1991

The Edison-ES programming language

John E. Davis
Lehigh University

AUTHOR:

Davis, John E.

TITLE:The Edison-ES Programming Language

DATE:January 1992

THE EDISON-ES PROGRAMMING LANGUAGE

by

John E. Davis

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of
Master of Science

in

Computer Science

Lehigh University

1991

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Dec 4, 1991
(date)

Professor in Charge

CS Division Chairman

CSEE Department Chairman

ACKNOWLEDGEMENTS

The author would like to express his appreciation to Professor Samuel Gulden for introducing him to parsing theory and compiler design and for his continued support throughout this project and the author's previous work with Edison. Appreciation is also expressed to Professor Edwin Kay for introducing the author to his first parser.

Expressions of gratitude are also extended to Janet Laubenstein, a colleague at Northampton Community College who previously worked with the author on the Edison system. Her continued interest and support throughout the development of the one-pass compiler was greatly appreciated.

A special note of appreciation and thanks to my wife, Sally, my children, John, Megan, and Bryan, and my mother. Your understanding and support has been appreciated even though at times during the work on this project I appeared to be a permanent fixture in front of the computer.

TABLE OF CONTENTS

The Edison-ES Programming Language

John E. Davis

ABSTRACT

This paper describes the Edison-ES programming language, and the construction of a one pass compiler for the language.

Edison-ES is an extension of a subset of the Edison programming language described by Per Brinch Hansen. Similar in appearance to Pascal, Edison was originally designed for writing the Edison Operating System for use on personal computers. Except for modules, procedure parameters and library procedures, Edison-ES contains all of the Edison constructs for writing sequential programs. None of the Edison constructs for writing concurrent programs have been included in Edison-ES. Character and numerical input and output have been added as extensions to this subset of Edison.

The one-pass compiler for Edison-ES is for use on a MS DOS microcomputer and is written in TURBO PASCAL. The compiler, which consists of three TURBO PASCAL UNITS and the main program, emits assembly code for the INTEL 8086 microprocessor which can be assembled and

linked using any 8086/88 assembler and MS DOS linker to obtain an executable file.

CHAPTER 1 THE EDISON LANGUAGE

Edison is designed as a programming language which is simple to use and understand, while supporting modular construction of sequential as well as concurrent programs for use on a personal computer.[1] The subset of Edison which is to be described is one which supports the creation of sequential programs. Since the programs are to be executed under MS DOS rather than the Edison system, constructs are omitted from the grammar which do not appear to be needed in the MS DOS environment. The extension of the subset to include basic input and output functions appear to be necessary if any use is to be made of the language and the compiler described later. This extended subset of Edison is referred to throughout this paper as Edison-ES.

Edison is a LL1 block structured programming language similar in appearance and semantics to Pascal. The design of Edison has taken the necessary, basic structures from Pascal and used them as a basis for Edison. Edison contains some constructs similar to those of Pascal, but omits others which are thought to be either redundant or unnecessary. Since simplicity

1. Per Brinch Hansen, Programming a Personal Computer. Englewood Cliffs NJ: Prentice Hall, Inc, 1982, pg. 8.

is a main goal in the design of Edison, structures whose function could be accomplished using previously defined structures are not included in Edison.

The similarities with Pascal are numerous. Programs are constructed of procedures, which communicate with other procedures through the use of both variable and value parameters. Some procedures may be assigned a value themselves in which case Pascal and Edison refer to the procedure as functions. Edison functions do not require the reserved word, **function**, but are declared like any other procedure except that it must have a data type. Like Pascal, Edison requires that all identifiers be assigned a data type. Edison supports the same data types as Pascal except for the pointer, variant record, and real types. Only one basic repetitive structure, the **WHILE** loop, is included in the language. The **REPEAT** and **FOR** loops are not included since they can be written as **WHILE** loops and therefore are not needed. **CASE** statements from Pascal are not included, since they can be replaced by the **IF-ELSE** in Edison.

Even with all of these similarities with Pascal, there are obvious differences. First, Edison can be used to write either sequential or concurrent programs. The sequential programs do indeed look similar to

Pascal programs. The concurrent programs, which cannot be written in Pascal, are made possible through the use of **WHEN, COBEGIN, ALSO, PROCESS** constructs. Since Edison is to be used for writing an operating system, library procedures are supported through the use of a **LIB PROC** construct. Input and output are not included since they are considered to be system dependent and therefore not able to be programmed in the high level language itself. Modules are also available in Edison. The module permits importing data from a surrounding block and exporting data to an enclosing block. The data received by a module or exported from a module is manipulated by operations entirely within the module. Since Edison programs are procedures called by other Edison programs, including the operating system, programs can have procedure parameters passed to them. In order to maintain consistency, any procedure is permitted to have procedure parameters in Edison. These procedure parameters, unlike those in Pascal, also include parameter lists. Edison uses an **int** function to replace the **ORD** function of Pascal. Thus **int('A')** in Edison yields the ASCII value of 'A' or 65. Edison uses the **char** function to replace Pascal's CHR function. Thus in Edison, **char(int(65))** = 'A'. Unlike Pascal, Edison also provides for an inverse mapping for

boolean values through the use of the **bool** function. Thus if **done** is an identifier of type **bool**,

bool(int(done)) = done.

Some semantic differences also exist between Pascal and Edison. Pascal and Edison both use the semicolon as an end of statement delimiter. Its use indicates another statement in the list follows the current statement. Pascal, however, permits the use of the blank or empty statement while Edison does not. Therefore in Edison a semicolon is not permitted prior to an **END** symbol, while Pascal does permit a semicolon prior to the **END** symbol. The use of the **END** symbol does not depend on a corresponding **BEGIN** symbol. In Edison, all **IF** and **IF-ELSE** constructs require an **END** symbol. This eliminates the classic "dangling else" problem found in Pascal with Pascal's **IF-THEN-ELSE** construct. All **WHILE** loops likewise require an **END** symbol. The only time a **BEGIN-END** construct is used is to indicate the beginning and end of the statement part of a procedure. Pascal's compound statement has been replaced with a statement list construct, which as mentioned earlier, concludes with a statement that is not followed by a semicolon.

CHAPTER 2 THE EDISON-ES LANGUAGE

2.1 DIFFERENCES WITH EDISON

Edison-ES as mentioned previously was designed as a subset of Edison, extended by including support for input and output. Concurrent programs were not of primary concern when defining Edison-ES, and were therefore not included in the definition of the language. Thus, the **COBEGIN**, **ALSO**, **WHEN**, and **PROCESS** constructs are not included in Edison-ES.

The target machine and environment played a large role in the design of the language. Because of these two factors, procedure parameters were not included in Edison-ES. Programs written in Edison-ES were expected to be run on INTEL 8086/88 based hardware in an MS DOS environment rather than the Edison system environment. In the Edison environment, programs, which are complete procedures, were called with parameters from the operating system. Procedure parameters were included in Edison because of the need to call programs with procedure parameters. In order to maintain generality, Hansen permitted all procedures to have parameter procedures. Since the main reason for supporting procedure parameters was no longer valid, this construct was not included in Edison-ES.

The elimination of the procedure parameter actually simplified the writing of Edison programs. The recommended method for writing Edison programs was to write programs with a standard program prefix. The program prefix was merely a procedure heading containing as parameters, all the operating system declarations and procedures that would be needed by any Edison program. When writing an Edison program the prefix with all its declarations and procedures would be the first procedure heading. All input, output, loading of library procedures, and disk drive operations were then passed to the program as procedure parameters when the program was called by the operating system. Because of the elimination of these procedure parameters, no program prefix is required. Instead, the main procedure has no parameters at all.

One of the parameters that was passed to an Edison program when called by the operating system was a load procedure. This procedure would be also used as a parameter for any call to a library procedure. Library procedures were loaded into memory as needed during execution of a program. The procedure was in memory only while being executed. When execution of the library procedure terminated, the memory it occupied was free to be used by another library procedure.

Library procedures were precompiled procedures that were loaded in memory above the program. This necessitated maintaining the top of code address at all times. Library procedures were not considered to be a necessity and were therefore not included in Edison-ES.

Edison includes an interesting structure called a module. A module is a collection of data entities and procedures that are either used within the module or exported to the immediately enclosing block. The exported procedures, variables, types, and constants may be called or referenced by the block immediately enclosing the module. Procedures which are exported are defined within the module and may use procedures and variables which are also defined within the module but not exported. This insures the integrity of the procedures in the module by permitting only locally defined entities to be operated on by these procedures.

Consider the classic **STACK** data structure, consisting of a stack top, a sequential collection of data items, and the operations to create the stack, test for an empty stack or a full stack, and push and pop items onto and from the stack. A **module** could be declared to export a variable **S** of type **stack** which could be an exported record type. The procedures for pushing and popping would likewise be exported. The

procedure for creating a stack need not be exported since this would be used to initialize the exported variable **S**. The functions for testing for an empty stack or full stack would not be exported, since they are needed by the pop and push procedures only. Thus **S**, a record type **stack**, and two procedures for operating on the stack would be available to the enclosing block. The enclosing block would not need to know how the procedures were defined or the actual definition of **S**, but just that the procedures and variable **S** of type **stack** were available. Exported entities are prefaced by an asterisk. The procedure, **halt**, is a standard Edison procedure for terminating the program and may be called anywhere in the program. The following is an example of such a module.

```
MODULE   "stacks"
   CONST
      max = 500
   ARRAY list[1:max] (int)
 * RECORD stack (l: list; top: int)
 * VAR
      S: stack
   PROC fullstack(VAR S: stack): BOOL
      BEGIN
         IF S.top = max DO
```

```
            VAL fullstack:= TRUE
         ELSE TRUE DO
            VAL fullstack:= FALSE
         END
      END        "fullstack"
   PROC is_mtstack(VAR S: stack): BOOL
     BEGIN
        IF S.top = 0 DO
           VAL is_mtstack:= TRUE
        ELSE TRUE DO
           VAL is_mtstack:= FALSE
        END
     END        "is_mtstack"
* PROC push(VAR S: stack; y: INT)
     BEGIN
        IF NOT fullstack(S) DO
           S.top:= S.top + 1;
           S.l[S.top]:= y
        ELSE TRUE DO
           HALT
        END
     END        "push"
* PROC pop(VAR S: stack; VAR y: INT)
      BEGIN
         IF NOT is_mtstack(S) DO
```

```
            y:= S.l[S.top];
            S.top:= S.top - 1
        ELSE TRUE DO
            HALT
        END
    END      "pop"
PROC createstk(VAR S: stack)
    BEGIN
        S.top:= 0
    END      "createstk"
BEGIN      "module stacks"
    createstk(S)
END
```

Each module is initialized by executing its statement part before creating the entities defined within the module. The notion of the module is most interesting, but not truly necessary. Thus they are not included in Edison-ES.

