruple Numerical Statement Identifiers	19
3. Internal Numerical Representation of Operators	25
4. Single Character Representation of CORE Operators	27

LIST OF FIGURES

Figures	Page
1. Information Flow of the Original SES	5
2. Overview of Current SES	11
3. The CORE Compiler	12
4. Creating the Lexical Analyzer with LEX	12
5. Generation of Token Sequence	13
6. Program Transformation in the SES	21
7. Pictorial DAG Representation of Array Information . . .	26
8. Phases of Equivalence Determination	28

CHAPTER I

INTRODUCTION

This paper investigates use of the computer to determine the correctness of a program segment designed to perform a specific task designated by a given set of instructions. The idea is to present the user with a problem statement and evaluate his/her response against a series of correct and incorrect templates. As opposed to requiring a direct match to an existing template, this system attempts to determine the functional equivalence of the program segment. Determining functional equivalence is a many faceted problem and is, in general, unsolvable. Using such an approach, however, provides freedom for expansion and enhancement of the user's current logical thinking patterns while developing a program to perform the given task. This is a definite advantage over conventional computer-based instructional techniques which limit the user to choosing or developing a single response as an exact match to a given answer. Should a system such as this be paired with the capabilities of artificial intelligence, a powerful instructional tool could be developed for use in and out of the conventional classroom setting. While the current system makes no use of artificial intelligence responses to the user, it provides an excellent base for such expansion. In addition to informing the user that his/her program was found to be functionally equivalent to a correct or incorrect response, the computer could detect errors in the

program segment, point these out to the user, and offer suggestions for correction.

The current system, hereafter referred to as the Statements Evaluation System (SES), is implemented to respond to the mini-language CORE, developed by Ledgard and Marcotty (12). Specifications of this language are provided in the appendix. The mini-language allows the three basic classes of statements (assignment, iteration, selection), but reduces the membership of the iteration class to one member, that of the while loop. The language also restricts the usage of types to the type integer. It does allow interactive programming through the use of the input/output statements, and allows nesting of if statements and while loops.

The system in this project is an expansion of the original SES developed by Tsang (3). There are a number of differences, including the approach taken to solve the problem, and the addition of the while loop to the system. The original system determined equivalence for assignment, declaration, and selection statements only.

As an example of the complexity of the problem of trying to determine the correctness of a program segment, consider the problem of summing the integers from one to ten. A number of different correct segments exist which satisfy this problem. One correct segment which might be used as a template against which responses are compared follows.

```
program
declare x,y;
begin
  x := 0;
  y := 0;
  while x < 10 do
    x := x + 1;
    y := y + x;
```

```
end;
end;
```

Two other program segments that might be generated to form the sum are given below.

```
1) program
   declare x,y;
   begin
     x := 1;
     y := 0;
     while x <= 9 loop
       y := y + x;
       x := x + 1;
     end;
     y := y + x;
   end;

2) program
   declare a,b,c,d;
   begin
     input a;
     b := 10;
     c := 1;
     d := 0;
     while (b >= a) loop
       d := d + a;
       a := a + c;
     end;
   end;
```

The above program segments are only two of the many variations that could be presented as correct responses. The problem is to determine whether a response program segment is "correct" or "incorrect". The student program segment is compared against a series of stored comparison templates. Multiple correct and multiple anticipated incorrect comparison templates are entered into the SES by an instructor. A student response segment is compared with correct comparison templates and if no match occurs, the response is compared with any anticipated incorrect templates.

When considering the functional equivalence of loops, major factors that must be considered are:

- a) the initial value of variables,
- b) the conditional operator,
- c) the value of the condition variable,
- d) the body of the loop,
- e) any assignment statements preceding and/or succeeding the loop,
- f) the number of iterations of the loop.

CHAPTER II

LITERATURE REVIEW

Functional Equivalence

In recent years interest and research in the areas of equivalence and functional equivalence have increased. The equivalence problem is, given two programs, to decide whether or not they produce the same outputs for identical inputs (1). It has been shown that equivalence is decidable for $\{x \leftarrow 1, x \leftarrow x + y, x \leftarrow x * y, x \leftarrow x/y\}$ -programs (straight-line programs) with one input variable (2). Work has also been done to determine the equivalence of conditional statements and loop statements. It has been stated that the equivalence problem for loops is unsolvable (1). This paper deals with the topic of functional equivalence, in particular with relation to the loop construct. Throughout this study, the problem of functional equivalence will be to decide whether two syntactically and semantically correct program segments, written in the same language to perform the same task, produce the same results.

Review of the Original SES

The original statements evaluation system was developed by Peter Yu Yee Tsang. The idea itself originated with Dr. Donald D. Fisher and the intent is to continually expand upon the system until it becomes one that can be used as a teaching aid in both academic and industrial

fields. The initial SES, developed to run on an IBM PC, tests the equivalence of the mini-language CORE declaration, assignment, and selection statements. The system contains a lexical analyzer and recursive descent parser written in the Pascal language. Expression tables are used to determine functional equivalence. The stages of the SES are as follows:

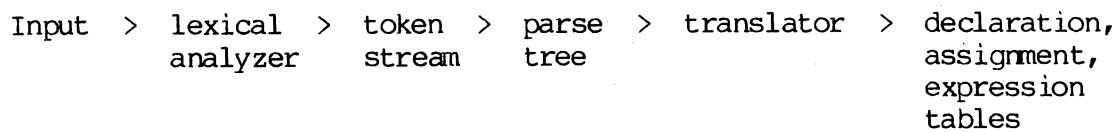


Figure 1. Flow of Control in Original SES

For example, to test the equivalence of the following two program segments, the translator would build three tables, A, B, and C.

```

1) declare x,y;           2) declare y;
   x := y + z * w;       declare x;
                           x := w * z + y;
  
```

(for program 1):

A. Declaration Table	B. Assignment Table	C. Expression Table
x integer	x	y z
y integer		w
		0 0

(for program 2):

A. Declaration Table	B. Assignment Table	C. Expression Table
y integer	x	w y
x integer		z
		0 0

The rows of the expression table represent addition while the columns represent multiplication. The last row is a sign bit for each column. This bit is set to zero if the column value is positive, and

set to one if the column value is negative. Two declaration tables are equivalent if they have the same variables in their tables, regardless of their order. Two expression tables are equivalent if they have the same elements, regardless of the order of the columns or order of the rows of each individual column. If given the template:

```

program
declare x,y,z;
begin
  x := z + y;
  if x > y then
    x := z * (y + 1);
  else
    x := z * (y - 1);
  end if;
end;

```

the original SES would be able to determine that the following program segments were equivalent to the template:

- | | |
|---|---|
| <pre> 1) program declare x,z,y; begin x := z + y; if x > y then x := (y + 1) * z; else x := y * z - 1 * z; end if; end; </pre> | <pre> 2) program declare y,x,z; begin x := y + z; if x < y then x := z * (y - 1); else x := (y + 1) * z; end if; end; </pre> |
|---|---|

The next segment would not be evaluated as equivalent to the template.

- ```

3) program
declare x,y.
begin
 x := y + z;
 if x > y then
 x := z * (y + 1);
 else
 x := z * (y - 1);
 end if;
end;

```

The problem with segment number three (above) is that there is a syntax error on the declaration line, that is, there is a period terminating that line instead of a semi-colon. The original SES was

integrated with the compiler so that syntax errors and/or detectable logic errors were output after a single pass through the program. The current SES is designed to allow the user to debug his/her program by sending it through a CORE compiler. When the program is syntactically correct, the user must request the computer to compare the program segment with the available templates.

### Computer-Based Instruction (CBI)

Just as the widespread use of automobiles, telephones, airplanes, and television changed the face of society and education, so too will the computer revolution alter (at least indirectly) the nature of instruction. Computer-Aided Instruction (CAI) offers many advantages over the conventional methods of teaching while maintaining equivalent levels of student performance. Currently, we may break down the areas computers are affecting education into five major categories. The first and most common is that of drill and practice. This technique starts the student at a particular skill level and presents a set of problems (similar to that of a workbook). The student types in his/her response, and the computer informs the student if the correct answer was received. If the answer was wrong, the computer generally instructs the student to try again. The second method of CAI is that of the tutorial. In this situation, the computer instructs the student in some area of knowledge, much in the same way an instructor would in a one-on-one situation. Of course, the programmer must anticipate most of the potential responses that the student might make in order to create a meaningful dialogue. This is extremely difficult to do, requires more memory than is available in most small computer systems, and takes a lot of field testing. The

third method, demonstration, makes use of one of the main features of traditional teaching. Utilizing the color, graphics, and sound potential of most small computers, software manufacturers are rapidly developing demonstration packages that will soon make the overhead projector obsolete. The fourth area is simulation. A simulation model imitates a real or imaginary system based on the theory of the operation of that system. Simulations focus the student's attention on certain aspects of the process under investigation. The design and programming of good computer simulations is very difficult. The last major category is that of instructional games. These games are designed to be 'fun' for students and thereby increase the chance of their learning the embedded concept.

The concept of using the computer as an aid to teaching is not new to the industrial community. Its emergence in the field of education, however, is relatively new. It is affecting the way students learn from kindergarden to college. This is important as society is becoming increasingly dependent upon computers. There is a growing need to produce computer-literate students that will be able to function in such a society.

Consider the use of computers in teaching computer programming. This is a new area of educational computer use that is drawing interest and research. There is evidence to suggest that students learn their cognitive network for a programming language through experience rather than actual classroom instruction. (8). This knowledge network is not only syntax and semantics, but also involves constructs and concepts which are not limited to a single programming language. Debugging skills are also difficult to teach in a classroom. Most instructors

provide a few simple exercises and then hope that the students discover the art of debugging on their own. It is a fact that much error diagnosing with beginning computer science students occurs on a one-to-one basis. It is not uncommon to find a student who feels completely inadequate until an experienced student or instructor volunteers their expertise. Such assistance requires an enormous amount of time. As shown in a study by Farrell, Anderson, and Reiser (9), however, a student who is working directly with the machine learns both more quickly and more deeply than students in classrooms. Therefore, a major goal of CAI in the field of computer science would be to capture the instructor's expertise by constructing an intelligent computer-based tutor that could help students. Such a program would give students the hands-on experience they need, the individual attention they require, and the encouragement to develop the proper knowledge network that is demanded. Such is the goal of the SES. While yet in its early stages of development, it already offers the student the ability to work directly with the computer and gives feedback on the correctness or incorrectness of the submitted response. A major advantage over current CAI packages is that the student does not have to give a specific predetermined response. The SES is flexible in that:

- a) it provides multiple templates that may be directly matched to the students response, and
- b) it does not require a direct match to any stored template, but is able to determine if the response is functionally equivalent to any stored templates.

