AN EXTENDED BASIC COMPILER WITH GRAPHICS INTERFACE FOR THE PDP---ETC(U)
JUN 77 M D ROBERTSON

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
<th>Column 6</th>
<th>Column 7</th>
<th>Column 8</th>
<th>Column 9</th>
<th>Column 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td>Row 2</td>
<td>Row 3</td>
<td>Row 4</td>
<td>Row 5</td>
<td>Row 6</td>
<td>Row 7</td>
<td>Row 8</td>
<td>Row 9</td>
<td>Row 10</td>
</tr>
<tr>
<td>Row 11</td>
<td>Row 12</td>
<td>Row 13</td>
<td>Row 14</td>
<td>Row 15</td>
<td>Row 16</td>
<td>Row 17</td>
<td>Row 18</td>
<td>Row 19</td>
<td>Row 20</td>
</tr>
<tr>
<td>Row 21</td>
<td>Row 22</td>
<td>Row 23</td>
<td>Row 24</td>
<td>Row 25</td>
<td>Row 26</td>
<td>Row 27</td>
<td>Row 28</td>
<td>Row 29</td>
<td>Row 30</td>
</tr>
<tr>
<td>Row 31</td>
<td>Row 32</td>
<td>Row 33</td>
<td>Row 34</td>
<td>Row 35</td>
<td>Row 36</td>
<td>Row 37</td>
<td>Row 38</td>
<td>Row 39</td>
<td>Row 40</td>
</tr>
<tr>
<td>Row 41</td>
<td>Row 42</td>
<td>Row 43</td>
<td>Row 44</td>
<td>Row 45</td>
<td>Row 46</td>
<td>Row 47</td>
<td>Row 48</td>
<td>Row 49</td>
<td>Row 50</td>
</tr>
<tr>
<td>Row 51</td>
<td>Row 52</td>
<td>Row 53</td>
<td>Row 54</td>
<td>Row 55</td>
<td>Row 56</td>
<td>Row 57</td>
<td>Row 58</td>
<td>Row 59</td>
<td>Row 60</td>
</tr>
<tr>
<td>Row 61</td>
<td>Row 62</td>
<td>Row 63</td>
<td>Row 64</td>
<td>Row 65</td>
<td>Row 66</td>
<td>Row 67</td>
<td>Row 68</td>
<td>Row 69</td>
<td>Row 70</td>
</tr>
<tr>
<td>Row 71</td>
<td>Row 72</td>
<td>Row 73</td>
<td>Row 74</td>
<td>Row 75</td>
<td>Row 76</td>
<td>Row 77</td>
<td>Row 78</td>
<td>Row 79</td>
<td>Row 80</td>
</tr>
<tr>
<td>Row 81</td>
<td>Row 82</td>
<td>Row 83</td>
<td>Row 84</td>
<td>Row 85</td>
<td>Row 86</td>
<td>Row 87</td>
<td>Row 88</td>
<td>Row 89</td>
<td>Row 90</td>
</tr>
<tr>
<td>Row 91</td>
<td>Row 92</td>
<td>Row 93</td>
<td>Row 94</td>
<td>Row 95</td>
<td>Row 96</td>
<td>Row 97</td>
<td>Row 98</td>
<td>Row 99</td>
<td>Row 100</td>
</tr>
</tbody>
</table>

UNCLASSIFIED

F/G 9/2

NAVAL POSTGRADUATE SCHOOL MONTEREY CALIF
THESIS

An Extended Basic Compiler with Graphics Interface for the PDP-11/50 Computer

by

Michael David Robertson

June 1977

Approved for public release; distribution unlimited.
An Extended Basic Compiler with Graphics Interface for the PDP-11/50 Computer.

Michael David Robertson

Naval Postgraduate School
Monterey, California 93940

June 1977

Naval Postgraduate School
Monterey, California 93940

Unclassified

Approved for public release; distribution unlimited.

Programming language
Compiler
Graphics

The design and implementation of an extension to the Basic programming language for use on the PDP-11/50 computer system has been described. The implementation consists of a compiler which generates code to be assembled and loaded into the computer system. An interface with C programs in the system library, which allows extended Basic to perform as an extensive graphics language, has been discussed. The design goals,
solutions, and recommendations for further expansion of the system have been presented. The compiler was implemented in
the C-programming language with the UNIX operating system as supported by the PDP-11/50 at the Naval Postgraduate School
Computer Laboratory.
Approved for public release; distribution unlimited

An Extended Basic Compiler with Graphics Interface for the PDP-11/50 Computer

by

Michael David Robertson
Lieutenant, United States Navy
B.S., University of Oklahoma, 1972

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1977

Author

Approved by:

Zygl V. Rich
Thesis Advisor

Larry M. Reetz
Second Reader

Chairman, Department of Computer Science

Dean of Information and Policy Sciences
ABSTRACT

The design and implementation of an extension to the Basic programming language for use on the PDP-11/50 computer system has been described. The implementation consists of a compiler which generates code to be assembled and loaded into the computer system. An interface with C programs in the system library, which allows extended Basic to perform as an extensive graphics language, has been discussed. The design goals, solutions, and recommendations for further expansion of the system have been presented. The compiler was implemented in the C-programming language with the UNIX operating system as supported by the PDP-11/50 at the Naval Postgraduate School Computer Laboratory.
1. Constant List ................................................. 35
1. External Files ............................................... 37
2. Parser .......................................................... 38
3. Code Generation ............................................... 38
   a. Extended Basic Language Structure ..................... 39
   b. Assignment Statements and Expressions ............. 40
   c. Control Statements ...................................... 42
   d. Declaration Statements .................................. 44
   e. Inout/Output Statements ................................. 45
   f. External Statements ..................................... 46

I. RECOMMENDATIONS FOR FUTURE DEVELOPMENTS .......... 49

II. CONCLUSIONS .................................................... 51

APPENDIX I - EXTENDED BASIC LANGUAGE MANUAL ............ 52
APPENDIX II - OPERATING IN UNIX WITH EXTENDED BASIC .... 113
PROGRAM LISTING - EXTENDED BASIC COMPILER .............. 116
PROGRAM LISTING - UNIX EXECUTIVE PROGRAM ............... 204
BIBLIOGRAPHY ..................................................... 209
INITIAL DISTRIBUTION LIST .................................... 210
ACKNOWLEDGEMENTS

The extensive work leading to the successful culmination of this thesis began with a class project for the students of Computer Science section CS52 while taking course CS 4113, Compilers. For their pioneering spirit in the inception of the Basic compiler, I am deeply indebted. LT Lyle Rich, their instructor, continued to work with the Basic compiler and is largely responsible for its successful implementation. Without his guidance and considerable knowledge of the POP-11 Computer System, this thesis would not be possible.
I. INTRODUCTION

A. HISTORY OF THE BASIC LANGUAGE

The Beginner’s All-Purpose Symbolic Instruction Code (BASIC) was developed at Dartmouth College to provide a simple, interactive language for liberal arts students with specific applications in scientific computation. In order to meet this goal, only a limited vocabulary of instructions was included in the original definition of Basic. There was no concept of data typing and there were no default conditions to memorize. The interactive nature of programming provided an ideal man/machine interface for creating and debugging programs, while the features of the language were well-suited for the expression of engineering and mathematics problems. Since this environment satisfied the needs of a wide range of potential computer users, Basic was procured for adaptation by a number of universities and commercial firms. In particular, timesharing service bureaus expanded computer usage among non-computer specialists by providing its customers with the Basic language. This led to the development of numerous dialects of Basic and to many extensions intended to satisfy the unique needs of various users [1].
As the use of Basic increased and extensions to the language became more widespread, the need for standardization became an industry wide concern. In 1974, this concern finally led to the formation of the X3J2 committee of the American National Standards Institute which was tasked with formulating a proposed standard for the Basic programming language. The result of an extensive effort was the Proposed American National Standards Institute (ANSI) report on a proposed standard for Minimal Basic [2]. The proposed standard established a minimum set of features which should be included in the implementation of a Basic language processor. While the proposed standard provided arithmetic and very simple string processing capabilities, it did not consider the more extensive features, i.e. multi-program interfacing and extensive predefined functions, which had initially led to the need for standardization. In a recent article [3], Lientz compared the different commercially available Basic language processors. This survey indicated that most Basic processors provided similar features and included extensive facilities beyond those in the proposed ANSI standard.

B. OBJECTIVES OF THE EXTENDED BASIC LANGUAGE

Extended Basic was designed to provide all the arithmetic processing features of the proposed standard for Basic as well as extensions and enhancements to the language for use at the Naval Postgraduate School. These extensions included
multi-dimensional arrays, logical operators for numeric and string quantities, string manipulation, and sequential access to external files. Further, extended Basic retained the original concepts of Dartmouth Basic while freeing the programmer from many of the original limitations. Enhancements included improved control structures and features to enhance increased readability. Extended Basic also attempted to maintain grammatical compatibility with existing extensions to Basic, particularly those in use at the Naval Postgraduate School.

An additional goal of extended Basic was to provide non-computer scientists with a more manageable high level language capable of interfacing with other subsystems supported on the PDP-11 at Naval Postgraduate School. Examples of such subsystems are the procedures which drive the various graphics devices found in the computer laboratory. The primary UNIX system graphics language is C [11] which provides support for the subsystems in the PDP-11.

Currently included within UNIX are a dialect of Fortran [12], the Fortran preprocessor [13] RAIFOR, an interpreter for a highly specialized dialect of Basic [14], produced by Bell Laboratories [4], Digital Equipment Corporation's FORTRAN IV PLUS, and the UNIX assembler [7]. None of these languages were entirely suited to this special graphics environment as they existed in the system. Extended Basic is an easily learned language which is readily adaptable to the
student environment and enhances the graphics capabilities in the laboratory.

Unlike many existing implementations, extended Basic was not implemented as a purely interpretive language. A source program is compiled, generating an assembly language file. This code is then assembled and loaded with the Basic library, and other libraries as specified by the user, including the C library, the various graphics device libraries, and any user designed libraries which may exist for particular implementations. The compilation, assembly and loading actions are called by a program, LBAX, which is resident in the UNIX system. Usage of the program is described in Appendix II.
II. LANGUAGE SPECIFICATION

In the following section, the Dartmouth Basic language and the ANSI proposed standard will be reviewed, followed by a discussion of the features of extended basic which differ from Dartmouth Basic and the proposed ANSI standard. These features include extended arithmetic processing, improved readability, expanded control structures, string manipulation, external file access, and program access to system software for graphics interface.

A. THE PROPOSED STANDARD FOR BASIC

1. Dartmouth Basic

Dartmouth Basic is a statement oriented language. Each statement consists of a line number and a command. Data is either numeric real or character string with no distinction being made between types of numeric data. Identifiers terminated by a dollar sign refer to string variables, while all other identifiers reference numeric quantities. Identifiers consist of only a single letter or a letter followed by a dollar sign. Arithmetic operations, defined on numeric data only, are represented by the infix operators +, -, *, /, and ^ (exponentiation). Unary operations are defined by the prefix operators + and -. Both data types may
be compared using the infix relational operators <, <=, >, >=, and <>.

One and two dimensional numeric arrays are supported. Finally, a limited number of predefined algorithms perform elementary function evaluation [5]. These include ABS, ATN, COS, EXP, INT, LOG, RND, SGN, SIN, SQR, and TAN.

A complete description of these predefined functions is presented in Appendix I.

Dartmouth Basic is intended to be an interactive language with both editing and program execution occurring in the same environment. Therefore, most Dartmouth style Basic implementations rely on line numbers to play an important part in the editing function of Basic.

2. The Proposed ANSI Standard

The proposed ANSI standard [2] incorporates all the features of Dartmouth Basic and adds the following statements:

```
ON RANDOMIZE DEF
OPTIONS STOP
```

With the exception of the OPTION statement, most existing Basic implementations include all of these additional features. These extensions are described as they exist in this implementation in Appendix I. The OPTION statement is used to specify whether the lower bound of an array is zero or one.
Most existing Basic language processors go well beyond the proposed ANSI standard to provide file-handling ability, formatted output, string manipulation, matrix operations, and a multitude of predefined functions. The survey by Lientz (3) documents these extensions for many large and mini-computer manufacturers, and for a number of timesharing services.

B. FEATURES OF THE EXTENDED BASIC LANGUAGE

Extended Basic was designed to maintain compatibility with the proposed ANSI standard while extending the language to incorporate such features as string processing and external file access. Enhancements were also included to provide additional control structures and increased readability. In this section the features of extended Basic which do not appear in the proposed ANSI standard will be discussed. Appendix I includes a complete description of the language.

1. Arithmetic Processing

Extended Basic adds to arithmetic processing by supporting multiple dimensional arrays. All arrays must be dimensioned prior to usage in the program and the same identifier may not serve as both an array, whose elements are subscripted, and a simple non-subscripted variable. Logical binary operators AND, OR, XOR (exclusive or), and the unary operator NOT are provided for the logical evaluation of numeric and string expressions. The relational operators
and \( \neq \) (not equal) have been added to the set of logical operators for compatibility with existing languages. User-defined functions, defined using a DEF statement, may have any number of parameters. However, as with FORTRAN, every function must have at least one parameter. Functions must be defined prior to appearance. While functions may refer to other functions within the body of the definition, recursive references are not permitted.