Console input and output of characters and integers was considered to be a necessity. With the unavailability of the input and output procedures of the Edison system, input and output procedures had to be added to the language or passed as parameters to the program. Passing them as parameters would once again require the use of a standardized procedure heading for

the main procedure, so extending the language to include these constructs seemed appropriate. These procedures are included as standard procedures with the names as reserved words. The procedures are coded using MS DOS interrupts and function calls.

Except for the constructs noted, and the inclusion of a condition that all constant, type, and variable declarations be made before local procedures are declared, Edison-ES is identical to the Edison grammar described by Hansen. Using standard algorithms for finding the null nonterminals, the follow sets,[2] and the first sets[3] for each of the nonterminals in the grammar, the resulting grammar can be verified to be LL1. A listing of the grammar symbols, the first sets, the follow sets, and the set of all null nonterminals is included in Appendix A1.

2.2 CHARACTERS, IDENTIFIERS, AND COMMENTS

Identifiers in Edison-ES are similar to those in Pascal. One change is the inclusion of the underscore character anywhere in the identifier name after the

---

2. C.N. Fischer and R.J. LeBlanc, Crafting A Compiler. Menlo Park, CA: Benjamin Cummings Pub. Co. Inc., 1988, pp. 103-105.
3. W. Barrett, R. Bates, D. Gustafson, J.Couch, Compiler Construction. 2nd Ed. Chicago: SRA, Inc., 1979, pp. 154-156.

first character. Identifiers must have a letter as the first character followed by any sequence of characters containing letters, underscore characters, or digit characters. The Edison-ES compiler will accept identifiers up to eighty characters long, but recognizes only the first ten characters. Identifiers longer than ten characters should be unique within the first ten characters. The compiler is not case sensitive, so that upper and lower case characters may be mixed throughout a program.

Identifiers may be used to represent constant data or variable data. All identifiers are of a specific type which is either a standard type or a user defined type.

A constant identifier may represent any numerical value, character symbol, truth symbol, enumeration symbol, or another constant identifier. Character constants and control characters are to be enclosed in single quotes when used within a program. The truth symbol constants are **TRUE** and **FALSE**.

Comments are to be enclosed in double quotes. Any text enclosed within double quotes is ignored by the compiler.

2.3 DATA TYPES

Data entities are grouped into classes depending on the set of values the entity may assume. These classes are the type to which the entity belongs. Types are considered to be elementary types and structured types. The elementary types are the standard types or enumeration types. The structured types permit definition of user defined types consisting of one or more elementary or other structured type.

For known types, standard types or types which have been declared, a variable is assigned a type either in a parameter list or in a variable declaration. A variable declaration begins with the word, **var**. The syntax for a variable declaration would be written

```
var
    v1: type_i;
    v2, v3: type_j;
    v4: type_k        .
```

The types may be the same or they may be different. Note that a semicolon is used to separate the variable declarations in the variable declaration list. The

last variable declaration in the list is not followed by a semicolon.

2.31 ELEMENTARY TYPES

The elementary types in Edison-ES are type **bool**, for boolean entities; type **int**, for non-fractional numeric entities; type **char**, for entities with character values; and type **enum**, for entities consisting of enumerated symbols. There is no type for fractional numeric entities. Hence all numeric data must be integer.

Identifiers of type **bool** have values that are either true or false. When printing, the Edison-ES compiler emits a value of 5 for true and a value of 4 for false. This is due to the assignment of constant names in the compiler. Identifiers of type **bool** may be operands for the boolean operators: **and**, **or**, and **not** where the usual logical rules apply.

An identifier of type **int** has values that are in the set of integers and which are between -32,768 and 32,767, the **-maxint** and **maxint** values for TURBO PASCAL on the MS DOS microcomputers. Identifiers of type **int** may be operands for the integer operations of +, -, *, **div**, and **mod**, which correspond respectively to addition, subtraction, multiplication, integer

division, and modulo. These identifiers may be compared using the usual numerical relational operators, =, <>, <, <=, >, >= which are used for equal, not equal, less than, less than or equal, greater than, and greater than or equal respectively. A comparison results in a value of type **bool**.

The identifiers of type **char** may be used to represent characters from the system's character set. Characters may include a printable character such as a letter or digit, or a control character. Printable characters are designated by enclosing the character in single quotes, 'A', for example. Control characters are designated by enclosing the character's ordinal value in parenthesis and prefacing the enclosed number with the symbol, **char**. Identifiers of type **char** may be compared using the usual numerical relational operators, =, <>, <, <=, >, >=, since the ordering of the characters is dependent on the numerical code used to represent the character set.

An enumeration type assigns a list of identifiers to a type name. Each identifier in the list is mapped to the ordinal position of the identifier in the list. The declaration of an enumeration type is made as a type declaration where the reserved word, **enum** would be followed by the type name being defined. This is

followed by the enumerated list of names enclosed in parenthesis. For example, to declare an enumeration type named **colors**, the construct

enum colors (red, blue, green, yellow)

would be used. In this example, **int(red)** = 0, **int(blue)** = 1, **int(green)** = 2, and **int(yellow)** = 3. Values of an enumeration type may be compared using the usual numerical relational operators, =, <>, <, <=, >, >= since the ordering is based on the ordinal position of the value in the enumeration list.

2.32 CONSTANTS

Constants are declared using a construct that starts with the symbol, **const**. Following this symbol is a list of statements assigning values to constant identifiers. Each of the statements in the list is of the form **const_id** = **value**, where **const_id** is the constant identifier being defined and **value** is the value assigned to the **const_id**. Each of the statements in the list is separated from the previous statement by a semicolon. The end of the list is determined by the absence of a semicolon after the last statement.

2.33 STRUCTURED TYPES

A structured type is a data type that can be constructed of components of other known types. The structured types in Edison are the array, the record, and the set.

2.331 ARRAYS

An entity of type array is a finite collection of data, all of the same type, where each data item is accessed by its position in the collection. A position within the collection is denoted by an index. An index is a value in a set of successive values bounded below and above by constant symbols. This set of bounded values is called a range. For example, if **L** is an array with indices in the range 5 through 25, the range for **L** would be denoted as **5:25**. Accessing item 6 in the collection **L** would be accomplished through reference to **L[6]**. Since the indices for **L** start at 5, item 6 would actually be the second item in the collection. These definitions are compatible with those of an array and range found in Pascal and other high level languages. Edison has no subrange types. To declare an array type called **list** with a range 1

through 25 where the items in list are integers, the construct

array list [1:25] (int)

would be used. The type name being defined is **list** and **int** is referred to as the base type. The range is enclosed in brackets with the lower bound, which is listed first, separated from the upper bound by a colon. The base type is always enclosed in parenthesis. Identifiers of type **array** may be compared, provided they are of the same type, using the relational operators of = and <> for equal and not equal respectively. Two arrays of the same type are equal if the corresponding elements in each array at each position in the range of the array are equal.

2.332 RECORDS

The record type permits the construction of a finite collection of data of different known types where each data item in the collection is known by an identifier called a field name. A record's field is accessed by using the record name followed by a period which is followed by the name of the field to be accessed. For example, if a variable, **FIGUR**, of type record had fields: **fig**, **length**, **width**, and **height**, the **height** field of the record would be accessed using the

notation **FIGUR.HEIGHT**. This again is similar to Pascal. The record type described above would be declared using

enum figkind (cube, rectangular, cylinder)
record figur(fig: figkind; length, width, height: int).

Two records of the same type may be compared using the equal and not equal relational operators. Two records of the same type are equal if all of the fields in one record are equal to each of the corresponding fields in the other record.

2.333 SETS

The final structured type is the set which is a finite collection of values of the same known elementary type, known as the set's base type. Elements of a set must have ordinal values in the range 0 through **SETLIMIT**, a system dependent constant. In Edison-ES the **SETLIMIT** is defined to be 127. To declare **CHARSET**, as a set of characters, the construct

set CHARSET (char)

would be used. The set may be empty or it may contain any number of elements with values in the range 0 through **SETLIMIT**. The usual set operations of union, intersection, and difference are defined for use with set operands in Edison. These are indicated by the

symbols +, *, and - respectively. Set inclusion is determined by the **IN** relational operator. An element **x** is in a set **S** means that **x** is a value in the base type of set **S**, and the ordinal value of **x** is between 0 and **SETLIMIT** inclusive.

Each identifier of one of the structured types is declared prior to the beginning of the statement part of the procedure in which it is declared. The order of declaration of variables, constants, and structured types is not dictated by the grammar. The declarations of these entities may be in any order, and may in Edison be mixed with the procedure and module declarations. In Edison-ES there is also no specification on the order of the declarations for constants, variables, and types. However, in Edison-ES these declarations must be completed before declaring procedures and modules within a procedure.