Such a system will allow a student to develop his/her own style and logical constructs. The student may submit multiple correct but different responses to the same problem. Such flexibility is needed in teaching the fundamentals of programming.



## CHAPTER III

### STATEMENT OF THE PROBLEM

The purpose of this study is to develop a portable program that will determine whether two input segments, written in the mini-language CORE and designed to perform the same task, will produce the same results.

## CHAPTER IV

### DESIGN AND IMPLEMENTATION

#### The CORE Program Segment

Figure 2 presents a broad overview of the current SES design.

CORE program segment -> CORE compiler -> Equivalence Determination -> Analysis of Program Segment

Figure 2. Overview of the Current SES

The input to the system is a program segment written in the Legard and Marcotty mini-language CORE. Specifications of the CORE syntax are in Appendix C. The general design of a CORE program is as follows:

```
program
declare ...;
begin
CORE statements;
end;
```

The program must be written in lowercase letters and all variables must be declared in the 'declare' statement. The maximum length of a variable name is ten characters. The variable name may be composed of any combination of letters, digits, and underscores (\_), as long as the first character is a letter. Variable names may include capital letters but the programmer must remain consistent with the original name throughout the program, i.e. the compiler will not make a distinction between upper- and lower-case letters. The body of a CORE program may

consist of any combination of if, while, and assignment statements. Each statement must be terminated with a semi-colon. There are no procedures allowed. Nesting of if and while statements is allowed.

### The CORE Compiler

The CORE compiler operates in the given manner:

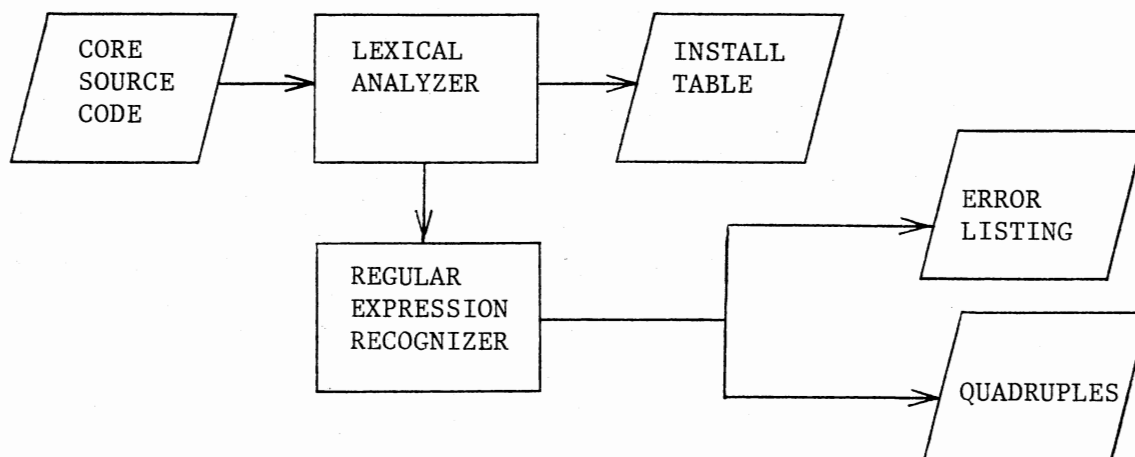


Figure 3. The CORE Compiler

### The Lexical Analyzer

The lexical analyzer is implemented using LEX, an automatic lexical analyzer generator. LEX generates a program which recognizes regular expressions. The LEX source (consisting of the user's regular expressions and actions) is passed through the LEX compiler which produces the lexical analyzer written in the C language (lex.yy.c).

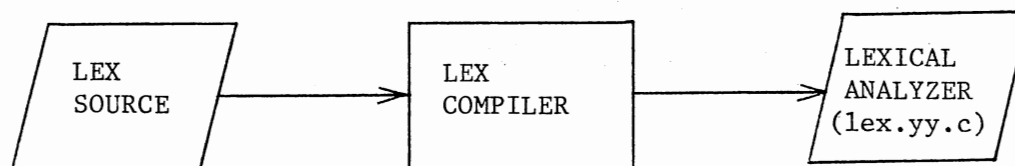


Figure 4. Creating the Lexical Analyzer with LEX

The input text (CORE program) is then passed through the lexical analyzer and a sequence of tokens is produced.

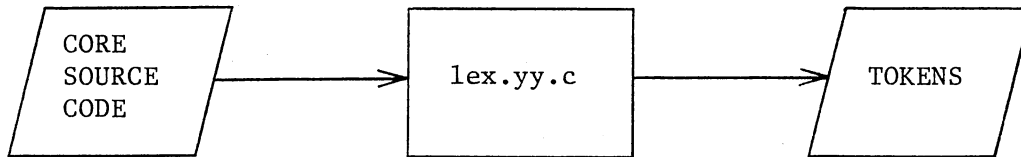


Figure 5. Generation of Token Sequence

Each regular expression in the LEX source file has an associated action which is to be executed when a token specified by that regular expression is recognized. Execution occurs in the following manner. The input is read, one character at a time, until it has found a match to a regular expression (a token). Once a token has been found, the analyzer executes the associated action. After completion of the action, control returns to the parser where the next input characters are read (5). In the case of the current SES, each action specified passes a numerical indication of the token found to the parser and makes an entry in the symbol table when appropriate. For example, the following is a regular expression and associated action sequence contained in the LEX source of the SES.

```

[a-zA-Z](?[_[a-zA-Z0-9]])* { inbuf(yytext);
 ylval = install(yytext,101,declr,
 stmtctr,linectr);
 return(114);}

```

The regular expression defined above will recognize any string of input symbols that begins with an upper- or lower-case letter of the alphabet. The character must be followed by zero or more instances of the sequence: one (optional) underscore character followed by an upper- or

lower-case character or a digit. The associated action sequence places the recognized token (yytext) into the input buffer, makes an entry in the symbol table (install table), and returns the value of 114. This value (114) is the numerical representation of an identifier name. A complete listing of all recognized tokens and their numerical representations is given in Table 1.

TABLE 1  
NUMERICAL REPRESENTATION OF TOKEN SEQUENCE

| TOKEN<br>NAME | NUMERICAL<br>REPRESENTATION |
|---------------|-----------------------------|
| BEGIN         | 101                         |
| DECLARE       | 102                         |
| ELSE          | 103                         |
| END           | 104                         |
| ENDIF         | 105                         |
| ENDLOOP       | 106                         |
| IF            | 107                         |
| INPUT         | 108                         |
| LOOP          | 109                         |
| OUTPUT        | 110                         |
| PROGRAM       | 111                         |
| THEN          | 112                         |
| WHILE         | 113                         |
| ID            | 114                         |
| CONST         | 115                         |
| OPEN          | 116                         |
| CLOSE         | 117                         |
| MULT          | 118                         |
| PLUS          | 119                         |
| MINUS         | 120                         |
| DIV           | 121                         |
| LT            | 122                         |
| LE            | 123                         |
| NE            | 124                         |
| GE            | 125                         |
| GT            | 126                         |
| EQ            | 127                         |
| CEQ           | 128                         |
| SCOLON        | 129                         |
| COMMA         | 130                         |

The symbol table, hereafter referred to as the install table, keeps a separate record for each identifier name and constant that is used in the program. Each record consists of four fields: name, value, stmtnum, and linenum. A description of the content of each field follows.

- Name** — holds the name of an identifier or constant. The maximum length of an identifier is ten characters. In the case of an identifier, this field contains the actual identifier name, in the case of a constant, it holds the character equivalent of the constant, such as '10'.
- Value** — Gives the index position in the install table of the first occurrence of the constant or variable. Entry is a -1 if the current entry is the first entry.
- Stmtnum** — Gives the statement number in the source program that contains the current variable or constant reference.
- Linenum** — Gives the line number in the source program that contains the current variable or constant reference.

As an example, given the program statements below, the corresponding entries would be made in the install table.

```

program
declare x;
begin
 x := 0;
 x := x + 1;
end;

```

| ( Index<br>Position) | Name | Value | Stmtnum | Linenum |
|----------------------|------|-------|---------|---------|
| 0                    | x    | -1    | 1       | 4       |
| 1                    | 0    | -1    | 1       | 4       |
| 2                    | x    | 0     | 2       | 5       |
| 3                    | x    | 0     | 2       | 5       |
| 4                    | 1    | -1    | 2       | 5       |

The data stored in the install table is used in the equivalence determination section of the system. Therefore, its contents are output to an external file. The name of the student's file is 'install.stud'. Because the instructor may enter many templates, each template file's

prefix is appended to an extension of '.nst'. For example, if the instructor entered a template stored in the file 'templatel.cor' then the name of the corresponding install table data file would be 'templatel.nst'.

### YACC - Input Parsing

In order to complete the CORE compiler, the tool YACC was used. YACC is an automatic parser generator and is an acronym for "Yet Another Compiler Compiler". YACC controls the parsing of the input. It makes successive calls to the lexical analyzer, obtaining a token as the result of each call. These tokens are organized according to the grammar rules supplied by the programmer. When a rule has been recognized, an associated programmer-defined action takes place. YACC requires that every specification file consist of three sections: the declarations, grammar rules, and programs. The sections are separated by double percent (%%) marks. An overview of a full specification file would look like (6):

```

declarations
%%
grammar rules
%%
programs

```

The declaration section may be left empty. However, YACC requires token names to be declared as such. This is done by simply specifying:  
%token name1 name2...

