AN INTERPRETER FOR THE BASIC PROGRAMMING LANGUAGE

THESIS

Presented to the Graduate Council of the
North Texas State University in Partial
Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

Min-Jye S. Chang, B. S.
Denton, Texas

May, 1975

Chang, Min-Jye S., An Interpreter for the Basic Programming Language. Master of Science (Computer Sciences), May, 1975, 82 pp., 3 tables, 8 illustrations, 14 appendices, bibliography, 12 titles.

In this thesis, the first chapter provides the general description of this interpreter. The second chapter contains a formal definition of the syntax of BASIC along with an introduction to the semantics. The third chapter contains the design of data structure. The fourth chapter contains the description of algorithms along with stages for testing the interpreter and the design of debug output.

The stages and actions are represented internally to the computer in tabular forms. For statement parsing working syntax equations are established. They serve as standards for the conversion of source statements into object pseudocodes. As the statement is parsed for legal form, pseudocodes for this statement are created. For pseudocode execution, pseudocodes are represented internally to the computer in tabular forms.

TABLE OF CONTENTS

Page

LIST OF TABLES. v

LIST OF ILLUSTRATIONS vi

LIST OF APPENDICES. vii

Chapter

I. INTRODUCTION. 1

Definition of the Problem
Purpose of the Study
Procedure
Limitation and Future Work
Organization

II. A DESCRIPTION OF BASIC. 6

Formal Specification
Formal Grammar
Hierarchy of Language
Backus Normal Form
Definition of the Source Language, BASIC

III. DATA STRUCTURE. 25

SOURCE
ATOMS
ERRORES, ERROREP, and ERROREE
Symbol List
Line Number List
Pseudocode List
IMAGE

IV. INTERPRETER 41

Lexical Analysis
Parsing
Execution
Listing
Debug Output
Testing

V. CONCLUSION. 63

Page
APPENDICES. 66
BIBLIOGRAPHY. 8

LIST OF TABLES

Table Page

I. List of Atoms. 27-28

II. Pseudocode List. 35-39

III. Debug Output 61

LIST OF ILLUSTRATIONS

Figure Page

1. Interpreter System Flow. 26
2. Atom Layout. 29
3. Symbol Node. 31
4. Line Number Node 32
5. Pseudocode Node. 33
6. Finite-state Machine for Lexical Analysis. 43
7. Parsing Block Chart. 47
8. Executing Block Chart. 58

LIST OF APPENDICES

Appendix		Page
I.	Reserved Words of BASIC	66
II.	Backus Normal Form of This BASIC Programming Language	67
III.	Trace Level 1 of SCANNER	69
IV.	Trace Level 1 of PARSER	70
V.	Trace Level 1 of EXECUTE	71
VI.	Trace Level 2 of SCANNER	72
VII.	Trace Level 2 of PARSER	73
VIII.	Trace Level 2 of EXECUTE	74
IX.	Source and Execution Listing of Test Program 1	75
X.	Source and Execution Listing of Test Program 2	76
XI.	Source and Execution Listing of Test Program 3	77
XII.	Source and Execution Listing of Test Program 4	78
XIII.	Source and Execution Listing of Test Program 5	79
XIV.	Source and Execution Listing of Test Program 6	80

CHAPTER I

INTRODUCTION

This thesis is an application of the top-down translator (3) to the interpretation of a modified version of the BASIC (6) programming language.

The BASIC programming language was chosen as the target language because, although it is a relatively simple language, it is complex enough to display many of the quality and implementation difficulties of more advanced high-level languages.

This interpreter is written in PL/1 (5) to be executed on the IBM/360 computer.

This interpreter is organized as four different segments, each of which makes a pass over some form of the source. The four segments are lexical analysis, parsing, executing, and listing. A main program calls each segment, which is implemented as a separate PL/1 procedure.

Definition of the Problem

Gries (3) used the term "interpreter" for a program which performs two functions:

1. Translate a source program written in the source language (BASIC in this application) into a pseudocode.

2. Execute (interpret, simulate) the program in this pseudocode.

The first part of the interpreter is like the first part of a multi-pass compiler. The main difference between an interpreter and a compiler is that the former executes the pseudocode and the latter eventually transforms the pseudocode into machine code.

The pseudocode into which a source language is translated should be designed to make the execution proper as efficient as possible.

A pseudocode representation could be interpreted as the machine language of some pseudocomputer.

A computer and associated routines that behave as such a pseudocomputer are referred to as an interpreter of the corresponding pseudocode.

Purpose of the Study

This interpreter is designed with the following purposes:

1. To explore the design of an interpreter for a batch processing environment.
2. To build efficient tools in the interpreter for correcting and detecting errors.

Procedure

The first step in preparation for this interpreter was library research, including readings related to the translator, BASIC and PL/1 programming languages, and the Job Control Language (4).

Donovan (2) has stated the general problem of designing software. Listed below are six steps in the design of this interpreter:

1. Specify the problem.
2. Specify data structure.
3. Define format of data structure.
4. Specify algorithm.
5. Look for modularity (i.e., capability of a complex program to be subdivided into independent more simple programming units).
6. Repeat 1 through 5 on modules.

Limitation and Future Work

The BASIC matrix commands preceded by "MAT" are not included in this interpreter. These commands may be added in the future.

One feasible method is to create new pseudocodes representing different matrix operations.

Organization

This thesis is organized into five chapters. The first chapter provides the general description of this

intrepreter. The second chapter contains a formal definition of the syntax of BASIC along with an introduction to the semantics. The third chapter contains the design of data structure. The fourth chapter contains the description of algorithms used in this interpreter. Also contained in Chapter Four are stages for testing this interpreter and the design of debug output. The fifth chapter contains the conclusions drawn from analysis of this work.

The program can be examined by obtaining report number NTCSCI74001 entitled "An Interpreter for the BASIC Programming Language" from the Department of Computer Sciences at North Texas State University (1).

CHAPTER BIBLIOGRAPHY

1. Chang, Min-Jye S., "An Interpreter for the BASIC Programming Language," Department of Computer Sciences, North Texas State University, Denton, Texas 1974.

2. Donovan, John J., Systems Programming, New York, McGraw-Hill Book Company, 1972

3. Gries, David, Compiler Construction for Digital Computers, New York, John Wiley & Sons, Inc., 1971.

4. International Business Machines, IBM Systems 360 Operating Systems: Job Control Language Reference, Form No. GC28-6704-2.

5. International Business Machines, IBM System 360 PL/1 Reference Manual, Form No. C28-8201-0.

6. Smith, Robert E., Discovering BASIC, New York, International Timesharing Corporation, 1970.

CHAPTER II

A DESCRIPTION OF BASIC

This chapter defines the syntax of the BASIC programming language based on the descriptions of Smith (5).

Before going into the description of BASIC, it is useful to analyze some of the problems in formally defining a language (2).

Formal Specification

A language may be thought of as a set of sentences with well-defined structures (2). The set of rules specifying valid constructions of a language is its syntax. The syntax of a language describes its form.

A language called a meta-language is employed to explain a language called an object language. A meta-language is a system of definitions of symbols and rules for their combination. Symbols of the object language are called terminal symbols. Symbols of a meta-language that denote strings in the object language are called nonterminal symbols.

The most elementary object in a formal language is a symbol. Symbols are concatenated to form strings, which may or may not belong to the language. Generally, a language does not include all possible strings on its alphabet (2). Only certain strings are valid sentences in the language.

Formal Grammar

The symbols which are in the object sentence when generation of the sentence is completed are referred to as terminal symbols. Those symbols which only appear in the intermediate steps are referred to as nonterminal symbols. One nonterminal symbol, the starting symbol, is distinguished as the source sentence symbol with which the generation process begins (2).

The process of generation of object language from source language consists of applying, at each step, any one of the set of rewriting rules or productions (2). A production is a string transformation rule having a left-hand side that is a pattern to match a substring (possibly all) of the string to be transformed, and a right-hand side that indicates a replacement for the matched portion of the string. This process transforms the string into a new string; the process stops when there is no production that can be applied or when the string consists solely of terminal symbols.

It is important to realize that any substring of the current string may be replaced by an applicable production and that only that part of the string matched by the left-hand side of the production is affected. Productions can totally replace substrings, or they may merely rearrange the symbols of the matched substring.

Abramson (1) defined a sentence as a sentential form containing only terminal symbols. A sentential form is any string which can be derived from the starting symbol.

Hierarchy of Language

The definition of production allows for a wide variety of string transformations. Certain restrictions on the form of productions give grammars producing subclasses of the class of formal languages. Noam Chomsky (2) has constructed a system of four language types that classify some languages according to such restrictions.

The most general type of grammar imposes no restrictions on the productions. In particular, productions that eliminate symbols are permitted. This allows the intermediate strings to expand and contract. A grammar without restrictions is called a type 0 grammar.

The simplest restriction which produces a strictly smaller class of languages is to require the right-hand side of every production to have at least as many symbols as the left-hand side. A grammar with this restriction is called a type 1 or noncontracting context-sensitive grammar.

If the left-hand side of the production is restricted to a single nonterminal symbol, its application cannot be dependent on the context in which the symbol occurs. Grammars with this restriction (and nonblank right-hand strings) are called type 2, context-free or simple phrase-structure grammars.

A third type of restriction on productions restricts the number of terminal and nonterminal symbols that each step can create. When, at most, one nonterminal symbol is

used in both the right-hand and left-hand sides of a production, the production is said to be linear.

Each of the above restrictions includes those above it. These types form a hierarchy. No type 3 grammar can generate the language defined by type 2 grammar. Similarly, no type 2 grammar can generate the language defined by type 1 grammar. Finally, type 1 is a strict subset of type 0. The BASIC in this interpreter is defined as a type 2 language.

Backus Normal Form

The metalanguage used in describing BASIC is Backus-Naur Form or Backus Normal Form (4). Terminal symbols represent themselves. Nonterminal symbols are enclosed in meta brackets, " " and " ". The symbol "::=" is read "is composed of" and is used to separate the defined symbol on the left of a production from the definition of the right. The symbol " " is read "or" and is used to separate alternate definitions in a production.

Definition of the Source Language, BASIC

The BASIC (Beginner's All Purpose Symbolic Instruction Code) was originally developed at Dartmouth College, New Hampshire, under the direction of Professor J.G. Kemeny (1).

The source language grammar adopted here is similar to the definition of BASIC by Smith (5). Some of the data structure meanings or execution-time actions of the various

BASIC statements were not unambiguously described by Kemeny (1). The interpretations which have been placed on such statements in this interpreter may have caused some discrepancies between this implementation and the original Dartmouth implementation.

Letters, Digits, Special Characters

The twenty-six letters of the English alphabet are used in constructing variables and in strings which may appear as comments or as messages in printed output.

Digits, just as letters, are used in forming variables and strings. In addition, they are used to form numbers and statement line numbers. There are ten digits. They are "1", "2", "3", "4", "5", "6", "7", "8", "9", and "0".

There are fifteen special characters. They are " ", "+", "-", "*", "/", "=", "(",")", " ", "'", ",", ";", " ", "$" and "?". They are used in forming strings.

Variables

"Variables" are names for a dual role: to represent in the source language the names of numerical values and at the same time, names of the computer cells where these numbers are located at execution time. A source-language variable of BASIC identifies an execution-time data structure of a fullword of storage (3) of the IBM/360 computer. The value of a variable may change during the execution of a BASIC program.

A source-language variable must conform to certain rules:

1. One to twenty characters may be used for any variable.
2. A variable may contain letters and digits.
3. A variable must begin with a letter.
4. A variable must not be a reserved word.

A reserved word is a source-language word that looks like a variable but which has special significance to this interpreter.

Examples.--The following are examples of source-language variables:

1. A
2. B234
3. ABCDEFGHIJKLMNOPQ
4. A1B2
5. B234P

Numbers

Numbers are used in the source-language as constants in expressions and in the lists of numbers used by the DATASTATEMENT.

Examples.--The following are examples of numbers:

1. 234
2. 5.89
3. 45.99

Operations

The execution time sequence of operations is generally in the same order as from the source-language reading from left to right, except for the following precedence of operations:

Priority 1

- \- Prefix minus
- \+ Prefix plus

Priority 2

- ** Exponentiate

Priority 3

- / Divide
- * Multiply

Priority 4

- \- Infix minus
- \+ Infix plus

To override this normal order of execution precedence, parentheses are inserted around the source expression that is to be evaluated first at execution.

Example.--The following are examples of complex expressions with operations:

1. 5 + 10 / 5 * 2 ** 3 - 6

First find exponent value	5+10/5*8-6
Next, divide	5+2*8-6
Next, multiply	5+16-6
Last, add and subtract	15

2. (2 - 1) * 5 / 5 + 2

First, find value of "2-1"	1*5/5+2
Next, multiply	5/5+2
Next, divide	1+2
Last, add	3

Relations

Six relations are available for test purposes. They are:

Equal to	=
Not equal to	¬=
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

Examples.--The following are examples of relations:

1. IF A = 4 THEN 123
2. IF ABCD > (X-Y9**2) THEN 321

An alternate source-language form of "not equal" sometimes used is "<>"; this form is not used in the present work.