2.34 CONSTRUCTORS

Edison and Edison-ES have a structure referred to as a constructor. A constructor is used with both elementary types and all structured data types. In an elementary type, a constructor indicates a mapping between the values listed in the constructor and the name in that ordinal position in the elementary type.

Thus for the elementary types **bool**, **int**, and **char**, the mappings **y:= bool(x)**, **y:= int(x)**, and **y:= char(x)** are constructors which assign to y the value whose ordinal value is x in the set of **bool** values, **int** values, and **char** values respectively. In a structured type, the constructor indicates a mapping of the values listed into the component of that structured type that is at the same ordinal position in the structured type. Thus a constructor is used to assign values to a variable whose type is the same as that of the constructor.

Consider the following declarations.

```
enum figkind (cube, rectangular, cylinder)
record figur(fig: figkind; length, width, height: int)
array list[1:5] (char)
set numset (int)
```

If **y:= figkind(2)**, then **y** must be an identifier of type **figkind** and it has just been assigned the value **cylinder**. If **z:= figur(rectangle, 3, 5, 6)**, **z** is a record of type **figur** and the **fig** field has been assigned **rectangle** and **length**, **width**, and **height** have been assigned 3, 5, 6 respectively. If **L** is an identifier of type **list**, the assignment

```
L:= list('a', 'b', 'c', 'd', 'e')
```

is equivalent to the 5 assignments L[1]:= 'a', L[2]:= 'b', ..., L[5]:= 'e'. One expression must be included for each position in the array. The assignment,

S:= numset(1, 2, 5, 50)

creates a set **S** containing the integers 1, 2, 5, and 50 as elements.

2.35 RETYPING IDENTIFIERS

Identifiers may be retyped when passed as parameters to another procedure or when used in an expression. The retyping is accomplished by specifying the new type for the identifier when the identifier is referenced either in an expression or as a parameter in an actual parameter list. This could be considered as a construct that has semantics somewhat similar to that of a record variant. Consider

```
enum grade(A, B, C, D, F)
```

and

```
var
    final_grade: grade;
    cred_hrs, qual_pts: int.
```

Here **final_grade** is of type **grade** and the variables **cred_hrs** and **qual_pts** are of type **int**. Rather than convert **final_grade** to an integer value, the statement

```
qual_pts:= cred_hrs * (4 - final_grade: int)
```

would result in an accepted construct in Edison. **Final grade** would be considered to be of type **int** in this expression, with an integer value which is **int(final_grade)**. Thus the value of a retyped identifier is value of a data item of the new type that has the same internal numerical representation. This would lead to a value of 0 for **final_grade** if **final_grade** had the grade value of **A**. Similarly, if **ch** is an identifier of type **char** and **ch:= '9'** then

ch:int - int('0') = 57 - 48

= 9

assuming the ASCII character set.

The only prerequisites for retyping a variable are that the original type and the new type have the same length and the new type be a known type. Length in this context, refers to the amount of internal storage required to represent a variable of the type being considered. Thus any elementary type may be retyped as any other elementary type, since all elementary types have the same length. It is also possible for a record to be retyped as an array if the total length of all of the fields was the same as the length of the array.

2.4 STATEMENTS

As mentioned previously, Pascal's compound statement has been replaced with the notion of a **statement list**. A **statement list** is a sequence of statements. Each statement in the sequence except the last is terminated by a semicolon. Hence the semicolon serves as a continuation symbol as in Pascal. This construct coupled with the requirement that **WHILE** and **IF** statements require an **END** symbol, eliminate the need for the **BEGIN-END** statement pair found in Pascal. The **WHILE** statement is the only repetitive construct in Edison. The control statement is the **IF-ELSE** where the **ELSE** is optional. Assignment to variables is through the use of an assignment statement. The **SKIP** statement and procedure call complete the list of supported statement constructs.

2.41 THE ASSIGNMENT STATEMENT

The syntax and semantics of the assignment statement is identical to that of the Pascal assignment statement. A variable of known type is followed by the assignment symbol, :=, which is followed by an expression. The expression is evaluated and the value assigned to the variable. Hence the syntax is

VARIABLE := EXPRESSION

where the expression's value and the variable must be of the same type. Assignment of a value to a component of a structured type, assigns a value to only that component while leaving the other components unchanged. Some examples of assignment statements are

```
SUM:= SUM + NUM

L[i]:= L[i+1] - 1

X.LENGTH:= 5
```

A constructor may also be used to assign values to a variable. (Section 2.34). If the declaration for a set, **set numset (int)** were made and **S** is of type **numset**, then **S:= numset(50)** would create a set **S** with the integer 50 as an element. The statement

S:= S + numset(25)

would assign to set **S** the set which is the union of set **S** and the set containing the integer 25.

2.42 THE CONTROL STATEMENT

The control statement in Edison is the **IF-ELSE** statement, where the **ELSE** clause is optional. The statement must end with an **END**. The syntax is

```
IF condition DO
    statement list
END .
```

If the **ELSE** clause is used the syntax is,

IF condition$_1$ DO
statement list$_1$
ELSE condition$_2$ DO
statement list$_2$
ELSE condition$_3$ DO
statement list$_3$
.
.
.
ELSE condition$_n$ DO
statement list$_n$
END

Thus any number of **ELSE** clauses may be used. **Condition$_i$** is an expression that has a value of type **bool**. When executing a control statement, **condition$_i$** is evaluated, and if true, **statement list$_i$** following the symbol **DO** is executed. If **condition$_i$** evaluated to false, control moves to the next **ELSE** clause or to the **END** if no **ELSE** clause exists. If the **condition$_{i+1}$** in this **ELSE** clause evaluates to true, **statement list$_{i+1}$** is executed, otherwise control moves to the next **ELSE** clause or the **END**. If no conditions evaluate to true, no statement lists are executed. When a statement list

has been executed, control moves to the **END**. Thus, at most one statement list is executed, the first statement list for which the preceding condition evaluated to true. Thus the clause, **ELSE TRUE DO statement list$_n$ END** , would have **statement list$_n$** executed if no other **condition$_i$**, $1 \leq i < n$, evaluated to true. Consider two examples.

```
IF (x > 10) OR (x < 0)  DO
  p:= p*x;
  x:= x-1
ELSE true DO
  writenum(p)
END
```

In this example, if **x > 10** or **x < 0** then the statements **p:= p*x** and **x:= x - 1** are executed in order after which control moves to the statement that follows the **END**. For any value of **x**, $0 \leq x \leq 10$, the value of **p** is printed to the output screen.

For the second example, consider the following Pascal **CASE** statement where A, B, C are procedure calls.

```
CASE x OF
    1: A;
    2: B;
    3: C
END
```

If $x \geq 4$ or $x \leq 0$, either of two situations could arise at runtime, depending on the implementation of Pascal that is in use. In one case no procedures are called and control moves to the statement following the **end**. In the other instance, the program terminates with a runtime error. The could be written in Edison so that no runtime error would occur if the value for x were out of range. Since no **CASE** exists in Edison it would be written using the **IF-ELSE** in the following way.

```
IF x = 1 DO
   A
ELSE x = 2 DO
   B
ELSE x = 3 DO
   C
END
```

If $x < 1$ or $x > 3$, then none of the procedure calls are made and control is moved to the statement following **END**. If $1 \leq x \leq 3$, then exactly one of the procedures A, B, or C is called, depending on the value x.

2.43 THE WHILE LOOP

The **WHILE** loop is the only repetitive structure in Edison because the other repetitive structures, **FOR** and **REPEAT** loops, can be written as **WHILE** loops. Edison's **WHILE** loop has interesting syntax and semantics, both of which are similar to those of Edison's **IF-ELSE** construct.

The syntax of the **WHILE** loop is

WHILE condition$_1$ DO
  statement list$_1$
ELSE condition$_2$ DO
  statement list$_2$
  .
  .
  .
ELSE condition$_n$ DO
  statement list$_n$
END

where the **ELSE** clause is optional.

In executing the **WHILE** loop, Edison continues to iterate the loop until all conditions have been evaluated as false. If any **condition$_j$**, $1 \leq j \leq n$, evaluates to true, **statement list$_j$**, following **condition$_j$** is executed. Control then returns back to

the beginning of the loop to begin testing the conditions again. Hence, if no conditions evaluate to true on the initial execution of the loop, no statement lists are executed.

Assume **condition**$_j$, $1 \leq j \leq n$, is the first condition following the **WHILE** to evaluate to true on an iteration of the loop. **Statement list**$_j$ will be the only statement list executed on that iteration of the loop. Care must be taken to insure that one of the conditions will eventually evaluate to false, otherwise the loop cycles forever. Thus an infinite loop would occur if the construct

```
readnum(x);
WHILE x > 0 DO
  A;
  readnum(x)
ELSE x = 0 DO
  B;
  readnum(x)
ELSE x < 0 DO
  C;
  readnum(x)
END
```

were used. No way exists to terminate this loop through testing the value for x, so that a condition

which is not totally dependent on the value of x must be used to eliminate the infinite loop. One possibility would be the through the use of a variable, **done: bool.**

```
readnum(x);
done:= false;
WHILE NOT done DO
    IF x > 0 DO
      A;
      done:= false;
      readnum(x)
    ELSE x = 0 DO
      B;
      done:= true;
      readnum(x)
    ELSE x < 0 DO
      C;
      done:= true;
      readnum(x)
    END          "IF"
END      "WHILE"
```

If the **ELSE** clause is omitted, the **WHILE** loop is semantically equivalent to a **WHILE** loop in Pascal.

2.44 THE SKIP STATEMENT

The **SKIP** statement is used for a statement that does nothing. Executing the **SKIP** has no effect on the program. It is used to indicate an empty statement.