The OPTION statement is not implemented. Since the lower bound of every array is always zero and there are \( n+1 \) elements allocated by the compiler for every array, the user is provided the OPTION feature by default. Due to the manner in which the UNIX system effects external system calls, undimensioned subscripted variables should not be used, as is conditionally allowed in Dartmouth Basic and the proposed ANSI standard.

Arithmetic constants may be written in either integer or decimal form. All constants are viewed internally as double precision floating point numbers. Scientific notation is not implemented. Numeric constants are output in decimal form only. The columnar width of numeric output may be specified using the COL function. If columnar width is not specified, COL defaults to 10 columns. If the value exceeds the prescribed width, the field is filled with a string of question marks.
2. Readability

Readability has been improved by increasing variable name length, permitting free form input with statement continuation, and by not requiring line numbers on all statements in the program. Historically, Basic permitted variable names consisting of a single letter or a letter followed by a number. This makes large programs difficult to understand and debug. Extended Basic allows variable names to consist of up to four alpha-numeric characters of both upper and lower case, except string variables which should include '§' in the second or third character position. Predefined functions may be written in upper or lower case; however, all characters in the name must be of the same case.

Basic traditionally has restricted each statement to one line. Extended Basic provides the "at" sign ( @ ) as a continuation character, allowing multiple program lines to appear as one statement to the compiler. This is particularly valuable when using nested IF statements with the ELSE clause followed by another IF statement. All of the members of the primary IF statement could not be physically contained on one line on conventional timesharing input/output devices. The following example demonstrates the improved readability provided by continuation:
if $x = y$ then $z$

$z = x(i, i) \theta$

else $\varepsilon$

if $x > y$ then $\theta$

$z = w(i, i) - x \varepsilon$

else $\varepsilon$

$z = w(i, i) - y$

Both Dartmouth and the proposed ANSI Basic include mandatory statement labeling because of the interactive editing feature of Basic. Extended Basic does not use internal interactive editing and subsequent program execution. Changes are made to the program source code, using the UNIX text editor and subsequently recompiling the program. Thus line labels are only necessary for use in control structures. Examples of limited line labeling are found in the example programs at the end of this section.

The TAB function has not been implemented. The use of commas and semicolons to force columnation is not effective. Partial consistency with the proposed standard has been maintained by providing a continuation flag for output. When a semicolon appears at the end of a print statement, newline is not invoked, and the next output from a print statement will immediately follow the existing output.
3. Control Structures

Extended Basic has expanded the control structures included in standard Basic. These structures consist of the FOR, IF, GOTO, GOSUB, ON, STOP and RANDOMIZE statements. Extended Basic significantly increases the power of the IF statement by providing an optional ELSE clause and by allowing an executable statement to follow the THEN and the ELSE. An executable statement is further defined in Appendix I. Any such executable statement may be used within an IF statement. Additionally, the IF statement, which is classified as a simple statement, may be used in the same manner as an executable statement in the ELSE clause. Thus IF statements may be nested to an infinite depth; however, only one executable statement may exist at the deepest level.

4. String Processing

Extended Basic contains features which provide for general string manipulation. Strings are created dynamically, may vary in length to a maximum of 255 characters, and may be subscripted to one dimension to create a vector of strings. The predefined function LEN returns the current length of a string. All string variables and string array elements are initialized as null strings with a length of zero. Strings may be created and associated with a variable using the replacement operator (=), an INPUT statement, or a READ statement. A string entered from the console or read from an external file may not be enclosed in quotation marks.
marks, but should be delimited by newlines. A string entered from the console or redirected by system editing through an external file may be terminated by a quotation mark or the newline symbol, '\n', which is equivalent to the ASCII line feed control character. Strings appearing in a data statement within the program must be enclosed in quotation marks since they form an integral part of the program. An additional feature of extended Basic allows comparison of string variables and extraction of substring segments.

Strings are compared using the same relational operators used for numeric data. Two strings are equal if and only if the strings have the same length and contain identical characters.

Substring extraction is accomplished using substring notation, i.e. A$(m;n). This expression returns the sub-string of string variable A$ beginning at character position m and extending for a length of n characters.

Other predefined functions are provided to facilitate processing strings. The CHR$ function converts a numeric argument into a single ASCII character while ASC converts the first character of a string argument into a numeric value.

5. Files

Data may be transferred between an extended Basic program and external storage using the file processing
feature. The OPEN statement identifies files and prepares them for access. The general form of an OPEN statement is:

```
OPEN (<external file number>,<access mode>) <file name>
```

where the <file name> is a character string, which is called a pathname in the UNIX hierarchical file system. If a file exists in the external file system with the name represented by the pathname, then that file is opened. Otherwise, a file is created with that name provided the <access mode> specifies writing. Each file currently in use is assigned a unique <external file number> by the programmer. This file number is used for all further references to the file while it remains open for access. Data is transmitted between the external file and the extended Basic program using the READ and PRINT statements with the <file option>:

```
READ # <file option>; <read list>
PRINT # <file option>; <expression list>
```

The <file option> specifies the file desired by referencing the <external file number> defined by a preceding OPEN statement. Access to a file may be terminated by the CLOSE statement. End-of-file may be determined with an IF END statement which has the following form:

```
IF END # <external file number> THEN <valid statement>
```

The <valid statement> may be any statement or expression which is permissible with a standard IF statement.
b. Standard Input/Output

Standard input and output files are organized sequentially. The standard input file is a linear sequence of numeric and string data items separated by commas and newlines. Each reference to a sequential file retrieves the next data item with READ #, or writes another data item with PRINT #. With each READ, the variables in the read list are assigned values from the input. Line terminators are treated as record terminators. There is no concept of a traditional record since each record may be of indefinite length, limited only by the medium through which the record is created.

Likewise, with each PRINT command, values from the expression list are written to the file. The expressions are written to the standard output as ASCII strings separated by spaces except for the last data item in the list which is followed by a newline. The use of newlines in this manner allows files to be displayed using system utilities and also allows files created with a text editor to be read by extended Basic programs.

Since data type-checking is not accomplished, the sequence of item data types in the expression list should match the sequence of item data types in the external file. Mismatched data types will return undesirable values. Numeric data types reading string values will return a sequence of zeros. String data types reading numeric values
will return a string of numbers.

Data may be appended to external files by specifying the append access mode when an OPEN statement is used. This allows additional data items to be written at the end of the specified file. An OPEN specifying write access will create a new file if one does not already exist, or will reopen an existing file, overwriting and destroying any pre-existing data.

7. External Interface

This version of extended Basic was designed primarily to enhance user ability to program with a simplistic language which could interface with other subsystems available within the UNIX environment. This was accomplished by creating the EXTERN and CALL statements.

The EXTERN statement defines, within the Basic program, those existing external subroutines which will be used for any software implementation.

Examples of subroutines which may be used are POW and PRINTF [9]. POW returns the value of the variable x raised to the power of v, performing floating point exponentiation. PRINTF converts, formats and prints all arguments after the first argument, and under the control of the first argument.
These subroutines would be defined in a Basic program by:

```
extern pow(double, double)
extern printf(char, double, integer)
```

While these example procedures exist in the UNIX system library, it is not necessary to use only existing procedures. The user may create procedures for specific needs by writing and compiling unique procedures in the C language [11], and including the loadable version of the procedure as a parameter when the system compile command for Basic, LBAX, is issued.

Once a procedure has been defined as external, it may be used in the Basic program by using the CALL statement. It would appear in the program as:

```
call pow(x, y)
call printf(a$, sum, prod)
```

Examples of programs using the EXTERN and CALL statements are provided in the next section.

C. EXAMPLE PROGRAMS

1. Quadratic Factors

This example program computes the factors of a quadratic equation.
rem quad factors of 6th degree polynomial, Bairstow method

dim a(9), b(9), c(9)
data 0, 1, -17, 8, 99.41, -251.218
data 352, 611, -134.109
data 0, 0, 0.00001, 20.5
print "Demonstration program output"
for i = 3 to 9
read a(i)
next i
read rl, sl, test, lim, n
print "The original polynomial = "
print "Power of x Coefficient"
j = 9 - n
for i = j to 9
m = 9 - i
print "", a(i)
next i
print "The quadratic factors are = "
a(1) = 0
b(2) = 0
c(1) = 0
c(2) = 0
r = rl
s = sl
5 knt = 1
6 for j = 3 to 9
\[ b(j) = a(j) + r*b(j-1) + s*b(j-2) \]
\[ c(j) = b(j) + r*c(j-1) + s*c(j-2) \]
next j
dnm = c(7)*c(2) - c(8)*c(6)
if dnm = 0 goto 1
r1 = r1 + 1
s1 = s1 + 1
1 delr = (-b(8)*c(7)+c(6)*b(9))/dnm
dels = (-c(7)*b(9)+b(8)*c(8))/dnm
r = delr + r
s = dels + s
if (abs(delr) + abs(dels) - test) <= 0 goto 3
if (knt = lim) < 0 goto 2
print "Does not converge after ", lim, " iterations."
stop
2 knt = knt + 1
go to 6
3 print "x^2 + ", r, " x + ", s
n = n - 2
tval = n - 2
if tval < 0 then @
\[ print b(6), " x + ", b(7) \]
if tval = 0 then @
\[ print b(5), " x + " , b(6), " x + ", b(7) \]
if tval > 0 go to 4
stop
4 for k = 3 to 9
2. Magic Figures

This program draws random symmetric figures on the TEKTRONIX graphics device. It uses four externally defined graphics routines which are located in the TEKTRONIX library. They are NWPAG, ANMODE, INITT, and FINITT. NWPAG erases the screen and returns the alphanumeric cursor to the HOME position, the upper left hand corner of the screen. ANMODE sets the cursor to the alphanumeric mode. INITT requires one argument parameter specifying the character transmission rate between the computer and terminal to determine the delay to the screen when erasure is being performed. FINITT clears the buffers and moves the pointer to the position indicated by the two parameters.

The externally defined procedure PLOT moves the pointer to the x,y coordinates indicated by the arguments and plots a point at that location. The sixth externally defined routine is MOVE. These procedures are user defined, and are located in the user's external file area. MOVE causes the pointer to be moved across the screen without drawing on the surface.

extern newpagn()
extern anmode()
extern init(integer)
extern finit(integer, integer)
extern plot(integer, integer)
extern move(integer, integer)

print "welcome to Magic -- enter your two numbers"

100 input "number one "; fm
input "number two "; fm2

call init(960)
call newoag()
d=10

h
c=fm
z=0
i=0

3  h=rad(z-90)
x=cos(h)*d+512
y=sin(h)*d+380
if i<>0 go to 4
    call move(4*x,4*y)
    go to 5

4  call plot(4*x,4*y)

5  z=z+c
  c=(-1)*c*fm2
  fm2=1/fm2
  i=1
  d=d+fm/90
  a=a+fm
  if a <27500 go to 3
  call move(0,4*780)
call anmode()
go to 100

end
I. IMPLEMENTATION

A. SYSTEM DESIGN

The extended Basic compiler was designed around a table-driven parser which checks statements for correct syntax and generates assembly code written into a UNIX file. This code is assembled and loaded together with requested and required libraries, and other user defined program segments, by the assembler and loader when called by the executive program, LBAX, located in the system library.

The decision to compile the source program and then assemble the intermediate language was based on the following consideration: formal parsing techniques could be used to analyze the syntax of the source program making extensions to the language relatively easy. In this case, an LALR parser-generator YACC [6], was used to automatically generate the parse tables for the language.

The following sections discuss the design of the extended Basic compiler and the implementation of the system executive program. Source listings of the programs are contained in the Program Listing section of this thesis.
B. COMPILER STRUCTURE

1. Compiler Organization

The compiler structure requires one pass through the source program to produce an intermediate assembly language file. This pass writes all numeric constants to the numeric constant list, determines the size of the symbol table and inserts symbols with associated attributes, outputs intermediate level code to a file based upon parse actions and semantics, resolves external calls and produces the code for access to external files.

The intermediate level code is the UNIX assembly language. The formatted output program, to be loaded and executed, is in the proper format for an assembly program. The format consists of text, data, and bss segments [7].

The text segment contains all the executable instructions and unmodified data. The data segment may contain text, but always contains initialized data which may be modified during execution. The bss segment contains uninitialized data areas and is an extension of the data segment.

The data segment contains the buffers for external file manipulation as illustrated in Figure 1. The number of buffers may not exceed fifteen and is determined by the OPEN actions in the parser. The length of each buffer is 513 bytes, six of which are utilized by the system Input/Output commands and 512 of which contain the string of data.
Compiler Output
Memory Map

- counters and uninitialized symbols
- stack pointer
- stack
- i/o buffers for external files
- initialized symbols
- string data
- numeric data
- instructions

bss

data

text

Figure 1
The Basic run-time "stack" is established in the bss segment by the compiler and is fifty bytes in length. It uses the "last in, first out" concept and grows downward toward the data segment.