READSTATEMENT

The READSTATEMENT is used to assign a specific numerical value to a simple or subscripted variable. The numerical value must have been previously assigned by a DATASTATEMENT.

Examples.--The following are examples of READSTATEMENTs:

1. 10 READ X
2. 20 READ Y,Z,AB

DATASTATEMENT

The DATASTATEMENT provides storage area for data. A declarative statement may be used to introduce signed numerical data into a BASIC program and may appear anywhere in a BASIC program.

One may think of all the DATASTATEMENTs in a program as being assigned to a data bank. When the program is run, the first READSTATEMENT uses the first number in the data bank.

The replacement of data actually occurs at the time of execution. One must be careful to make sure that there are sufficient data and that they are in the proper order.

Examples.--The following are examples of DATASTATEMENTs:

1. 10 DATA 45, 20
2. 20 DATA 20, 40

   30 READ XY, WY

In the second example, the variable XY is assigned a value of 20 and the variable WY is assigned a value of 40.

PRINTSTATEMENT

The PRINTSTATEMENT is used for outputting data. The PRINTSTATEMENT may be used for printing the value of a variable or computation, for printing a heading, or simply for skipping a line.

Each line is divided horizontially into five twenty-character zones. When only one value is printed, it is placed in zone 1. When more than one value is printed, the second is placed in zone 2, the third in zone 3, etc. If more than five values are printed, the first five are placed in the five zones in order. The sixth value is printed on the next line in zone 1, the seventh in zone 2, etc. It is also possible to print messages in a manner similar to the formatting of values.

Examples.--The following are examples of PRINTSTATEMENTs:

1. 10 PRINT 'X=', X
2. 20 PRINT X+Y**2, ABC, DD
3. 30 PRINT 'PAY RATE', 'HOURS', 'GROSS', 'NET'

The symbol "'" is used in pairs to represent strings which are printed in the program output but is not part of it.

LETSTATEMENT

The LETSTATEMENT is the principal computational statement in a BASIC program.

The "=" sign in BASIC is not a mathematical equal sign; it means "replaced by". Therefore, this statement is interpreted to mean, "The value of the arithmetic expression on the right of the "replaced by" sign replaces the value of the variable on the left.

Examples.--The following are examples of LETSTATEMENTs:

1. 10 LET N = N +1
2. 20 LET X = 3
3. 30 LET R = A + B -63

The first example results in the value 1 being added to the value N in storage. The new sum replaces the original value of N.

The second example causes the number 3 to be stored in the location assigned for the variable X.

In the third example the value of B is added to the value of A and 63 is subtracted from the sum. The final value is stored in the location assigned to R.

GOTOSTATEMENT

The simplest BASIC statement for altering the sequence of execution is the GOTOSTATEMENT.

Example.--The following is an example of a GOTOSTATEMENT:

```
1.  10  GOTO 100
    --  -----
    100 ------
```

In the above example, 10 and 100 represent line numbers; line numbers identify source statements and are composed of positive numbers with five or less digits.

ONSTATEMENT

The ONSTATEMENT permits transfer of control to one of a group of statements, with the particular one chosen during the run on the basis of results computed in the execution of the program. The statement is of the form

ln ON expression THEN ln1, ln2, ln3, - - - -

where the "expression" is any valid BASIC expression and the

subscripts on the line numbers of statements in the program indicate their sequence in the ONSTATEMENT. Executions of this statement causes statement lni to be executed next, where i is the integer value of the expression. The "expression" in the ONSTATEMENT must produce a result of at least 1 and no more than the number of line-number labels contained in the statement.

Examples.--The following are examples of ONSTATEMENTs:

1. 80 ON A+B THEN 100, 110, 120, 130, 140
2. 33 ON X-Y+2 THEN 10, 360, 44, 60

In the first example, if the expression has a value of 4, control is transferred to statement number 130 when the statement is executed.

In the second example, the expression "X-Y+2" must produce at execution an integer value in the range 1 to 4. If outside this range, execution continues with the next in-line statement.

IFSTATEMENT

The IFSTATEMENT permits one to make the transfer of control depending on the results of a computation, the comparison of expressions. Such a statement, called a conditional transfer statement, transfers control only if a certain condition is met. The GOTOSTATEMENT is called an unconditional transfer statement, since it always transfers control.

Examples.--The following are examples of IFSTATEMENTs:

1. 10 IF X = 10 THEN 200

2. 20 IF X = A*20 THEN 80

In the first example, if the value of X at execution is equal to 10, transfer program control to line number 200. If not, execution continues with the next in-line statement.

In the second example, if the value of X is equal to the value of "A*20", transfer control to line number 80. If not, execute the next in-line statement.

In this interpreter, six relations are available for IFSTATEMENT. Please refer to "Relations".

FORSTATEMENT and NEXTSTATEMENT

Looping, one of the most important techniques in programming (6), makes it possible to perform the same calculation on more than one set of data. A loop consists of the repetition of a section of a program, substituting new data each time, so that each pass through the loop is different from the preceding one.

The combination of the FORSTATEMENT and NEXTSTATEMENT is the most powerful two-instruction set in the BASIC language. The loop starts with the FORSTATEMENT and ends with the NEXTSTATEMENT (inclusive).

The general format of the FORSTATEMENT is

ln FOR variable = a TO b STEP c

where the "variable" is the index, "a" is the initial value of the index, "b" is the terminal value of the index, and

"c" is the value by which the index is modified for each pass. The values of "a", "b", and "c" may be any valid expressions.

A few rules which apply to the FORSTATEMENT are:

1. Every FORSTATEMENT must have an associated NEXTSTATEMENT which names the same "variable". FORSTATEMENT and NEXTSTATEMENT form a pair.
2. The number of BASIC statements that may appear between the FORSTATEMENT and NEXTSTATEMENT is unlimited.
3. Transfer out of a loop can be accomplised by using an IFSTATEMENT or a GOTOSTATEMENT, but transfer back into the loop is not correct.
4. FORSTATEMENTs may be nested; that is, an inner loop may be completely contained within an outer loop.
5. The STEP c may be omitted if "c" is understood to be 1.

The general form of the NEXTSTATEMENT is

ln NEXT variable.

Examples.--The following are examples of FORSTATEMENT and NEXTSTATEMENT pairs:

1. 10 FOR N = 1 TO 100 STEP 1

   -----

   90 NEXT N

2. 20 FOR I = 1 TO 2
   30 FOR J = 1 TO 2

      -----
   70 NEXT J
   81 NEXT I

GOSUBSTATEMENT and RETURNSTATEMENT

A subroutine is essentially an independent program, but it is written in such a way that it can be executed only when called by another program (6). Subroutines are used at execution to perform tasks that are needed on more than one occasion. A subroutine call can be written at any place in a program. A main program may call upon a subroutine to perform a certain operation. A subroutine may be called any number of times, and reentry to the main program at the proper point is automatically controlled by the calling program. However in this implementation of the interpreter, a subroutine may not directly or indirectly call itself.

A subroutine in BASIC may consist of any number of statements, but its last one must be a RETURNSTATEMENT. The subroutine calling statement is the GOSUBSTATEMENT. A subroutine can also call another subroutine.

Examples.--The following are examples of GOSUBSTATEMENTs and RETURNSTATEMENTs:

```
1.  10  GOSUB 20
        -----
    20  -----
        -----
    40  RETURN
2.  21  GOSUB 30
        -----
    30  -----
        -----
    140 GOSUB 50
    40  RETURN
    50  -----
        -----
    60  RETURN
```

DIMSTATEMENT

The source DIMSTATEMENT assigns names to vectors and arrays and specifies the object data structure, i.e., how many storage locations are to be reserved for them.

In the storage allocation, vectors and arrays will be allocated contiguous locations in the data area and are placed in an ascending order.

Examples.--The following are examples of DIMSTATEMENTs:

1. DIM A(2,2)
2. DIM B(6)

The first example assigns A as the name of an array which requires four storage locations: A(1,1), A(1,2), A(2,1) and A(2,2).

The second example assigns B as the name of a vector which requires six storage locations: B(1), B(2), B(3), B(4), B(5), and B(6).

DEFSTATEMENT

The general form of the user-defined function is

ln DEF FNa(v) = expression

where "a" is any letter of the alphabet, "v" is a variable and "expression" is an expression that uses the variable "v".

The user-defined function is useful when a particular one-statement computation is required at several different points in a program.

Examples.--The following are examples of DEFSTATEMENTs:

1. 10 DEF FNA(X) = X - 4 + 5 - X ** 2
2. 20 DEF FNB(Y) = Y / Y - 4 + 2 - Y * 6

STOPSTATEMENT

The STOPSTATEMENT is used to terminate the execution of a program.

ENDSTATEMENT

The final statement in each BASIC program must be the ENDSTATEMENT. It tells the interpreter the source program is complete, and if executed it stops the execution.

Built-in Functions

The BASIC language has nine built-in functions in this interpreter. They are

1.	SIN(X)	Sine of X
2.	COS(X)	Cosine of X
3.	TAN(X)	Tangent of X
4.	ATN(X)	Arctangent of X
5.	EXP(X)	Exponentiation, e^X
6.	ABS(X)	Absolute value of X
7.	LOG(X)	Natural logarithm of X
8.	SQR(X)	Square root of X
9.	INT(X)	Integer part of the value of X (the sign of X remains unchanged)

CHAPTER BIBLIOGRAPHY

1. Abramson, Harvey, Theory and Application of a Bottom-Up Syntax-Directed Translator, New York, Academic Press Inc., 1973.

2. Donovan, John J., Systems Programming, New York, McGraw-Hill Book Company, 1972.

3. International Business Machines, IBM Systems 360 PL/1 Reference Manual, Form No. C28-8201-0.

4. Naur, Peter and others, "Revised Report on the Algorithmic Language. Algol 60," Programming Systems and Languages, edited by Saul Rosen, New York, McGraw-Hill Book Company, 1967.

5. Smith, Robert E., Discovering BASIC, New York, International Timesharing Corporation, 1970.

6. Spencer, D.D., A Guide to BASIC Programming: A Time-Sharing Language, the United States of America, Addison-Wesley Publishing Company, Inc., 1970.

CHAPTER III

DATA STRUCTURE

In this chapter the data bases and their formats for this interpreter are depicted in detail. Figure 1 shows the manner in which the various data bases are used (1).

SOURCE

SOURCE is an external file built by the SCANNER and is used as an input to the LISTER. It contains a copy of the BASIC source program.

ATOMS

ATOMS is an external file built by the SCANNER and is used as an input to the PARSER. The entries in the file ATOMS are numbers which identify a certain atom-type followed by a variable which may point to an entry in the symbol list or in the line number list. The atom-type may also be followed by a value of NULL (a special PL/1 value) or by a number representing the trace level for the PARSER or the SCANNER.

Table I lists the various atom-types along with their respective operands, if any. Figure 2 shows the three types of entries.