2.45 PROCEDURE CALLS AND FUNCTION CALLS

Procedure calls in Edison-ES are similar to those of Pascal, a **procedure name** followed by an optional actual parameter list enclosed in parenthesis. The procedure name must be known to the program through a previous declaration. The optional parameter list contains parameters which are mapped one-to-one to the parameters in the parameter list of the procedure heading. In the mapping, each parameter in the actual parameter list is mapped with the formal parameter in the same ordinal position in the formal parameter list. Parameters mapped to each other must be of the same type, or they may be retyped in the actual parameter list so as to match the type of the corresponding formal parameter.

Consider the construct

findnum(num, ch: int)

where **ch** was declared as type **char** and the procedure heading for **findnum** is

proc findnum(var x: int; y: int).

The actual parameter **num** is mapped to the formal parameter **x** and the actual parameter **ch** is mapped to the formal parameter **y**. Since **ch** was declared as type **char**, a type conflict was avoided through the use of retyping **ch** to type **int** when the call was made. The parameter **y** receives the **int(ch)** as its value, while **x** references the value of the variable **num**.

The formal parameters in the procedure heading are either variable parameters, preceded by the symbol, **var**, or they are value parameters. Variable parameters reference the actual parameter mapped to it, so that altering the value of a variable parameter, alters the corresponding actual parameter in exactly the same manner. A value parameter is a completely new variable, with an initial value given by the current value of the corresponding actual parameter.

When executing a procedure call, the compiler

(a) pushes the base pointer of the current procedure environment,

(b) pushes the base pointer of the enclosing procedure environment,

(c) pushes the return address which will be the address of the instruction following the procedure call

(d) creates space for each parameter in the order listed in the call

(e) initializes each parameter. If an actual parameter maps to a variable parameter, the formal parameter is initialized with the address of the actual parameter. In the case of the actual parameter mapping to a value parameter, the formal parameter is initialized with the value of the actual parameter.

(f) Control is then transferred to the procedure.

A function is a procedure which is used as an operand in an expression. The function name is assigned a known type when it is declared. Within the function, a value is assigned to the function name using an assignment statement, where the function name is preceded by the symbol, **val**. For example, consider a call to a function which computes the value of the factorial of an integer n,

x:= fact(n) .

Assuming the value of the factorial is to be computed using the function **fact**, upon completion of the call to **fact**, the value of the factorial of **n** is assigned to an integer variable, **x**. The only difference in the way the compiler interprets the call to a function from a call to any other procedure is that space for the value to be returned by the function name is placed on the

stack before saving the base pointer of the calling block. Within the function, a recursive call can be made to the function fact, using the construct

val fact:= n * fact(n-1).

The use of the symbol **val** preceding the assignment to the procedure name, **fact**, is required for Edison to interpret the assignment to **fact** as an assignment to the function name.

2.5 PROCEDURES

Procedures in Edison-ES are quite similar to those in Pascal. Procedures are available as complete procedures or as split procedures. A split procedure is similar to a **forward** procedure in Pascal. It consists of a procedure heading preceded by the symbol, **pre**. The declaration of the complete procedure definition for this split procedure follows later and is preceded by the symbol, **post**. A complete procedure is declared using a procedure heading followed by the body of the procedure. The body of the procedure consists of declarations followed by the statement part.

2.51 PROCEDURE HEADING

A procedure heading begins with the symbol, **proc**, followed by an identifier which is the name of the

procedure. This is followed by an optional formal parameter list enclosed by parenthesis. If the procedure is a function, this optional parameter list would be followed by a colon and the type of the value to be returned by the function. As previously mentioned, (Section 2.45), the parameter list contains variable parameters, which are preceded by the **var** symbol, and value parameters which are not preceded by a **var** symbol. The type for each identifier is specified after listing parameters of the same type. These parameters of the same type are separated by commas and the list ends with a colon followed by the type. If more than one list is used within the parameter list, the lists are separated by semicolons. For example,

```
proc findsum(var sum: int; num: int)
```

would be a procedure heading with two parameters for a procedure that is not a function.

```
proc fact(n: int): int
```

would be a function procedure heading with one parameter.

Pascal and Edison-ES treat the procedure heading differently for split procedures. In Edison-ES, the split procedure has a heading that starts with a **pre** symbol similar to

```
pre proc A(x: char; y: int) .
```

The definition of **proc A**, which follows later, must be a complete procedure preceded by the **post** symbol. Thus the definition of **A** as a complete procedure has the heading

```
post proc A(x: char; y: int) .
```

The differences between the Edison-ES procedure heading and that of Pascal are obviously the repeating of the parameter list in Edison-ES in addition to the use of the symbol, **post**.

Finally, the block level changes in the procedure heading. The block level increases by one at the end of the procedure name. This new block level remains as the current block level until the **END** symbol is found at the end of the procedure's statement part.

2.52 THE PROCEDURE BODY

The procedure body consists of declarations and the statement part. The declarations are those of entities which are known only to the procedure. Thus the scope of an entity is the block in which it is declared and all blocks declared within that block. These locally defined entities may be constants, types, variables, and other procedures. Edison-ES requires that all constants, types, and variables be declared

within the procedure before the local procedures are declared. The grammar for Edison however, does not require this. In both Edison and Edison-ES the order of the listing of declarations of constants, types, and variables is not fixed by the grammar. Thus constants may be declared after some of the variables, but before other variable declarations.

Constants are declared following the reserved word **const**. A constant declaration is of the form

const_id = value

as mentioned previously (Section 2.32). Types are declared by listing the structured type to be defined followed by the definition as mentioned previously (Section 2.31). Variable declarations as previously explained (Section 2.3) begin with the symbol **var**.

Procedure declarations follow the declarations of the constants, types and variables in Edison-ES. The procedures are either split procedures or complete procedures. The declarations terminate and the statement part begins with the **begin** symbol and ends with the **end** symbol.

The statement part is a statement list containing statements separated by semicolons. The statement before the **end**, which is the last in the list is not followed by a semicolon.

Execution of the procedure starts at the **begin** symbol, and continues to the **end** symbol. Procedures and data entities declared within the procedure are local to the procedure and may be referenced from any statement within the procedure.

2.6 INPUT OUTPUT

Edison-ES has two system dependent input procedures and two system dependent output procedures. These are defined and added as standard procedures in order to provide some practicality to the language since the Edison system calls for input and output are no longer available as explained in Section 2.1. The two input procedures are **readch** and **readnum**. Each has one variable parameter.

The standard procedure, **readch**, is called through a procedure call of the form, **readch(ch_sym)**, where **ch_sym** is any variable of type **char**. Execution of this procedure requires that the actual parameter be a variable since the character read from the keyboard is returned in the actual parameter. To read an end of line which has a return and a line feed, two calls must be made, one for each character.

To read an integer from the keyboard, the procedure **readnum**, is called using a procedure call of

the form **readnum(int_sym)**, where **int_sym** is any variable of type **int**. Upon execution of the procedure, an integer entered from the keyboard is read and returned in the actual parameter. The entry is terminated with a space or a return.

Character and integer output are available through the use of the standard procedures, **writech** and **writenum**. Each procedure has a single value parameter thus permitting the use of any identifier or literal value to be used for the actual parameter.

To write a character, the procedure call, **writech(ch_sym)**, is made where **ch_sym** is any identifier of type **char** or a character literal value. To write to a new line on the output screen, two calls must be made, one with a **char(13)** and the other with a **char(10)**. This will result in printing a return and a line feed on the output screen.

Writing an integer to the output screen is performed using the procedure call, **writenum(int_sym)**, where **int_sym** is any integer identifier or literal. The integer is written at the current cursor position, using only the number of characters required to represent the integer value. Values are expected to be in the range -32,768 through 32,767. Negative numbers will be preceded by a minus sign. If a number does not

fit on the current line on the output screen it will wrap around to the beginning of the next line, leaving on the first line those characters that fit on that line and printing the remaining characters at the beginning of the next line.

CHAPTER 3 THE EDISON-ES COMPILER

The Edison-ES compiler which is described in this paper is a one-pass compiler which emits assembly language code for the INTEL 8086/88 based family of microcomputers. The compiler creates an output assembly code file, **EOBJ1**. The assembly code is then assembled using any assembler for this microprocessor. The resulting object code is linked using any MS DOS linker to create an executable file. All test programs are assembled using the TURBO ASSEMBLER and are linked using the TLINK[4] linker. The source code for the compiler is available in Packard Laboratory Room 325 on the campus of Lehigh University.

3.1 DESCRIPTION

The compiler is written in Pascal, using TURBO PASCAL 4.0. It consists of a main program, EC.PAS which uses four other TURBO PASCAL 4.0 units. In addition to the DOS unit from TURBO PASCAL 4.0, EC.PAS uses SYMBOLIO.PAS, WORD_TAB.PAS, and EMITCODE.PAS. Two major benefits are derived from the use of units in writing the Edison-ES compiler. First, it provides the

---

4. TURBO ASSEMBLER, TURBO PASCAL, and TLINK are trademarks of BORLAND INTERNATIONAL.

means for modularization of this large program for improved readability. Second, it provides a solution to the size limit problem of the TURBO PASCAL editor.

The DOS unit is used for reading the command line when the compiler is called. It retrieves the name of the file to be compiled from the command line tail and makes it available to the program.

The unit, SYMBOLIO.PAS, is used for opening and closing files, reading from the Edison-ES source file, and writing to the output files. Two output files are created. First is the assembly code file, EOBJ1. The other is ECODE, a file that lists the file read, showing the entire file if compilation was completed. If compilation was not completed because of an error, ECODE contains only the portion of the file processed before the error occurred as well as an error message.