For example, the token PROGRAM is specified as:

```
%token PROGRAM
```

As was stated earlier, the lexical analyzer returns a numerical indication of the recognized token. For this reason, the specification

of tokens as they appear in the CORE compiler are similar to:

```
%token PROGRAM 123
```

where 123 is the numerical indication passed by the lexical analyzer.

When specifying the grammar rules, each rule has the form:

```
A : BODY ;
```

where A represents a nonterminal name, and BODY represents a sequence of names and literals. The names represent terminals or nonterminals, and may be composed of letters, periods, underscores, and digits. A literal is a character enclosed in single quotes. In general, a grammar involves four quantities: terminals, nonterminals, a start symbol, and productions. The productions define the ways in which the nonterminals may be built up from the terminals and other nonterminals. The start symbol denotes the language of interest and is presumed to be the left hand side of the first grammar rule in the rules section. The central idea in defining a language is to repeatedly apply the productions to expand the nonterminals into a string of nonterminals and terminals.

For example, consider the following specification:

```
(1) A -> X
(2) X -> a | b
```

The symbol '->' is read as 'derives'. Therefore, production number one (above) tells us that A derives X. This means that we can replace one instance of X with any string that X derives. For example, we might replace X in production number one with the 'a' from production number two. A sequence of such replacements is termed a derivation. In the YACC specification, the '->' symbol is replaced by the colon. The '|' symbol separates string that may be derived from a single nonterminal. The following specification appears at the beginning of the grammar rules section in the CORE compiler.



```

prog : program
 : decseq
 : begin
 : stmtseq
 : end SCOLON
 | empty
 ;
program : PROGRAM
 { quadgen(quadnum++,1,0,0,0,0); }
 ;

```

The start symbol is 'prog'. It is defined as being the derivation of each of the nonterminals: program, decseq, begin, stmtseq, and end, and the terminal SCOLON. Alternately it may be empty, which means it has no program body and is therefore deemed an error. The nonterminal 'program' derives the terminal PROGRAM which would be matched by a token sent from the lexical analyzer (identified in the example by capital letters). In each grammar rule, associated actions are specified in brackets {}. In the above example, there is an action associated with the nonterminal 'program'. It is a temporary transfer of control to the procedure 'quadgen'. This procedure is coded in the 'programs' section of the YACC specification file. Its purpose is to generate the intermediate code and is discussed subsequently in this paper.

If at any point the parser is not able to complete its derivation sequence, this indicates that a token has been found for which there is no corresponding match. This indicates an error. All errors are passed to the error routine with a number indicating the type of error. An appropriate error message will be generated to the user.

In addition to the install table data, the compiler also produces a file of intermediate code in the form of quadruples. The student quadruples appear in the file 'outquad.stud'. The instructor's quadruples appear in the file with the original program's prefix and the extension '.oq'. For example, if the original program were stored in

the file 'templatel.cor' then the corresponding quadruples would be stored in the file 'templatel.oq'. Each quadruple is implemented as a record with four fields: operator, argument one, argument two, and result.

Quadruples are a form of three-address code which is a sequence of statements, generally in the form of  $A := B \text{ op } C$ . A, B, and C are either user-defined names, constants, or compiler-generated temporary names. 'Op' is any operator or logical operator. In the statement ' $A := B + C$ ', B would be stored in argument one, C would be stored in argument two, A would be stored in the result field, and the appropriate representation of addition would be stored in the op field. In the current CORE compiler, what is actually stored in the quadruple record is the install table index of the current operand. That is, in the last example, the install table index of B would be stored in argument one. Each operation also has its own numerical representation. For example, the representation of '+', or addition, is '11'. A complete listing of 11 possible quadruple statement and operation representations is given in Table 2.

TABLE 2  
QUADRUPLE NUMERICAL STATEMENT IDENTIFIERS

| IDENTIFIER | STATEMENT REPRESENTATION |
|------------|--------------------------|
| 1          | 'Program'                |
| 2          | 'Begin'                  |
| 3          | 'End Loop'               |
| 4          | Assignment               |
| 5          | 'If/Then'                |

TABLE 2 (continued)

| IDENTIFIER | STATEMENT<br>REPRESENTATION |
|------------|-----------------------------|
| 6          | Greater Than                |
| 7          | Equal                       |
| 8          | Not Equal                   |
| 9          | Less Than                   |
| 10         | Multiplication              |
| 11         | Addition                    |
| 12         | Subtraction                 |
| 13         | While Loop                  |
| 15         | 'Else'                      |
| 16         | 'Input'                     |
| 17         | 'Output'                    |
| 18         | Less Than/Equal             |
| 19         | Grtr Than/Equal             |
| 20         | End                         |

#### Equivalence Determination

While the above section, that of the CORE compiler, was completed in the C language, the system section that determines functional equivalence is written in ADA. It consists of twelve separately compiled packages, and a main program (this does not include the student or instructor menus). The first package, package type\_definitions, consists of all the user-defined types needed to complete the system. The system is supported entirely by arrays, that is, there are no dynamic variables. In all, eight arrays are implemented. The program reads in the quadruples produced by the compiler and converts them into expression tables. The process of transformation is shown in Figure 6.

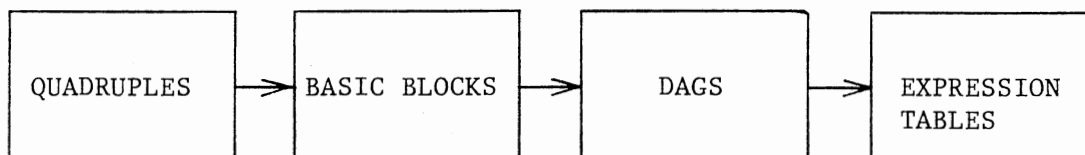


Figure 6. Program Transformation in the SES

The `type_definitions` package is followed by an initialization package which initializes each element in many of the arrays. Also, a utility package exists which holds various utility programs that are helpful such as `char_to_integer` which converts a character string to an integer value. The actual equivalence determination begins with the reading of each external file that holds the quadruples. Each file is read into an array of type `'quadprog'`. Each entry in the array is a record which contains the following fields:

```

op : integer
arg1 : integer
arg2 : integer
rslt : integer
markr : integer

```

The fields `'op'`, `'arg1'`, `'arg2'`, and `'rslt'` each hold the install table index of their corresponding operands. All fields are initially set to zero. The `'markr'` field will be used at a later time to mark the leaders of basic blocks. After the `quadprog` has been successfully filled, the quadruples are optimized and decomposed into basic blocks. Package `quadruple_transformer` contains the procedures and functions necessary to make these transformations. This is best explained with an example. Consider the following CORE program and its corresponding set of quadruples.

| <u>CORE Program</u> | <u>Quadprog Array (Quadruples)</u> |      |      |      |       |
|---------------------|------------------------------------|------|------|------|-------|
|                     | Op                                 | Arg1 | Arg2 | Rslt | Markr |
| program             | (1)                                | 1    | 0    | 0    | 0     |
| declare a,b,c;      | — no quadruple for 'declare'       |      |      |      |       |
| begin               | (2)                                | 2    | 0    | 0    | 0     |
| (1) a := 5;         | (3)                                | 4    | 2    | 0    | 1     |
| (2) b := 5;         | (4)                                | 4    | 2    | 0    | 3     |
| (3) c := a + b;     | (5)                                | 4    | 1    | 3    | 8     |
| end;                | (6)                                | 4    | 8    | 0    | 5     |
|                     | (7)                                | 16   | 0    | 0    | 0     |

### Install Table

| Index | Name | Value | Stmntnum | Linenum |
|-------|------|-------|----------|---------|
| 1     | a    | -1    | 1        | 4       |
| 2     | 5    | -1    | 1        | 4       |
| 3     | b    | -1    | 2        | 5       |
| 4     | 5    | 2     | 2        | 5       |
| 5     | c    | -1    | 3        | 6       |
| 6     | a    | 1     | 3        | 6       |
| 7     | b    | 3     | 3        | 6       |
| 8     | #1   | -1    | 0        | 0       |

Statement number one in the above program (a := 5) corresponds to quadruple number three. The 'op' field contains the value of four which indicates an assignment. 'Arg1' contains the value two. Index position two of the install table contains data on the constant five. The 'rslt' field contains the value of one which refers to index position one of the install table which holds data about variable 'a'.

The first optimization routine removes the 'trivial' quadruples. These are the first two and the last quadruple of every set. These represent the 'program', 'begin', and 'end' statements respectively. The second optimization technique handles the situation that occurs as a result of an assignment such as the one in statement number three of the program. If one assignment statement has two (or more) operands, there is an extra quadruple produced. In statement number three, 'c := a + b', 'a' and 'b' are added together and stored in a compiler-generated temporary position. These temporary positions are designated in the

install table by a '#' character prefix. After the sum of 'a' and 'b' is stored in temporary '#1', an additional quadruple is generated that places the value in temporary '#1' into the original result 'c' (or index position five in this case). Optimization takes place by eliminating this extra quadruple. The temporary reference (in the 'rslt' field of quadruple number five) is replaced by the 'rslt' field of quadruple number six. The resulting set of quadruples appears as:

| Op | Arg1 | Arg2 | Rslt | Markr |
|----|------|------|------|-------|
| 4  | 2    | 0    | 1    | 0     |
| 4  | 2    | 0    | 3    | 0     |
| 4  | 1    | 3    | 5    | 0     |

After the quadruples are optimized, they are divided into basic blocks. A basic block is a sequence of statements which is entered only from the beginning, that is, there is no possibility of branching within the block. In order to partition a set of quadruples into basic blocks, the leaders (first statement of each basic block) must be determined.

The rules for determining leaders are as follows (4):

- a) the first statement of a set of quadruples is a leader;
- b) any statement which is the target of a conditional or unconditional goto is a leader;
- c) any statement which immediately follows a conditional goto is a leader.

Each leader is found and marked with a one in its 'markr' field.

After all have been found, they are entered into an array called 'leaders'.

The next step was to represent each basic block with a directed acyclic graph (DAG). DAGs are useful data structures for analyzing basic blocks. They reveal common subexpressions within a block, the names that are used inside the block but evaluated outside the block,

and which statements of the block could have their value used outside the block. A DAG is constructed as follows:

- a) leaves are labeled by unique identifiers, either variable names or constants;
- b) interior nodes are labeled by operator symbols
- c) nodes are assigned a set of identifiers as labels.