1. Scanner

The scanner analyzes the source program, returning a sequence of tokens to the parser. In addition, the scanner processes data statements and recognizes continuation characters. Analysis of the first non-blank character in the input stream determines the general class of the next token. The remainder of the token is then scanned, placing each successive character into one of the accumulator vectors, ID or NUMSTR, used for identifier and numeric items respectively.

If the scanner recognizes an identifier, it searches the reserved word list to determine if the identifier is a reserved word. If found, the token associated with that reserved word is returned to the parser.

In the event the token is not a reserved word, it is validated from the symbol table returning an error code, if not defined, or the symbol table location index number, if defined. In order to be a valid member of the symbol table, an identifier must be a numeric-identifier, string-identifier, function-identifier, array identifier, or built-in function. Whenever a symbol not defined in the
symbol table is encountered, it is verified to be a proper identifier, occurring in a valid position in the input string, and is then inserted into the symbol table.

If the scanner recognizes a token as a numeric constant, the number list is searched to determine if the number is already stored. If the number is not an element of the list, it is inserted into the literal numbers table with its appropriate identifying attributes.

2. Symbol Table

The symbol table contains attributes of program and compiler generated entities such as identifiers and function names. The information stored in the symbol table is created and referenced by the compiler to verify that the program is semantically correct and to assist in code generation. Access to the symbol table is provided through a number of procedures operating on the globally defined symbol table variables.

The symbol table is a C language structure as illustrated in Figure 2. It may contain up to 200 individual elements which are accessed as members of an array, or may be identified by the attributes stored in each structure element vector.

The final elements of the symbol table contain the names of the built-in (or predefined) functions. The symbol table grows downward with subsequent symbols preceding
Symbol Table Structure

structure "symtable"

attribute

Figure 2
the built-in function symbol names. Individual elements of
the symbol table are located by any of a number of attrib-
utes as illustrated in Figure 2. Each entry in the symbol
array refers to a structure consisting of six elements.
Symbols may be selected based upon the entries in any one of
the elements or any combination of elements.

The attributes of a symbol are:

Symbol. The null terminated string of characters representing the symbol.

Type. A numeric value which characterizes a symbol (-1 through 10)

- the null parameters of external variables
- a numeric identifier
- a numeric array
- a string identifier
- a string array
- a programmer defined function
- a numeric built-in function
- a string built-in function
- a simple format
- a numeric format
- a numeric string built-in function
- an external variable

Dimension. The dimension of an array, the number of parameters for a function.
Length. The length of a string.

Dope Vector. The index of the first element of the array's dope vector as found in the dope array called DOPE.

Amount. When used with built-in functions, this indicates whether or not the built-in function is being used. For arrays, this contains the number of elements in a numeric array, or the number of bytes in a string array.

The symbol table is operated on using specialized procedures. LOOKUP is called with a pointer which identifies a symbol string. It invokes COMPARE repeatedly, working upward from the first symbol through the built-in function list. COMPARE compares two string arguments. If the string is found, LOOKUP returns the element number of the symbol. Otherwise -1 is returned. INSERT is called with a pointer argument to a symbol string. The string is copied into the next available table element and all the attribute elements are set to zero. When the scanner determines the symbol type, the attributes are set to the appropriate values.
1. Constant List

The constant list stores literal numbers in a C language structure as illustrated in Figure 3. It may contain up to 200 different literal numbers which may be accessed as members of an array, or by determining the characteristics of each element’s unique attributes. Each entry in the constant list refers to a structure of five elements, which contain the various attributes.

The attributes of a constant are:

Value. The actual value of the constant, stored in both double precision floating point and integer form.

Declaration. This identifies the context in which a number was first encountered which may be of type floating point or integer, determined by the presence of a decimal point in the input string. For code generation only the floating point form is used.

Use. This determines whether the value has been used as a number, a statement label, which may precede any statement, or a label, which is the statement label to which a branch statement or control structure refers.

In the C environment, a real number which is read as data for an integer variable is truncated to integer form. Similarly, an integer number read as data for a real variable is transformed to real notation. When a value is
Constant Table Structure

structure "numbers"  "attributes"

\[\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c} \quad \text{address i} \\
\text{d}
\end{array}\]

\[\begin{array}{c}
\text{i} \\
\text{number i} \\
\text{use} \\
\text{luse} \\
\text{dec}
\end{array}\]

Figure 3
stored in the Basic constant list, it is stored in both forms, thus requiring a flag indicating the proper form to be accessed when the number is used during execution of the program. While the compiler produces output which performs arithmetic operations with double precision floating point numbers only, labels and statement labels should be of integer form.

1. External Files

External file management is implemented using the UNIX system calls OPEN and CLOSE, and system routines GETC and PUTC [9].

Each time the parser encounters an OPEN statement, a flag is set in an element of the compiler array FDS, which contains a file descriptor status for each external file. The element number corresponds directly to the referenced external file. In the event a command to CLOSE a previously unopened file occurs, an error flag is set for the corresponding file. Similarly, efforts to READ from or PRINT to an unopened file will cause an error flag to be set in the FDS array. These errors are reported after the scanner completes its function, during the acceptance actions of the compiler.

While the parser is generating assembly code, the string name of each referenced file is inserted as a constant in the assembly source program. This provides the
string argument which is required as one of the parameters for the UNIX system routine OPEN.

2. Parser

The parser is a table-driven pushdown automaton. It receives a stream of tokens from the scanner and analyzes them to determine if they form a sentence in the extended Basic grammar. As the parser accepts tokens, one of three actions will be performed. It may stack the token and continue to analyze the source program by fetching another token, or the parser may determine that it has recognized the right part of one of the productions of the language and cause a reduction to take place. Finally, the parser may determine that the current string of tokens does not produce a valid right part for a production and thus produces a syntax error message.

3. Code Generation

In addition to verifying the syntax of source statements, the parser also acts as a transducer by associating semantic actions with reductions. Each time the parser determines that a reduction should take place, the procedure SEMANT is called with the number of the production passed as a parameter. The constant list contains the information required to perform the semantic action associated with the selected production. The action may include generation of assembly language code and operations such as symbol table
manipulations and updating of the parse arrays. Some productions have no semantic actions associated with them.

In the following section, the syntax of the language is listed in BNF notation [8]. A listing of the grammar with appropriate semantic actions is provided in the program listing following the appendices of this thesis. The token 'cr' means carriage return.

a. Extended Basic Language Structure

The overall structure of the extended Basic language is defined by the following syntax equations:

(1) <program> ::= <statement list> <end statement>

(2) <statement list> ::= <simple statement>
(3) :<statement list> <simple statement>

(4) <end statement> ::= <statement label> END cr
(5) ;END cr

(6) <simple statement> ::= <statement label> <exec state> cr
(7) :<statement label>
    <if statement> cr
(8) :<statement label>
    <data statement> cr
(9) :<statement label>
    <def statement> cr
(10) :<statement label>
    <rem statement> cr
(11) :<statement label>
    <extern statement> cr
(12) :<for statement>
(13) :<dim statement>
(14) :<exec state> cr
(15) :<if statement> cr
(16) :<data statement> cr
(17) :<def statement> cr
(18) :<rem statement> cr
(19) :<extern statement> cr
(20) :<error> cr
(21) :cr
b. Assignment Statements and Expressions

The following syntax equations are for properly formed assignment statements and expressions. The types of operands which are acceptable with each of the binary operators is shown in Table 1. The operand for the unary operators + and - must be numeric quantities. The operand for the unary operator NOT must be a logical quantity. The grammar rules cause a check to be made, insuring that the above semantic rules are followed.

Checks are also made to insure that subscripted variables are dimensioned before being used, that the correct number of subscripts is provided, that each subscript is of type numeric, and that a subscripted variable is not used as a FOR loop index. Likewise, checks are made on the number and type of parameters in a function call to insure they match the function definition. In rule (4b) the ';' appears literally in the equation.

(36) <let statement> ::= <string let>
(37)            |<numeric let>
(38) <string let> ::= LET <string ref> = <string exp>
(39) ;<string ref> = <string exp>

(40) <string ref> ::= <string id>
(41) ;<substring ref>
(42) ;<string array ref>
(43) ;<array subst ref>

(44) <substring ref> ::= <string ref lo> <substring spec>
(45) <string ref lo> ::= <string id> (  
(46) <substring spec> ::= <numeric exp> | <numeric exp> )

(47) <numeric exp> ::= <term>
(48) ;<numeric exp> + <term>
(49) ;<numeric exp> - <term>
(50) ;+ <term>
(51) ;<term>

(52) <term> ::= <primary>
(53) ;<term> * <primary>
(54) ;<term> / <primary>

(55) <primary> ::= <primary element>
(56) ;<primary> + <primary element>

(57) <primary element> ::= <numeric ref>
(58) ;<number>
(59) ;<bif>
(60) ;( <numeric exp> )
(61) ;<func ref>

(62) <numeric ref> ::= <numeric id>
(63) ;<array ref>

(64) <array ref> ::= <array ref head> <numeric exp> )
(65) <array ref head> ::= <array id> (  
(66) ;<array ref head> <numeric exp> ,

(67) <bif> ::= <string bif ref> <string exp> )
(68) ;<numeric bif ref> <numeric exp> )
(69) ;<numeric bif noarm>

(70) <string bif ref> ::= <string bif> (  
(71) ;<numeric bif ref> <numeric exp> ,

(72) <numeric bif ref> ::= <numeric bif> (  
(73) ;<numeric bif ref> <numeric exp>

(74) <string exp> ::= <string ref>
(75) ;<string>
(76) ;<str num bif> ( <numeric exp> )
c. Control Statements

The control statements in extended Basic are defined by the following syntax equations:

\[
\text{(104) } \text{<for statement>} \ ::= \text{ <statement label> <for clause> <statement list> <next clause>}
\]

\[
\text{(105) } \text{<statement label>} \ ::= \text{ <number>}
\]
(106) for clause ::= for head cr
(107)   ); for head STEP numeric exp cr
(108) for head ::= FOR for init TO numeric exp
(109) next clause ::= statement label NEXT
(110) NEXT numeric id cr
(111) NEXT cr
(112) statement label NEXT cr
(113) for init ::= numeric id = numeric exp
(114) if statement ::= if clause exec statement
(115) if clause ::= if clause else clause exec statement
(116) if clause ::= if clause else clause if statement
(117) if head ::= if head goto number
(118) if clause ::= if clause number
(119) if clause ::= if clause else clause number
(120) else clause ::= exec statement ELSE
(121) ELSE cr
(122) if clause ::= if head THEN
(123) if head ::= IF rel exp
(124) IF END # number
(125) stop statement ::= STOP
(126) rem statement ::= REM cr
(127) on statement ::= on head label
(128) on head ::= on begin
(129) on head ::= on begin
(130) on begin ::= ON numeric exp on case sel
(131) ON numeric exp on selector
(132) on case sel ::= GOSUB
(133) GOSUB
(134) on selector ::= THEN
(135) GOTO
(136) GOTO
(137) label ::= number
(138) branch statement ::= gosub label
(139) gosub label
(140) RETURN
d. Declaration Statements

All subscripted quantities in Basic should be declared prior to use in the program. The declaration statements in extended Basic are given by the following syntax equations:

(146) <dim statement> ::= <dim head> cr
(147)    |<dim head> cr
(148) <dim head> ::= <dim array head> <number> )
(149)    |<dim head alp> number> )
(150) <dim head> ::= <dim head slo> <number> )
(151) <dim array head> ::= <dim head> (  
(152) <dim head lp> ::= <statement label> DIM
(153)    |DIM
(154)    |<dim head>
(155)    |<dim head>
(156) <dim head slo> ::= <dim head lp> <string id> (  
(157) <dim head alp> ::= <dim head lp> <numeric id> (  
(158)    |<dim head alp> <number> ,
(159) <data statement> ::= <data head> <number>
(160)    |<data minus> <number>  
(161)    |<data head> <string>
(162) <data head> ::= DATA
(163)    |<data head> <number> ,
(164)    |<data minus> <number> ,
(165)    |<data head> <string>  
(166) <data minus> ::= <data head> -  
(167) <def statement> ::= <def left part> = <numeric exp>  
(168) <def left part> ::= DEF <def head> <numeric id> )
(169) <def head> ::= <function id> (  
(170)                        ;<def head> <numeric id>,  

\[ e. \] Input/Output Statements

The input/output statements in extended Basic are consistent with the ANSI proposed standards. Care should be exercised in the use of punctuation in input/output statements as defined by the following syntax equations:

(171) <open statement> ::= <open head> <number> ) <string>  
(172) <open head> ::= OPEN ( <number>,  
(173) <read statement> ::= <read head> <numeric ref>  
(174)                        ;<read head> <string ref>  
(175) <read head> ::= READ  
(176)                        ;<read head> <numeric ref>,  
(177)                        ;<read head> <string ref>,  
(178) <input statement> ::= <input head> <numeric ref>  
(179)                        ;<input head> <string ref>  
(180) <input head> ::= INPUT  
(181)                        ;<input head> <string exp>;  
(182)                        ;<input head> <numeric ref>,  
(183)                        ;<input head> <string ref>,  
(184) <readfil statement> ::= <readfil head> <numeric ref>  
(185)                        ;<readfil head> <string ref>  
(186) <readfil head> ::= <read file>  
(187)                        ;<readfil head> <numeric ref>,  
(188)                        ;<readfil head> <string ref>,  
(189) <read file> ::= READ & <number>, <numeric exp>;  
(190)                         ;READ & <number>;  
(191) <print statement> ::= PRINT  
(192)                        ;<print head> <numeric exp>  
(193)                        ;<print head> <string exp>  
(194)                        ;<print head> <format element>  
(195)                        ;<print head> <numeric exp>;  
(196)                        ;<print head> <string exp>;  
(197)                        ;<print head> <format element> ;

45
(198) <print head> ::= PRINT
(199)   |<print head> <numeric exp> ,
(200)   |<print head> <string exp> ,
(201)   |<print head> <format element> ,
(202)   |<print head> <numeric exp> ;
(203)   |<print head> <string exp> ;
(204)   |<print head> <format exp> ;
(205) <write statement> ::= <write head> <numeric exp>
(206)   |<write head> <string exp>
(207) <write head> ::= <write file>
(208)   |<write head> <numeric exp> ,
(209)   |<write head> <string exp> ,
(210) <write file> ::= PRINT # <number> , <numeric exp> ;
(211)   |PRINT # <number> ;
(212) <format element> ::= <simple format>
(213)   |<format left part> <numeric exp>)
(214) <format left part> ::= <numeric format> ( 
(215) <restore statement> ::= RESTORE 
(216)   |RANDOMIZE 
(217)   |RANDOMIZE ( <numeric exp> )
(218) <close statement> ::= CLOSE ( <number> )

f. External Statements

The external and call statements in extended Basic are the basis of the uniqueness of this implementation. These statements provide interface capability with other system programs and procedures. They are defined by the following syntax equations:

(219) <extern statement> ::= <extern head>
(220) <extern head> ::= EXTERN TYPE <numeric id> <parm def>
(221) |EXTERN <numeric id> <parm def>
(222) |EXTERN & TYPE <numeric id>
(223) |<extern head> , <numeric id> <parm def>
(224) <parm def> ::=
(225) \ \ \ ( )
(226) \ \ \ <\text{parm head}> \ \ \ \ \ \ \ \text{TYPE} )
(227) \ \ \ <\text{parm head}> \ \ \ \ & \ \ \ \ \ \ \ \text{TYPE} )

(228) \ \ \ <\text{parm head}> \ ::= \ ( \\
(229) \ \ \ \ \ \ \ \ <\text{parm head}> \ \ \ \ \ \ \ \text{TYPE} , \\
(230) \ \ \ \ \ \ \ \ <\text{parm head}> \ \ \ \ & \ \ \ \ \ \ \ \text{TYPE} ,

(231) \ \ \ <\text{call statement}> \ ::= \ <\text{call head}> \\
(231) \ \ \ <\text{call nhead}> \\
(232) \ \ \ \ \ <\text{call head}> \ <\text{numeric exp}> ) \\
(233) \ \ \ <\text{call head}> <\text{array id}> ) \\
(234) \ \ \ <\text{call head}> <\text{string exp}> ) \\
(235) \ \ \ <\text{call head} > \ & \ <\text{numeric id}> )

(236) \ \ \ <\text{call head}> \ ::= \ <\text{call nhead}> ( \\
(237) \ \ \ \ \ <\text{call nhead} > = <\text{numeric id} > ( \\
(238) \ \ \ <\text{call shead} > = <\text{numeric id} > ( \\
(239) \ \ \ <\text{call head}> <\text{numeric exp}> , \\
(240) \ \ \ <\text{call head} > <\text{array id} > , \\
(241) \ \ \ <\text{call head} > <\text{string id} > , \\
(242) \ \ \ <\text{call head} > \ & \ <\text{numeric id} > ,

(243) <\text{call nhead}> :: = \text{CALL} <\text{numeric id}>

(244) <\text{call shead}> :: = \text{CALL} <\text{string ref}>
<table>
<thead>
<tr>
<th>Type 1 operands</th>
<th>Type 2 operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td><code>+</code></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td><code>↑</code></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td><code>-</code></td>
</tr>
<tr>
<td><code>&lt;&gt;</code></td>
<td><code>and</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>*</code></td>
</tr>
<tr>
<td><code>=</code></td>
<td><code>or</code></td>
</tr>
<tr>
<td><code>= (assignment)</code></td>
<td><code>/</code></td>
</tr>
<tr>
<td>string</td>
<td>numeric</td>
</tr>
<tr>
<td>string</td>
<td>type 1</td>
</tr>
<tr>
<td>numeric</td>
<td>error</td>
</tr>
<tr>
<td>numeric</td>
<td>type 1, type 2</td>
</tr>
</tbody>
</table>
I. RECOMMENDATIONS FOR FUTURE DEVELOPMENTS

A number of additional extensions to this Basic language could be made. These include formatted input/output, a TRACE statement for debugging, additional string processing features, scientific notation, and random access for external files.

Basic processors have traditionally implemented formatted input/output by modifying the print statement as shown below:

```
PRINT USING <format string> ; <expression>
```

The format string contains a description of the format into which the values in the expression list are to be placed. This might be implemented using the PRINTF routine in the UNIX library or by allowing the user to directly use PRINTF vice the CALL and EXTERN statements.

A TRACE instruction, similar to that provided in many COBOL implementations, would list the source program line numbers as each statement was executed and optionally print the current values of selected variables. An accompanying UNTRACE statement would disable the trace. This could be easily implemented using flags.

Additional string operators could include a search function which would determine the position of one string
within another, and a substring replacement operation which would replace a substring with another (possibly null) string. String concatenation could be implemented for use in building strings by buffered input/output and using the UNIX routines GETC and PUTC.

Random access to elements of external files would be enhancing for file management, but would not greatly increase the flexibility of the existing file management methods used in graphics work. This might be accomplished by creating an array of data vectors at the beginning of each external file. Each vector would contain the beginning address of each record and the length of the record.

Scientific notation would enhance numeric output by expanding the range of numbers which could be comfortably printed on an output page.
II. CONCLUSIONS

The extended Basic compiler presented in this thesis is a working software package. It has demonstrated that it is capable of performing graphics work in the Naval Postgraduate School Computer Laboratory, and will provide a measurable improvement to graphics efforts of both Computer Science and non-Computer Science students than was previously afforded by the UNIX system library of programming languages.

Improvements noted in the Recommendations section do not represent all possible improvements, but only those developed or generated during development and testing of this Basic compiler.
Elements of extended Basic are listed in alphabetical order in this section of the thesis. A synopsis of each element is given, followed by a description and examples of its use. The intent is to provide a reference for the features of this implementation of BASIC and not to teach the BASIC language.

A program consists of one or more properly formed extended Basic statements. An END statement, which must be present, terminates the program, and additional statements are ignored. The ASCII character subset, consisting of alphanumerics and the specified special characters, is accepted.

In this section, the "synopsis" presents the general form of the element. Square brackets, [], denote an optional feature, while braces, {}, indicate that the enclosed section may be repeated zero or more times. Terms enclosed in <> are either non-terminal elements of the language, which are further defined in this section, or terminal symbols. All special characters and capitalized words are terminal symbols.
ELEMEN T:  

ABS predefined function

SYNOPSIS:

ABS ( <expression> )

DESCRIPTION:

The ABS function returns the absolute value of the <expression>. The argument should evaluate to a floating point number.

EXAMPLES:

ABS(X)

ABS(X*Y)
ELEMENT:

ASC predefined function

SYNOPSIS:

ASC ( <expression> )

DESCRIPTION:

The ASC function returns the ASCII numeric value of the first character of the <expression>. The argument should evaluate to a string.

EXAMPLES:

ASC(A$)

ASC("x")
ELEMENT:

ATAN predefined function

SYNOPSIS:

ATAN ( <expression> )

DESCRIPTION:

The ATAN function returns the arctangent of the <expression>. The argument should evaluate to a floating point number.

EXAMPLES:

ATAN(X)
ATAN(SQRT(SIN(X)))

PROGRAMMING NOTE:

All other inverse trigonometric functions may be computed from the arctangent using simple identities.
ELEMENT:
CALL statement

SYNOPSIS:

[[line number]] CALL [[variable] = ]
  <identifier> [ ( <expression>
  {~ <expression> ) ]

DESCRIPTION:
The CALL statement references an externally defined C procedure or function. The optional <variable> may be either a numeric identifier or a string reference. The CALL <identifier> may be up to 9 characters in length. If the <variable> is present, then the <identifier> references a function and returns a value. If the <variable> is absent, the <identifier> references a procedure and returns no value.

A CALL statement should be preceded by an EXTERN statement defining the form and nature of the <identifier>.

A CALL statement may have an infinite number of arguments which should each be valid <expressions> evaluating to numeric or character values. Arguments may further evaluate to array pointers if previously declared as such in the EXTERN statement. If the argument is declared to be of type char, then the argument value may consist of one character. To pass a string of characters as an argument, the argument may be of type X char, which implies a vector of characters, or a character string.

If a CALL statement has no arguments, then the entire argument list may be omitted from the statement.

EXAMPLES:

CALL j = test1 ("test Y")
call sink( ship )
call a$3(5) = strq( less )
CALL list
call movabs (x, y)
PROGRAMMING NOTE:

<identifiers> may be up to nine characters in length.
ELEMENT:

CHR$ predefined function

SYNOPSIS:

CHR$ ( <expression> )

DESCRIPTION:

The CHR$ function returns a character string of length 1 consisting of the character whose ASCII equivalent is the <expression> truncated to an integer modulo 128. The argument may evaluate to a floating point number.

EXAMPLES:

CHR$(A)
CHR$(12)
CHR$((A+B/C)*SIN(X))

PROGRAMMING NOTE:

CHR$ can be used to send the standard ASCII control characters such as a formfeed to the output device. The following statement would accomplish this:

PRINT CHR$(10)
ELEMENT: 
CLOSE statement

SYNOPSIS:

[
<line number>]
CLOSE (<constant>)

DESCRIPTION:

The CLOSE statement causes the file specified by its <constant> to be closed. Before the file may be referenced again it should be reopened using an OPEN statement.

A terminal error occurs if the specified file has not previously appeared in an OPEN statement.

EXAMPLES:

CLOSE (1)

PROGRAMMING NOTE:

On normal completion of a program all open files are closed. If the program terminates abnormally it is possible that files created by the program may be lost.
COL predefined function

SYNOPSIS:

COL ( <expression> )

DESCRIPTION:

The COL function defines the column width for a numeric output. The default width value is 10 digits, including the sign and the decimal point.

The COL function should be used only in a PRINT statement.

EXAMPLES:

print COL(12)
print COL(i*j)
ELEMENT:

<constant>

SYNOPSIS:

[<sign>] <integer> [.] [ <integer> ]

["] <character string> ["]

DESCRIPTION:

A <constant> may be either a numeric constant or a string constant. All numeric constants are stored as floating point numbers. Strings may contain any ASCII character except >, which may be represented as >.

Numeric constants may be either a signed or unsigned integer or decimal number. String constants may be up to 255 characters in length. Strings entered from the console for an INPUT statement may not contain quotes, however, a double quote or a newline may be used to terminate a string during INPUT or READ. Strings entered from a data statement should be enclosed in quotes, since they are found in the program. Strings read from a file may not contain quotes.

EXAMPLES:

10

-100.75639

"THIS IS THE ANSWER"

PROGRAMMING NOTE:

The line continuation character ($) may not be used in the program for carrying string constants to another line.
ELEMENT:

COS predefined function

SYNOPSIS:

COS( <expression> )

DESCRIPTION:

COS is a function which returns the cosine of the <expression>. The argument should evaluate to a floating point number expressed in radians.

EXAMPLES:

COS(θ)
COS(SQRT(x-y))
ELEMENT:

COSH predefined function

SYNOPSIS:

COSH ( <expression> )

DESCRIPTION:

COSH is a function which returns the hyperbolic cosine of the <expression>. The argument should evaluate to a floating point number.

EXAMPLES:

COSH(X)

COSH(X^2+Y^2)
ELEMENT:

DATA statement

SYNOPSIS:

\[ \langle \text{line number} \rangle \text{ DATA } \langle \text{constant} \rangle \{, \langle \text{constant} \rangle \} \]

DESCRIPTION:

DATA statements define floating point and string constants which are assigned to variables using a READ statement. Any number of DATA statements may occur in a program. Strings and numeric elements are stored separately. The ordering of string and number elements in a data statement need not match the ordering in the corresponding read statement. The first occurrence of an element type will be read when demanded. The constants are stored consecutively for each type in a data area as they appear in the program and are not syntax checked by the compiler. Character strings should be enclosed in quotes. Data elements should be separated by commas.

Should either type of data be exhausted, a restore for that type only is generated. If a type is requested when no data is defined, a terminal error results.