The WORD_TAB.PAS unit maintains the symbol table. In this unit, the procedure **nextsym(sym)** uses SYMBOLIO.PAS to provide the next character from the input file. Strings of characters are processed and are recognized as grammar tokens or identifiers. Identifiers are placed in the symbol table, if they are not already declared within the current block. The symbol table is searched frequently in this unit. The table search is necessary to determine if an identifier

already exists in the block, in the case of a name declaration, or if the name exists locally or globally when a name is used in a statement. The parser from the main program calls a procedure CHECK in this unit that verifies the next follow symbol. During the semantic analysis, types are checked in this unit using the procedure, CHECKTYPE. The beginning of each block is initialized in this unit with the BEGINBLOCK procedure. The termination of a block is accomplished through the use an ENDBLOCK procedure. All error messages are emitted through this unit. All stacks and lookup tables are initialized in WORD_TAB.PAS. WORD_TAB.PAS uses the EMITCODE.PAS and SYMBOLIO.PAS units.

EMITCODE.PAS as the name implies, emits the assembly code to the EOBJ1 file. A large majority of the procedures in this unit are exported to the main program. The symbol table record, **wordattr**, is declared in this unit, but the table of records is declared and completely maintained in the WORD_TAB.PAS unit. This is required because of the need to have procedures in EMITCODE.PAS use values that are kept in the symbol table and hence are passed records from the table. EMITCODE.PAS uses the SYMBOLIO.PAS unit.

The main program, EC.PAS uses all of the units mentioned. All of the constants, types, and variables, used in EC.PAS are declared in the units. EC.PAS contains the code for a recursive descent parser, a semantic analyzer, and calls to EMITCODE.PAS for emitting code.

3.2 CONSTRUCTION

The construction of the compiler was completed in four phases. The parser was constructed initially and then enhanced in the second phase to perform the scope analysis. The third phase consisted of adding the code to perform the semantic analysis. The addition of procedures for generating the assembly code completed the construction of the Edison-ES compiler.

3.21 THE PARSER

The parser is at the heart of the compiler. It is a typical recursive descent parser for an LL1 grammar, with a procedure created for each nonterminal in the grammar.

The construction of the compiler began with the parser and the unit SYMBOLIO.PAS. This read strings from input, identified the string as a grammar symbol or an identifier before moving to the next input string. If the expected grammar symbol was not found,

an error message, ***EXPECTED FOLLOW SYMBOL NOT FOUND*** was emitted. Parsing continues until an error occurs or the program to be parsed is accepted.

The parser was expanded through the inclusion of WORD_TAB.PAS to include scope analysis of the identifiers. Each identifier was recorded in the symbol table along with attributes that were determined by the parser. Identifier types such as constant, procedure, type, or variable were recorded with the name of the identifier, the symbol number, and the minimum block level.

Each identifier found by the parser, required searching the symbol table. Every identifier found was initially entered in the table with a **sym_kind** attribute of **undeclared**. The **undeclared sym_kind** attribute was used to determine if the record being compared in the search was the current record or a previously declared record for the same name. This entry was updated to the correct **sym_kind** or deleted once the table was searched. The entry was deleted from the table if it previously existed at the same level and the parser was not parsing a declaration. If while parsing a declaration, the identifier was found to already be declared in the current level, an error resulted, with the appropriate error message.

References to an identifier made within the statement part of a procedure resulted in searching the current block for the name. If it was not in the table at the current level, the table was searched from the current block level down to level 0 or until the name was found. If the name was only found with **sym_kind** of **undeclared**, an error resulted with the appropriate message.

Searching by block level was performed through the use of stacks. Each time a new level was created, a new stack was created for its identifiers. When a block ended, its stack was removed. Thus the only stacks in existence at any time during the parse were the stacks for the current block and each of its enclosing blocks. The existing stacks were maintained through the use of an array of pointers. Each pointer in the array points to the top of a stack. This array has 8 entries, numbered 0 through 7, which correspond to the block level for the stack pointed to by the array entry. Thus at most 7 levels of nesting procedures are available in Edison-ES. The entries in the stacks are records with a pointer to the previous entry in the stack and a pointer to the identifier's record in the symbol table.

In the semantic analysis stage, each initial reference to an identifier, required additional information be added to the symbol table entry for the identifier. This additional information included identifier type such as procedure, value or variable parameter, record field, structured or elementary type, identifier length, and value if a constant. This was used, at each future reference to the identifier, to insure that expressions consisted of operands of the same type and that they were assigned to variables of the same type. Displacements of all variables from the beginning of the block in which they were declared were calculated and likewise recorded.

The final stage of construction was the generation of the assembly code. The unit, EMITCODE.PAS, was added with the sole purpose of writing the code segments to **EOBJ1**. Each segment of code to be emitted was written into a procedure that was called from EC.PAS. The code, data, and stack segments, along with tables of labels are initialized in the INIT_ASM procedure. The jump to the final return address for an exit to DOS and the input output procedures are in the ENDCODE procedure.

3.22 THE SYMBOL TABLE

The symbol table is a large structure that retains all needed information about each identifier found by the parser. It is significantly different from the structures used for holding information about identifiers in the four pass version of the Edison compiler.

In Hansen's four pass compiler, Pass 2[5] used an array of integers that was mapped to an array of characters. The position in the array of characters of the last character in an identifier name was placed in an array of integers. The position of insertion was always the next identifier id number location. Thus if the fifth identifier of a program had its characters entered sequentially into the array of characters, and the name occupied locations 15 through 18, the integer array would have the number 18 placed in position 5.

In Hansen's Pass 3[6], the identifier number is used as an index into an array of records. The records contain seven fields which correspond to the attributes of the identifiers. Hansen used the array structure because pointer data types were not available in

---

5. Per Brinch Hansen. pp. 306-306.
6. Per Brinch Hansen. pp. 324-325.

Edison. Every record had the same seven attributes, some of which were retyped as needed. For some identifiers, seven attributes were unnecessary, but because of the absence of variant records, every record needed the same number of fields.

The symbol table in Edison-ES begins as an array of 0 through **maxword** of pointers. The current version has **maxword** set to 750. Each pointer is either nil or points to a linked list of one or more variant records. Records are inserted into the table through the use of a hash function applied to the identifier string. The declaration of the record is

```
wordattr = record
     alias,  minlevl, val: integer;
     sym_kind: namekind;
     sorc_sym: nme;
     typelen: integer;
     case nametype: namekind of
          recordtype: (fieldptr: wrdptr);
          enumtype:   (cptr: wrdptr);
          arraytype:  (upbnd, lowbnd: integer;
                       indxtype, entrytype: wrdptr);
          settype:    (typeelement: wrdptr);
          constant:   (constval: integer;
                                 consttype: wrdptr);
```

```
            field,
            valparam,
            varparam,
            variable,
            proctype:  (displace: integer; nextone,
                                        varkind: wrdptr);
   end;                (* wordattr *)  .
```

In this declaration, **namekind** is an enumerated type declared as

```
     namekind = (undeclared, incomplete, constant,
          tipe, field, variable, split, partial,
          complete, recordtype, arraytype,
          elemtype, settype, enumtype, proctype,
          valparam, varparam)
```

and

```
                    wrdptr = ^ wordattr.
```

The **alias** is the identifier number. **Minlevl** is the minimum block level for which the identifiers are declared. **Sym_kind** refers to the identifier's type. Constants are obviously **constant**. Procedures are **split, complete, incomplete,** or **partial**. All parameters and variables are **variable**. **Tipe** is used for type identifiers. The **nametype** is then used for additional information about the identifier. The actual string must be available for searching for

global identifiers. This is kept in **sorc_sym**. **Typelen** is the length of the identifier in words. **Nextone** is a pointer to the next record in the linked list pointed to by the hash table pointer.

In the variant part of the **wordattr** record, the **fieldptr** field for the recordtype variant is a pointer to the first field of an identifier of type record. This pointer points back into the symbol table to the **wordattr** for the first field. This first field's **wordattr** record has fields **displace, nextone,** and **varkind.** The first of these refer to the field's displacement in words from the beginning of the record containing the field. The other two fields in the field's **wordattr** record point to the field that follows the current field and the field's type.

For the variables and parameters, **displace, nextone, varkind** are used in a manner similar to the use with the field. **Displace** is the number of words from the beginning of the procedure, **nextone** points back into the symbol table to the **wordattr** of the next variable or parameter in the list, and **varkind** points to the identifier's type.

The **arraytype** has fields for upper and lower bounds for the indices, and two pointer fields that point back into the symbol table. **Indxtype** points to

the record for the type of index used, while **entrytype** points to the **wordattr** of the basetype of the array.

The remaining variants have fewer fields, but their use of pointer fields are similar. **Typeelement** points to the **wordattr** of the a set's basetype. The variant **enumtype** has a field that points back into the symbol table to the first constant name in the enumeration list. Each **constant** has fields for a value and a pointer to the next constant name in the enumeration list.

Readability was the main reason for the use of the variant record here. The arraytype needed the most fields with four. Rather than have several fields with zero or nil values, the decision was made to have meaningful names for each namekind with only the number of fields needed.

As mentioned previously (Section 3.21), an array of pointers to the tops of currently active identifier stacks was used to retain currently active declarations. Each stack entry has a data field consisting of a pointer into the symbol table. The pointer points to the **wordattr** record for the declared variable. This allows for rather fast searching of the current block and enclosing blocks for an identifier

name by traversing the stacks and also rapid searching by name using the hash table.

3.23 CODE GENERATION

The generation of the assembly code is performed in the EMITCODE.PAS unit. Each block of code required by a construct in Edison is emitted through a separate procedure. Global constants are used for generating the various labels required throughout the assembly code program. The input and output procedures are assembly procedures that follow the main procedure. The main procedure contains all of the emitted code for the Edison-ES program except the input and output procedures which are called from the main procedure.

The length of each instruction in bytes is added to a running total of bytes kept in the **byte_cnt** variable. Two other global integer variables, **labl_cnt, inst_cnt,** are updated throughout the code generation process. **Labl_cnt** is used to generate labels within the code for loops and branching instructions. It is updated each time its current value is used in the generation of a loop or a branch instruction. **Inst_cnt** is used to generate labels for procedure headings and jumps to the beginning of the statement part of a procedure. The assembler treats

all forward jumps as 3 byte instructions, placing a **no opcode** directive in the third byte if the jump is a short jump.