As an example, consider the following two expressions:

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

The expression for 'b' is a duplicate of the expression for 'a' (assuming for the sake of the example that the statements are consecutive and the values of 'x' and 'y' remain constant). The attached identifier list for the expression 'x + y' would consist of both identifiers 'a' and 'b'. The expression would only be represented once within the DAG.

The DAG's were constructed using algorithm 9.2 of Aho, Sethi, and Ullman (4). Procedures and functions used to create the DAGs are contained in the create\_DAGS package. As a result two arrays, 'identifiers' and 'nodes', are filled with data about the basic block. Both arrays contain records, the contents of each are explained below.

#### Identifier Array

```
nndx : an index into the node array which represents the node to who's
 list this identifier belongs;
name : the name of the identifier (represented as an index into the
 install table);
block : the number of the current basic block.
```

#### Node Array

```
nname : the name of the node (represented as an integer, see table 3);
left : the left child of the node;
right : the right child of the node;
block : the number of the current basic block.
```

Continuing with the previous example, the contents of the arrays

would be:

| <u>Identifier Array</u> |      |       | <u>Node Array</u> |       |      |       |       |
|-------------------------|------|-------|-------------------|-------|------|-------|-------|
| Nndx                    | Name | Block | Index             | Nname | Left | Right | Block |
| 1                       | 1    | 1     | 1                 | -4    | 2    | 0     | 1     |
| 1                       | 3    | 1     | 2                 | -11   | 1    | 3     | 1     |
| 2                       | 5    | 1     |                   |       |      |       |       |

Recall that the contents of the 'name' field in the identifier array is actually an index into the install table, likewise with the 'left' and 'right' fields of the node array. The negative four and negative eleven in the 'nname' fields of entries one and two in the node array represent operators. As shown in Table 3, they stand for assignment and addition respectively.

TABLE 3

INTERNAL NUMERICAL REPRESENTATION OF OPERATORS

| OPERATOR | OPERATION          |
|----------|--------------------|
| - 4      | Assignment         |
| - 6      | Greater Than       |
| - 7      | Equal              |
| - 8      | Not Equal          |
| - 9      | Less Than          |
| -10      | Multiplication     |
| -11      | Addition           |
| -12      | Subtraction        |
| -16      | Input              |
| -17      | Output             |
| -18      | Less Than/Equal    |
| -19      | Greater Than/Equal |

Since the program works exclusively with integers as opposed to actual names (such as the identifier 'a') there arose a need to distinguish between constants and identifiers whose integers represented install table indices, and the integers which represent operators. The logical



solution to this was to make the operator representatives negative numbers, since an index can never be negative. In order to demonstrate, shown below is the contents of the arrays as they would appear if the actual identifier, constant, and operator names were used.

Identifier Array

Node Array

| Nndx | Name | Block | Index | Nname | Left | Right | Block |
|------|------|-------|-------|-------|------|-------|-------|
| 1    | a    | 1     | 1     | :=    | 5    | 0     | 1     |
| 1    | b    | 1     | 2     | +     | a    | b     | 1     |
| 2    | c    | 1     |       |       |      |       |       |

The information represents the following DAGs (in pictorial form):

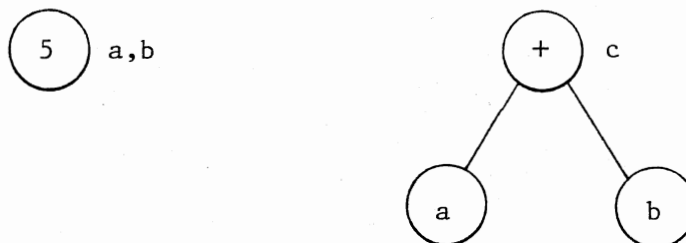


Figure 7. Pictorial DAG Representation of Array Information

From this new information we are able to construct expression tables for both the instructor and student programs. Functional equivalence is determined through a comparison of these two tables. An expression table is actually an updated version of the quadruples that were read previously. Once again, the expression table is an array of records, each of which has the following structure:

- id : holds the install table index value of the resultant identifier (in the statement 'a := b + c', 'a' would be represented by this field);
- left : the left operand corresponding to the 'id' field;
- right : the right operand corresponding to the 'id' field;
- op : the statement operator;
- block : the block number in which this statement appears;

match : this field tells if a match has been found for this statement (explained subsequently in this document).

In addition to install table indices, the 'id' field also designates the beginning and end of control statements (if and while statements). The 'id' field appears as a negative three (-3) at the beginning of a while statement and a negative thirteen (-13) at the end of the loop. Likewise, the beginning of an if statement is marked by a negative five (-5) and the end is signalled by a negative fifteen (-15). The 'op' field is a type character field which holds a single character. Therefore, for operands such as '<>', ':=', '<=', '>=', a single character representation was required. Table 4 shows the representations of each existing operator used in this program.

TABLE 4  
SINGLE CHARACTER REPRESENTATION  
OF CORE OPERATORS

| OPERATOR | REPRESENTATION |
|----------|----------------|
| +        | +              |
| -        | -              |
| *        | *              |
| :=       | ~              |
| <        | <              |
| >        | >              |
| <>       | /              |
| <=       | (              |
| >=       | )              |

Continuing with our example, the contents of the expression table would become:

| Id | Left | Right | Op | Block | Match |
|----|------|-------|----|-------|-------|
| a  | 5    | 0     | := | 1     | 0     |
| b  | 5    | 0     | := | 1     | 0     |
| c  | a    | b     | +  | 1     | 0     |

Or Equivalently (actual internal representation):

| Id | Left | Right | Op | Block | Match |
|----|------|-------|----|-------|-------|
| 1  | 2    | 0     | ~  | 1     | 0     |
| 3  | 2    | 0     | ~  | 1     | 0     |
| 5  | 1    | 3     | +  | 1     | 0     |

After the expression tables have been filled, equivalence determination can begin. In all, the program makes three distinct passes through the data in an effort to determine equivalence:

```
expression preliminary basic functional
tables > matching > matching > equivalence > results
```

Figure 8. Phases of Equivalence Determination

### Preliminary Matching

The procedure responsible for the preliminary matching is one called 'mark\_matches', found in the match\_maker package. It attempts to match those expressions that are 'direct' matches, that is, they need little checking. In this and all succeeding determination attempts, expressions with the operator of '~' (or ':=') are not compared. This is because this operator only appears with direct assignment statements such as 'a := 5'. To require a direct match would exclude statements such as 'a := 3 + 2' which is equivalent. There are points in the determination process when the knowledge of an initial value of a variable needs to be known, but this can be determined by the combination of two functions, find\_index and compute, to be discussed later. In order for a match to occur at this point in the process, two expressions must have the same operators and have the same corresponding basic block number. Throughout the remaining examples, the operand 'x'

will be assigned as the instructor's variable and the operand 'y' will appear as the student's variable. The operand 'z' will appear as an universal identifier.

The first check that is made is one that determines if the identifier being calculated also appears as an operand in the same statement. For example, 'x := x + z'. If so, the student statement must also conform to this rule as in 'y := y + z'. Assuming for the moment that both statements fall into this category, the following possible operand combinations must be considered.

|             |             |      |
|-------------|-------------|------|
| x := x + z; | y := z + y; | -OR- |
| x := x + z; | y := y + z; | -OR- |
| x := z + x; | y := z + y; | -OR- |
| x := z + x; | y := y + z; |      |

This is, of course, assuming the operator is that of addition or multiplication which allows the operands to be in any order because of the commutative property of both operators. In the case of subtraction, corresponding operands must be in corresponding positions.

|             |             |
|-------------|-------------|
| x := x - z; | y := y - z; |
| x := z - x; | y := z - x; |

Also, since the CORE language allows for run-time assignment of variables through the 'input' statement, the possibility that x, y, or z may be input by the user also must be considered. Given that the variable 'z' has a direct assignment of ten in the instructor's program and the student's variable 'z' is input during run-time, the benefit of the doubt is given to the student. That is, it is possible to input ten when prompted. Therefore, any input variable is considered to be equivalent to a directly assigned variable throughout the program.

When checking control statements during this phase, the conditional variables must be the same.

```

while (x < z) loop
while (x < z) loop
while (z > x) loop
while (z > x) loop
if (x < z) then
if (x < z) then
if (z > x) then
if (z > x) then
while (y < z) loop
while (z > y) loop
while (y < z) loop
while (z > y) loop
if (y < z) then
if (z > y) then
if (y < z) then
if (z > y) then

```

For any of the above statements to be considered equivalent, the last assignment into the variable 'x' must match the last assignment into the variable 'y' and the last assignments into both variables 'z' must match. That is, the programs in segment 'A' below would be considered equivalent, while the programs in segment 'B' would not.

|                                                                                                                                                          |                                                                                                                                                   |
|----------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> (A) program (instructor's)   declare x,z;   begin     x := 1;     z := 10;     while (x &lt; z) loop       x := x + 1;     end loop;   end; </pre> | <pre> program (student's)   declare y,z;   begin     y := 1;     input z;     while (z &gt; y) loop       y := 1 + y;     end loop;   end; </pre> |
| <pre> (B) program (instructor's)   declare x,z;   begin     x := 1;     z := 10;     while (x &lt; z) loop       x := x + 1;     end loop;   end; </pre> | <pre> program (student's)   declare y,z;   begin     y := 1;     z := 11;     while (z &gt; y) loop       y := y + 1;     end loop;   end; </pre> |

The function that aids in determining the current variable at a given point in the program is called `find_index`. It performs a search of the expression table and returns one of the values given below.