EXAMPLES:

10 DATA 10.0, 11.7, 100

DATA "This is a string.", 5, 10.4, "The End"

PROGRAMMING NOTE:

The RESTORE command may be used to reread a data line.
ELEMENT:
DEF statement

SYNOPSIS:

\[
(\text{line number}) \text{ DEF } \langle \text{function name} \rangle (\langle \text{variable} \rangle \\\n(, \langle \text{variable} \rangle)) = \langle \text{expression} \rangle
\]

DESCRIPTION:

The DEF statement specifies a user defined function which returns a floating point number. One or more arguments are passed to the function and are used in evaluating the expression. The values may be in floating point form. Recursive calls are not permitted.

The \langle expression \rangle in the define statement may reference \langle variables \rangle other than the dummy arguments, in which case the current value of the \langle variable \rangle is used in evaluating the \langle expression \rangle.

The first two alphanumerics of the \langle function name \rangle should be FN, Fn, fN or fn. The \langle function name \rangle may not exceed a total of four characters.

EXAMPLES:

10 DEF FNA(X,Y) = X + Y - A

DEF FNC(A,B) = A + B - FNA(A,B) + D
DEG predefined function

SYNOPSIS:
DEG ( <expression> )

DESCRIPTION:
The DEG function converts the floating point value of the <expression> into degrees. The <expression> should evaluate to a floating point value in radians.

EXAMPLES:
DEG ( 3.14159 * j )
ELEMENT:

DIM statement

SYNOPSIS:

1) \[
[\text{line number}] \text{ DIM } \langle\text{identifier}\rangle \ (\langle\text{subscript list}\rangle) \ \]
   \[
   \langle\langle\text{identifier}\rangle \ (\langle\text{subscript list}\rangle)\rangle
   \]

2) \[
[\text{line number}] \text{ DIM } \langle\text{identifier}\rangle \ (\langle\text{constant}\rangle) \ \]
   \[
   \langle\langle\text{identifier}\rangle \ (\langle\text{constant}\rangle) \ (\langle\text{subscript list}\rangle)\rangle
   \]

DESCRIPTION:

The dimension statement statically allocates space for floating point or string arrays. String array elements may be of any length up to 32767 characters. String array length should be specified. Initially, all floating point arrays are set to zero and all string arrays are null strings. An array may be dimensioned explicitly; no default options are provided except for string arrays which default to 1 element if the \langle\text{subscript list}\rangle is absent. Arrays are stored in row major order. The \langle\text{subscript list}\rangle may consist of integers. All subscripts have a lower bound of 0 and an upper bound of \( n \), for a total of \( n+1 \) elements.

The type 1 \text{DIM} statement above refers specifically to an array of numeric elements. Type 2 refers to string arrays. Both types of arrays may be combined in one \text{DIM} statement, however all the required elements in the synopsis may be present for each type.

\langle\text{constant}\rangle may be included for all string arrays and may not be present for floating point arrays. String array elements point to vectors of character strings with a maximum number of characters, or string length, equal to \langle\text{constant}\rangle. The \langle\text{subscript list}\rangle for a string array may not have more than one element.

EXAMPLES:

\text{DIM A(10,20), B(10)}

\text{DIM BS(2)(5), C(7)}
ELEMENT:

END statement

SYNOPSIS:

[<line number>] END

DESCRIPTION:

An END statement indicates the end of the source program. If any statements follow the END statement they are ignored.

EXAMPLES:

10 END

END

PROGRAMMING NOTE:

If a STOP statement does not precede an END statement somewhere in the program, a STOP statement is automatically inserted before the END statement.
ELEMENT:

<exec statement>

SYNOPSIS:

[<line number>] CALL statement <cr>
[<line number>] CLOSE statement <cr>
[<line number>] END statement <cr>
[<line number>] EXTERN statement <cr>
[<line number>] GOSUB statement <cr>
[<line number>] GOTO statement <cr>
[<line number>] INPUT statement <cr>
[<line number>] LET statement <cr>
[<line number>] NEXT statement <cr>
[<line number>] ON statement <cr>
[<line number>] OPEN statement <cr>
[<line number>] PRINT statement <cr>
[<line number>] PRINT # statement <cr>
[<line number>] RANDOMIZE statement <cr>
[<line number>] READ statement <cr>
[<line number>] READ # statement <cr>
[<line number>] RESTORE statement <cr>
[<line number>] RETURN statement <cr>
[<line number>] STOP statement <cr>

DESCRIPTION:

An <exec statement> is the only allowable executable statement in an IF statement construct. <exec statements> may appear as <simple statements> throughout the program.

NOTE:

See <statement>.
ELEMENT:

EXP predefined function

SYNOPSIS:

EXP( <expression> )

DESCRIPTION:

The EXP function returns e (2.71828... ) raised to the power of the <expression>. The argument should evaluate to a floating point number.

EXAMPLES:

EXP( X)
EXP( LOG( X) )
ELEMENT:

<expression>

DESCRIPTION:

Expressions consist of algebraic combinations of variables, constants, and operators. The hierarchy of operators is:

1) ()
2) ±
3) *, /
4) +, -, unary +, unary -
5) relational ops <, <=, >, >=, =, <>, ~=, !=, LT, LE, GT, GE, EQ, NE
6) NOT(expression)
7) AND
8) OR, XOR

Relational operators result in a 0 if false and nonzero (1) if true. String variables may be operated on only by relational operators. Mixed string and numeric comparisons are not permitted.

The three types of expressions are string, arithmetic and boolean.

EXAMPLES:

X + Y

(A <= R) OR (CS > D$) / (A - B AND D)
EXTERN

ELEMENT:

EXTERN statement

[<line number>] EXTERN [<type>] <identifier> ( [<type>
    (, <type>)])

DESCRIPTION:

The EXTERN statement declares the type of procedure or function referenced by the <identifier> in a CALL statement. The <identifier> is from an externally defined library and cannot be internally redefined by the user. The EXTERN statement should precede, and may appear at any point prior to, the CALL statement.

If the first optional <type> is missing, then that <type> defaults to integer.

The five varieties of <type> are integer, float, double, char and addr. These types may alternately be declared as arrays by preceding the type by &, as in & integer, & float, & double, & char and & addr.

The EXTERN statement may declare an infinite number of arguments for the procedure or function.

EXAMPLES:

extern ginit(integer)
extern integer move(integer, integer)
extern & char Amt( & float, & char)
extern gerase( )
extern newpq
ELEMENT:

FOR statement

SYNOPSIS:

```plaintext
[<line number>] FOR <index> = <expression> TO
  <expression> [STEP <expression>]
  <statement list>
[<line number>] NEXT [<index>]
```

DESCRIPTION:

Execution of all statements between the FOR statement and its corresponding NEXT statement is repeated until the indexing variable reaches the exit criteria. If the step is positive, the loop exit criteria is that the index exceeds the value of the TO <expression>. If the step is negative, the index should be less than the TO <expression> for the exit criteria to be met.

The <index> may be an unsubscripted variable and is initially set to the value of the first <expression>. If the exit criteria is met on initial entry, 0 executions of the loop are performed. If the STEP clause is omitted, a default value of 1 is assumed. A step of 0 may be used to loop indefinitely.

EXAMPLES:

```plaintext
FOR I = 1 TO 10 STEP 3
FOR IND = J*K-L TO 10*SIN(X)
FOR I = 1 TO 2 STEP 0
```
ELEMENT:

<function name>

SYNOPSIS:

FN<identifier> or fn<identifier>

DESCRIPTION:

Any <identifier> starting with fn, fn, FN, or Fn refers to a user-defined function. The <function name> should appear in a DEF statement prior to appearing in an <expression>.

There may not be any spaces between the FN or fn and the <identifier>.

EXAMPLES:

FNA(x) = x+2
fnAr(i,j) = i*j
ELEMENT:

GOSUB statement

SYNOPSIS:

[<line number>] GOSUB <line number>
[<line number>] GO SUB <line number>

DESCRIPTION:

The address of the next sequential instruction is saved on the run-time stack, and control is transferred to the subroutine labeled with the <line number> following the GOSUB or GO SUB.

EXAMPLES:

10 GOSUB 300
GO SUB 100
ELEMENT:

GOTO statement

SYNOPSIS:

[<line number>] GOTO <line number>

[<line number>] GO TO <line number>

DESCRIPTION:

Execution continues at the statement labeled with the <line number> following the GOTO or GO TO.

EXAMPLES:

100   GOTO 50

GO TO 10
ELEMENT:

<identifier>

SYNOPSIS:

<letter> { <letter> or <number> } [$]  

DESCRIPTION:

An identifier begins with an alphabetic character followed by three alphanumeric characters. If the second or third character is a dollar sign the associated variable is of type string, otherwise it is of type floating point.

EXAMPLES:

A

R$s

XYb

PROGRAMMING NOTE:

All non-reserved identifiers may consist of any mixture of upper and lower case letters and numerics.
ELEMENT:

IF statement

SYNOPSIS:

[<line number>] IF <expression> GO TO <line number>

[<line number>] IF <expression> THEN <exec statement>

[<line number>] IF <expression> THEN <exec statement>
ELSE <exec statement>

or

ELSE IF statement

DESCRIPTION:

If the value of the <expression> is not 0, the following occurs:
1) the GOTO causes an unconditional branch to <line number>, or
2) the <exec statement> following the THEN is executed.

If the value of the <expression> is 0, the following occurs:
1) either the <exec statement> or the IF statement following the ELSE is executed, or
2) the next sequential statement in the program is executed.

EXAMPLES:

IF A$ < B$ THEN X = Y*Z

IF (A$<B$) AND (C OR D) GO TO 300

IF J AND K THEN GOTO 11 ELSE GOTO 12

PROGRAMMING NOTE:

The line continuation symbol (\) may be used following the THEN or ELSE symbols to produce more readable code:

\if x = y then \a
  z = z + 1 \a
else \a
  print x = y
\a
ELEMENT:

[INPUT statement

SYNOPSIS:

[<line number>] INPUT [<prompt string> ;] <variable> {, <variable> }
{, <prompt string>; <variable> }
{, <variable>}

DESCRIPTION:

The <prompt string>, if present, is printed on the console. A prompt string may be followed by a semi-colon. A line of input data is read from the console and assigned to the variables as they appear in the variable list. Data items preceded by prompt strings should be separated by a carriage return. Strings may not be enclosed in quotation marks.

EXAMPLES:

10 INPUT A,B
   INPUT "SIZE OF ARRAY?"; N, "DEFAULT VALUE?"; X
   INPUT "VALUES?"; A(I),B(I),C(A(I))
   input a$, a(i)
ELEMENT:

INT predefined function

SYNOPSIS:

INT (<expression>)

DESCRIPTION:

The INT function returns the largest integer less than or equal to the value of the <expression>. The argument should evaluate to a floating point number.

EXAMPLES:

INT (AMNT / 100)
INT(3 * X * SIN(Y))
LEN

ELEMENT:

LEN predefined function

SYNOPSIS:

LEN ( <expression> )

DESCRIPTION:

The LEN function returns the actual length of the string <expression> passed as an argument. Zero is returned if the argument is the null string.

EXAMPLES:

LEN(A$)
LET STATEMENT

SYNOPSIS:

\[ \text{[line number]} \text{ (LET) variable = expression} \]

DESCRIPTION:

The \textit{expression} is evaluated and assigned to the \textit{variable} appearing on the left side of the equal sign. The type of the \textit{expression}, either floating point or string, should match the type of the \textit{variable}.

EXAMPLES:

100 LET A = B + C

\[ x(3,A) = 7.32 + Y + x(2,3) \]

73 \[ w = (A<B) \text{ OR (C$>D$)} \]
ELEMENT:

<line number>

SYNOPSIS:

<digit> {<digit>}

DESCRIPTION:

<line numbers> are optional on all statements and are ignored by the compiler except when they appear in a GOTO, GOSUB, or ON statement. In these cases, the <line number> should appear as the label of one and only one <statement> in the program.

<line numbers> should be less than 32767.

EXAMPLES:

100

4635
ELEMENT:

LOG predefined function

SYNOPSIS:

LOG ( <expression> )

DESCRIPTION:

The log function returns the natural logarithm of the value of the <expression>. The argument should evaluate to a non-zero floating point number.

A negative value will produce undesirable results.

EXAMPLES:

LOG (X)

LOG((A + B)/D)

LOG10 = LOG(X)/LOG(10)
ELEMENT:
MOD predefined function

SYNOPSIS:
MOD ( <expression> , <expression> )

DESCRIPTION:
The MOD function evaluates the first <expression> modulo the second <expression> and returns a float point value. Both <expressions> should evaluate to floating point numbers.

EXAMPLES:
MOD (x, y)
MOD (SQRT (LOG (X)), X + Y)
ELEMENT:

NEXT statement

SYNOPSIS:

[<line number>] NEXT [<identifier>]

DESCRIPTION:

A NEXT statement denotes the end of the closest unmatched FOR statement. If the optional <identifier> is present it should match the index variable of the FOR statement being terminated. The <line number> of a NEXT statement may appear in an ON or GOTO statement, in which case execution of the FOR loop continues with the loop variables assuming their current values.