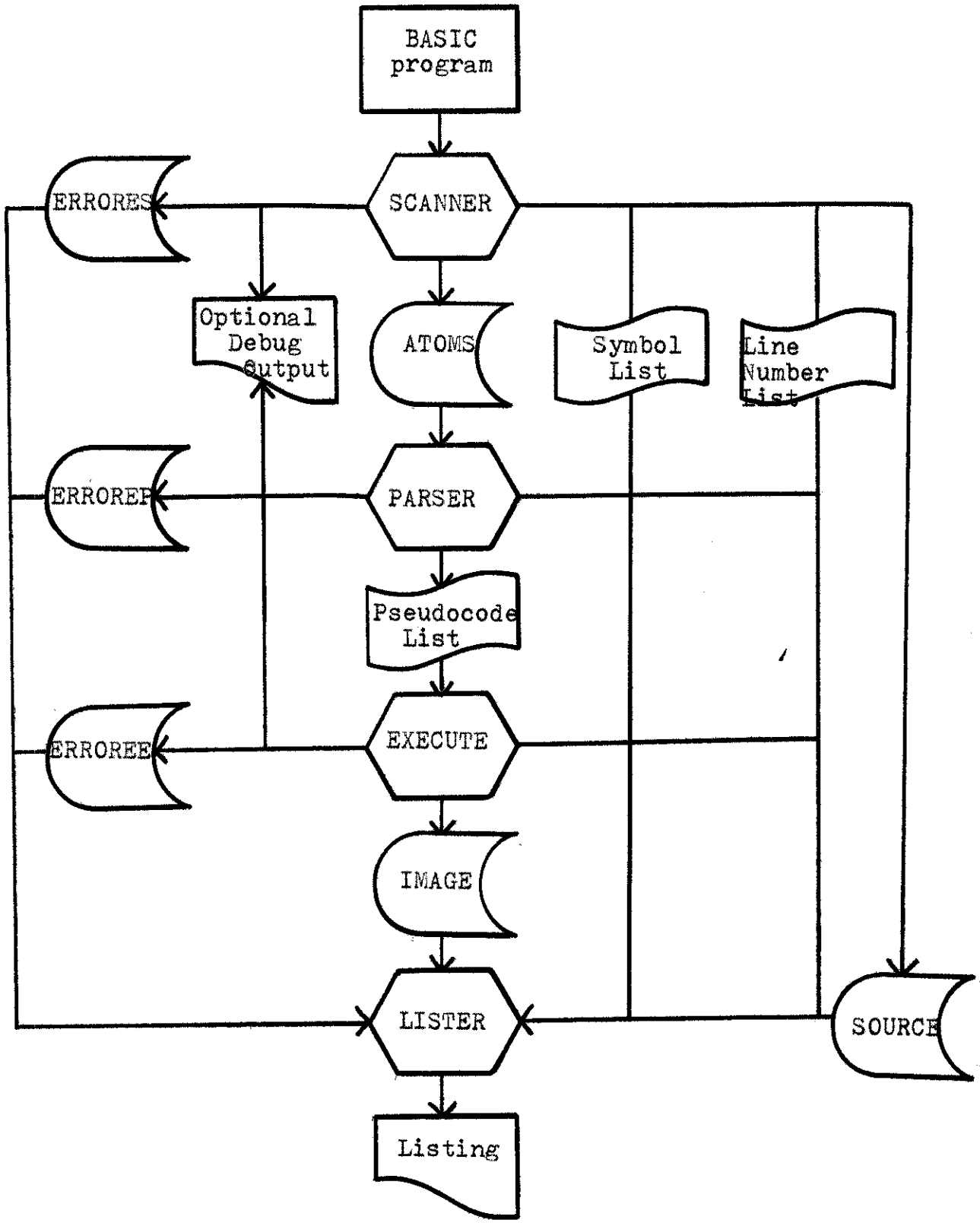

Fig. 1--Interpreter System Flow

TABLE I

LIST OF ATOMS

ATOM	OPERAND
Debug	Trace level for the PARSER and the EXECUTE.
Line Number	Pointer to the entry in the line number list.
Identifier	Pointer to the symbol list entry.
Constant	Pointer to the symbol list entry.
Edit	Pointer to the symbol list entry.
Left Paren	
Right Paren	
Comma	
Bar	
Dollar	
Plus	
Minus	
Multiply	
Divide	
Less Than	
Greater Than	
Equal	
Not	
IF	
ON	
TO	

TABLE I--Continued

ATOM	OPERAND
DIM	
END	
FOR	
LET	
REM	
DATA	
GOTO	
NEXT	
READ	
STEP	
STOP	
THEN	
GOSUB	
PRINT	
RETURN	
SIN	
COS	
TAN	
ATN	
EXP	
ABS	
LOG	
SQR	
INT	
DEF	

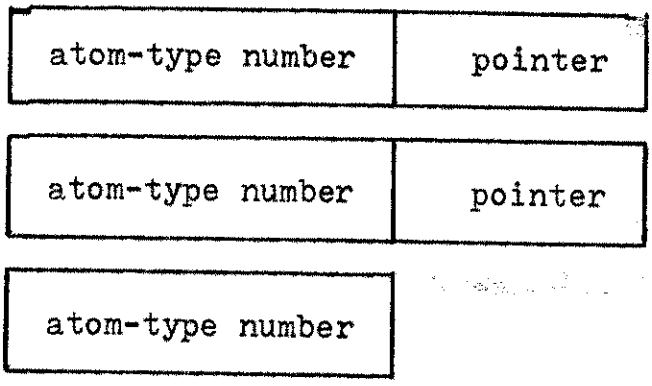

Fig. 2--Atom layout

ERRORES, ERROREP, and ERROREE

ERRORES, ERROREP, and ERROREE are external files built by the SCANNER, the PARSER and the EXECUTE respectively. All the files contain the error messages generated during processing of the BASIC program. All the files are merged into the output listing produced by the LISTER. Each entry in these files consists of a line number on which the error occurred and up to one hundred characters of text describing the error.

Symbol List

The symbol list is a simply linked list. The nodes of the list are dynamically allocated as they are needed. Each node contains eight fields. Figure 3 shows the lay-out of a node. A node is variable in size since the NAME field is variable in size.

The field called CHAIN always points to the next entry in the symbol list. The last entry in the symbol list has a CHAIN value of NULL, a special PL/1 value.

The field called TYPE is an integer value which indicates the node type. TYPE has different sets of values for each segment of this interpreter. The node types built by the SCANNER are

1. Identifier,
2. Constant,
3. Edit.

The node types recognized by the PARSER are:

1. Subscripted variable,
2. Simple variable,
3. Constant
4. Function
5. Edit
6. Dummy variable.

The node types recognized by the EXECUTE are

1. Subscripted variable
2. Simple variable
3. Constant.

The field called LOC is filled in by the PARSER and the EXECUTE. The PARSER fills the LOC when parsing a user-defined function. When the node is the type of user-defined function, the LOC is a pointer to an entry in the line number list. When the node is the type of dummy variable, the LOC is a pointer to a simple variable where the value of the dummy variable is stored. The EXECUTE sets the LOC during storage allocations.

The field called LNGH is filled in when the node is allocated. It contains the number which is the length of the NAME field in characters.

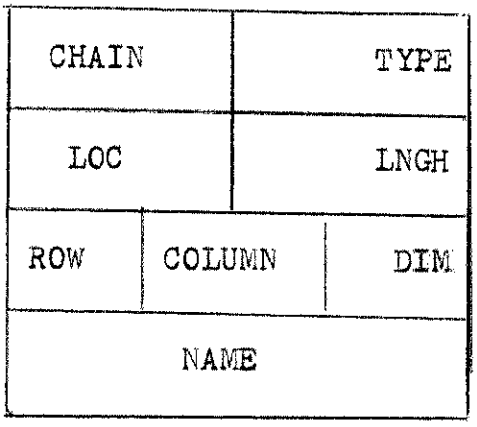

Fig. 3--Symbol Node

The field called ROW is filled in by the PARSER. It points to an entry in the symbol list where the row value is stored.

The field called COLUMN is filled in by the PARSER. It points to an entry in the symbol list where the column value is stored.

The field called DIM is filled in by the EXECUTE. It contains the number of elements in an array.

The field called NAME is filled in when the node is allocated. It contains an identifier which names an array, a simple variable, an edit, a defined function or a constant.

Nodes are allocated by the SCANNER and the PARSER. The SCANNER builds nodes for every identifier, edit and constant

found in the BASIC program. The PARSER builds nodes for temporaries. The PARSER also builds nodes for function values and dummy variable values.

Line Number List

The line number list is a one-way linked list. The nodes of the list are dynamically allocated as they are needed. Each node contains four fields. A node is variable in size since the NAMEL field is variable in size. Figure 4 shows the node layout in the line number list.

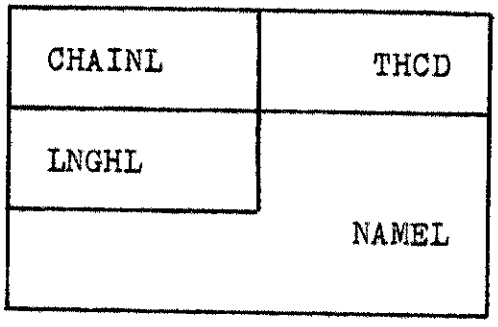

Fig. 4--Line number node

The field called CHAINL is always a pointer to the next entry in the line number list. The last entry in the line number list has a CHAINL value of NULL, a special PL/1 value.

The field called THCD is a pointer to an entry in the pseudocode list. It is used to transfer the program control when executing the pseudocode.

The field called LNGHL is filled in when the node is allocated. It contains a value which is the length of the NAMEL field in characters.

The field called NAMEL is filled in when the node is allocated. It contains a line number which names a source statement.

Pseudocode List

The pseudocode list is named LOGTAC in this interpreter. LOGTAC is a simply linked list built by the PARSER and is used as an input to the EXECUTE. It is the internal representation for the BASIC programming language. The entries in the pseudocode list are numbers which identify the pseudocode type optionally followed by one to three pointers or numbers. Figure 5 shows the five types of entries in the pseudocode list.

code	chaint			
code	number	chaint		
code	pointer	chaint		
code	pointer	pointer	chaint	
code	pointer	pointer	pointer	chaint

Fig. 5--Pseudocode node

The field called CHAINT always points to the next entry in the pseudocode list.

Table 2 defines the various codes and lists their respective operands, if any. Pointer operands point to the entries in the symbol list or in the line number list. A code defining an action on a pointer actually means the action applied to the entity defined by the symbol list entry or line number list entry pointed by the pointer.

IMAGE

IMAGE is an external file built by the EXECUTE and is used as an input to the LISTER. The entries in the file IMAGE are numbers which identify a certain output action followed by a variable which may point to an entry in the symbol list or point to a value in the storage. The output action may also be followed by a value of NULL (a special PL/1 value).

The three output actions are:

1. SKIP : Skip to a new line.
2. OUT0 : Output the value of a variable or constant.
3. OUT1 : Output messages.

TABLE II

PSEUDOCODE

CODE	OPERAND AND CODE DEFINITION
ATN	P1, P2 P1 is the real argument passed to built-in function ATN and the result is stored in P2.
EXP	P1, P2 P1 is the real argument passed to built-in function EXP and the result is stored in P2.
ABS	P1, P2 P1 is the real argument passed to built-in function ABS and the result is stored in P2.
LOG	P1, P2 P1 is the real argument passed to built-in function LOG and the result is stored in P2.
SQR	P1, P2 P1 is the real argument passed to built-in function SQR and the result is stored in P2.
INT	P1, P2 P1 is the real argument passed to built-in function INT and the result is stored in P2.

TABLE II--Continued

CODE	OPERAND AND CODE DEFINITION
OUTO	P1 OUTPUT P1 (number).
OUT1	P1 OUTPUT P1 (Character).
STOPCD	No operand. Stop execution.
SUBTRACT	P1, P2, P3 Subtract P1 by P2 and store the result in P3.
EXPONENTIATION	P1, P2, P3 Exponentiate P1 by P2 and store the result in P3.
SIN	P1, P2 P1 is the real argument passed to built-in function SIN and the result is stored in P2.
COS	P1, P2 P1 is the real argument passed to built-in function COS and the result is stored in P2.
TAN	P1, P2 P1 is the real argument passed to built-in function TAN and the result is stored in P2.

TABLE II--Continued

CODE	OPERAND AND CODE DEFINITION
EXIT	P1 Exit a function by returning the function value, P1.
GO	P1 Branch to P1.
GOIF-FALSE	P1, P2 If P1 is false, branch to P2; otherwise, execute the next code.
IN	P1 Input P1.
INDX	P1, P2, P3 Calculate the address of P1 subscripted by P2, and store the resulting address in P3.
MOV	P1, P2 Move P1 to P2.
NEGATE	P1, P2 Negate P1 and store the result in P2.
COMPARE-GT	P1, P2, P3 If P1 is greater than P2, P3 is set to true; otherwise, P3 is set to false.

TABLE II--Continued

CODE	OPERAND AND CODE DEFINITION
COMPARE-LE	P1, P2, P3 If P1 is less than or equal to P2, P3 is set to true; otherwise, P3 is set as false.
COMPARE-LT	P1, P2, P3 If P1 is less than P2, P3 is set to true; otherwise, P3 is set to false.
DODIVIDE	P1, P2, P3 Divide P1 by P2 and store the result in P3.
DOTIMES	P1, P2, P3 Multiply P1 by P2 and store the result in P3.
ENTER	P1, P2 Enter a function by storing the return address at P1, the function exit, and the argument value at P2, the dummy argument.
END-TAC	No operands. Indicates end of internal representation.
GDEBUG	Number. Set the EXECUTE trace level.

TABLE II--Continued

CODE	OPERAND AND CODE DEFINITION
GLNNO	Number. Sets a new line number.
ADD	P1, P2, P3 Add P1 to P2 and store the result in P3.
CALL	P1, P2, P3 The function, P1 is invoked, the real argument is stored in P2 and the returned value is stored in P3.
COMPARE-EQ	P1, P2, P3 If P1 is equal to P2, P3 is set to true; otherwise, P3 is set to false.
COMPARE-GE	P1, P2, P3 If P1 is greater than or equal to P2, P3 is set to true; otherwise, P3 is set to false.