A positive number : indicates an index in the install table (meaning the variable has a direct assignment as in 'x := 5');

-10 : indicates the value is computed, as in the case of 'x' in the expression 'x := 5 + z';

-16 : indicates the value of the variable is input during run time as in the statement 'input x';

```

-1 : indicates that the value of the variable is
 : computed within a control statement as is 'x' in:
 : while (z < 10) loop
 : z := z + 1;
 : x := z + x;
 : end loop;

```

During this preliminary phase, all values must have direct assignments in order to be considered equivalent. In other words, `find_index` must return a positive number for each variable. To illustrate, the following programs would not be considered equivalent (at this point):

|                                                                                                                                     |                                                                                                                                     |
|-------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <pre> program (instructor's) declare x,z; begin   x := 1;   z := 10   while (x &lt; z) loop     x := x + 1;   end loop; end; </pre> | <pre> program (student's) declare y,z; begin   y := 1;   z := 5 + 5;   while (y &lt; z) loop     y := y + 1;   end loop end; </pre> |
|-------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|

If a match between two statements is found, then the 'match' field of the student's expression table is set to one. The 'match' field of the instructor's expression table is set to the index position of the statement in the student's expression table that it matches.

### Basic Matching

The second attempt at matching the two program segments is driven by the procedure `compare_tables` in the `program_evaluator` package. The matching that occurs in this section of the program is an extension of that done in the preliminary matching. It passes through each instructor statement that has a zero in its 'match' field (thereby indicating that this statement has no match) and attempts to locate a similar statement in the student's program. In order to do this, it calls a function `find_an_expr_to_match` which returns an index into the student's expression table of a possible statement. If the instructor

student's expression table of a possible statement. If the instructor statement in question is located within a control statement, then an attempt is made to find a similar statement within the corresponding student's control statement. (Obviously if the student has no control statement no statement is suggested for matching). If the instructor's statement is not within a control statement, the function first checks the corresponding block number for a match possibility. If none is found, the function returns any statement (that is not within a control block) that is similar. If a student statement is located and an unsuccessful attempt is made at matching, the program continues down the student's available candidates and the process is repeated until a match is found or all possibilities have been exhausted.

The matching at this phase is more complex than that of the previous phase due, in part, to the capabilities of the function compute. If the find\_index function returns a negative ten, compute makes an attempt to determine the value computed for a particular variable. Take the following two segments as examples.

|                                                                                                                |                                                                                                                 |
|----------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| <pre>(A) program     declare x,y,z;     begin       x := 5;       y := x + 3;       z := y + x;     end;</pre> | <pre>(B) program     declare x,y,z;     begin       input x;       y := x + 3;       z := y + x;     end;</pre> |
|----------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|

The compute function would be able to determine that the variable 'z' in segment 'A' had a value of thirteen, and the variable 'z' in segment B could have any possible value. Compute returns an actual value as opposed to an index. The problem that arises with this is that it needed to return a value that would properly indicate an unsuccessful attempt to locate an actual value. In segment 'B' above, the value

returned by compute is negative sixteen, which indicates the value was input at some point and may hold any integer. The problem lies in determining whether negative sixteen indicates an input variable or is the actual value assigned to some variable. The problem was solved through the use of a boolean status flag. This flag holds a single character with the following meanings:

'R'        - returning a value assigned to a variable;  
 'I'        - value was input from the user at some point;  
 'N'        - last value assigned was inside a control block therefore  
             is not computable;  
 'X'        - value can not be determined.

Compute would return an 'I' for cases such as the last example, and an 'X' for situations such as the one presented below.

```
program
declare x,y,z;
begin
 x := 1;
 input y;
 while (x < y) loop
 x := x + 1;
 end loop;
 z := x; ← (a)
end;
```

In statement 'a' above, neither the value of 'x' nor the value of 'z' is computable.

Another factor that gives power to this phase of determination is that involved with finding the last reference to a variable. Consider the following two segments:

```
(A) program (instructor's)
declare x,z;
begin
 x := 1;
 while (x < 10) loop
(a) z := z + x;
 x := x + 1;
 end loop;
 output z;
end;
```

```
(B) program (student's)
declare y,z;
begin
 y := 1;
 while (10 > y) loop
(b) z := z + y;
 y := y + 1;
 end loop;
 output z;
end;
```



When trying to match the output statements, the value of 'z' is not directly computable. If, however, it can be found that the last reference to 'z' in the instructor's program was a match to the last reference of 'z' in the student's program, then it can be assumed that the current 'z's are equivalent. Since the 'match' field of the instructor's expression table holds the index of the statement it matches in the student's expression table, the best way to determine the last reference is to search the instructor's expression table to see if the last reference to 'z' had a student statement match. That is, if statement 'a' in the above instructor's program was already matched to statement 'b' in the student's program, then it can be assumed that the output statements work with the same variable.

The ability to continue a search aids in the comparison of control statements, as shown:

|                                                                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                                                                                                                                                                   |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>(A) program (instructor's)     declare w,x,y,z;     begin       x := 1;       y := 0;       z := 0;       (1) while (x &lt; 10) loop           y := y + x;           x := x + 1;         end loop;       w := 10;       (2) while (w &lt; 20) loop           w := w + 1;           z := z + w;         end loop;     end;</pre> | <pre>(B) program (student's)     declare a,b,c,d;     begin       a := 10;       b := 0;       c := 0;       (1) while (a &lt; 20) loop           a := a + 1;           b := b + a;         end loop;       c := 1;       (2) while (c &lt; 10) loop           d := d + c;           c := c + 1;         end loop;     end;</pre> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

The above program segments are equivalent. The SES is able to decide this by matching loop number one of the instructor's program to loop number two of the student's program, and vice versa.

### Functional Equivalence

The third and final phase of equivalence determination attempts to match any instructor's statement that has not been previously matched. There are separate attempts at matching the expression, select, and iteration statements. The idea at this step is to allow for different thought patterns in the design of the program segments. The package that contains the functions and procedures for this phase is the equiv\_chk package. It considers variations that may occur in the if, while, and expression statements. For example, when considering a loop, the following variation may occur:

|                                                                                                                                                                  |                                                                                                                                                                             |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> program (instructor's) declare x,y,z; begin   x := 0;   y := 0;   z := 10;   while (x &lt; 10) loop     x := x + 1;     y := y + z;   end loop; end;</pre> | <pre> program (student's) declare a,b,c; begin   a := 1;   b := 0;   c := 10;   while (a &lt; 10) loop     a := a + 1;     b := c + b;   end loop;   b := c + b; end;</pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

In the above example, the student's loop executes one less time than the instructor's. The program is equivalent, however, because the pertinent statements in the loop body have been executed outside the loop to make up the difference. In order to determine equivalence, the number of times each loop executes is evaluated. Then data on the loop control variable must be gathered. Among other things, it must be determined if the loop control variable is used in the calculations of other statements within the body. If so, its position relative to the other statements must be considered. For example, consider the template segment that follows.

```

x := 0;
while (x < 10)
loop
 x := x + 1;
 y := y + 1;
end loop;

```

The two following segments are equivalent variations:

```

a) x := 1;
 while (x < 10)
 loop
 y := y + x;
 x := x + 1;
 end loop;

```

```

b) x := 0;
 while (x <= 10)
 loop
 x := x + 1;
 y := y + x;
 end loop;
 y := y - x;

```

The number of iterations of the student and instructor loops must be determined. If the student's loop executes more times than the instructor's, there must be statements outside the loop that will negate the action(s) that occurred within the loop. If the student's loop executes fewer times than the instructor's, there must be statements outside the loop that complete the actions of the body.

When questioning the equivalence of expression statements at this phase, the different ways of writing the same expression must be considered. In the preceding phases, the equivalence of operands and operators were considered. The sequence 'x := 10; y := 5; z := x + y;' was considered equivalent to 'y := 5; x := y + y; z := x + y;' because the values of 'x', 'y', and 'z' were equivalent, as was the operator of the expression evaluation into 'z'. When evaluating the following two statements: 'x := y + y' and 'x := y \* 2', the values of the 'x' variables must be evaluated. Before evaluation, it must be confirmed that the last references to the resultant identifiers were matched. If this can be determined and the evaluation of the expressions are the same, it may be assumed that the statements are equivalent.

The factor to be considered when determining the functional

equivalence of the 'if/else' statement is that of the reverse positioning of the condition operands. If the operator is the same but the operands have reversed positions, then the expressions that occur within the body of the statement must be reversed respective to the operands. As an example, consider the following:

```

a) if (x > y) then
 z := z + x;
else
 z := z + y;
end if;

b) if (y > x) then
 z := z + y;
else
 z := z + x;
end if;

```

It should be injected at this point that there may be numerous distinct correct answers to a programming problem. It is not realistic to expect the SES to recognize all correct solutions. Many common (and some uncommon) variations that may occur in student responses were studied previous to the development of the current SES. As a result, it is able to detect and accept many functional equivalences. A factor that aids its detection capabilities is the capacity to store many templates. This significantly raises the probability that a match (between a student response and a stored template, whether correct or incorrect) may be found. Consider the stored templates that follow. Each is designed to compute the product of the numbers one to ten.

Correct:

```

a) x := 1;
 y := 1;
 while (x < 11) loop
 y := y * x;
 x := x + 1;
 end loop;

b) x := 2;
 y := 1;
 while (x < 11) loop
 y := x * y;
 x := x + 1;
 end loop;

c) x := 1;
 y := 1;
 while (x <= 10) loop
 x := x + 1;
 y := y * x;
 end loop;

d) x := 2;
 y := x;
 while (x <= 10) loop
 x := x + 1;
 y := y * x;
 end loop;

```

Incorrect:

```
e) x := 0;
 y := 0;
 while (x < 11) loop
 x := x + 1;
 y := y * x;
 end loop;
```

```
f) x := 1;
 y := 1;
 while (x <= 11) loop
 x := x + 1;
 y := y * x;
 end loop;
```

The following program segment, which is a correct response to the problem, would not be correctly matched to templates a, b, or c. It would, however, be matched to template d. As a result, the student would be informed that he/she had submitted a correct answer to the given problem.

```
x := 2; y := 2;
while (x < 11) loop
 x := x + 1;
 y := y * x;
end loop;
```

The next segment is an example of a response which would be matched to an incorrect response. The segment would match template e and the student would be informed that the submitted response was not a correct solution to the given problem.

```
x := 0; y := 0;
while (x <= 10) loop
 y := y + x;
 x := x + 1;
end loop;
```

## CHAPTER V

### RESULTS OF THE STUDY

#### Equivalent and Nonequivalent Programs

Given are some examples of programs that are determined to be equivalent to their corresponding templates. In the samples that follow, each instructor's program is considered to be a correct template. It should be stated that the instructor is also allowed to submit an incorrect template for comparison. If a student's input program segment is equivalent to a correct template, a message informing him/her of a correct response is displayed along with the point value of a match to the template. If an incorrect response is submitted and matched, a message informing the student that his/her response is incorrect is displayed along with any points that may be available for the match.