While it is possible to branch into a loop, it is undesirable since the loop will not be properly executed. Those statements occurring at and after the addressed statement will be executed, and the NEXT statement will be ignored.

EXAMPLES:

10 NEXT

NEXT I
ELEMENT:

ON statement

SYNOPSIS:

(1) [<line number>] ON <expression> GOTO <line number> {, <line number>}

(2) [<line number>] ON <expression> GO TO <line number> {, <line number>}

(3) [<line number>] ON <expression> GOSUB <line number> {, <line number>}

(4) [<line number>] ON <expression> GO SUB <line number> {, <line number>}

(5) [<line number>] ON <expression> THEN <line number> {, <line number>}

DESCRIPTION:

The <expression>, truncated to the nearest integer value, is used to select the <line number> at which execution will continue. If the <expression> evaluates to 1 the first <line number> is selected and so forth. In the case of an ON ... GOSUB statement the address of the next instruction becomes the return address. ON ... THEN produces the same results as ON ... GO TO

If the <expression> after truncating is less than one or greater than the number of <line numbers> in the list, the program continues with the next executable statement.

EXAMPLES:

10 ON I GOTO 10, 20, 30, 40

ON J*K=M GOSUB 10, 1, 1, 10
ELEMENT:
OPEN statement

SYNOPSIS:

[<line number>] OPEN (<file number>,<mode>) <file name>

DESCRIPTION:

The OPEN statement opens the <file number> for random access (<mode> 0), reading (<mode> 1), writing (<mode> 2), appending (<mode> 3). <file name> is a string of ASCII characters which represents the file specified by <file number>. A file is created by the first OPEN statement for <file name> and <file number> with the write <mode> specified. Attempts to open a non-existent file for reading will cause a fatal error.

Although the programmer may have uncountably many files, limited only by the number of <file names> available; a maximum of 15 files may be open at any one time. <file numbers> are restricted to the sequence 0-14 inclusively. No two <file numbers> for open files may be the same, but should be unique for each open file.

The <file number> will be used for input/output and closing of files. It is the sole uniform reference between the above statements.

EXAMPLES:

10 open (1,1) "datal"

open (5,0) "filed"
ELEMENT:

PAGE predefined function

SYNOPSIS:

PRINT PAGE

DESCRIPTION:

The PAGE function causes a new page command to be issued. The page print function should not be used on the console, since it will cause undesirable effects on the CRT screen.

EXAMPLES:

print page

PROGRAMMING NOTE:

PRINT PAGE is the same as PRINT CHR$(10). It should be used in the same manner as TAB or CCL, which is only in a PRINT statement.
ELEMENT:

PRINT statement

SYNOPSIS:

[<line number>] PRINT <expression> <delim>

( <expression> <delim> )

DESCRIPTION:

A PRINT statement sends the value of the expressions in the expression list to the console. A space is appended to all numeric values and if the numeric item exceeds the right margin then the print buffer is dumped before the item is printed. The <delim> between the <expressions> may be either a comma or a semicolon.

If the <delim> is a comma, the output of elements is sequential on an output line. If the semicolon is used, the print buffer is dumped upon encountering the semicolon, and the next line is begun. If, however, the semi-colon occurs at the end of the list of elements to be printed, no newline is issued, and subsequent printing will begin at the next position on the line.

EXAMPLES:

PRINT A, B, "THE ANSWER IS"; x
ELEMENT:

PRINT # statement

SYNOPSIS:

[<line number>] PRINT # <file number> ;
<expression> , <expression>

DESCRIPTION:

PRINT # causes the output for a program to be directed to the indicated file number. Before a transaction may take place, a file should be opened using the OPEN command with mode 2 or 3. The file is an external file in the user's directory. This allows the user to store program results externally, and to eventually output the results to an external device, such as the line printer.

EXAMPLES:

PRINT # 2
ELEMENT:

RAD predefined function

SYNOPSIS:

RAD ( <expression> )

DESCRIPTION:

The RAD function converts the value of the <expression> into a radian value. The <expression> should evaluate to a floating point number.

EXAMPLES:

RAD (180 * i)
ELEMENT:

RANDOMIZE statement

SYNOPSIS:

[<line number>] RANDOMIZE [<numeric expression>]

DESCRIPTION:

A RANDOMIZE statement seeds the random number generator with 1301, if no <numeric expression> argument is supplied, and <numeric expression> modulo 2**15 - 1 if specified.

EXAMPLES:

10 RANDOMIZE

    RANDOMIZE (1013)
READ statement

SYNOPSIS:

\[ \langle \text{line number} \rangle \] READ
\[ \langle \text{variable} \rangle , \langle \text{variable} \rangle \]

DESCRIPTION:

A READ statement assigns values to variables in the variable list from DATA statements. Fields may be floating point or string constants and are delimited by a comma.

DATA statements are processed sequentially as they appear in the program. An attempt to read past the end of the last data statement produces an error, and automatically generates an appropriate RESTORE. An attempt to read non-existent data will produce a terminal error.

EXAMPLES:

100 READ A,B,C$
ELEMENT:

READ # statement

SYNOPSIS:

\[ <\text{line number}> \] READ # <\text{file number}> ;
\  \  \ <\text{variable}> \ (, <\text{variable}>)

DESCRIPTION:

A READ # statement assigns values to variables in the variable list. Values are read from sequential records from the external file specified by the <file number>. Fields may be floating point or strings.

EXAMPLES:

200 \  \ READ # 1; \ PAYR, \ PAYO, \ HKSP, \ HRSO
\  \ READ # 2; \ x, \ y, \ z5
ELEMENT:

REM statement

SYNOPSIS:

[<line number>] REM [<remark>]

[<line number>] REMARK [<remark>]

DESCRIPTION:

A REM statement is ignored by the compiler and compilation continues with the statement following the next carriage return. The REM statement may be used to document a program. REM statements do not affect the size of program that may be compiled or executed.

A REM statement may be the object of either a GOTO or GOSUB statement.

EXAMPLES:

10 REM THIS IS A REMARK

20 REMARK This is another remark.
ELEMENT:

reserved word list

SYNOPSIS:

<letter> { <letter> } [ $ ]

DESCRIPTION:

The following words are reserved by extended Basic and may not be used as <identifiers>:

ABS  AND  ASC  ATAN  CALL
CHR$ CLOSE COL COS COSH
DATA  DEF  DEG  DIM ELSE
END  EQ  EXP  FILE FOR
GE  GO  GOSUB GOTO GT
IF  INPUT INT LE LEN
LET  LOG  LT  MOD NE
NEXT  NOT  ON OPEN OR
PAGE  PRINT  RAD RANDOMIZE READ
REM  RESTORE RETURN RND SIN
SINH  SQRT  STEP STOP TAB
TAN  THEN  TO VAL

Reserved words may be preceded and followed by either a special character or a space. Spaces may not be embedded within reserved words. Reserved word identifiers should consist of upper or lowercase letters exclusively.
ELEMENT:

RND predefined function

SYNOPSIS:

RND

DESCRIPTION:

The RND function generates a uniformly distributed random number between 0 and 1.

EXAMPLE:

RND
ELEMENT:

<simple statement>

SYNOPSIS:

[<line number>] DATA statement <cr>
[<line number>] DEF statement <cr>
[<line number>] DIM statement <cr>
[<line number>] <exec statement> <cr>
[<line number>] FOR statement <cr>
[<line number>] IF statement <cr>
[<line number>] REM statement <cr>

DESCRIPTION:

All <simple statements> are elements of a <statement list> and are executable. All <simple statements> end with a carriage return <cr>. 
ELEMENT:

SIN predefined function

SYNOPSIS:

SIN ( <expression> )

DESCRIPTION:

SIN is a predefined function which returns the sine of the <expression>. The argument should evaluate to a floating point number in radians.

EXAMPLES:

x = SIN(y)
SIN(A - B/C)
ELEMENT:

SINH predefined function

SYNOPSIS:

SINH ( <expression> )

DESCRIPTION:

SINH is a function which returns the hyperbolic sine of the <expression>. The argument should evaluate to a floating point number.

EXAMPLES:

SINH(Y)

SINH(B + C)
special characters

DESCRIPTION:

The following special characters are used by Extended Basic:

†  circumflex  
(  open parenthesis  
)  closed parenthesis  
[  open square bracket  
]  closed square bracket  
"  double quote  
*  asterisk  
+  plus  
−  minus  
/  slant  
;  semicolon  
<  less-than  
>  greater-than  
=  equal  
,  comma  
CR  carriage return (new line)  
!  exclamation point  
@  line continuation  
~  tilde  
\  substring  
\  space  
#  number sign  
$  dollar  
&  ampersand  
.  period  

Any special character in the ASCII character set except >, which may appear as \>, may appear in a string. Special characters other than those listed above, if they appear outside a string, will generate an error.
ELEMENT:

SQRT predefined function

SYNOPSIS:

SQRT ( <expression> )

DESCRIPTION:

SQRT returns the square root of the absolute value of the <expression>. The argument should evaluate to a floating point number. Negative numbers will return 0.

EXAMPLES:

SQRT (Y)

SQRT(X^2 + Y^2)
ELEMENT:

<statement list>

SYNOPSIS:

<simple statement>

{(simple statement)}

DESCRIPTION:

A <statement list> is a sequence of executable statements. All extended Basic statements are terminated by a carriage return (<cr>).
ELEMENT:

STOP statement

SYNOPSIS:

[<line number>] STOP

DESCRIPTION:

Upon execution of a <STOP statement>, program execution terminates and all open files are closed. The print buffer is emptied and control returns to the host system. Any number of STOP statements may appear in a program.

A STOP statement is appended to all programs by the compiler.

EXAMPLES:

10 STOP

STOP
ELEMENT:

<subscript list>

SYNOPSIS:

<integer> {, <integer> }

DESCRIPTION:

A <subscript list> may be used as part of a <DIM statement> to specify the number of dimensions and extent of each dimension of the array being declared or as part of a <subscripted variable> to indicate which element of an array is being referenced.

Elements of a subscript list in a DIM statement may be integers.

EXAMPLES:

x(10,20,20)
ELEMENT:

TAB predefined function

SYNOPSIS:

TAB (<expression>)

DESCRIPTION:

TAB moves the text pointer to the absolute column indicated by the evaluated <expression>. If the expression evaluates to a value greater than 80, the TAB value is defaulted to <expression> - 80 and will not cause text to wrap around on the same line at the console.

EXAMPLES:

TAB (10)

TAB (i + j)
ELEMENT:

TAN predefined function

SYNOPSIS:

TAN (<expression>)

DESCRIPTION:

TAN is a function which returns the tangent of the expression. The argument should evaluate to a floating point number in radians.

If the <expression> is a multiple of pi/2 radians, the value returned is the largest or smallest number in the system, depending upon which side of zero is approached by the function.

EXAMPLES:

10 TAN(A)

TAN(X - 3*COS(Y))
ELEMENT:

VAL predefined function

SYNOPSIS:

VAL(<expression>)

DESCRIPTION:

The VAL function converts the string number in ASCII passed as a parameter into a floating point number. The <expression> should evaluate to a string.

Conversion continues until a character is encountered that is not part of a valid number or until the end of the string is encountered. The maximum length for a string is 22 digits.

EXAMPLES:

VAL(A$)
VAL("3.789")
VAL("This returns zero")
ELEMENT:

<variable>

<variable>

SYNOPSIS:

<identifier> [ ( <subscript list> ) ]
<string identifier> ( <beginning position> ! <string length> )

DESCRIPTION:

A <variable> in extended Basic may either represent a floating point number or a string depending on the type of the <identifier>. All string variables should appear in a DIM statement before being used as a <variable>.

String variables may be broken down into substring units by indicating string name, starting character and length of substring. The element <beginning position> is an <expression> and refers to the first character position of the substring. It should evaluate to a number. The element <string length> is an <expression> and should evaluate to a number. It is the absolute length of the substring. String character count begins at 1.

EXAMPLES:

X
YS(3;10)
ABS$(8;20)
ABS$(x(I),y(I),s(I-1))
APPENDIX II - OPERATING IN UNIX WITH EXTENDED BASIC

Ibax is the shell command call for the extended Basic compiler in the PDP-11/50 UNIX computer system at the Naval Postgraduate School. It is of the form:


The system call accepts three types of arguments:

Flags defined below: an argument whose name ends with ".o" which is taken to be a Basic source program and is compiled; arguments ending in ".o" which are taken as object files to be passed to the loader.

The following flags are interpreted by Ibax:

- C Include the standard C library when loading the results of the compilation.
- S Compile the named Basic program, and leave the assembly-language output on a corresponding file suffixed ".s".
- c Include the graphics library for the CONOGRAPHICS graphics device.
- r Include the graphics library for the RAMTEK graphics device.
- o Compile the named Basic program, and leave the object file on a corresponding file suffixed ".o".
- t Include the graphics library for the Tektronix graphic device.
- v Include the graphics library for the Vector General graphics device.

Whenever a graphics library is included for loading with the compiled source program, the standard C library is appended to the loader library list. Other arguments are taken to be either C compatible object programs, typically produced by an earlier C compilation, or perhaps libraries of Basic or C compatible routines. These programs, together with the results of any specified compilation, are loaded (in the order given) to produce an executable program with the name a.out. Libraries with the same file name as the source program, and which end in ".o", should not be used since they will not be retained upon creation of file.o by the executive program.