CHAPTER BIBLIOGRAPHY

1. Donovan, John J., Systems Programming, New York, McGraw-Hill Book Company, 1972.

CHAPTER IV

INTERPRETER

Lexical Analysis

The segment concerned with lexical analysis of the source is implemented as an independently complied procedure called SCANNER. Reasons for separating lexical from syntactical analysis are discussed below (2):

1. A large portion of compile-time is spent in scanning characters. Separation allows sole concentration on reducing this time.
2. The syntax of symbols can be described by very simple grammars. Separating scanning from syntax recognition makes it possible to develop efficient parsing techniques.
3. Since the SCANNER returns a symbol instead of a character, the syntax analyzer actually gets more information about what to do at each step.
4. Development of high-level languages requires attention to both lexical and syntactic properties.
5. Separation makes it possible to write one syntactic analyzer and several scanners

which are simpler and easier to write. Each scanner translates the symbols into the same internal form used by the syntactic analyzer.

Problem

The problem of lexical analysis is to recognize certain strings as basic elements. The basic elements are placed into the symbol list or the line number list. As other segments recognize the use and meaning of the elements, further information is entered into these lists.

Data Structure

The input to the SCANNER is the BASIC source program. The BASIC program is on punched cards.

The outputs produced by the SCANNER are:

1. The symbol list.
2. The line number list
3. SOURCE
4. ATOMS
5. ERRORES
6. Optional debut listing.

Algorithm

The SCANNER procedure is an implementation of a finite-state machine (2) which breaks the source input into atoms and builds a symbol list and a line number list in the process. A state diagram for the finite-state machine is shown in Figure 6.

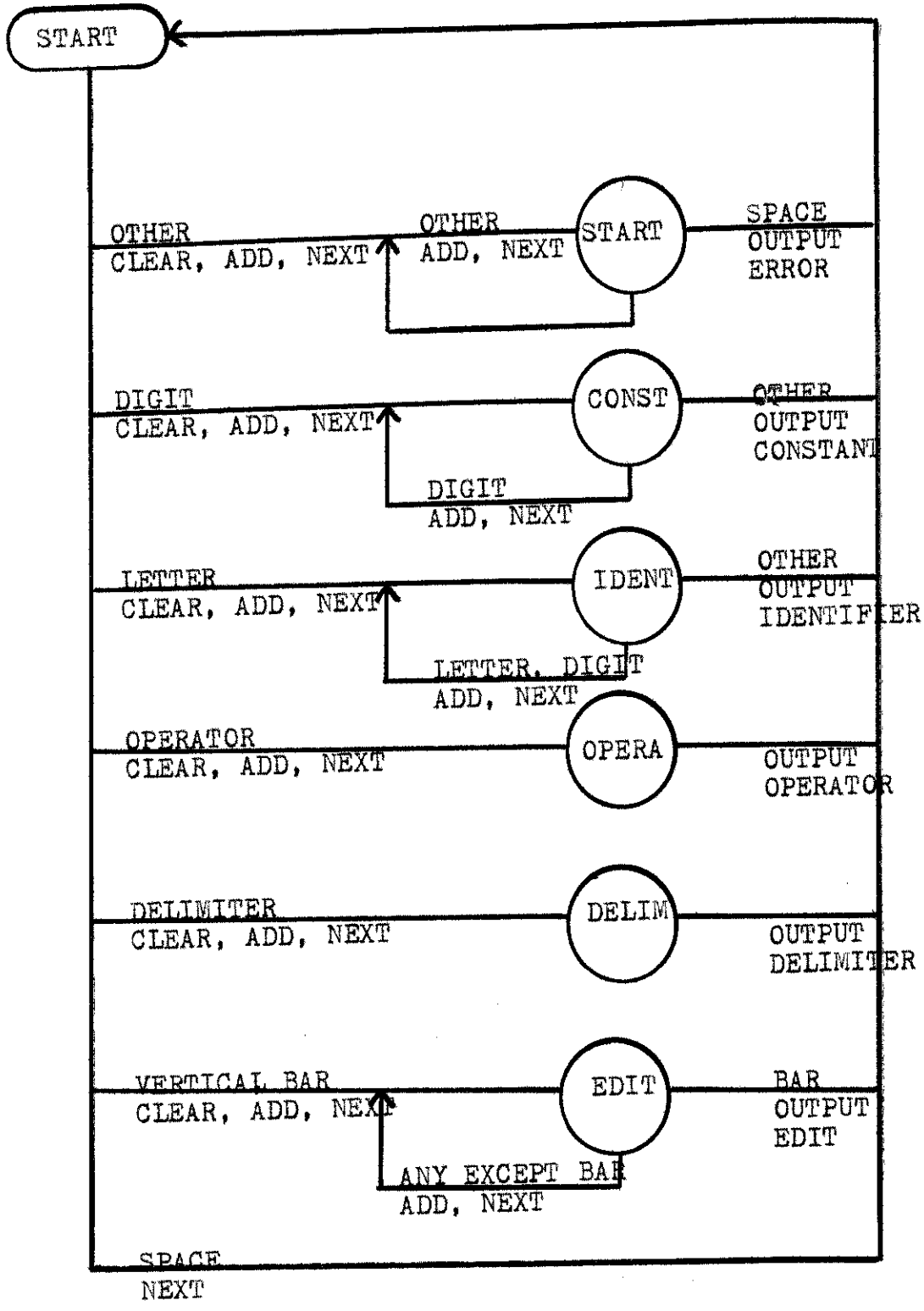

Fig. 6--Finite-state machine for lexical analysis

The 256 possible character codes are broken into classes using a translate-table. There are seven different character classes.

The finite-state machine simulation is done in the usual manner with two tables. One table which defines the next state function, and one table which defines the action associated with each state transition.

There are six different states in the finite-state machine; that is, START state, CONSTANT state, IDENTIFIER state, OPERATOR state, DELIMITER state and EDIT state.

In the lexical analysis comments are discarded since they have no effect on the processing of the program.

Parsing

The parsing segment is implemented as an independently compiled procedure called PARSER.

The parsing segment is an application of the top-down parsing (2) in this interpreter. Gauthier (1) defined the term "top-down parsing" as a procedure that creates goals and subgoals in attempting to relate a statement to its syntax environment.

The method of recursive descent (2) is used. The PARSER has one recursive procedure for each nonterminal symbol which parses phrases for the nonterminal symbol. The procedure is told where in the program to begin looking for a phrase (2) for the nonterminal symbol; hence it is goal oriented or predictive. The procedure finds its phrase by comparing the

source program at the point indicated by a cursor with right parts of rules for the nonterminal symbol, calling other procedures to recognize subgoals when necessary.

Problem

The problem in the parsing segment is to recognize the phrases and interpret the meaning of the constructions.

This process is known as syntax analysis. The PARSER also notes syntactic errors and assures some sort of recovery so that the interpreter can continue to look for other syntactic errors which were originally in the source.

Data Structure

The input to the PARSER is an external file called ATOMS.

The output produced by the PARSER are:

1. Pseudocode list
2. ERROREP
3. Optional debug listing.

The symbol list and line number list are used to support the syntactic analysis.

Algorithm

In general the PARSER segment, when called by the system, converts statements in the BASIC program to the internal representation and enters them in the pseudocode list. Atom by atom and statement by statement, a BASIC program is parsed.

Fundamentally the parsing segment is comprised of PROGRAM-HEAD and PROGRAM-TAIL. PROGRAM-HEAD is responsible for parsing DEFSTATEMENT and DIMSTATEMENT. PROGRAM-TAIL is responsible for parsing the other BASIC statements.

PROGRAM-HEAD has two main slaves. They are DEFINED-FUNCTION and DIM-STATEMENT. These three control programs as well as their service routines produce the appropriate pseudocodes and insert them in the pseudocode list for the DEFSTATEMENT and DIMSTATEMENT.

PROGRAM-TAIL consists of PROGRAM-MAIN and PROGRAM-CONTROLLER. PROGRAM-CONTROLLER is written as a recursive program. PROGRAM-MAIN is designed to drive the PROGRAM-CONTROLLER. When called, PROGRAM-CONTROLLER passes control to one of its fourteen slaves. These fourteen programs as well as their service routines are in charge of creating the appropriate pseudocodes. The block chart is shown in Figure 7.

DATASTATEMENT and READSTATEMENT

In processing the DATASTATEMENT, no pseudocode is produced. For each signed number appearing in the DATASTATEMENT, an entry is created in a linked list called DATA-INPUT. The DATA-INPUT is arranged on a first-in-first-out basis.

In processing the READSTATEMENT requests are made to the DATA-INPUT. For each simple or subscripted variable in the READSTATEMENT, a set of two pseudocodes is created. They are IN and MOV.

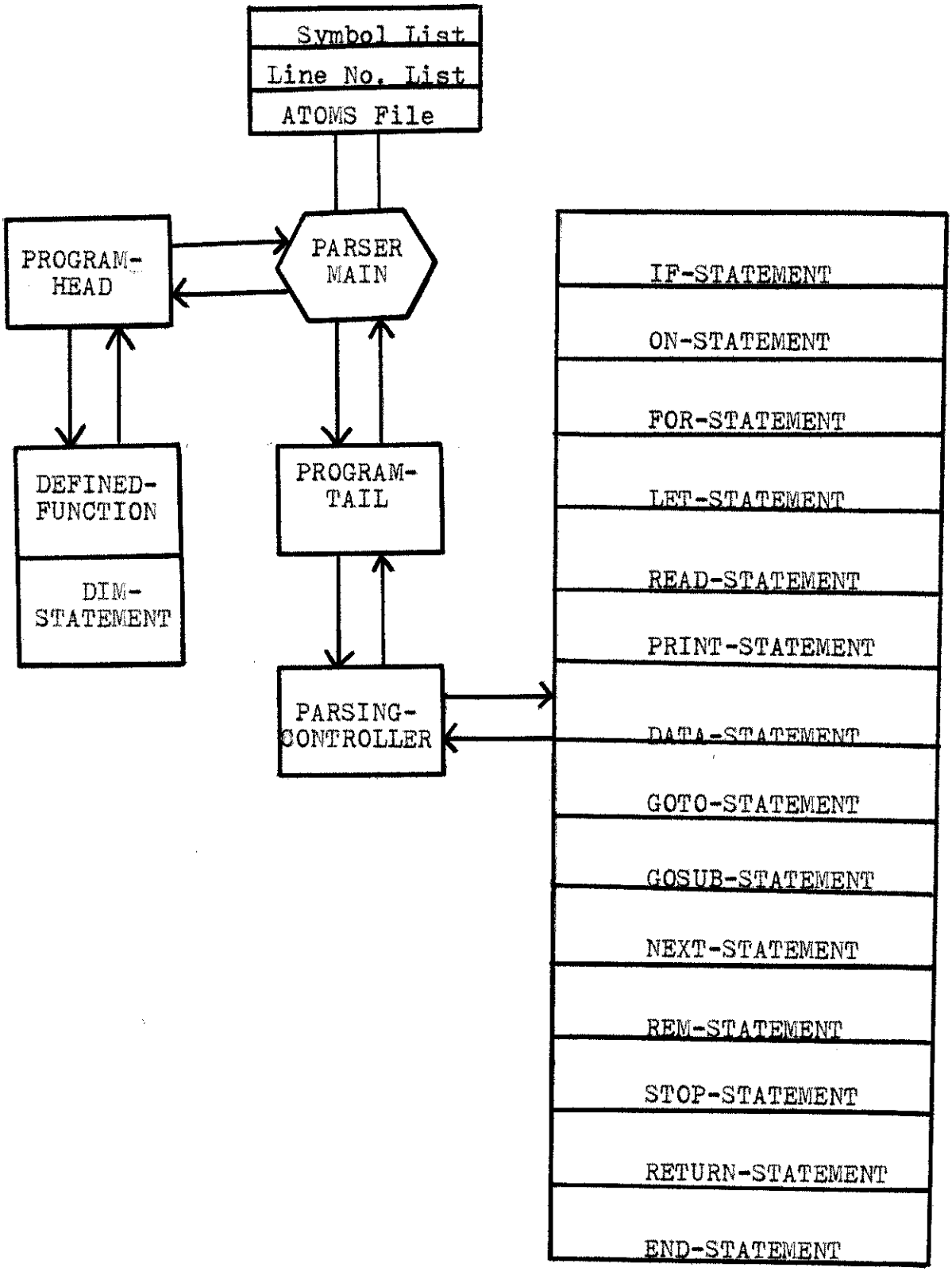

Fig. 7--Parsing Block Chart

Example.--The following is an example of the DATASTATEMENT and READSTATEMENT.

```
10  DATA  10, 25, 39
38  READ  X, Y, Z
```

After parsing these two statements, three entries were created in the DATA-INPUT list. Three sets of IN and MOV internal representation were added to the pseudocode list.

PRINTSTATEMENT

The BASIC of this interpreter provides for five zones of twenty characters each per line. Listed below are pseudocodes associated with the PRINTSTATEMENT:

1. SKIP: Skip to a new line
2. OUT0: Output the value of a variable
3. OUT1: Output heading or label messages.

Examples.--The following are examples of PRINTSTATEMENTs:

1. `10 PRINT`
2. `30 PRINT 'X= ', X`

In the first example, SKIP is the only pseudocode to be created in the parsing segment. In the second example, two entries are produced in the pseudocode list. The are OUT1 and OUT0.