#### Instructor's Template

```
a) program
 declare x,y;
 begin
 x := 0;
 while (x < 10)
 loop
 x := x + 1;
 input y;
 if (y < 0) then
 output y;
 end if;
 end loop;
 end;
```

#### Student's Program

```
program
 declare x,y;
 begin
 x := 1;
 while (x <= 10)
 loop
 x := x + 1;
 input y;
 if (0 > y) then
 output y;
 end if;
 end loop;
 end;
```

- b) program  
 declare x,y;  
 begin  
 x := 0;  
 y := 5;  
 while (x < 10)  
 loop  
 x := x + 1;  
 y := y \* 2;  
 end loop;  
 output y;  
 end;
- c) program  
 declare x,y,z;  
 begin  
 x := 2;  
 y := 5;  
 z := 5 \* (x + y);  
 output z;  
 end;
- d) program  
 declare x,y,z;  
 begin  
 x := 10;  
 z := 5;  
 y := x + z;  
 if (y > x) then  
 z := 20 + x;  
 else  
 z := 20 + z;  
 end if;  
 end;
- e) program  
 declare w,x,y,z;  
 begin  
 w := 1;  
 x := 0;  
 y := 1;  
 z := 2;  
 while (x < 10)  
 loop  
 x := x + 1;  
 y := y + x;  
 end loop;  
 while (w < 5)  
 loop  
 w := w + 1;  
 z := z \* 2;  
 end loop;  
 end;
- program  
 declare a,b;  
 begin  
 a := 5;  
 b := 0;  
 while (b < 10)  
 loop  
 a := a + a;  
 b := b + 1;  
 end loop;  
 output a;  
 end;
- program  
 declare a,b,c;  
 begin  
 a := 2;  
 b := 5;  
 c := (5 \* a) + (5 \* b);  
 output c;  
 end;
- program  
 declare x,y,z;  
 begin  
 z := 5;  
 x := z \* 2;  
 y := x + z;  
 if (x > y) then  
 z := 20 + z;  
 else  
 z := 20 + x;  
 end if;  
 end;
- program  
 declare a,b,c,d;  
 begin  
 a := 1;  
 b := 2;  
 while (a < 5)  
 loop  
 b := b + b;  
 a := a + 1;  
 end loop;  
 c := 0;  
 d := 1;  
 while (c < 10)  
 loop  
 c := c + 1;  
 d := c + d;  
 end loop;  
 end;

While the SES is capable of matching many general program segments, it is not reasonable to expect perfection. The determination of functional equivalence is, in general, unsolvable. The SES assumes that the instructor will provide a variety of program templates, both correct and incorrect, of expected responses. It is also assumed that the student will put thought and consideration into the design of his/her program segment.

Given below are some examples of the types of program segments that the SES will not compute as matches.

```

program (instructor's)
declare x;
begin
 x := 0;
 while (x < 5) loop
 x := x + 1;
 end loop;
end;

```

```

program
declare a;
begin
 a := 0;
 a := a + 1;
 a := a + 1;
 a := a + 1;
 a := a + 1;
 a := a + 1;
end;

```

The program will not extend the body of a loop to match a sequence of expressions.

```

program (instructor's)
declare x,y,z;
begin
 x := 1;
 y := 5;
 z := 5;
 while (x < 10)
 loop
 x := x + 1; <-- a
 y := y + z; <-- b
 end loop;
end;

```

```

program
declare x,y,z;
begin
 x := 1;
 input y;
 input z;
 while (x < 10)
 loop
 y := y + z; <-- c
 x := x + 1; <-- d
 end loop;
end;

```

The problem with the above samples is that caused by the assumption placed on input variables. It must be assumed that the user will input the correct values for the identifiers, therefore, an input variable is an automatic match to any value. Since the program attempts to match



expressions in the order in which they are presented, statement 'a' is first paired with statement 'c'. This produces a match because both the variables in expression 'c' have their values input. It is possible that these statements match. As a consequence, however, the statements 'b' and 'd' are not matched because they are not equivalent.

As another precaution, it should be noted that the nesting of if and while statements will produce unpredictable results. The nesting of a single if statement within a single while statement is permitted. Depending on the arrangement of statements, however, the combination and nesting of these statements causes inconsistencies in the program.

#### Suggestions for Future Study

It is highly recommended that development of the SES continue. This project has much potential. When deciding whether two program segments will produce the same output values no matter their design, the question comes to mind "Is that not the purpose of a compiler"? Perhaps, but the compiler fails in many ways when compared to the possibilities of the SES. A compiler may aid in the development of basic debugging skills, but it offers no help to the student who is struggling to develop the knowledge network required for efficient structured programming. In short, it is not a teacher. The SES on the other hand allows the user to read a problem instruction and toy with the many solutions to it. This will encourage exploration and development of basic thought patterns as well as the knowledge of basic constructs (such as condition and iteration statements). It is suggested that the next step in development is the incorporation of the current SES into an expert system. The program developed in this study

will provide an excellent base for such a system. Due to the design of the current SES, logic errors may easily be detected during the evaluation of the program segment. It is suggested that an appropriate dialogue be developed that will inform the student of his/her error and suggest methods of correction. It would be advantageous to the student if the SES were interactive and would allow changes to be made as the errors were detected.

### Summary

The purpose of this study is to expand the original Statements Evaluation System to include the evaluation of the 'while loop' statement. The system allows an instructor to enter multiple questions and template answers. These templates may be (logically) either correct or incorrect. The syntax, however, should be consistent with that of the mini-language CORE. These templates are compiled into intermediate code and stored for use later as a functional equivalence evaluation factor. The student user is presented a menu (as developed by the instructor) from which he/she can choose one of a maximum of twenty available options. Once an option is selected, the student is presented a set of instructions to follow in developing a program. Once the student has developed a syntactically correct program segment, he/she may submit it for evaluation. An appropriate response informing the student of a match to a correct template, a match to an incorrect template, or the inability to match to either a correct or incorrect template is displayed.

## BIBLIOGRAPHY

- 1) Ibarra, Oscar H., and Rosier, Louis E., "The Equivalence Problem and Correctness Formulas for a Simple Class of Programs", Univ. of Texas at Austin, Dept. of Computer Science, Tech. Rep. No. 83-23 (1983).
- 2) Ibarra, Oscar H., and Leininger, Brian S., "On the Simplification and Equivalence Problems for Straight-line Programs", J. ACM 30,3 (July 1983), pp. 641-656.
- 3) Tsang, Peter Yu Yee, "A Statements Evaluation System for Functionally Equivalent Responses", Okla. State Univ., Dept. of Computer Science (1984).
- 4) Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D., Compilers - Principles, Techniques, and Tools, Addison-Wesley, Reading, Mass., 1985.
- 5) Lesk, M. E. and Schmidt, E., "Lex - A Lexical Analyzer Generator", Bell Laboratories, Murray Hill, N.J., (1978).
- 6) Johnson, S. C., "Yacc: Yet Another Compiler Compiler", Computing Science Tech. Rep. No. 32, Bell Laboratories, Murray Hill, N.J., (1978).
- 7) Baker, Louis, "ADA & AI Join Forces", AI Expert, (Apr. 1987) pp. 38-43.
- 8) Swigger, Dr. Kathleen M., and Evans, Donald, "A Computer-Based Tutor for Assembly Language", Journal of Computer-Based Instruction, (Winter 1987), Vol. 14, No. 1, pp. 35-38.
- 9) Jarvis, Mildred D., "Computer Based Training: Lessons Learned", Proceedings of the Human Factors Society, 28th Annual Meeting, (1984), pp. 515-519.
- 10) Swigger, Kathleen, and Wallace, Layne, "Use of Appropriate Looping Structures: Expert vs. Novice", Proceedings of the Human Factors Society, 26th Annual Meeting, (1982), pp. 999-1003.
- 11) United States Department of Defense, Reference Manual for the ADA Programming Language, Springer-Verlag, New York, 2nd Ed., 1984.
- 12) Ledgard, Henry, and Marcotty, Michael, The Programming Language Landscape, SRA Inc., Chicago, Ill., (1981).

- 13) Coburn, Peter, Kelman, Peter, et al., Practical Guide to Computers in Education , Addison-Wesley Publ., Reading, Mass., 2nd ed., (1985).

## APPENDIX A

### SES INSTRUCTOR'S OPERATION MANUAL

#### Purpose

The purpose of this manual is to guide the instructor in developing a menu and set of templates for the OSU Statements Evaluation System.

## PREFACE

## Developing the SES

There are three distinct steps that must be taken when developing templates for comparison. The instructor must:

- a) write and debug the CORE source code of the template;
- b) develop a menu option and instructions that will describe for the user the program he/she must write;
- c) give a brief description of the template to the computer.

Each step is fully documented in this manual.

## CHAPTER I

### THE CORE TEMPLATE

#### Format and Content

The template may be developed by using any standard editor. The syntax of the source code must be in accordance with the standards set forth for the Legard and Marcotty mini-language CORE. The CORE language is designed for simple data manipulation. It is not a powerful language and should not be expected to perform large computational problems. It will perform best when used with its purpose and abilities in mind.

The following guidelines should be observed when developing the CORE template.

- 1) CORE allows only one data type, that of INTEGER. All variable names must be declared and the length should not exceed ten characters.
- 2) The source code should be written in lowercase letters. Identifier names may contain a combination of letters, digits, and underscores. The compiler, however, does not distinguish between upper and lowercase letters. Therefore, if an identifier name of 'Tax' is declared, the user must not use 'TAX' or 'tax' within the program body. The compiler sees these as three separate variables, the last two of which are undeclared.
- 3) There are three mathematical operators, those being addition (+), subtraction (-), and multiplication (\*). Division is not allowed.
- 4) There are six logical operators: equal, not equal, less than, greater than, less than or equal, and greater than or equal.
- 5) All statements must be terminated with a semi-colon.