Jumps to previously defined labels presented an interesting problem. Jumps to known labels resulted in the assembler emitting either two byte or three byte instructions depending on the actual displacement of the known label from the jump instruction. Since this displacement varies from program to program, a lookup table, **clabl**, is used to map the value of **byte_cnt** to each label emitted. When a jump instruction is required that refers to a known label, the displacement of the label from the byte_cnt at the jump instruction is calculated. If the displacement is between -127 to 127, a two byte instruction results otherwise the instruction requires three bytes. The appropriate instruction length is then added the **byte_cnt**.

When a procedure heading is found in the parse, a label, **PLxx**, is generated where **xx** is the current **inst_cnt**. **Inst_cnt** is then incremented by 2 in preparation for the next procedure heading. A jump instruction to label, **PLyy**, is emitted after the parameters have been initialized. The value of **yy** in the label, **PLyy**, is the value, **inst_cnt + 1**. This label is emitted at the beginning of the statement part

of the procedure. Hansen used variables, **paramcount**, **paramlength**, and **displacement** to get to the beginning of the statement part of each of his procedures. These he calculated in his fourth pass[7]. The variable **inst_cnt** thus eliminates the need for these other variables. A table, **plabl**, is used as a lookup table for finding the label to which to jump when the procedure is actually called.

When creating space for variable lists and parameter lists, a **PUSH AX** instruction is emitted. The number of pushes required is calculated from the number of variable symbols of each type and the length of each variable symbol. For instance if a procedure has 5 parameters of type integer and an array of 200 characters, 5 pushes would be required for the integer parameters and 200 for the array. In each of these a simple loop

```
INSTxx:
        PUSH  AX
        LOOP  INSTxx
```

would be used where the CX register was initialized to 5 for the integer parameters and 200 for the array. The current value for **labl_cnt** is used for the value **xx**.

---

7. Per Brinch Hansen. pp. 356-367.

Similarly, in the **IF-ELSE** and **WHILE-ELSE** constructs, the **labl_cnt** is used to generate labels for the jump that is emitted at the **ELSE** clause. One label is emitted for a jump to the end of the construct, followed by the emitting of a label for the start of the else clause. Labels are also emitted using **labl_cnt** when evaluating a **bool** expression, which involves emitting a conditional jump and an unconditional jump.

The initialization of the emitted program is performed in the procedure, INIT_ASM. This procedure initializes the global counters and emits the code required by MS DOS for successful execution and termination of the program. The CODE, DATA, and STACK segments are also initialized with this code and the correct return address placed on the stack. The return instruction, used to return to DOS is emitted here also. The address of this return instruction is placed on the stack. This solved one of the largest problems encountered in previous work with emitting assembly code. Previously, the assembly code was scanned a second time and the proper label for the return instruction was issued on the second pass. This second pass of the assembly code is no longer required. Since each Edison-ES program is initialized in the same way,

the return instruction's displacement from the beginning of the code segment is easily calculated. Using the sum of the instruction lengths of all instructions prior to the return instruction, the address of the return is calculated and pushed onto the stack. Thus it is available on the top of the stack at the end of all the initialization and at the end of the program. Popping of the stack into CX at the end of the program is done with the return address being assigned to CX. A **JMP CX** performs the jump back to the address of the return instruction.

The procedure ENDCODE provides the code required to end the main procedure of emitted code and the code for the input, output, and HALT procedures. The instruction to pop the address of the return to DOS instruction into CX and the actual jump are issued here along with **ENDP** directive. The standard procedures for input, output, and HALT are written as procedures that follow the main procedure and thus can be called from the main procedure. All of these procedures utilize the INT 21h interrupt for MS DOS function calls.

When an Edison-ES procedure is called, the emitted code pushes the BP register, the current base pointer. The BP register is then updated to the current value of the stack pointer. This is followed by a pushing of

the base of the enclosing block, which is either the current base or on the stack at the address, **BP-2**. Then a push to leave space for the return address is made. All parameter space requirements are then handled by pushing onto the stack using a loop as mentioned previously. The return address is at this point the current value in **byte_cnt**. This is then moved into the stack at **BP-4**. Finally the jump is made to the label that was issued when the procedure heading was parsed.

In the procedure, the parameters are initialized with either the values or addresses as needed. Space is allocated for the local variables using the previously described loop. All variables are initialized to zero when initially created. A jump is then made to the label that was emitted at the beginning of parsing the statement part of the procedure.

Local variables and value parameters are accessed by their displacement within the current environment block whose base address is in the BP register. Variables which are global to a procedure are accessed through calculation of their address in the enclosing block. A loop which moves to the base of the enclosing block where the global variable was declared is emitted

in this case. The displacement within the defining block is then used to calculate the variable's address within the block. This address is then pushed onto the stack.

Variable parameters present several different situations. The stack location for a variable parameter contains the address of the corresponding actual parameter. If the actual parameter was a variable or value parameter in an enclosing block, the address is calculated in a manner similar to that for global variables. The variable parameters which correspond to actual parameters that are themselves variable parameters simply have the address from the actual parameter's stack entry copied onto the stack.

In evaluating **bool** expressions, all booleans are compared with the constant value 4, which is the constant used by the compiler for the identifier, **FALSE**. Values which are not equal to **FALSE** are true. No checking is performed at runtime to insure that value on the top of the stack are in fact the values for **FALSE** or **TRUE**, which are 4 and 5 respectively. Thus **TRUE** is determined as not **FALSE**.

The remaining emitted code is generated in a fairly standard manner. Set operations are performed using masks with the shift and logical instructions.

Arithmetic operations are performed using the stack to hold the operands and perform the appropriate operation. If several registers were available for an instruction sequence, choices were usually made so as to optimize the time for the instruction sequence. This is the only optimization performed on the generated code.

3.24 ERROR HANDLING

No attempt at error recovery is made in the compiler. Upon discovery of a compilation error, an error message is printed to the screen with the line number of the line where the error was encountered. A similar message is entered in the **ECODE** file which is a text file containing the source code that compiled prior to the occurrence of the error. The compiler halts upon discovery of an error.

3.25 LIMITATIONS

The known limitations of the Edison-ES compiler have for the most part been previously mentioned where appropriate. Below is a summary of those which were mentioned and others which were not.

1. The stack and code segments each have a 64K size limit. No segment swapping is supported. The

data segment and stack segment are initialized to the same segment address. The bottom of the stack segment is used for the data declarations used by the input and output procedures. This requires approximately the first 25 bytes of the stack segment, the stack is slightly less than 64K in size.

2. No runtime checking for values in specific ranges is performed. Thus as mentioned, values which are not false are defaulted to true. **TRUE** is represented by the integer 5 and **FALSE** by 4. Sets may contain only values in the range 0 through **SETLIMIT**, which is set at 127. Hence a set of integer may only contain integers in that range. Unpredictable results can occur if values are used outside this range. Range checking is performed however on the indices of an array.

3. Nesting of procedures is supported only up to 7 levels. Beyond that, the range checking in TURBO PASCAL will cause the program to terminate with a runtime error. This occurs because of an out of range error in the array of stack pointers to the stacks for each level's variables.

4. A maximum of 10 characters are used for uniquely identifying an identifier name. Longer names

are truncated on input, but do not cause a problem for the compiler.

5. The maximum of 750 identifier names are permitted. This limit is due to the size of the lookup table that keeps the label numbers for the procedures.

3.3 USING THE COMPILER

The compiler is used by issuing a command of the form

EC filename

at the MS DOS prompt. The **filename** argument is expected to be the complete path and file name for the file to be compiled. Upon successful compilation, the compiler will create a file, **EOBJ1**, on the default disk drive. This file contains the assembly code emitted by the compiler. A second file, **ECODE**, is also emitted to the default drive. This is a text file containing a copy of the source program just compiled. If errors occurred, **ECODE** will contain the source code up to the point the error occurred and a copy of the same error message that was displayed on the screen. **EOBJ1** will contain the code generated up to the error.

Once successful compilation of a program has occurred, the **EOBJ1** file is assembled using an assembler for the INTEL 8086/88 microprocessor

instruction set. The resulting object code file is then linked using any MS DOS linker to obtain an executable MS DOS file with an **EXE** extension. This executable file may now be used from the DOS system prompt. All assembly and linking for the programs tested with the Edison-ES compiler were performed using the TURBO ASSEMBLER from Borland International. Assembly was performed using the TURBASM software and linking performed using the accompanying TLINK software. Other assemblers and linkers were tested, but the software from BORLAND performed much faster than the others.

3.4 ADDITIONAL WORK

The work on this paper has resulted in a compiler which compiles executable programs for the extended subset of EDISON. As mentioned, problems were solved that were encountered previously while attempting to emit assembly code. Other work on the compiler could be performed, resulting in a compiler for the Edison language itself. First, the module construct must be implemented. Concurrency is a major issue which needs to be addressed. The issues of procedure parameters, library procedures and code optimization could also be addressed. These issues all provide additional

projects to be completed before the Edison system can be fully implemented on the MS DOS microcomputers.

BIBLIOGRAPHY

Barrett, W., Bates, R., Gustafson, D., Couch, J. Compiler Construction. 2nd Ed. Chicago: SRA, Inc., 1979.

Fischer, C.N. and LeBlanc, R.J., Crafting A Compiler. Menlo Park, CA: Benjamin Cummings Pub. Co. Inc., 1988.

Hansen, Per Brinch. Programming a Personal Computer. Englewood Cliffs NJ: Prentice Hall, Inc, 1982.

_____. iAPX86/88, 186/188 User's Manual, Programmer's Reference. Santa Clara CA: INTEL Corp., 1986.

Lemone, Karen A., Assembly Language and Systems Programming for the IBM PC and Compatibles. Glenview, IL: Scott Foresman and Co., 1985.