LETSTATEMENT

The LETSTATEMENT is the assignment statement in the BASIC programming language.

Listed below are pseudocodes associated with the LETSTATEMENT:

1. ADD
2. SUBTRACT
3. DOTIMES
4. DODIVIDE
5. EXPONENTIATION
6. NEGATE
7. CALL

Elements involved in the LETSTATEMENT are constant, simple variables, subscripted variables and defined functions. It is slower to analyze and interpret subscripted variables than simple variables. Each subscripted variable needs three extra pseudocodes. They are SUBTRACT, DOTIMES and ADD.

Examples.--The following are examples of LETSTATEMENTs:

1. 10 LET XYZ = 10
2. 20 LET X = Y - Z + FNA(2) * 4 / 2

In the first example, MOV is the only internal representation created in the pseudocode list. In the second example, "FNA" is assumed to be the name of a defined function in a BASIC program. Six pseudocodes are produced in the pseudocode list. They are SUBTRACT, CALL, DOTIMES, DODIVIDE, ADD and MOV.

GOTOSTATEMENT

In processing the GOTOSTATEMENT, the "GO" internal representation is produced in the pseudocode list. In this internal representation there is a pointer to an entry in the pseudocode list, to which the program is supposed to transfer the control.

The GOTO-STATEMENT routine fetches this pointer in the line number list and uses it as the operand of the "GO" pseudocode.

ONSTATEMENT

When processing the ONSTATEMENT, the ON-STATEMENT routine will create an index variable and set its initial value to 1. Pseudocodes associated with the ONSTATEMENT are:

1. ADD
2. SUBTRACT
3. DOTIMES
4. DODIVIDE
5. EXPONENTIATION
6. COMPARE-EQ
7. GOIF-TRUE
8. MOV

Examples.--The following are examples of ONSTATEMENTs:

1. ON X THEN 10,20
2. ON X+Y THEN 100, 200, 300

In the first example six entires were created in the pseudocode list. They are MOV, COMPARE-EQ, GOIF-TRUE, ADD, COMPARE-EQ, and GOIF-TRUE. In the second example, pseudocodes created in the pseudocode list are MOV, COMPARE-EQ, GOIF-TRUE, ADD, COMPARE-EQ, GOIF-TRUE, ADD, COMPARE-EQ and GOIF-TRUE.

IFSTATEMENT

The IFSTATEMENT may be used to conditionally alter the execution flow of a BASIC program. An IFSTATEMENT has the form:

ln IF expression1 relation expression2 THEN ln1

ln2 -----

ln1 -----

The effect of the IFSTATEMENT is to transfer control from the current statement to the statement numbered ln1. If the relation is satisfied, the program control is transferred to statement ln1. Otherwise, statement ln2 will take over program control.

Pseudocodes are first generated to compute the values of expression1 and expression2 respectively. One of the following pseudocodes is then generated:

1. COMPARE-EQ
2. COMPARE-GE
3. COMPARE-GT
4. COMPARE-LE
5. COMPARE-LT
6. COMPARE-NEQ

Finally, the pseudocode "GOIF-TRUE" is added to the pseudocode list.

Examples.--The following are examples of IFSTATEMENTs:

1. 10 IF X+Y = X*Y THEN 100
2. 20 IF X/Y**2 = X-Y*2 THEN 200

In the first example four entries are created in the pseudocode list. They are ADD, DOTIMES, COMPARE-EQ and GOIF-TRUE. In the second example, pseudocodes generated in the pseudocode list are EXPONENTIATION, DODIVIDE, DOTIMES, SUBTRACT, COMPARE-EQ and GOIF-TRUE.

FORSTATEMENT and NEXTSTATEMENT

The FORSTATEMENT and the NEXTSTATEMENT are used in pairs to govern the repeated execution of several BASIC statements. When processing the FORSTATEMENT, the FOR-STATEMENT first initializes the index variable. It then stacks the index variable, increment value, final value and the line number of the first statement following the FORSTATEMENT. An index variable is basic to the control at execution. At each execution-time iteration of the FOR range the index variable is updated by an increment value. Iteration continues until the index variable reaches its final value. At that time execution drops down through the NEXTSTATEMENT.

When processing the NEXTSTATEMENT, the NEXT-STATEMENT routine first unstacks the index variable, increment value, final value and the line number of the first statement following the paired FORSTATEMENT. It then updates the index variable and examines the value of the index variable against the final value.

To check for proper program sequence, the PARSER keeps a counter. Whenever a FORSTATEMENT is encountered, this counter is incremented by one. It is decremented by one for a NEXTSTATEMENT. At the completion of nesting pairs of FORSTATEMENTs and NEXTSTATEMENTs, the counter is expected to have the value of zero.

Examples.--The following are examples of FORSTATEMENTs and NEXTSTATEMENTs:

```
1.  10  FOR X = 2 TO 8 STEP 2
    20  LET Y = Y + 2 + 4
    30  NEXT X
2.  40  FOR Y = 1 TO 10 STEP 5
    50  FOR Z = 1 TO 10 STEP 5
    60  LET XYZ (Y,Z) = Y + Z
    70  NEXT Z
    80  NEXT Y
```

In the first example, X is the index variable. Its initial value is assigned to be two. The increment value is two and the final value is four. Six entries are created in the pseudocode list. They are MOV, ADD, ADD, MOV, COMPARE-LE and GOIF-TRUE.

In the second example pseudocodes generated in the pseudocode list are MOV, MOV, ADD, SUBTRACT, DOTIMES, ADD, MOV, COMPARE-LE, GOIF-TRUE, COMPARE-LE and GOIF-TRUE.

GOSUBSTATEMENT and RETURNSTATEMENT

Parameterless subroutines are allowed in BASIC programs through the GOSUBSTATEMENT and the RETURNSTATEMENT.

In order to allow the nesting of pairs of the GOSUBSTATEMENTs and the RETURNSTATEMENTs, a linked list called RETURN-STACK is used to save the return addresses on a last-in-first-out basis.

When processing the GOSUBSTATEMENT, the GOSUB-STATEMENT routine pushes down the return address. When processing

the RETURNSTATEMENT, the RETURN-STATEMENT routine pops up the return address.

Examples.--The following are examples of the GOSUBSTATEMENTs and RETURNSTATEMENTs:

```
1.  10  GOSUB 100
    20  -----
    100 -----
        -----
    200 RETURN
2.  10  GOSUB 100
    20  -----
    100 GOSUB 200
    101 -----
    105 RETURN
    200 -----
        -----
    300 RETURN
```

In the first example, the RETURN-STACK has only one return address. Statement 100 stacks it and statement 200 unstacks it. In the second example, the RETURN-STACK contains two return addresses. When executing the statement 300, the return address popped up by the RETURN-STATEMENT routine is line number 101.

DIMSTATEMENT

When processing the DIMSTATEMENT, the DIM-STATEMENT routine identifies the source-language names of vectors and

arrays. It also saves the values of row parameters and column parameters.

DEFSTATEMENT

When processing the DEFSTATEMENT, the DEF-STATEMENT routine identifies the name of the defined function and the name of the dummy variable. The first psuedocode created by the DEF-STATEMENT is ENTER. The last pseudocode created by the DEF-STATEMENT is EXIT. Control transfer and argument replacement in function call are performed by pseudocode CALL and ENTER. The result is passed back through the EXIT pseudocode.

STOPSTATEMENT

When processing the STOPSTATEMENT, the STOP-STATEMENT routine generates the pseudocode STOPCD.

ENDSTATEMENT

When processing the ENDSTATEMENT, no pseudocode is created in the pseudocode list. In this interpreter, as soon as the ENDSTATEMENT is encountered, the program control is transferred back to the main control program.

Execution

The execution segment is implemented as an independently compiled procedure called EXECUTE. EXECUTE consists of two major activities: storage allocation and pseudocode execution.

Problem

The problem in the execution segment is to reserve the proper amounts of storage required by the BASIC program. Once the interpreter has created the pseudocode list and reserved the proper amounts of storage, it may start to execute the pseudocode list.

Data Structure

The input to the EXECUTE is the pseudocode list. The pseudocode list is a one-way linked list built by the parsing segment.

The outputs produced by the EXECUTE are:

1. IMAGE: It is a one-way linked list used as an input to the LISTER.
2. ERROREE.
3. Optional debug listing.

The symbol list and line number list support the execution segment. Their addresses are passed as external addresses.

Algorithm

When called by the main program, EXECUTE starts to assign storages and execute pseudocodes.

Storage allocation is performed by scanning the symbol list and reserving the appropriate amount of storage for each constant, simple variable and subscripted variable. Similarly, storage is assigned for the temporary locations that will contain intermediate results.

It is the responsibility of EXECUTE to maintain the pseudocode execution sequence. In order to carry out the execution in the correct sequence, EXECUTE must keep track of where it is in processing the pseudocode list. To this end, a pointer called PSW is used to hold the next pseudocode to be executed. As pseudocode is processed, the PSW is advanced to the next pseudocode. This segment may be summarized in block chart form as shown in Figure 8.

Listing

The listing segmentation is implemented as an independently compiled procedure called LISTER.

Problem

The problem in the listing segment is to produce a source listing with line numbers. Error messages from other segmentations are merged into the source listing.

Data Structure

The inputs to the LISTER are:

1. SOURCE
2. ERRORES
3. ERROREP
4. ERROREE
5. IMAGE

The output produced by the EXECUTE is the source listing.

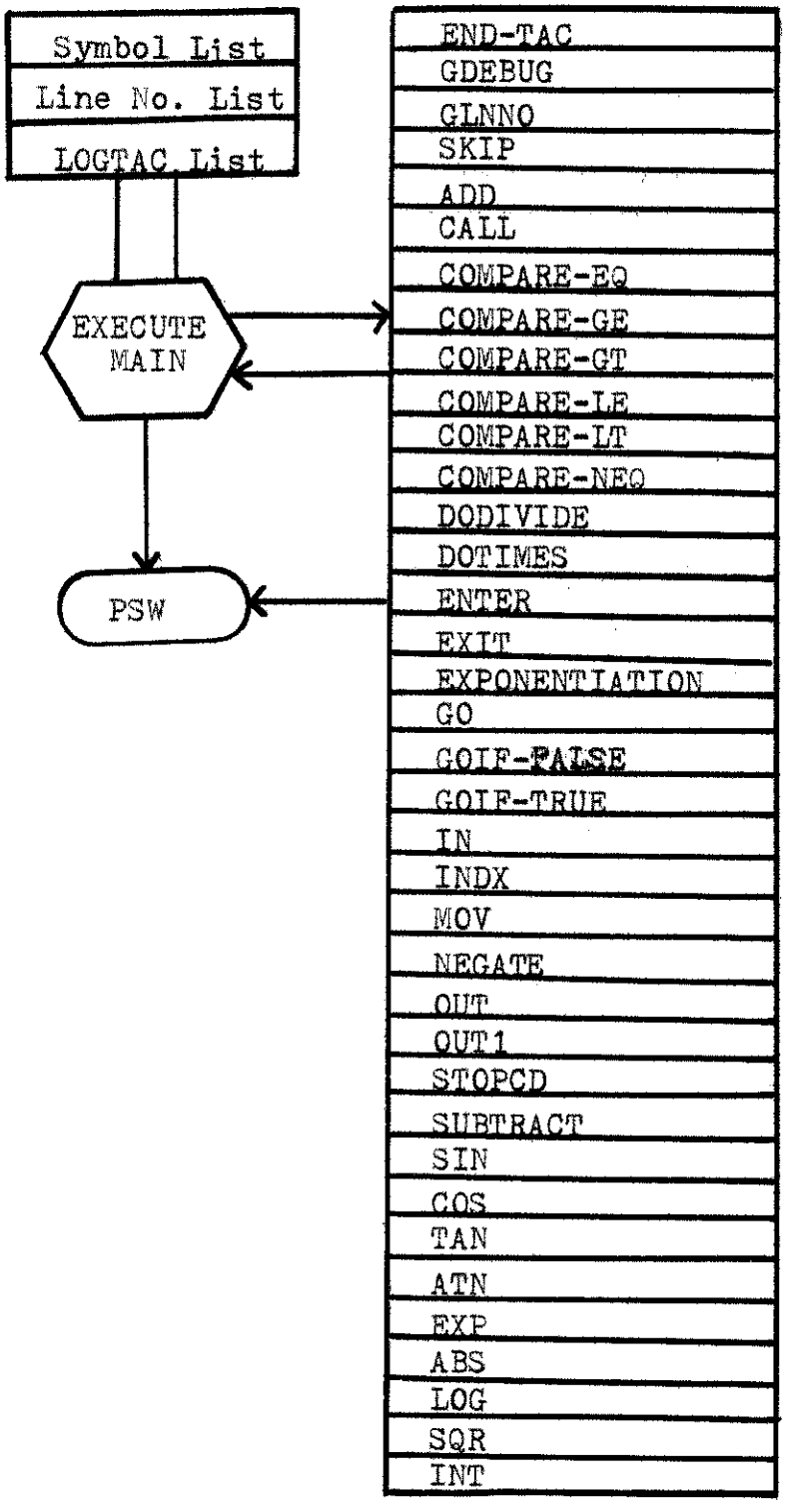

Fig. 8--Executing Block Chart

Algorithm

The inputs are processed by the LISTER in the following order: SOURCE, ERRORES, ERROREP, ERROREE and IMAGE.