- 6) 'If' statements may be embedded. Each statement should be terminated with a corresponding 'end if'.
- 7) 'While/loop' statements may be embedded. Each statement must be terminated with an 'end loop'.
- 8) The use of parentheses is allowed to manipulate the order of operation in expression (assignment) statements.

### Compiling the Template

Once the template has been written and saved, it may be compiled by typing:

```
$ICORE filename
```

where 'filename' is the name of the source code. The length of the 'filename' should not exceed ten characters (this includes the file extension). The CORE compiler may generate any of nine error codes. Following is a list of the codes and their descriptions.

| Code | Problem                                 |
|------|-----------------------------------------|
| 1    | Illegal Variable Name                   |
| 2    | IF/END IF statements not sequenced      |
| 3    | BEGIN/END statements not sequenced      |
| 4    | WHILE/END LOOP statements not sequenced |
| 5    | Missing PROGRAM statement               |
| 6    | Variable already declared               |
| 7    | Undeclared variable name                |
| 8    | Variable name exceeds maximum length    |
| 9    | Syntax Error: ';','(',')' expected      |
| 10   | No Errors or Warnings                   |

Definitions are as follows:



| Code | Definition                                                                                                  |
|------|-------------------------------------------------------------------------------------------------------------|
| 1    | Illegal character(s) in the variable name.                                                                  |
| 2    | An 'IF' statement was found with no corresponding 'END IF' statement.                                       |
| 3    | A 'BEGIN' statement was found with no corresponding 'END' statement.                                        |
| 4    | A 'WHILE' statement was found with no corresponding 'END LOOP' statement.                                   |
| 5    | Either the wrong file was submitted for compilation, or the source code is missing the 'PROGRAM' statement. |
| 6    | The same variable name appears twice in a declaration sequence.                                             |
| 7    | A variable name has been found in the program body that was not found in the declaration sequence.          |
| 8    | An identifier name exceeds the length (in characters) of the maximum length (10 characters).                |
| 9    | Check the given line number for missing parentheses, semicolons, etc.                                       |
| 10   | The program is syntactically correct.                                                                       |

After the template has been successfully compiled, the instructor should retain the name of the file for input into the menu.

## CHAPTER II

### DEVELOPING THE SES MAIN MENU

This chapter details how to set up the SES main menu. This is the menu that will be displayed to the student. When the student chooses an option, a set of instructions will be given on the development of a program. The student should write a program that meets the specifications given in the instructions, compile his/her program segment, and submit it for comparison against the menu option's templates.

To begin the SES Development Menu, type:

```
$ IMENU
```

The following menu will appear:

```
* STATEMENT'S EVALUATION SYSTEM *
*** Instructor Menu ***
1) Modify Menu
2) Develop Templates
3) Exit System
Option # ? __
```

Before inserting the templates, the instructor should add the menu option. Therefore, option number two should be selected from this menu. This will produce the following menu:

```
*** MENU MAINTENANCE ***
1) Add New Menu Options
```

- 2) View Existing Menu Options
- 3) Modify Existing Menu Options
- 4) Delete Existing Menu Options
- 5) Return To Main Menu

Option # ? \_\_\_\_

Option number one should be selected at this point. A description of it and the other options is following.

#### Adding Menu Options

When adding menu options, two things need to be input. The first is the menu 'title', or the entry that will appear on the actual menu screen when displayed to the user. The second is the accompanying instructions. If a student should pick a particular option from the menu (based on the 'title' above), he/she should receive some instructions giving specifications for a CORE segment that is to be written.

The computer will prompt the instructor for the menu's title and instructions. The menu option number (number one for the first entry) is generated automatically. THERE IS AN UPPER LIMIT OF 20 AVAILABLE MENU OPTIONS. Also, the instructions given to the student may not exceed 240 characters (three eighty-character lines). When the instructor has completed the typing of his/her instructions into the computer, he/she should terminate them with an ampersand '&'. This signals the end of the instructions to the computer. A sample dialogue between machine and instructor is given for example. Capital letters indicate instructor input.

Menu Option Number => 1

Enter Menu Title - 30 character limit => SUMMATION OF NUMBERS 1 TO 10

Enter student instructions, there is a 3 line (240 character) limit  
Enter a '&' to terminate the instructions:

YOU ARE TO WRITE A PROGRAM THAT WILL SUM THE NUMBERS FROM ONE TO TEN.  
THE USE OF A 'WHILE LOOP' IS RECOMMENDED FOR SOLVING THIS PROBLEM. &

The computer will then display the information received for the instructor's verification. Corrections are allowed if necessary. After the data is verified it is saved to a file for future reference. The computer will then return to the 'MENU MAINTENANCE' menu and await further input.

#### Viewing Existing Menu Options

When option number two is selection from the 'MENU MAINTENANCE' menu, the instructor is allowed to view the menu options and accompanying instructions that are currently available to the user. The instructor is first shown the full selection of student menu options as displayed to the student. There are two methods of viewing these options. The instructor may either automatically view all student menu options, or he/she may view selected options. If the instructor chooses to view all the available options, the computer begins with student option number one, displays it and its accompanying instructions, and continues with each successive student menu option until all have been displayed for viewing. Should the instructor choose only to view selected options, the computer prompts for the desired option and complies by displaying it for view. After the viewing has been completed, the computer returns to the 'MENU MAINTENANCE' menu.

### Modifying Menu Options

We know that instructors do not make mistakes, but they may occasionally change their minds. For this reason, the instructor is allowed to modify the title and/or accompanying instructions of any student menu option. To begin, the student menu, as displayed to the student, is displayed for the instructor. The computer then prompts for the menu option number that he/she wishes to modify. When the correct option is received, the computer displays the current information in the following format:

```

1) Menu Option => 1
2) Menu Title => SUMMATION OF NUMBERS
 1 TO 10
3) Instructions =>
 YOU ARE TO WRITE A PROGRAM THAT WILL SUM
 THE NUMBERS FROM ONE TO TEN. THE USE OF
 A 'WHILE LOOP' IS RECOMMENDED FOR SOLVING
 THIS PROBLEM.

 Modify Which Number ? =>
 - Enter 0 (zero) to Quit -

```

The instructor may then enter a one, two, or three, depending on the data he/she wishes to modify. IT IS RECOMMENDED THAT THE INSTRUCTOR NOT CHANGE THE MENU OPTION NUMBER. This number is generated automatically, a change may disrupt the sequence.

If the instructor decides not to modify the existing data, a zero may be entered and control will pass back to the 'MENU MAINTENANCE' menu.

### Deleting Menu Options

The instructor is also allowed the opportunity to delete any existing student menu option from the SES. Before deletion, the

student menu is displayed for the instructor. The computer then prompts for the number to be deleted. The instructor may enter a zero at this point to return to the 'MENU MAINTENANCE' menu and avoid deleting any options. Should the instructor wish to proceed, however, he/she is offered no second chance. The instructor should type in the menu option number of the data to be deleted and press return. This marks the record, in effect allowing a new record to be written over the old.

## CHAPTER III

### DEVELOPING THE SES TEMPLATES

After the instructor has entered the student's menu option(s) he/she is now ready to attach the template references. Option number one should be chosen from the main menu (figure # ?). The following menu will be displayed to the instructor.

**\*\* TEMPLATE MAINTENANCE \*\***

- 1) Add New Templates
- 2) View Existing Templates
- 3) Modify Existing Templates
- 4) Delete Existing Templates
- 5) Return to Main Menu

Option # ? \_\_\_\_

The instructor is allowed to enter any number of templates, correct or incorrect, for any existing student menu option. Following are the details of each above option.

#### Adding New Templates

In order to add new templates to the system, select option number one from the templates menu. When adding a template, the instructor is prompted for the student menu option with which the template should be associated, the external filename that contains the source code of the template, the point value that is assigned for a correct match to this

template, and information stating whether the template is a correct or incorrect response. A sample dialogue follows. Data in capital letters represents instructor responses. Before the interaction begins, the student menu is displayed for the instructor's viewing.

Enter Menu Option Number to  
which this template belongs           => 1

Enter Full Name of Template file,  
10 Characters Maximum Length       => TEMP1.COR

Is the Template Correct or  
Incorrect ? (C or I)               => C

How Many Points are Available for  
a Match to this Template ?       => 25

The information that was received from the instructor is then displayed for verification. The instructor is allowed to make changes and/or corrections. After the data is verified, it is stored in an external file for future reference, and control is passed back to the Templates Menu.

#### Viewing Existing Templates

In order to view any or all of the existing templates, the instructor should select option number two from the Templates Menu. The program allows the user to view all the currently existing templates that are stored in the system, or to view only the templates associated with a given student menu option. The student menu is displayed to the instructor and the computer asks if the instructor wishes to view all or selected templates. Should the user wish to view all the system templates, the program begins with student option number one, displays all information regarding each associated template, and continues to do so until the last student option templates have been



displayed. On the other hand, should the user desire to view only selected template(s), the computer requests the student menu option number to which the template(s) belong and displays information concerning the related template(s). Control is then returned to the Template Menu.

### Modifying the Templates

In order to correct information stored on a particular template, the instructor should select the third option from the Template Menu. The student menu is displayed for the instructor's view, and the computer requests that the user enter the student menu option that is associated with the template to be modified. A screen will then appear that displays each template that is associated with the given student option. The computer requests the instructor to indicate which template he/she wishes to modify. The information is then displayed as follows:

- 1) Menu Option => 1
- 2) Filename => TEMPL.COR
- 3) Template is => CORRECT
- 4) Point Value => 25

Modify Which Number?  
Enter 0 (zero) to Quit =>

The instructor is given a chance to return to the Templates Menu without making any modifications by entering a zero. If, however, changes need to be made, the corresponding number should be entered (for example, to change the file name, enter number two) and the computer will prompt for the correct information. When all changes have been made, the instructor should enter a zero to return to the

Templates Menu.