Sethi, Ravi, Programming Languages: Concepts and Constructs. Reading MA: Addison Wesley, 1989.

Swan, Thomas, Mastering Turbo Pascal 4.0 . 2nd ed., Indianapolis IN: Hayden Books, 1988.

APPENDIX
A.1 THE EDISON-ES GRAMMAR

A.11 TERMINAL SYMBOLS

and1, array1, becomes1, begin1, char1, colon, coma, const1, div1, do1, dot, else1, end1, enum1, eq, gc1, ge, gt, idc, idf, idp, idt, idv, if1, in1, lambda, lbrac, le, lparen, lt, minus, mod1, module1, ne, not1, numeral1, or1, plus, post1, pre1, proc1, quote1, rbrac, rdch1, rdnum1, record1, rparen, semi, set1, skip1, star, val1, var1, while1, writch1, writnum1

A.12 NONTERMINAL SYMBOLS

A, AD, AE, AEX, AFU, AL, ALX, AO, ASE, ASEX, AT, ATU, ATX, C, CD, CDL, CDLX, CHS, CP, CPX, CS, CSL, CSLX, CX, D, E, ED, EL, ELX, ESL, ESLX, EX, EXD, F, FG, FGX, FL, FLX, FU, FX, KS, MD, MDX, MO, NL, NLX, P, PC, PCX, PD, PDL, PDLX, PG, PH, PHX, PL, PLX, PTX, PX, RD, RO, RS, S, SD, SE, SEP, SEX, SL, SLX, SP, T, TD, TX, VD, VDX, VG, VGX, VL, VLX, VS, VSN, VSX

A.13 PRODUCTIONS

A ---> AE
AD ---> array1 idt lbrac RS rbrac lparen idt rparen
AE ---> ASE AEX
AEX ---> RO ASE
AEX ---> lambda
AFU ---> lparen AE rparen
AL ---> A ALX
ALX ---> coma A ALX
ALX ---> lambda
AO ---> minus
AO ---> or1
AO ---> plus
ASE ---> SEP AT ASEX
ASEX ---> AO AT ASEX
ASEX ---> lambda
AT ---> FU FX
AT ---> MO ATX
AT ---> AFU FX
AT ---> not1 ATU
ATU ---> FU
ATU ---> AFU
ATX ---> AFU TX
ATX ---> FU TX
C ---> idt CX
CD ---> idc eq KS
CDL ---> const1 CD CDLX

```
CDLX ---> lambda
CDLX ---> semi CD CDLX
CHS ---> char1 lparen numeral1 rparen
CHS ---> quote1 gc1 quote1
CP ---> PH CPX PDLX SP
CPX ---> D CPX
CPX ---> lambda
CS ---> E do1 SL
CSL ---> CS CSLX
CSLX ---> else1 CS CSLX
CSLX ---> lambda
CX ---> lambda
CX ---> lparen EL rparen
D ---> CDL
D ---> TD
D ---> VD
E ---> SE EX
ED ---> enum1 idt lparen ESL rparen
EL ---> E ELX
ELX ---> coma E ELX
ELX ---> lambda
ESL ---> idc ESLX
ESLX ---> coma idc ESLX
ESLX ---> lambda
EX ---> lambda
EX ---> RO SE
EXD ---> star D
F ---> FU FX
F ---> PC FX
F ---> lparen E rparen
F ---> not1 F
FU ---> C
FU ---> KS
FU ---> VS
FG ---> idf FGX colon idt
FGX ---> coma idf FGX
FGX ---> lambda
FL ---> FG FLX
FLX ---> lambda
FLX ---> semi FG FLX
FX ---> colon idt FX
FX ---> lambda
KS ---> CHS
KS ---> idc
KS ---> numeral1
MD ---> module1 MDX SP
MDX ---> D MDX
MDX ---> EXD MDX
MDX ---> lambda
MO ---> and1
```

```
MO ---> divl
MO ---> modl
MO ---> star
NL ---> idc NLX
NL ---> idf NLX
NL ---> idv NLX
NLX ---> coma NL
NLX ---> lambda
P ---> PX CP
PC ---> idp PCX
PCX ---> lambda
PCX ---> lparen AL rparen
PD ---> CP
PD ---> postl CP
PD ---> prel PH
PDL ---> PD
PDL ---> MD
PDLX ---> PDL
PDLX ---> lambda
PG ---> varl VG
PG ---> VG
PH ---> procl idp PHX PTX
PHX ---> lambda
PHX ---> lparen PL rparen
PL ---> PG PLX
PLX ---> lambda
PLX ---> semi PG PLX
PTX ---> colon idt
PTX ---> lambda
PX ---> CDL PX
PX ---> lambda
PX ---> TD PX
RD ---> recordl idt lparen FL rparen
RO ---> eq
RO ---> ge
RO ---> gt
RO ---> inl
RO ---> le
RO ---> lt
RO ---> ne
RS ---> KS colon KS
S ---> ifl CSL endl
S ---> PC
S ---> rdchl lparen VL rparen
S ---> rdnuml lparen VL rparen
S ---> skipl
S ---> VS becomesl E
S ---> whilel CSL endl
S ---> writchl lparen NL rparen
S ---> writnuml lparen NL rparen
```

```
SD ---> setl idt lparen idt rparen
SE ---> SEP T SEX
SEP ---> lambda
SEP ---> minus
SEP ---> plus
SEX ---> AO T SEX
SEX ---> lambda
SL ---> S SLX
SLX ---> lambda
SLX ---> semi S SLX
SP ---> beginl SL endl
T ---> F TX
TD ---> AD
TD ---> ED
TD ---> RD
TD ---> SD
TX ---> lambda
TX ---> MO F TX
VD ---> varl VG VDX
VDX ---> lambda
VDX ---> semi VG VDX
VG ---> idv VGX colon idt
VGX ---> coma idv VGX
VGX ---> lambda
VL ---> idv VLX
VLX ---> coma idv VLX
VLX ---> lambda
VS ---> VSN VSX
VSN ---> idv
VSN ---> vall idp
VSX ---> lbrac E rbrac VSX
VSX ---> dot idf VSX
VSX ---> colon idt VSX
VSX ---> lambda
```

A.14 THE NULL NONTERMINALS

```
AEX  ALX  ASEX CDLX CPX  CSLX CX   ELX  ESLX EX   FGX
FLX  FX   MDX  NLX  PCX  PDLX PHX  PLX  PTX  PX   SEP
SEX  SLX  TX   VDX  VGX  VLX  VSX
```

A.15 THE FIRST SETS

```
A = { andl, charl, divl, idc, idt, idv, lparen, minus,
        modl, notl, numerall, plus, quotel, star, vall}
AD = { arrayl}
AE = { andl, charl, divl, idc, idt, idv, lparen, minus,
        modl, notl, numerall, plus, quotel, star, vall}
AEX = { eq, ge, gt, inl, lambda, le, lt, ne}
AFU = { lparen}
```

```
AL = { andl, charl, divl, idc, idt, idv, lparen, minus,
       modl, notl, numeral1, plus, quotel, star, val1}
ALX = { coma, lambda}
AO = { minus, orl, plus}
ASE = { andl, charl, divl, idc, idt, idv, lparen,
       minus, modl, notl, numeral1, plus, quotel,
       star, val1}
ASEX = { lambda, minus, orl, plus}
AT = { andl, charl, divl, idc, idt, idv, lparen, modl,
       notl, numeral1, quotel, star, val1}
ATU = { charl, idc, idt, idv, lparen, numeral1, quotel,
       val1}
ATX = { charl, idc, idt, idv, lparen, numeral1, quotel,
       val1}
C = { idt}
CD = { idc}
CDL = { const1}
CDLX = { lambda, semi}
CHS = { charl, quotel}
CP = { procl}
CPX = { array1, const1, enum1, lambda, record1, set1,
       var1}
CS = { charl, idc, idp, idt, idv, lparen, minus, notl,
       numeral1, plus, quotel, val1}
CSL = { charl, idc, idp, idt, idv, lparen, minus, notl,
       numeral1, plus, quotel, val1}
CSLX = { elsel, lambda}
CX = { lambda, lparen}
D = { array1, const1, enum1, record1, set1, var1}
E = { charl, idc, idp, idt, idv, lparen, minus, notl,
       numeral1, plus, quotel, val1}
ED = { enum1}
EL = { charl, idc, idp, idt, idv, lparen, minus, notl,
       numeral1, plus, quotel, val1}
ELX = { coma, lambda}
ESL = { idc}
ESLX = { coma, lambda}
EX = { eq, ge, gt, inl, lambda, le, lt, ne}
EXD = { star}
F = { charl, idc, idp, idt, idv, lparen, notl,
       numeral1, quotel, val1}
FG = { idf}
FGX = { coma, lambda}
FL = { idf}
FLX = { lambda, semi}
FU = { charl, idc, idt, idv, numeral1, quotel, val1}
FX = { colon, lambda}
KS = { charl, idc, numeral1, quotel}
MD = { module1}
```

```
MDX = { array1, const1, enum1, lambda, record1, set1,
        star, var1}
MO = { and1, div1, mod1, star}
NL = { idc, idf, idv}
NLX = { coma, lambda}
P = { array1, const1, enum1, proc1, record1, set1}
PC = { idp}
PCX = { lambda, lparen}
PD = { post1, pre1, proc1}
PDL = { module1, post1, pre1, proc1}
PDLX = { lambda, module1, post1, pre1, proc1}
PG = { idv, var1}
PH = { proc1}
PHX = { lambda, lparen}
PL = { idv, var1}
PLX = { lambda, semi}
PTX = { colon, lambda}
PX = { array1, const1, enum1, lambda, record1, set1}
RD = { record1}
RO = { eq, ge, gt, in1, le, lt, ne}
RS = { char1, idc, numeral1, quote1}
S = { idp, idv, if1, rdch1, rdnum1, skip1, val1,
      while1, writch1, writnum1}
SD = { set1}
SE = { char1, idc, idp, idt, idv, lparen, minus, not1,
       numeral1, plus, quote1, val1}
SEP = { lambda, minus, plus}
SEX = { lambda, minus, or1, plus}
SL = { idp, idv, if1, rdch1, rdnum1, skip1, val1,
       while1, writch1, writnum1}
SLX = { lambda, semi}
SP = { begin1}
T = { char1, idc, idp, idt, idv, lparen, not1,
      numeral1, quote1, val1}
TD = { array1, enum1, record1, set1}
TX = { and1, div1, lambda, mod1, star}
VD = { var1}
VDX = { lambda, semi}
VG = { idv}
VGX = { coma, lambda}
VL = { idv}
VLX = { coma, lambda}
VS = { idv, val1}
VSN = { idv, val1}
VSX = { colon, dot, lambda, lbrac}
```