Debug Output

SCANNER, PARSER and EXECUTE generate a debug listing when requested. The debug output options are controlled from the source input stream. The options can be turned on and off at any point during execution with single source cards. There are several levels of debug output that can be selected.

Table III shows the output for each level. Samples of debug outputs are shown in Appendix C.

Testing

This interpreter was tested in several stages. The stages were (3):

1. Stage 1--SCANNER output was hand-checked using the debug output.
2. Stage 2--PARSER output was hand-checked using the debug output. The PARSER and the SCANNER were integrated for this stage.
3. Stage 3--EXECUTE output was hand-checked using the debug output. EXECUTE was integrated with PARSER and SCANNER for this stage.

4. Stage 4--LISTER output was hand-checked using the debug output. LISTER was integrated with PARSER, SCANNER, and EXECUTE for this stage.

Test programs for stage 1 through 3 were designed to test the interpreter rather than go into execution. The programs were designed to test each major feature of the interpreter on a statement by statement basis. Programs for stage 4 were designed to produce output that could be interpreted as correct or incorrect depending on whether or not the interpretation had been correct. Samples of testing programs are shown in Appendixes 9-14.

TABLE III

DEBUG OUTPUT

PHASE	LEVEL	OUTPUT
SCANNER	0	None
	1	Source, Errors, Atoms
	2	Source, Errors, Atoms, Detailed trace of finite-state machine
PARSER	0	None
	1	Source, Errors, Pseudo-code
	2	Source, Pseudocode, Atoms, Detailed subroutine trace of PARSER
EXECUTE	0	None
	1	Execution sequence of pseudocode
	2	Execution sequence of pseudocode, value of location before and after execution.

CHAPTER BIBLIOGRAPHY

1. Gauthier, Ponto, Designing Systems Programs, New Jersey, Prentice-Hall, Inc., 1970.

2. Gries, David, Compiler Construction for Digit Computers, New York, John Wiley & Sons, Inc., 1971.

3. Isaacson, Portia, A Compiler for This Programming Language, Department of Computer Sciences, North Texas State University, Denton, Texas, 1971 (unpublished).

CHAPTER V

CONCLUSION

For this BASIC translator, an interpretive approach was selected. Rather than translating the source statement to directly executable computer machine code, the statement is interpreted in object pseudocodes. Some particular advantages for the interpretive method are (1)

1. Easier alteration to a running program.
2. Reduction in object code size, especially when data type checking is involved.
3. Greater diagnostic capability.
4. Easier portability of object programs.

Its disadvantages can be slowness due to software decoding of pseudocodes and repetitive interpretation of unchanging elements. But even the slowness of execution is not a disadvantage for program development, where compile time greatly exceeds execution time--it is only a disadvantage for production programs.

Repeated attempts with various algorithms proved that the best approach is to divide characters into groups. In this lexical analysis six groups are devised. Class I represents the twenty-six characters of the English alphabet. Class II represents the digits zero through nine while

the operators are assigned to the third class. Classes 4, 5 and 6 represent delimiters, blank, and quotation mark respectively.

This allows quick adaptation to finite-machine simulation; hence, the process requires less time. Also, the number of data bases is reduced. To accomplish this, the terminal and identifier tables are combined with the resulting table having a pointer that points to either the symbol table or line number table. The line number table acts as the bridge within the statements of the program and allows for continuous execution of the pseudocode list. These adaptations decrease the amount of storage required.

Through the careful design of pseudocodes and elimination of the use of unnecessary repetitions of the pseudocode, the interpreter is made more efficient and effective. The storage space for the pseudocode list is reduced, thus reducing the execution time.

The interpreter is also equipped to detect and correct minor errors before continuing execution. Also, in order to facilitate program debugging, an optional trace listing is made accessible to the programmer. This debug output facilitates detection and correction of these errors.

In conclusion, no major problems are encountered in building the interpreter and an even more efficient one seems possible through further adaptations of the interpreter presented.

CHAPTER BIBLIOGRAPHY

1. Broadbent, J.K., "Microprogramming and System Architecture," The Computer Journal (Volume 17 Number 1), 1973.

APPENDIX I

RESERVED WORDS

IF	REM
ON	SIN
TO	SQR
ABS	TAN
ATN	DATA
COS	GOTO
DEF	NEXT
DIM	READ
END	STEP
EXP	STOP
FOR	THEN
INT	GOSUB
LET	PRINT
LOG	RETURN

APPENDIX II

BACKUS NORMAL FORM OF THIS BASIC PROGRAMMING LANGUAGE

```
<BASIC PROGRAM> ::= <PROGRAM-HEAD><PROGRAM-TAIL><TERMINAL ST>
                  |<PROGRAM-TAIL><TERMINAL ST>
<PROGRAM-HEAD> ::=<REM ST><DEF ST><DIM ST>|<REM ST><DEF ST>|<REM ST><DIM ST>
                                |<DEF ST><DIM ST>|<DEF ST>|<DIM ST>
<FUNCTION VARIABLE> ::= FN<LETTER>|FN<DIGIT>
<DEF ST> ::=<LINE NUMBER><BLANK>DEF<BLANK><FUNCTION VARIABLE><LEFT PAREN>
             <SIMPLE VARIABLE><RIGHT PAREN><EQUAL SIGN><EXPRESSION>
<DIM ST> ::=<LINE NUMBER><BLANK>DIM<BLANK><DIMENSION LIST>
<DIMENSION LIST> ::=<DIMENSION>|<DIMENSION LIST><COMMA><DIMENSION>
<DIMENSION> ::=<SIMPLE VARIABLE><LEFT PAREN><INTEGER><COMMA><RIGHT PAREN>
              |<SIMPLE VARIABLE><LEFT PAREN><INTEGER><COMMA><INTEGER><RIGHT PAREN>
<TERMINAL ST> ::=<END ST>
<PROGRAM TAIL> ::=<STATEMENT>|<PROGRAM TAIL><STATEMENT>
<STATEMENT> ::=<LET ST>|<READ ST>|<DATA ST>|<PRINT ST>|<GOTO ST>|<ON ST>|
                <IF ST>|<FOR ST>|<NEXT ST>|<DIM
                <IF ST>|<FOR ST>|<NEXT ST>|<GOSUB ST>|<RETURN ST>|<STOP ST>|
                <REM ST>
<LINE NUMBER> ::=<DIGIT>|<LINE NUMBER><DIGIT>
<REM ST> ::=<LINE NUMBER><BLANK>REM<BLANK><MESSAGE>
<LET ST> ::=<LINE NUMBER><BLANK><VARIABLE><EQUAL SIGN><EXPRESSION>
<EXPRESSION> ::=<MULTIPLY FACTOR>|<PREFIX OP><EXPRESSION>|
                <EXPRESSION><LOW PRIORITY><MULTIPLY FACTOR>
<MULTIPLY FACTOR> ::=<MULTIPLY FACTOR><HIGH PRIORITY><INVOLUTION FACTOR>|
                     <INVOLUTION FACTOR>
<PREFIX OP> ::=<PLUS SIGN>|<MINUS SIGN>
<HIGH PRIORITY> ::=<ASTERISK SIGN>|<SLASH SIGN>
<LOW PRIORITY> ::=<PLUS SIGN>|<MINUS SIGN>
<INVOLUTION FACTOR> ::=<TERM>|<TERM><ASTERISK SIGN><ASTERISK SIGN><TERM>
<TERM> ::=<CONSTANT>|<VARIABLE>|<FUNCTION REF>|<EXPRESSION>
<VARIABLE> ::=<SIMPLE VARIABLE>|<DIMENSION VARIABLE>
<DIMENSION VARIABLE> ::=<SIMPLE VARIABLE><LEFT PAREN><EXPRESSION><COMMA>
                        <EXPRESSION><RIGHT PAREN>
                       |<SIMPLE VARIABLE><LEFT PAREN><EXPRESSION><RIGHT PAREN>
<SIMPLE VARIABLE> ::=<LETTER>|<LETTER><LETTERDIGIT>
<LETTERDIGIT> ::=<ALPHABET>|<LETTERDIGIT><ALPHABET>
<ALPHABET> ::=<LETTER>|<DIGIT>
<CONSTANT> ::=<NUMBER>
<NUMBER> ::=<INTEGER>|<DECIMAL>
<INTEGER> ::=<DIGIT>|<INTEGER><DIGIT>
<DECIMAL> ::=<INTEGER><PERIOD><INTEGER>|<PERIOD><INTEGER>
<DIGIT> ::=0|1|2|3|4|5|6|7|8|9
<SIGN> ::=<PLUS SIGN>|<MINUS SIGN>
<FUNCTION REF> ::=<FUNCTION NAME><LEFT PAREN><EXPRESSION><RIGHT PAREN>
<FUNCTION NAME> ::=<BUILT-IN FUNCTION>|<USER FUNCTION>
<USER FUNCTION> ::=<FUNCTION VARIABLE>
<BUILT-IN FUNCTION> ::=SIN|COS|TAN|ATN|EXP|ABS|LOG|SQR|INT
<SIGNED NUMBER> ::=<SIGN><CONSTANT>|<CONSTANT>
<DATA LIST> ::=<SIGNED NUMBER>|<DATA LIST><COMMA><SIGNED NUMBER>
<DATA ST> ::=<LINE NUMBER><BLANK>DATA<BLANK><DATA LIST>
<GOTO ST> ::=<LINE NUMBER><BLANK>GOTO<BLANK><LINE NUMBER>
<LINE NUMBER LIST> ::=<LINE NUMBER>|<LINE NUMBER LIST><COMMA><LINE NUMBER>
<ON ST> ::=<LINE NUMBER><BLANK>ON<BLANK><EXPRESSION><BLANK>THEN
          <BLANK><LINE NUMBER LIST>
```

APPENDIX II--Continued

```
<GOSUB ST> ::=<LINE NUMBER><BLANK>GOSUB<BLANK><LINE NUMBER>
<RETURN ST> ::=<LINE NUMBER><BLANK>RETURN
<IF ST> ::=<LINE NUMBER><BLANK>IF<BLANK><BOOLEAN EXPRESSION><BLANK>THEN
          <BLANK><LINE NUMBER>
<BOOLEAN EXPRESSION> ::=<EXPRESSION><RELATION><EXPRESSION>
<RELATION> ::= =|>=|<=|>|<|<>
<FOR ST> ::=<LINE NUMBER><BLANK>FOR<BLANK><SIMPLE VARIABLE><EQUAL SIGN>
           <EXPRESSION><BLANK>TO<BLANK><EXPRESSION><BLANK>STEP
           <BLANK><EXPRESSION>
<NEXT ST> ::=<LINE NUMBER><BLANK>NEXT<BLANK><SIMPLE VARIABLE>
<STOP ST> ::=<LINE NUMBER><BLANK>STOP
<END ST> ::=<LINE NUMBER><BLANK>END
<VARIABLE LIST> ::=<VARIABLE>|<VARIABLE LIST><COMMA><VARIABLE>
<READ ST> ::=<LINE NUMBER><BLANK>READ<BLANK><VARIABLE LIST>
<PRINT LIST> ::=<PRINT ITEM>|<PRINT LIST><COMMA><PRINT ITEM>
<PRINT ITEM> ::=<EDIT>|<EXPRESSION>
<EDIT> ::=<BAR>|<MESSAGE>|<BAR>
<MESSAGE> ::=<CHARACTER>|<MESSAGE><CHARACTER>
<CHARACTER> ::=<LETTER>|<DIGIT>|<SPECIAL CHARACTER>
<PRINT ST> ::=<LINE NUMBER><BLANK>PRINT<BLANK><PRINT LIST>
<LETTER> ::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<SPECIAL CHARACTER> ::=+|-|*|/|=|)|((|<|>|'|,|:|<BLANK>|$|?
<PLUS SIGN> ::=+
<MINUS SIGN> ::=-
<EQUAL SIGN> ::==
<LEFT PAREN> ::=(
<RIGHT PAREN> ::=)
<ASTERISK SIGN> ::=*
<SLASH SIGN> ::=/
<PERIOD> ::=.
<COMMA> ::=,
```