### Deleting Existing Templates

Templates may be deleted from the system by choosing option number four from the Templates Menu. The computer will display the student's menu for the instructor and prompt for the student option to which the template belongs. After the instructor has entered the correct option number, the computer will display a numbered list of all the templates associated with the given option. The instructor is then allowed to enter the number of the template to be deleted. There is no turning back at this point. Once the number has been entered, the template is deleted. Control will then return to the Templates Menu.

Throughout the design of these menus, care has been taken to preserve the integrity of the data entered. For example, if the computer is expecting a numerical response (as in the selection of a menu option) and the user inputs a character, the computer displays an appropriate error message and allows the user to re-enter his/her response. The same is true if the computer is expecting a particular character or character string response (as in 'Y' or 'N'). It is believed that the instructor's menu is user friendly and will present no obstacles to the user.

## APPENDIX B

### SES STUDENT USER'S MANUAL

#### Purpose

The purpose of this manual is to guide the student user in the evaluation of program segments using the OSU Statements Evaluation System.

## PREFACE

### Evaluating a Program Segment

There are three steps that must be completed in order to evaluate a program segment. The user must:

- a) choose an option from the Student Menu to obtain instructions for the development of a program segment;
- b) code and debug the CORE source code program that will meet the specifications given in step one (above);
- c) submit the program segment for evaluation against the stored expected responses.

Each step is fully documented in this manual.

## CHAPTER I

### THE STUDENT MENU

The student menu contains a maximum of twenty options that are developed by an instructor to challenge a student user. To access the menu, type the following at the VAX/VMS prompt (do not type the '\$' prompt):

```
$ SMENU
```

The menu will appear and the student will be requested to choose an option that is of interest. An invalid response will be ignored, and the computer will continually prompt until a valid response is received. Once a correct input is obtained, the computer will display for the student the instructions that accompany his/her menu choice. Assuming the student chooses option six from the Student Menu, a screen similar to the sample below will be presented.

```
Menu Option Number => 6
```

```
Instructions: You are to write a program that will sum the
 numbers from one to ten. It is recommended
 that you use a 'WHILE' loop to implement
 your solution.
```

The student is then given the opportunity to continue with this request, return to the menu and make another choice, or exit the system. If the student elects to proceed, the following screen will appear (continuing with the previous example, menu option number six has been chosen):

```
INSTRUCTIONS TO PROCEED...
```

- 1) Your menu choice is number => 6  
REMEMBER this number.
- 2) Write a program using the mini-language CORE that will fulfill the given instructions. Compile and debug your program. To compile your program, type:

CORE filename

where 'filename' is the name of your source file.

- 3) When your program is working properly, type:

COMPARE

The computer will compare your program segment to various correct and incorrect templates that have been prepared for the menu option chosen. The comparison program will prompt you for the menu option that you chose. When it does, enter the number given the step one (1) above.

At this point, control will return to the VMS operating system, and the student should begin developing a program segment that will conform to the specifications set forth by the instructor. The program should be written the the Legard and Marcotty mini-language CORE. The syntax specifications for this language are provided. Instructions on developing and compiling a CORE program are given in the next section.

## CHAPTER II

### DEVELOPING AND COMPILING A CORE PROGRAM

#### Format and Content

The template may be developed by using any standard editor. The syntax of the source code must be in accordance with the standards set forth for the Legard and Marcotty mini-language CORE. The CORE language is designed for simple data manipulation. It is not a powerful language and should not be expected to perform large computational problems. It will perform best when used with its purpose and abilities in mind.

The following guidelines should be observed when developing the CORE program.

- 1) CORE allows only one data type, that of INTEGER. All variable names must be declared and the length should not exceed ten characters.
- 2) The source code should be written in lowercase letters. Identifier names may contain a combination of letters, digits, and underscores. The compiler, however, does not distinguish between upper and lowercase letters. Therefore, if an identifier name of 'Tax' is declared, the user must not use 'TAX' or 'tax' within the program body. The compiler sees these as three separate variables, the last two of which are undeclared.
- 3) There are three mathematical operators, those being addition (+), subtraction (-), and multiplication (\*). Division is not allowed.
- 4) There are six logical operators: equal, not equal, less than, greater than, less than or equal, and greater than or equal.
- 5) All statements must be terminated with a semi-colon.

- 6) 'If' statements may be embedded. Each statement should be terminated with a corresponding 'end if'.
- 7) 'While/loop' statements may be embedded. Each statement must be terminated with an 'end loop'.
- 8) The use of parentheses is allowed to manipulate the order of operation in expression (assignment) statements.

### Compiling the Program

Once the program has been written and saved, it may be compiled by typing:

```
$ICORE filename
```

where 'filename' is the name of the source code. The length of the 'filename' should not exceed ten characters (this includes the file extension). The CORE compiler may generate any of nine error codes. Following is a list of the codes and their descriptions.

| Code | Problem                                 |
|------|-----------------------------------------|
| 1    | Illegal Variable Name                   |
| 2    | IF/END IF statements not sequenced      |
| 3    | BEGIN/END statements not sequenced      |
| 4    | WHILE/END LOOP statements not sequenced |
| 5    | Missing PROGRAM statement               |
| 6    | Variable already declared               |
| 7    | Undeclared variable name                |
| 8    | Variable name exceeds maximum length    |
| 9    | Syntax Error: ';','(',')' expected      |
| 10   | No Errors or Warnings                   |

Definitions are as follows:



| Code | Definition                                                                                                  |
|------|-------------------------------------------------------------------------------------------------------------|
| 1    | Illegal character(s) in the variable name.                                                                  |
| 2    | An 'IF' statement was found with no corresponding 'END IF' statement.                                       |
| 3    | A 'BEGIN' statement was found with no corresponding 'END' statement.                                        |
| 4    | A 'WHILE' statement was found with no corresponding 'END LOOP' statement.                                   |
| 5    | Either the wrong file was submitted for compilation, or the source code is missing the 'PROGRAM' statement. |
| 6    | The same variable name appears twice in a declaration sequence.                                             |
| 7    | A variable name has been found in the program body that was not found in the declaration sequence.          |
| 8    | An identifier name exceeds the length (in characters) of the maximum length (10 characters).                |
| 9    | Check the given line number for missing parentheses, semicolons, etc.                                       |
| 10   | The program is syntactically correct.                                                                       |

After the program has been successfully compiled, the student may submit the program for evaluation. Instructions for the last step are in the next section.

## CHAPTER III

### SUBMITTING A PROGRAM FOR EVALUATION

After the student has carefully designed, edited, and compiled his/her program, it may be submitted for comparison by typing:

```
$COMPARE
```

at the VAX/VMS prompt (again, do not by the '\$' shown, it is the VMS prompt). The SES will ask for the menu option number to which the student has responded. Continuing with the above example, the student would enter a six in response. If the student has input the correct option number, the computer will proceed with its comparison. If the student input the wrong option number, he/she is given the opportunity to correct the entry.

The computer will compare the program segment against all existing templates that are stored with the given option number. The program may be matched to either an expected correct or incorrect template, or it may be that the program can not be matched to any template. In any case, the user will be informed of the outcome of the comparison. One of the following responses will be given:

- 1) Your program has been SUCCESSFULLY MATCHED TO A CORRECT template
- 2) Your program has been matched to an expected INCORRECT template, Please reconsider your approach and try again...
- 3) Your program can not be matched to either a CORRECT or an INCORRECT template. Program Correctness can not be determined at this time...

The student may change or correct his/her program and resubmit it

at any time. When resubmitting a program for evaluation against the same menu option number, it is not necessary to complete step one (displaying the student menu). The student simply may repeat steps two and three (compiling and submitting the program for comparison) as many times as desired.

## APPENDIX C

### MINI-LANGUAGE CORE SYNTACTIC CATEGORIES

|                      |     |                                                                                                                   |
|----------------------|-----|-------------------------------------------------------------------------------------------------------------------|
| program              | ::= | program<br>declaration...<br>begin<br>statement...<br>end;                                                        |
| declaration          | ::= | DECLARE identifier [ , identifier ] ...;                                                                          |
| statement            | ::= | assignment-statement<br> <br>if-statement<br> <br>loop-statement<br> <br>input-statement<br> <br>output-statement |
| assignment-statement | ::= | identifier := expression ;                                                                                        |
| if-statement         | ::= | IF comparison THEN<br>statement...<br>[ ELSE<br>statement... ]<br>END IF ;                                        |
| loop-statement       | ::= | WHILE comparison LOOP<br>statement...<br>END LOOP ;                                                               |
| input-statement      | ::= | INPUT identifier [ , identifier ] ...;                                                                            |
| output-statement     | ::= | OUTPUT identifier [ , identifier ] ...;                                                                           |
| comparison           | ::= | ( operand comparison-operator operand )                                                                           |
| expression           | ::= | [ expression + ] factor<br> <br>[ expression - ] factor                                                           |
| factor               | ::= | [ factor * ] operand                                                                                              |
| operand              | ::= | integer<br> <br>identifier<br> <br>( expression )                                                                 |
| comparison-operator  | ::= | <   <=   >   >=   =   <>                                                                                          |

```
identifier ::= letter [[_] letter] ...
integer ::= digit ...
```

VITA

Kay Ellen Slack

Candidate for the Degree of

Master of Science

Thesis: CORRECTNESS AND LEVEL OF INCORRECTNESS DETERMINATIONS OF PROGRAM SEGMENTS; FUNCTIONAL EQUIVALENCE OF WHILE-DO STATEMENTS

Major Field: Computing and Information Sciences

Biographical:

Personal: Born in Ardmore, Oklahoma, December 6, 1961, the daughter of Dr. Harold W. and Mary A. Slack.

Education: Graduated from Madill High School, Madill, Oklahoma, in May 1980; received Bachelor of Science Degree with double major in Business Management and Computer Information Systems from East Central Oklahoma State University in May, 1984; completed requirements for the Master of Science degree at Oklahoma State University in December, 1988.

Professional Experience: Programmer/Analyst, University Computer Center, Oklahoma State University, January, 1985 to August, 1987; Computer Science/Information Systems Instructor, Southeastern Oklahoma State University, August, 1987, to August, 1988.