A.16 THE FOLLOW SETS

```
A = { coma, rparen}
```

```
AD = { array1, begin1, const1, enum1, module1, post1,
       pre1, proc1, record1, set1, star, var1}
AE = { coma, rparen}
AEX = { coma, rparen}
AFU = { and1, colon, coma, div1, eq, ge, gt in1, le,
       lt, minus, mod1, ne, or1plus, rparen, star}
AL = { rparen}
ALX = { rparen}
AO = { and1, char1, div1, idc, idp, idt, idv1, paren,
       mod1, not1, numeral1, quote1, star, val1}
ASE = { coma, eq, ge, gt, in1, le, lt, ne, rparen}
ASEX = { coma, eq, ge, gt, in1, le, lt, ne, rparen}
AT = { coma, eq, ge, gt, in1, le, lt, minus, ne, or1,
       plus, rparen}
ATU = { coma, eq, ge, gt, in1, le, lt, minus, ne, or1,
       plus, rparen}
ATX = { coma, eq, ge, gt, in1, le, lt, minus, ne, or1,
       plus, rparen}
C = { and1, colon, coma, div1, do1, else1, end1, eq,
       ge, gt, in1, le, lt, minus, mod1, ne, or1,
       plus, rbrac, rparen, semi, star}
CD = { array1, begin1, const1, enum1, module1, post1,
       pre1, proc1, record1, semi, set1, star, var1}
CDL = { array1, begin1, const1, enum1, module1, post1,
       pre1, proc1, record1, set1, star, var1}
CDLX = { array1, begin1, const1, enum1, module1, post1,
       pre1, proc1, record1, set1, star, var1}
CHS = { and1, array1, begin1, colon, coma, const1,
       div1, do1, else1, end1, enum1, eq, ge, gt in1,
       le, lt, minus, mod1, module1, ne or1, plus,
       post1, pre1, proc1, rbrac, record1 rparen,
       semi, set1, star, var1}
CP = { begin1, lambda}
CPX = { begin1, module1, post1, pre1, proc1}
CS = { else1, end1}
CSL = { end1}
CSLX = { end1}
CX = { and1, colon, coma, div1, do1, else1, end1, eq,
       ge, gt, in1, le, lt, minus, mod1, ne, or1,
       plus, rbrac, rparen, semi, star}
D = { array1, begin1, const1, enum1, module1, post1,
       pre1, proc1, record1, set1, star, var1}
E = { coma, do1, else1, end1, rbrac, rparen, semi}
ED = { array1, begin1, const1, enum1, module1, post1,
       pre1, proc1, record1, set1, star, var1}
EL = { rparen}
ELX = { rparen}
ESL = { rparen}
ESLX = { rparen}
EX = { coma, do1, else1, end1, rbrac, rparen, semi}
```

```
EXD = { array1, begin1, const1, enum1, record1, set1,
        star, var1}
F = { and1, coma, div1, do1, else1, end1, eq, ge, gt,
        in1, le, lt, minus, mod1ne, or1, plus, rbrac,
        rparen, semi, star}
FG = { rparen, semi}
FGX = { colon}
FL = { rparen}
FLX = { rparen}
FU = { and1, colon, coma, div1, do1, else1, end1, eq,
        ge, gt, in1, le, lt, minus, mod1, ne, or1,
        plus, rbrac, rparen, semi, star}
FX = { and1, coma, div1, do1, else1, end1, eq, ge, gt,
        in1, le, lt, minus, mod1, ne, or1, plus, rbrac,
        rparen, semi, star}
KS = { and1, array1, begin1, colon, coma, const1, div1,
        do1, else1, end1, enum1, eq, ge, gt in1, le,
        lt, minus, mod1, module1, ne or1, plus, post1,
        pre1, proc1, rbrac, record1 rparen, semi, set1,
        star, var1}
MD = { begin1}
MDX = { begin1}
MO = { char1, idc, idp, idt, idv, lparen, not1,
        numeral1, quote1, val1}
NL = { rparen}
NLX = { rparen}
P = { lambda}
PC = { and1, colon, coma, div1, do1, else1, end1, eq,
        ge, gt, in1, le, lt, minus, mod1, ne, or1,
        plus, rbrac, rparen, semi, star}
PCX = { and1, colon, coma, div1, do1, else1, end1, eq,
        ge, gt, in1, le, lt, minus, mod1, ne, or1,
        plus, rbrac, rparen, semi, star}
PD = { begin1}
PDL = { begin1}
PDLX = { begin1}
PG = { rparen, semi}
PH = { array1, begin1, const1, enum1, module1, post1,
        pre1, proc1, record1, set1, var1}
PHX = { array1, begin1, colon, const1, enum1, module1,
        post1, pre1, proc1, record1, set1, var1}
PL = { rparen}
PLX = { rparen}
PTX = { array1, begin1, const1, enum1, module1, post1,
        pre1, proc1, record1, set1, var1}
PX = { proc1}
RD = { array1, begin1, const1, enum1, module1, post1,
        pre1, proc1, record1, set1, star, var1}
```

```
RO = { and1, char1, div1, idc, idp, idt, idv1, paren,
       minus, mod1, not1, numeral1, plus, quote1,
       star, val1}
RS = { rbrac}
S = { else1, end1, semi}
SD = { array1, begin1, const1, enum1, module1, post1,
       pre1, proc1, record1, set1, star, var1}
SE = { coma, do1, else1, end1, eq, ge, gt in1, le, lt,
       ne, rbrac, rparen, semi}
SEP = { and1, char1, div1, idc, idp, idt, idv1, paren,
       mod1, not1, numeral1, quote1, star, val1}
SEX = { coma, do1, else1, end1, eq, ge, gt, in1, le,
       lt, ne, rbrac, rparen, semi}
SL = { else1, end1}
SLX = { else1, end1}
SP = { begin1, lambda}
T = { coma, do1, else1, end1, eq, ge, gt, in1, le, lt,
       minus, ne, or1, plus, rbrac, rparen, semi}
TD = { array1, begin1, const1, enum1, module1, post1,
       pre1, proc1, record1, set1, star, var1}
TX = { coma, do1, else1, end1, eq, ge, gt, in1, le, lt,
       minus, ne, or1, plus, rbrac, rparen, semi}
VD = { array1, begin1, const1, enum1, module1, post1,
       pre1, proc1, record1, set1, star, var1}
VDX = { array1, begin1, const1, enum1, module1, post1,
       pre1, proc1, record1, set1, star, var1}
VG = { array1, begin1, const1, enum1, module1, post1,
       pre1, proc1, record1, rparen, semi, set1, star,
       var1}
VGX = { colon}
VL = { rparen}
VLX = { rparen}
VS = { and1, becomes1, colon, coma, div1, do1, else1,
       end1, eq, ge, gt, in1, le, lt, minus, mod1, ne,
       or1, plus, rbrac, rparen, semi, star}
VSN = { and1, becomes1, colon, coma, div1, do1, dot,
       else1, end1, eq, ge, gt, in1, lbrac, le, lt,
       minus, mod1, ne, or1, plus, rbrac, rparen,
       semi, star}
VSX = { and1, becomes1, colon, coma, div1, do1, else1,
       end1, eq, ge, gt, in1, le, lt, minus, mod1, ne,
       or1, plus, rbrac, rparen, semi, star}
```

VITA

The author, John E. Davis, was born on August 15, 1945 in Bethlehem, PA. He is the oldest of the 2 children of May F. (Rothrock) Davis and Frederick P. Davis. His mother and younger brother, Frederick, still reside in Bethlehem.

John was graduated from Liberty High School in 1963 and went on to attend what is now Kutztown State University. Majoring in mathematics and minoring in physics, John graduated with honors in 1967. He received the Kappa Mu Epsilon Mathematics Award at upon graduation from Kutztown. John went on as a teaching assistant to study full-time at the University of Delaware in Newark, DE. He received the Master of Science Degree in Mathematics in 1970. His thesis was ***A Uniqueness Theorem for the Generalized Axially Symmetric Helmholtz Equation*** and was under the supervision of Professor Richard Weinacht. Between 1980 and 1984, John attended Kutztown State University, Muhlenberg College, and Northampton Community College taking courses in computing science.

In 1968 John accepted a position as a high school mathematics teacher and Chairman of Department of Mathematics in the Southern Lehigh School District,

Center Valley, PA. He held this position until September 1983 when he accepted a position as Academic Computing Coordinator and Assistant Professor of Data Processing at Northampton Community College. He was promoted to Associate Professor of Computer Information Science at Northampton Community College in 1988. In Spring 1990 he resigned his position as Academic Computing Coordinator to devote more time to teaching. Since 1985 he has been an adjunct professor of Computer Information Science at Muhlenberg College in Allentown.

John is married to the former Sally A. Whitehead of Allentown. They reside in Allentown with their 3 children, John, Megan, and Bryan.

END

OF

TITLE