APPENDIX III

TRACE LEVEL 1 OF SCANNER

```
WRITE2 ATOM COMMA
WRITE0 ATOM CONSTANT        2
WRITE2 ATOM RIGHT_PAREN
WRITE2 ATOM COMMA
WRITE0 ATOM IDENTIFIER      A
WRITE2 ATOM LEFT_PAREN
WRITE0 ATOM CONSTANT        2
WRITE2 ATOM COMMA
WRITE0 ATOM CONSTANT        1
WRITE2 ATOM RIGHT_PAREN
WRITE2 ATOM COMMA
WRITE0 ATOM IDENTIFIER      A
WRITE2 ATOM LEFT_PAREN
WRITE0 ATOM CONSTANT        2
WRITE2 ATOM COMMA
WRITE0 ATOM CONSTANT        2
WRITE2 ATOM RIGHT_PAREN
WRITE2 ATOM COMMA
WRITE0 ATOM IDENTIFIER      B
WRITE2 ATOM LEFT_PAREN
WRITE0 ATOM CONSTANT        1
WRITE2 ATOM COMMA
WRITE0 ATOM CONSTANT        1
WRITE2 ATOM RIGHT_PAREN
WRITE2 ATOM COMMA
WRITE0 ATOM IDENTIFIER      B
WRITE2 ATOM LEFT_PAREN
WRITE0 ATOM CONSTANT        1
WRITE2 ATOM COMMA
WRITE0 ATOM CONSTANT        2
WRITE2 ATOM RIGHT_PAREN
WRITE2 ATOM COMMA
WRITE0 ATOM IDENTIFIER      B
WRITE2 ATOM LEFT_PAREN
WRITE0 ATOM CONSTANT        2
WRITE2 ATOM COMMA
WRITE0 ATOM CONSTANT        1
WRITE2 ATOM RIGHT_PAREN
WRITE2 ATOM COMMA
WRITE0 ATOM IDENTIFIER      B
WRITE2 ATOM LEFT_PAREN
WRITE0 ATOM CONSTANT        2
WRITE2 ATOM COMMA
WRITE0 ATOM CONSTANT        2
WRITE2 ATOM RIGHT_PAREN
102  GOSUB 104
WRITE3 ATOM LINE_NUMBER     102
WRITE2 ATOM GOSUB
WRITE0 ATOM CONSTANT        104
103  STOP
WRITE3 ATOM LINE_NUMBER     103
WRITE2 ATOM STOP
104  FOR I = 1 TO 2
WRITE3 ATOM LINE_NUMBER     104
WRITE2 ATOM FOR
WRITE0 ATOM IDENTIFIER      I
WRITE2 ATOM EQUAL
WRITE0 ATOM CONSTANT        1
WRITE2 ATOM TO
WRITE0 ATOM CONSTANT        2
```

APPENDIX IV

TRACE LEVEL 1 OF PARSER

```
GLNNC       103
STOPCD
GLNNC       104
MOV         I                I
GLNNC       105
MOV         I                J
GLNNC       557
SUBTRACT    I                1               T$17
DOTIMES     T$17             2               T$18
ADD         T$18             J               T$17
INDX        C                T$17            $C$
MOV         0                $C$
GLNNC       106
MOV         I                K
GLNNC       107
SUBTRACT    I                1               T$19
DOTIMES     T$19             2               T$20
ADD         T$20             J               T$19
INDX        C                T$19            $C$
SUBTRACT    I                1               T$21
DOTIMES     T$21             2               T$22
ADD         T$22             J               T$21
INDX        C                T$21            $C$
SUBTRACT    I                1               T$23
DOTIMES     T$23             2               T$24
ADD         T$24             K               T$23
INDX        A                T$23            $A$
SUBTRACT    K                1               T$25
DOTIMES     T$25             2               T$26
ADD         T$26             J               T$25
INDX        B                T$25            $B$
DOTIMES     $A$              $B$             T$27
ADD         $C$              T$27            T$28
MOV         T$28             $C$
GLNNC       108
COMPARE_LT  K                2               T$29
ADD         K                1               K
GOIF_TRUE   T$29             107
GLNNC       109
COMPARE_LT  J                2               T$30
ADD         J                1               J
GOIF_TRUE   T$30             997
GLNNC       050
COMPARE_LT  I                2               T$31
ADD         I                1               I
GOIF_TRUE   T$31             105
GLNNC       051
GO          025
GLNNC       090
GO          090
GLNNC       024
GLNNC       025
SKIP
GLNNC       027
SKIP
OUT1        C(1,1)
OUT1        C(1,2)
GLNNC       028
SKIP
INDX        C                1               $C$
```

APPENDIX V

TRACE LEVEL 1 OF EXECUTE

```
DOTIMES     T$17        2           T$18
ADD         T$18        J           T$17
INDX        C           T$17        $C$
MOV         0           $C$
GLNNC       100
MOV         1           K
GLNNC       107
SUBTRACT    I           1           T$19
DOTIMES     T$19        2           T$20
ADD         T$20        J           T$19
INDX        C           T$19        $C$
SUBTRACT    I           1           T$21
DOTIMES     T$21        2           T$22
ADD         T$22        J           T$21
INDX        C           T$21        $C$
SUBTRACT    I           1           T$23
DOTIMES     T$23        2           T$24
ADD         T$24        K           T$23
INDX        A           T$23        $A$
SUBTRACT    K           1           T$25
DOTIMES     T$25        2           T$26
ADD         T$26        J           T$25
INDX        B           T$25        $B$
DOTIMES     $A$         $B$         T$27
ADD         $C$         T$27        T$28
MOV         T$28        $C$
GLNNC       108
COMPARE_LT  K           2           T$29
ADD         K           1           K
GOIF_TRUE   T$29        107
```

APPENDIX VI

TRACE LEVEL 2 OF SCANNER

```
LINE =    10                    STATE =  1   COLUMN_TYPE =  4   ATOM = *
LINE =    10                    STATE =  5   COLUMN_TYPE =  1   ATOM = (
WRITE2 ATOM LEFT_PAREN
LINE =    10                    STATE =  1   COLUMN_TYPE =  1   ATOM = (
LINE =    10                    STATE =  2   COLUMN_TYPE =  3   ATOM = X
WRITE0 ATOM IDENTIFIER      X
LINE =    10                    STATE =  1   COLUMN_TYPE =  3   ATOM = X
LINE =    10                    STATE =  4   COLUMN_TYPE =  2   ATOM = +
WRITE2 ATOM PLUS
LINE =    10                    STATE =  1   COLUMN_TYPE =  2   ATOM = +
LINE =    10                    STATE =  3   COLUMN_TYPE =  4   ATOM = 1
WRITE0 ATOM CONSTANT        1
LINE =    10                    STATE =  1   COLUMN_TYPE =  4   ATOM = 1
LINE =    10                    STATE =  5   COLUMN_TYPE =  5   ATOM = )
WRITE2 ATOM RIGHT_PAREN
LINE =    10                    STATE =  1   COLUMN_TYPE =  5   ATOM = )
20   DEF FNB(Y) = Y**2 +2+Y+1
WRITE3 ATOM LINE_NUMBER    20
LINE =    20                    STATE =  1   COLUMN_TYPE =  5   ATOM = 20
LINE =    20                    STATE =  1   COLUMN_TYPE =  5   ATOM = 20
LINE =    20                    STATE =  1   COLUMN_TYPE =  1   ATOM = 20
LINE =    20                    STATE =  2   COLUMN_TYPE =  1   ATOM = D
LINE =    20                    STATE =  2   COLUMN_TYPE =  1   ATOM = DE
LINE =    20                    STATE =  2   COLUMN_TYPE =  5   ATOM = DEF
WRITE2 ATOM DEF
LINE =    20                    STATE =  1   COLUMN_TYPE =  5   ATOM = DEF
LINE =    20                    STATE =  1   COLUMN_TYPE =  1   ATOM = DEF
LINE =    20                    STATE =  2   COLUMN_TYPE =  1   ATOM = F
LINE =    20                    STATE =  2   COLUMN_TYPE =  1   ATOM = FN
LINE =    20                    STATE =  2   COLUMN_TYPE =  4   ATOM = FNB
WRITE0 ATOM IDENTIFIER      FNB
LINE =    20                    STATE =  1   COLUMN_TYPE =  4   ATOM = FNB
LINE =    20                    STATE =  5   COLUMN_TYPE =  1   ATOM = (
WRITE2 ATOM LEFT_PAREN
LINE =    20                    STATE =  1   COLUMN_TYPE =  1   ATOM = (
LINE =    20                    STATE =  2   COLUMN_TYPE =  4   ATOM = Y
WRITE0 ATOM IDENTIFIER      Y
LINE =    20                    STATE =  1   COLUMN_TYPE =  4   ATOM = Y
LINE =    20                    STATE =  5   COLUMN_TYPE =  5   ATOM = )
WRITE2 ATOM RIGHT_PAREN
LINE =    20                    STATE =  1   COLUMN_TYPE =  5   ATOM = )
LINE =    20                    STATE =  1   COLUMN_TYPE =  3   ATOM = )
LINE =    20                    STATE =  4   COLUMN_TYPE =  5   ATOM = =
WRITE2 ATOM EQUAL
LINE =    20                    STATE =  1   COLUMN_TYPE =  5   ATOM = =
LINE =    20                    STATE =  1   COLUMN_TYPE =  1   ATOM = =
LINE =    20                    STATE =  2   COLUMN_TYPE =  3   ATOM = Y
WRITE0 ATOM IDENTIFIER      Y
LINE =    20                    STATE =  1   COLUMN_TYPE =  3   ATOM = Y
LINE =    20                    STATE =  4   COLUMN_TYPE =  3   ATOM = *
WRITE2 ATOM MULTIPLY
LINE =    20                    STATE =  1   COLUMN_TYPE =  3   ATOM = *
LINE =    20                    STATE =  4   COLUMN_TYPE =  2   ATOM = *
WRITE2 ATOM MULTIPLY
LINE =    20                    STATE =  1   COLUMN_TYPE =  2   ATOM = *
LINE =    20                    STATE =  3   COLUMN_TYPE =  5   ATOM = 2
WRITE0 ATOM CONSTANT        2
LINE =    20                    STATE =  1   COLUMN_TYPE =  5   ATOM = 2
LINE =    20                    STATE =  1   COLUMN_TYPE =  3   ATOM = 2
LINE =    20                    STATE =  4   COLUMN_TYPE =  2   ATOM = +
```

APPENDIX VII

TRACE LEVEL 2 OF PARSER

```
PREFIX_SIGN
ATOM_IS
ATOM_IS
ATOM_IS
NEXTATOM                    CONSTANT
        ADD                   T$13          1          T$14
        EXIT                  T$14
FLUSH                       LINE_NUMBER
        GLNNO                 40
NEXTATOM                    LINE_NUMBER
ATOM_IS
ATOM_IS
PROGRAM_TAIL
PARSING_MAIN
PARSING_CONTROLLER
ATOM_IS
ATOM_IS
ATOM_IS
ATOM_IS
NEXTATOM                    LET
LET_STATEMENT
GOAL
NEXTATOM                    IDENTIFIER
ATOM_IS
NEXTATOM                    EQUAL
EXPRESSION
TERM
PRIORITY
PREFIX_SIGN
ATOM_IS
ATOM_IS
ATOM_IS
NEXTATOM                    IDENTIFIER
ATOM_IS
NEXTATOM                    LEFT_PAREN
EXPRESSION
TERM
PRIORITY
PREFIX_SIGN
ATOM_IS
ATOM_IS
ATOM_IS
NEXTATOM                    CONSTANT
ATOM_IS
NEXTATOM                    RIGHT_PAREN
        CALL                  10            2          T$15
ATOM_IS
NEXTATOM                    PLUS
TERM
PRIORITY
PREFIX_SIGN
ATOM_IS
ATOM_IS
ATOM_IS
NEXTATOM                    IDENTIFIER
ATOM_IS
NEXTATOM                    LEFT_PAREN
EXPRESSION
TERM
PRIORITY
```

APPENDIX VIII

TRACE LEVEL 2 OF EXECUTE

```
SUBTRACT        FNA$DA          1               I$1
BEFORE      2.000000E+00    1.000000E+00    0.000000E+00
AFTER       2.000000E+00    1.000000E+00    1.000000E+00

ADD             FNA$DA          1               T$2
BEFORE      2.000000E+00    1.000000E+00    0.000000E+00
AFTER       2.000000E+00    1.000000E+00    3.000000E+00

DOTIMES         T$1             T$2             I$1
BEFORE      1.000000E+00    3.000000E+00    0.000000E+00
AFTER       1.000000E+00    3.000000E+00    3.000000E+00

EXIT            T$3

CALL            20              2.56            T$16

GLNNO           20

ENTER           FNB$DA

EXPONENTIATION  FNB$DA          2               T$4
BEFORE      2.559999E+00    2.000000E+00    0.000000E+00
AFTER       2.559999E+00    2.000000E+00    6.553593E+00

ADD             T$4             2               T$5
BEFORE      6.553593E+00    2.000000E+00    0.000000E+00
AFTER       6.553593E+00    2.000000E+00    8.553593E+00

ADD             T$5             FNB$DA          T$6
BEFORE      8.553593E+00    2.559999E+00    0.000000E+00
AFTER       8.553593E+00    2.559999E+00    1.111359E+01

ADD             T$6             1               T$7
BEFORE      1.111359E+01    1.000000E+00    0.000000E+00
AFTER       1.111359E+01    1.000000E+00    1.211359E+01

EXIT            T$7

ADD             T$15            T$16            T$17
BEFORE      3.000000E+00    1.211359E+01    0.000000E+00
AFTER       3.000000E+00    1.211359E+01    1.511359E+01

CALL            30              89.56           T$18

GLNNO           30

ENTER           FNC$DA

DOTIMES         FNC$DA          3               T$8
BEFORE      8.956000E+01    3.000000E+00    0.000000E+00
AFTER       8.956000E+01    3.000000E+00    2.686799E+02

DOTIMES         3               FNC$DA          T$9
BEFORE      3.000000E+00    8.956000E+01    0.000000E+00
AFTER       3.000000E+00    8.956000E+01    2.686799E+02

DODIVIDE        T$9             2               T$10
BEFORE      2.686799E+02    2.000000E+00    0.000000E+00
AFTER       2.686799E+02    2.000000E+00    1.343400E+02
```