Basic programs may not be compiled for future use as libraries since every compiled Basic program includes a "main" section, which drives the program. Thus additional
libraries may be created in the C language, compiled using
the -c option for output as '.o' files, and then included in
the '.o' form as object libraries for the Basic loader [9].

If the -o option is exercised, the subsequent file.o
may be invoked by LBAX and will return an executable a.out
file. The effect of the -o option is to produce the source
program in object code, which is fully loadable. Caution
should be exercised to prevent usage of a -o option output
as a library file.

In addition to the features supported in standard
Basic, a number of special features are found in the NPS
version of extended Basic. These include:

- call References an externally defined C language
  procedure or function.
- chr$ Return a character string of length 1 determined
  by the ASCII equivalent of an expression argu-
  ment.
- close Causes the externally referenced file to be
closed.
- col Specifies column width of subsequently printed
  numeric values.
- dim In addition to numeric arrays, permits creation
  of a vector of strings.
- extern Declares type and arguments of external pro-
  cedure or function referenced by a call state-
  ment.
- len Returns the length of a string expression.
- mod Evaluates an expression with modulo arithmetic.
- open Causes the externally referenced file to be
  opened and indicates the mode for opening the
  file.
- read file Reads sequentially from the specified external
  file.
- val Converts a string of numbers to a floating point
  number.
- write file Write sequentially into the specified external
  file.

String manipulation is enhanced by use of substringing
constructs. Strings may be referred to in an Algol-like
manner to produce portions for reading, writing, or altera-
tion.

Since the UNIX environment does not support some of the
features of standard Basic without considerable system over-
head (and in some cases, not at all), the NPS version of ex-
tended Basic uses slightly different, although no less
specific, formats in some statement formations.
Importantly, the NPS extended Basic is a compiler version, and is not interpretive. Thus, the use of line numbers with every statement is not mandatory or recommended. Creation and subsequent editing of programs is effected by use of the UNIX editor. Execution of the program is accomplished through the a.out file, as with other UNIX compilers.

The files which are used by the system while executing the shell executive program are:

- `file.b` input file
- `file.o` object file
- `file.s` assembly-language output
- `a.out` loaded output
- `/usr/basic/haxcomp` compiler
- `/usr/basic/casiclib.a` Basic library
- `/usr/graph/conie.a` Oconographics library
- `/usr/graph/rmtksub.o` RAMTEK library, part I
- `/usr/graph/moresub.o` RAMTEK library, part II
- `/usr/graph/vq.a` Vector General library
- `/usr/lib/libt.a` Tektronics library
- `/lib/libc.a` C library; see section III
- `/lib/liba.a` Assembler library used by some routines in libc.a and basiclib.a

The diagnostics produced by Basic itself are intended to be self-explanatory. Occasionally messages may be produced by the assembler or loader. Of these the most mystifying are from the assembler, in particular "m", which means a multiple-defined external symbol (function or data).
PROGRAM LISTING - EXTENDED BASIC COMPILER

PARSING RULES

%{
#include "./hstruc.c"
#include "./bfun.c"
%

%token STEP DATA DEF DIM ELSE END FOR GOSUB GO TO GOTO IF
%token NEXT ON PRINT READ REM RESTORE RETURN STOP THEN TO
%token OPEN CLOSE SUB RANDOMIZE rele spec OR XOR NOT AND
%token number numeric+id array+id string+id function+id
%token numeric+format string+string+if numeric+if
%token simple+format str+num+if
%token EXTERN TYPE INPUT LET CALL

%left '+' '-'
%left '*' '/'
%left '^'

%% /* beginning of the rules section */
program: statement-list end-statement

statement-list: simple-statement

statement-list: statement-list simple-statement

end-statement: statement-label END '0'

statement-label: exec-statement '0'

statement-label: if-statement '0'

statement-label: data-statement '0'

statement-label: def-statement '0'

statement-label: rem-statement '0'

statement-label: extern-statement '0'

for-statement

dim-statement

exec-state '0'

if-statement '0'

data-statement '0'

def-statement '0'

116
rem+statement '0
extern+statement '0
element '0
'0

exec+state:
read+statement
restore+statement
open+statement
close+statement
input+statement
readf+statement
print+statement
write+statement
stop+statement
on+statement
branch+statement
let+statement
call+statement

for+statement:
statement+label for+clause statement+list
next+clause = (semant(41,52);)
for+clause statement+list next+clause =
         (semant(41,51));

statement+label:
   number =
       (semant(19,51);
if (numbers[51].use != 1)
numbers[51].use=2; )

label:
   number =
       (semant(20,51); if (numbers[51].use == 0)
numbers[51].use=1; )

for+clause:
   for+head '0 =
       ( $3=forctr; semant(39,51);)
for+head STEP numeric+exp '0 =
       ( $3=forctr; semant(40,51);)

for+head:
   FOR for+init TO numeric+exp = ( $3=$2;)

next+clause:
   statement+label NEXT numeric+id '0
   NEXT numeric+id '0
   NEXT '0
statement+label NEXT '0

for+init:
   numeric+id '=' numeric+exp =
       ( $3=$1; semant(38,51);)

dim+statement:
sdim+head '0
dim+head '0
dim+head:     dim+array+head number ')') =
  ( symtable[31].amt =
    (numbers[52].numberi+1) +
    symtable[31].length+1; )
; dim+head+arlo number ')') =
  ( j=doeek++; done[j]=numbers[52].numberi;
    symtable[31].dimen++;
    caldope(51, symtable[31].dimen, symtable[31].doov); )
; dim+head:     dim+head+slo number ')') = ( $5=51;
    symtable[31].length=numbers[52].numberi;)
; dim+array+head:     sdim+head '(' =
  ( $5=51; sytable[31].type=3;
    symtable[31].dimen=1; )
; dim+head+lp:     statement+label DIM
  ; DIM
  ; sdim+head '
  ; dim+head '
; dim+head+slo:     dim+head+lp string+id '(' = ( $3=$2;)
; dim+head+aro:     dim+head+aro numeric+id '(' =
  ( $S=$2; symtable[32].dimen=0;
    symtable[32].type=1;
    symtable[32].doov=doeept;)
; dim+head+slo:     dim+head+slo string+id '(' = ( $3=$2;)
; data+statement:     data+head number =
  ( data[datapt++] = numbers[52].numberf;)
; data+minus number = ( data[datapt++] =
  -numbers[52].numberf;)
; data+head string =
  ( strcopy(stiq,datastor);
    datastor+=stiqf+1; )
; data+head:     DATA
  ; data+head number ', ' = (data[datapt++] =
    numbers[52].numberf;)
; data+minus number ', ' = ( data[datapt++] =
  -numbers[52].numberf;)
; data+head string ', ' =
  (strcopy(stiq,datastor); datastor+=stiqf+1;)
; data+minus:     data+head '-'
; def+statement:     def+left+part '=' numeric+exp =
  ( semant(37, 31);)
; def+left+part:     DEF def+head numeric+id '(' =
  ( semant(35, 52); $S=$2;
def+head: function+id '(' = ( $s$=$l$; symtable[$l$].dimen=0); 
  | def+head numeric+id ',' = ( $s$=$l$; symtable[$l$].length=$s$2; symtable[$l$].dimen++)

read-statement: read+head numeric+ref =
  ( semant(33,-1));
  | read+head string+ref =
  ( semant(54,-1));

read+head: READ
  | read+head numeric+ref ',' =
  ( semant(33,-1));
  | read+head string+ref ',' =
  ( semant(54,-1));

restore-statement: RESTORE =
  ( semant(32,-1));
  | RANDOMIZE =
  ( semant(55,-1));
  | RANDOMIZE '(' numeric+exp ') ' =
  ( semant(55,33));

open-statement: open+head number ')' string =
  ( semant(51,32));

open+head: OPEN '(' number ',' =
  ( j=numbers[33].number; fns[j] = 1; semant(50,33));

close-statement: CLOSE '(' number ')' =
  ( j=numbers[33].number;
   if(fns[j] == 0) fns[j] = 2;
   semant(52,33));

input-statement: input+head numeric+ref =
  ( semant(48,-1));
  | input+head string+ref =
  ( semant(49,-1));

input+head: INPUT
  | input+head string+exp '; ' =
  ( semant(43,-1);
   stigl=0; stiq[stigl++]= ' ';
   stiq[stigl]='0'; semant(14,j);
   semant(43,-1));
  | input+head numeric+ref ',' =
  ( semant(48,-1));
  | input+head string+ref ',' =
  ( semant(49,-1));
readf+statement: readf+head numeric+ref =
  ( semant(69,-1); semant(71,-1); )
  readf+head string+ref =
  ( semant(70,-1); semant(71,-1); )
readf+head:
  readf+file
  readf+head numeric+ref ',,' =
  semant(69,-1);
  readf+head string+ref ',,' =
  semant(70,-1);
readf+file: READ '#' number ',,' numeric+exp ';'; =
  ( j=numbers[33].number;
    if(fds[j] == 0) fds[j] = 2;
    semant(68,53); )
  READ '#' number ',,' =
  ( j=numbers[33].number;
    if(fds[j] == 0) fds[j] = 2;
    semant(68,53); )
print+statement: PRINT
  ( semant(44,-1));
  print+head numeric+exp =
  ( semant(42,-1); semant(44,-1));
  print+head string+exp =
  ( semant(43,-1); semant(44,-1));
  print+head format+element =
  ( semant(44,-1));
  print+head format+element ',,' =
  ( semant(42,-1));
  print+head string+exp ';'; =
  ( semant(43,-1));
print+head:
  PRINT
  print+head numeric+exp ',,' =
  ( semant(42,-1));
  print+head string+exp ',,' =
  (semant(43,-1));
  print+head format+element ',,' =
  (semant(43,-1));
  print+head numeric+exp ';'; =
  ( semant(42,-1); semant(44,-1));
  print+head string+exp ';'; =
  ( semant(43,-1); semant(44,-1));
  print+head format+element ';';
write+statement: write+head numeric+exp =
  ( semant(72,-1); semant(74,-1); )
write+head string+exp =
  ( semant(73,-1); semant(74,-1); )
write+head:
  write+head numeric+exp ',,' =
{ semant(72, -1); }

; write-head string-exp ',' =
{ semant(73, -1); }

write-file: PRINT #' number ',' numeric-exp ';' =
{ j=numbers[53].numberi;
  if(fds[j] == 0) fds[j] = 2;
  semant(75, $3); }

; PRINT #' number ';'
{ j=numbers[53].numberi;
  if(fds[j] == 0) fds[j] = 2;
  semant(75, $3); }

; format-element:    simple-format =
{ semant($2, -1); }

; format-left-part numeric-format '(' =
{ semant($3, $1); }

; format-left-part:    numeric-format '(' =
{ $3 = $1; }

if-statement: if-clause exec-state =
{ semant($2, -1); }

; if-clause else-clause exec-state =
{ semant($2, -1); }

; if-clause else-clause if-statement =
{ if-head goto number =
  { semant($3, $3); }
  if-clause number =
  { semant($10, -1); semant($3, $2); }
  if-clause else-clause number =
  { semant($10, -1); semant($3, $3); }

; else-clause: exec-state ELSE =
{ semant($1, -1); }

; number ELSE =
{ semant($10, -1); semant($3, $1); }

if-clause: if-head THEN

; if-head:    IF rel-exp =
{ semant($27, -1); }

; IF END #' number

rel-exp: rel-exp XOR rel-term =
{ semant($56, -1); }

; rel-exp OR rel-term =
{ semant($57, -1); }

; rel-term

rel-term: rel-term AND rel-primary =
{ semant($58, -1); }

121
rel+primary

rel+primary: numeric+exp rel numeric+exp =
( semant(25,32));

string+exp rel string+exp =
( semant(20,52));

'(' rel+exp ')'

NOT '(' rel+exp ')'
( semant(59,-1));

rel: '=' := ($$=0;)

'!' '=' := ($$=4;)

'>' := ($$=8;)

'<' := ($$=12;)

'>' '=' := ($$=16;)

'=' := ($$=20;)

'<' '>' := ($$=4;)

'=' := ($$=4;)

relspec := ($$=1;)

stop+statement: STOP =
( semant(24,-1);)

rem+statement: REM

on+statement: on+head label =
( semant(23,-1));

on+head:
on+begin

on+head label '',' =
( semant(22,-1));

on+begin: ON numeric+exp on+case+sel =
( semant(21,0));

ON numeric+exp on+selector =
( semant(21,-1));

on+selector:

THEN

GOTO

GO TO

on+case+sel: GOSUB

GO SUB

branch+statement: gosub label

goto! label

RETURN =
( semant(18,-1));