APPENDIX IX

SOURCE AND EXECUTION LISTING OF TEST PROGRAM 1

```
                    DEPARTMENT OF COMPUTER SCIENCE
                    NORTH TEXAS STATE UNIVERSITY
                BASIC INTERPRETER SOURCE STATEMENT

                                                    DATE : 74/10/19
                                                    TIME : 16:50:52
??    S = 2, P = 2, G = 2;
100   REM TEST LETSTATEMENT
102   LET X =2.56+4.78/2+2.7**2
56    LET Y=---+-(4*(2/2*2.76/4))+(22.6*0.8-10.58)
58    LET Z=X/2+Y*X-656.52/4+6.98
59    LET L=X+Y+(X+Y)*(X-1)+Z-(X+Y+Z)+4.5**2-8.5*1.2
60    LET M=-(((X+Y)-(X-Y+Z-L)+(Z+L+L)-Y+X)/2-5)/2
909   LET N = 5+2+1-3-3*5+5**2/2
62    PRINT "VALUE OF X","VALUE OF Y","VALUE OF Z"
64    PRINT X,Y,Z
621   PRINT
500   PRINT
620   PRINT "VALUE OF L","VALUE OF M","VALUE OF N"
622   PRINT L,M,N
77    STOP
78    END
```

```
VALUE OF X          VALUE OF Y          VALUE OF Z
 1.223999E+01        1.025999E+01       -2.544780E+01

VALUE OF L          VALUE OF M          VALUE OF N
 2.629492E+02       -2.003368E+02        2.500000E+00
```

APPENDIX X

SOURCE AND EXECUTION LISTING OF TEST PROGRAM 2

```
                    DEPARTMENT OF COMPUTER SCIENCE
                    NORTH TEXAS STATE UNIVERSITY
                    BASIC INTERPRETER SOURCE STATEMENT
                                                    DATE : 74/10/19
                                                    TIME : 16:56:47

      ??   S = 2, P = 2, G = 2;
      100  REM TEST ONSTATEMENT AND READSTATEMENT
      188  DATA 1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,6,9,5,4,6,7,8,10
      500  READ X
      991  IF X = 10 THEN 250
      609  ON X THEN 10,20,30,40,50,60,70,80,90
      5    LET ZERO = ZERO + 1
      6    GOTO 500
      10   LET ONE = ONE + 1
      11   GOTO 500
      20   LET TWO = TWO + 1
      21   GOTO 500
      30   LET THREE = THREE + 1
      31   GOTO 500
      40   LET FOUR = FOUR + 1
      41   GOTO 500
      50   LET FIVE = FIVE + 1
      51   GOTO 500
      60   LET SIX = SIX + 1
      61   GOTO 500
      70   LET SEVEN = SEVEN + 1
      71   GOTO 500
      80   LET EIGHT = EIGHT + 1
      81   GOTO 500
      90   LET NINE = NINE + 1
      91   GOTO 500
      250  PRINT |# OF ZERO|,|# OF ONE|,|# OF TWO|,|# OF THREE|,|# OF FOUR|
      251  PRINT ZERO,ONE,TWO,THREE,FOUR
      252  PRINT
      888  PRINT |# OF FIVE|,|# OF SIX|,|# OF SEVEN|,|# OF EIGHT|,|# OF NINE|
      300  PRINT FIVE,SIX,SEVEN,EIGHT,NINE
      301  STOP
      400  END
```

```
# OF ZERO          # OF ONE           # OF TWO           # OF THREE
 2.000000E+00      3.000000E+00       2.000000E+00       2.000000E+00

# OF FIVE          # OF SIX           # OF SEVEN         # OF EIGHT
 3.000000E+00      4.000000E+00       3.000000E+00       3.000000E+00


# OF FOUR
 3.000000E+00

# OF NINE
 3.000000E+00
```

APPENDIX XI

SOURCE AND EXECUTION LISTING OF TEST PROGRAM 3

```
                    DEPARTMENT OF COMPUTER SCIENCE
                    NORTH TEXAS STATE UNIVERSITY
                    BASIC INTERPRETER SOURCE STATEMENT

                                                        DATE : 74/10/19
                                                        TIME : 16:58:12
1       77    S = 2, P = 2, G = 2;
        100   REM TEST IFSTATEMENT AND GOTOSTATEMENT
3       25    LET X = 4*2/2-5+7+10-2**2
4       26    LET Y = 5-4+(6-7*2**2/2)+20*2
5       27    IF X <> Y THEN 15
6       28    PRINT !HAVE EXECUTED 28!
7       30    IF Y>X THEN 17
8       15    PRINT !HAVE EXECUTED 27!
9       16    GOTO 28
10      17    PRINT !HAVE EXECUTED 30!
11      300   IF X<Y THEN 40
12      38    PRINT !HAVE EXECUTED 41!
13      39    IF Y>=X THEN 50
14      40    PRINT !HAVE EXECUTED 300!
15      41    IF X<=Y THEN 38
16      50    PRINT !HAVE EXECUTED 39!
17      52    IF X*2+Y**2 = X*2+Y**2 THEN 60
18      54    PRINT !HAVE EXECUTED 62!
19      55    STOP
20      60    PRINT !HAVE EXECUTED 52!
21      62    GOTO 54
22      80    END
```

```
HAVE EXECUTED 27
HAVE EXECUTED 28
HAVE EXECUTED 30
HAVE EXECUTED 300
HAVE EXECUTED 41
HAVE EXECUTED 39
HAVE EXECUTED 52
HAVE EXECUTED 62
```

APPENDIX XII

SOURCE AND EXECUTION LISTING OF TEST PROGRAM 4

```
                    DEPARTMENT OF COMPUTER SCIENCE
                    NORTH TEXAS STATE UNIVERSITY
                    BASIC INTERPRETER SOURCE STATEMENT

                                                        DATE : 74/10
                                                        TIME : 17:01:
001       ??   S = 2, P = 2, G = 2;
002       100  REM TEST GOSUBSTATEMENT AND RETURNSTATEMENT
003       104  DATA 44.56,55.87,100.28
004       50   GOSUB 400
005       54   PRINT !VALUE OF X!,!VALUE OF Y!, !VALUE OF Z!
006       56   PRINT X,Y,Z
007       60   STOP
008       400  READ X
009       401  GOSUB 500
010       402  RETURN
011       500  READ Y
012       501  GOSUB 600
013       504  RETURN
014       600  READ Z
015       605  RETURN
016       700  END
```

```
 VALUE OF X          VALUE OF Y          VALUE OF Z
 4.456000E+01        5.587000E+01        1.002800E+02
```

APPENDIX XIII

SOURCE AND EXECUTION LISTING FOR TEST PROGRAM 5

```
                    DEPARTMENT OF COMPUTER SCIENCE
                    NORTH TEXAS STATE UNIVERSITY
                    BASIC INTERPRETER SOURCE STATEMENT

 77   S = 1, F = 1, G = 1:
 100  REM TEST DEFSTATEMENT                                   DATE : 74/10/19
 10   DEF FNA(X) = (X-1)* (X+1)                               TIME : 20:28:24
 20   DEF FNB(Y) = Y**2 +2+Y+1
 30   DEF FNC(Z) = Z*3 + 3*Z/2 +2*Z+1
 40   LET L = FNA(2)+FNB(2.56)+FNC(89.56)
 50   LET M = FNA(1.63)*FNB(20.96)*FNC(56.85)
 800  LET N = FNA(FNB(FNC(2))) + FNB(FNA(FNC(2)))
 750  PRINT |VALUE OF L|,|VALUE OF M|,|VALUE OF N|
 740  PRINT L,M,N
 102  PRINT
 801  PRINT |VALUE OF FNA(2.1)|,|VALUE OF FNB(3)|,|VALUE OF FNC(2)|
 802  PRINT FNA(2.1),FNB(3),FNC(2)
 900  PRINT
 901  LET X1 = 4.5
 902  LET Y1 = 5.7
 910  PRINT |VALUE OF X1+Y1|,|VALUE OF X1-Y1|,|VALUE OF X1*Y1|
 920  PRINT X1+Y1,X1-Y1,X1*Y1
 730  STOP
 710  END
```

```
VALUE OF L              VALUE OF M              VALUE OF N
 5.962532E+02            2.844179E+05            8.359094E+04

VALUE OF FNA(2.1)       VALUE OF FNB(3)         VALUE OF FNC(2)
 3.400557E+00            1.499999E+01            1.400000E+01

VALUE OF X1+Y1          VALUE OF X1-Y1          VALUE OF X1*Y1
 1.020000E+01           -1.200000E+00            2.564999E+01
```

APPENDIX XIV

SOURCE AND EXECUTION LISTING FOR TEST PROGRAM 6

```
                    DEPARTMENT OF COMPUTER SCIENCE
                     NORTH TEXAS STATE UNIVERSITY
                  BASIC INTERPRETER SOURCE STATEMENT   DATE : 74/10/19
                                                       TIME : 20:32:56

      ??   S = 1, P = 1, G = 1;
      100  REM TEST FORSTATEMENT,NEXTSTATEMENT,READSTATEMENT,DATASTATEMENT
      900  DIM A(2,2),B(2,2),C(2,2)
      110  DATA 2,3,4,5,2.2,3.5,4.6,5.7
      101  READ A(1,1),A(1,2),A(2,1),A(2,2),B(1,1),B(1,2),B(2,1),B(2,2)
      102  GOSUB 104
      103  STOP
      104  FOR I = 1 TO 2
      105  FOR J = 1 TO 2
      957  LET C(I,J) = 0
      106  FOR K = 1 TO 2
      107  LET C(I,J) = C(I,J) + A(I,K) *B(K,J)
      108  NEXT K
      109  NEXT J
      050  NEXT I
      051  GOSUB 025
      090  RETURN
      024  REM PRINT MATRIX C
      025  PRINT
      027  PRINT TAB(1);C(1,1);TAB(15);C(1,2)
      028  PRINT C(1,1),C(1,2)
      029  PRINT
      035  PRINT TAB(1);C(2,1);TAB(15);C(2,2)
      040  PRINT C(2,1),C(2,2)
      015  RETURN
      044  END
```

```
C(1,1)              C(1,2)
 1.82CC00E+01        2.409999E+01

C(2,1)              C(2,2)
 3.179997E+01        4.249998E+01
```

BIBLIOGRAPHY

Books

Abramson, Harvey, Theory and Application of a Bottom-Up Syntax-Directed Translator, New York, Academic Press Inc., 1973.

Donovan, John J., Systems Programming, New York, McGraw-Hill Book Co., 1972.

Gauthier, Ponto, Designing Systems Programs, New Jersey, Prentice-Hall, Inc., 1970.

Gries, David, Compiler Construction for Digital Computers, New York, John Wiley & Sons, Inc., 1971.

International Business Machines, IBM Systems 360 Operating Systems: Job Control Language Reference, Form No. GC28-6704-2.

International Business Machines, IBM System 360 PL/1 Reference Manual, Form No. C28-8201-0.

Naur, Peter and others, "Revised Report on the Algorithmic Language. Algol 60," Programming Systems and Languages, edited by Saul Rosen, New York, McGraw-Hill Book Company, 1967.

Smith, Robert E., Discovering BASIC, New York, International Timesharing Corporation, 1970.

Spencer, D.D., A Guide to BASIC Programming: A Time-Sharing Language, the United State of America, Addison-Wesley Publishing Company, Inc., 1970.

Article

Broadbent, J.K., "Microprogramming and System Architecture," The Computer Journal (Volume 17 Number 1), 1973.

Unpublished Material

Chang, Min-Jey S., "An Interpreter for the BASIC Programming Language," Department of Computer Sciences, North Texas State University, Denton, Texas 1974.

Isaacson, Portia, A Compiler for this Programming Language, Department of Computer Sciences, North Texas State University, Denton, Texas 1971.