122
gostub: GOSUB =
\( \text{semant}(17,-1); \)
\( \text{GO SUB} = \)
\( \text{semant}(17,-1); \)
\( \text{ goto } 1: \text{goto} \)
\( \text{goto: GOTO} = \)
\( \text{semant}(16,-1); \)
\( \text{GO TO} = \)
\( \text{semant}(16,-1); \)
let=statement: string=let
\( \text{numeric=let} \)
string=let: LET string=ref ' = ' string=exp =
\( \text{semant}(15,-1); \)
\( \text{string=ref ' = ' string=exp} = \)
\( \text{semant}(15,-1); \)
string=exp: string=ref
\( \text{string} = \)
\( \text{semant}(14,j); \)
\( \text{string=ref} \) \( \text{(' numeric=exp ')} \) =
\( \text{semant}(53,51); \)
numeric=let: LET numeric=ref ' = ' numeric=exp =
\( \text{semant}(13,-1); \)
\( \text{numeric=ref} \) \( \text{=' numeric=exp} \) =
\( \text{semant}(13,-1); \)
numeric=exp: term
\( \text{numeric=exp} \) \( \text{+' term} \) =
\( \text{semant}(9,-3); \)
\( \text{numeric=exp} \) \( \text{-' term} \) =
\( \text{semant}(10,-4); \)
\( \text{+' term} \) =
\( \text{semant}(11,-1); \)
\( \text{-' term} \) =
\( \text{semant}(12,-1); \)
term: primary
\( \text{term} \) \( \text{+' primary} \) =
\( \text{semant}(7,-1); \)
\( \text{term} \) \( \text{/' primary} \) =
\( \text{semant}(8,-2); \)
primary: primary=element
\( \text{primary} \) \( \text{+' primary=element} \) =
\( \text{semant}(6,-1); \)
primary=element: numeric=ref
\( \text{number} = \)
\( \text{semant}(3,51); \) \( \text{numbers}[51].\text{use}=1; \)
bif

'(' numeric+exp ')
func+ref =
{ semant(1,$1); }

numeric+ref: numeric+id =
{ semant(1,$1); }
array+ref =
{ semant(2,$1); }

array+ref+head: array+id ')' =
{ semant(1,$1); 
  $3=$1; dfunar[++dfunar]=0; }
array+ref+head numeric+exp ')' =
{ $3=$1; dfunar[dfunar]++; semant(45,$2); }

array+ref+head: array+id '(' =
{ semant(1,$1); 
  $3=$1; dfunar[++dfunar]=0; }

string+ref: string+id =
{ semant(46,$1); }

substring+ref:
  string+array+ref
  array+substr+ref

string+ref+lo: string+id '(' =
{ semant(46,$1); }

substring+ref: string+ref+lo substring+spec

string+array+ref: string+ref+lo numeric+exp ')') =
{ if (symtable[31].type != 3) 
    error(symtable[31].symbol,msg[8]); 
    semant(50,-1); }

array+substr+ref: array+substr+lo substring+spec

array+substr+lo: string+array+ref '(' ;

substring+spec:
  numeric+exp ';' numeric+exp ')' =
{ semant(61,-1); }

bif: string+bif+ref string+exp ')' =
{ if (dfunar[do+funar-++]! = symtable[31].dimen) 
  { error(symtable[31].symbol,msg[6]); 
    semant(53,$1); 
  } 
  numeric+bif+ref numeric+exp ')' =
{ if (dfunar[do+funar-++]! = symtable[31].dimen) 
  { error
error(symtable[$1].symbol, msg[0]);
}  
\{ semant(53,$1); \}
\{ numeric+bif+no parm \}

string+bif+ref: string+bif '(' =
\{ $s = s1; dfunar[++dfunar] = 0; \}
\{ string+bif+ref string+ref ',$' =
\{ $s = $1; dfunar[dfunar++] = 0; \}

numeric+bif+ref: numeric+bif '(' =
\{ $s = s1; dfunar[dfunar++] = 0; \}
numeric+bif+ref: numeric+bif '(' =
\{ $s = s1; dfunar[dfunar++] = 0; \}
numeric+bif+no parm: numeric+bif
\{ semant(53,$1); \}

func+ref+head: func+ref+head numeric+exp ')' =
\{ if (dfunar[++dfunar] == symtable[$1].dimen)
\{ error(symtable[$1].symbol, msg[0]); \}
\{ semant(54,$1); \}

func+ref+head: function+id '(' =
\{ $s = s1; dfunar[++dfunar] = 0; \}
\{ func+ref+head numeric+exp ',$' =
\{ dfunar[dfunar++] = 0; $s = $1; \}

call+statement: call+head ')' =
\{ semant(66,$1); \}
\{ call+head =
\{ semant(66,$1); \}
\{ call+head numeric+exp ')' =
\{ oncct++; semant(63,$1-oncct); $s = $1; semant(66,$1); \}
\{ call+head array+id ')' =
\{ oncct++; semant(63,$1-oncct); $s = $1; semant(66,$1); \}
\{ call+head string+exp ')' =
\{ oncct++; semant(63,$1-oncct); $s = $1; semant(66,$1); \}
\{ call+head '& numeric+id ')' =
\{ oncct++; semant(63,$1-oncct); $s = $1; semant(66,$1); \}

call+head: call+head '(' =
\{ semant(64,-1); semant(65,$1); $s = $1; \}
\{ call+head '=' numeric+id '(' =
\{ semant(1,$3); semant(65,$3); $s = $3; \}
\{ call+head '=' numeric+id '(' =
\[
(\text{semant}(1, 3); \text{semant}(63, 3); 3s = 3s; ) \\
\text{call+head numeric+exo '1' } = \\
(\text{oncnt}++; \text{semant}(63, 3 - \text{oncnt}); 3S = 3s; ) \\
\text{call+head array+id '1', } = \\
(\text{oncnt}++; \text{semant}(63, 3 - \text{oncnt}); 3S = 3s; ) \\
\text{call+head string+exo '1', } = \\
(\text{oncnt}++; \text{semant}(63, 3 - \text{oncnt}); 3S = 3s; ) \\
\text{call+head 'A' numeric+id '1', } = \\
(\text{oncnt}++; \text{semant}(63, 3 - \text{oncnt}); 3S = 3s; )
\]

call+head: \text{CALL numeric+id } = \\
(\text{semant}(1, 3); 3S = 3S; )

\text{call+head: \text{CALL string+ref } = } \\
(\text{semant}(1, 3); 3S = 3S; )

\text{extern+statement: extern+head } \\
\text{extern+head: \text{EXTERN TYPE numeric+id parm+def } = } \\
(\text{symtable}[3].\text{length} = 2; 3S = 2; \\
\text{symtable}[3].\text{dimen} = \text{oncnt}; 3S = 2; \\
\text{symtable}[3].\text{type} = 10; \text{oncnt} = 0; ) \\
\text{EXTERN numeric+id parm+def } = \\
(\text{symtable}[2].\text{length} = 0; \\
\text{symtable}[2].\text{dimen} = \text{oncnt}; 3S = 0; \\
\text{symtable}[2].\text{type} = 10; \text{oncnt} = 0; ) \\
\text{EXTERN 'A' TYPE numeric+id parm+def } = \\
(\text{symtable}[4].\text{length} = 3 + \text{DISP; } \\
\text{symtable}[4].\text{dimen} = \text{oncnt}; 3S = 3 + \text{DISP; } \\
\text{symtable}[4].\text{type} = 10; \text{oncnt} = 0; ) \\
\text{extern+head '1' numeric+id parm+def } = \\
(\text{symtable}[3].\text{length} = 1; \\
\text{symtable}[3].\text{dimen} = \text{oncnt}; 3S = 1; \text{oncont} = 0; \\
\text{symtable}[3].\text{type} = 10; )

\text{parm+def: } \\
\text{'}(\text{'}\text{')}; \\
\text{parm+head TYPE '1'; } = \\
(\text{oncnt}++; \text{insert}("1"); \\
\text{symtable}[j].\text{length} = 2; \\
\text{symtable}[j].\text{type} = 10; ) \\
\text{parm+head '1' TYPE '1'; } = \\
(\text{oncnt}++; \text{insert}("1"); \\
\text{symtable}[j].\text{length} = 2 + \text{DISP; } \\
\text{symtable}[j].\text{type} = 10; ) \\
\text{parm+head: '}(\text{'}\text{')}, \\
\text{parm+head TYPE '1'; } =
{ oncnt++; j = insert("±±");
symtable[j].length = 2 * 2;
symtable[j].type = -1;
}
parm+head '8' TYPE ','
{ oncnt++; j = insert("±±");
symtable[j].length = 3 + 2 + CDISP;
symtable[j].type = -1;
}

#include "./bscan.c"
```c
semant(ca,i) int ca,i;
{ int k,k1;
    switch (ca)
    {
        case 1: printf("mov $S%d, -(r4)/n",i); return;
        case 2: j = symtable[i].type;
            if (j == 0)
                error(symtable[i].symbol, msg[5]);
            printf("a0b1 DOPCAL/n");
            printf("mov $S%d, -(r4)/n");
            printf("jsr cc, DOPCAL/n", i);
            return;
        case 3: printf("mov $V%d, -(r4)/n", i); return;
        case 6: printf("movf *(r4)+,fr1/n");
            printf("movf *(r4)+,fr0/n");
            printf("mov *(r4)+,fr0/n");
            printf("movf fr0, *(r4)/n");
            return;
        case 7: j = temocnt++ % 20;
            printf("mov $T%d, -(r4)/n", j);
            printf("mov fr0, *(r4)/n");
            return;
        case 8: j = temocnt++ % 20;
            printf("movf *(r4)+,fr1/n");
            printf("movf *(r4)+,fr0/n");
            printf("movf fr0, *(r4)/n");
            printf("mov *(r4)+,fr0/n");
            return;
        case 9: j = temocnt++ % 20;
            printf("mov *(r4)+,fr1/n");
            printf("mov *(r4)+,fr0/n");
            printf("mov fr1, fr0/n");
            printf("mov $T%d, -(r4)/n", j);
            printf("mov fr0, *(r4)/n");
            return;
        case 10: j = temocnt++ % 20;
            printf("movf *(r4)+,fr1/n");
            printf...
```
printf("movf *(r4)+,fr0\n");
printf("subf fr1,fr0\n");
printf("mov $1%a,*(r4)\n",j);
printf("mov fr0,*(r4)\n");
return;
case 11: j=numcnt++ % 20;
printf("movf *(r4)+,fr0\n");
printf("ahsf fr0\n");
printf("mov $1%a,-(r4)\n",j);
printf("movf r0,*(r4)\n");
return;
case 12: j=numcnt++ % 20;
printf("movf *(r4)+,fr0\n");
printf("mov $1%a,*(r4)\n");
printf("mov fr0,*(r4)\n");
return;
case 13: printf("mov $1f,-(r4)\nor 2+f\n");
printf("1: <\%s\0\n\neven\%r\",stig);
printf("2: mov $%o,-(r4)\n",stig1);
return;
case 14: printf(".globl strmv\n");
printf("jsr pc,strmv\n");
return;
case 15: printf("jmp \n"); return;
case 16: printf("jmr \n"); return;
case 17: printf("jsr pc,\n"); return;
case 18: printf("rts pc\n");
return;
case 19: j=numbers[i].numberi;
printf("\nL%d\n",j); return;
case 20: j=numbers[i].numberi;
printf("\nL%d\n",j); return;
case 21: oncnt=0;
printf("movf *(r4)+,fr0\n");
printf("movf r0,r3\n");
printf("dec r3\ncmp $0,r3\n");
printf("jgt 5f\n");
printf("jmp b+7\n": as) r3\n");
if (i == -1)
  printf("jmp *df(r3)\n");
else
  {
    printf("jsr pc,*df(r3)\n");
    printf("jmp 5f\n");
  }
printf("\n8:\n");
return;
case 22: oncnt++; return;
case 23: printf("\nb: cmp $%a,r3\n",oncnt);
printf("bge 7b\n5:\n");
return;
case 24: printf("jmp ENDER\n");
return;
case 25: printf("movf *(r4)+,fr0\n");
printf("movf *(r4)+,fr0\n\n");
printf("movf *,r3\n",i);
printf("movf fr0,fr0\n");
printf("cf#\n");
printf(".globl COMP\n");
printf("jsr pc,COMP\n");
return;

case 26:
printf(".globl cmmp\n");
printf("jsr pc,cmmp\n");
printf("movf %d,fr0\n");
printf("cmp %d,(r4)+\n");
printf(".globl COMP\n");
printf("jsr pc,COMP\n");
return;

case 27:
printf("tst (r4)+\nbeq 4\n");
return;

case 28:
printf("\n4: \n");
return;

case 29:
printf("\n9: \n");
return;

case 30:
j=numbers[i],numbers[i];
printf("L%d\n\n",j);
return;

case 31:
printf("jmp 9f\n4:\n");
return;

case 32:
printf("mov $DATA-%s,DATA\n");
printf("mov $STRDATA,STR\n");
return;

case 33:
printf(".globl danrd\n");
printf("jsr pc,danrd\n");
return;

case 34:
printf("jsr pc,FX%\n",i);
return;

case 35:
printf("jmp FX%\n\nF X%:\n");
return;

case 36:
symtable[i].dimen++;
k=i-symtable[i].dimen;
for(j=k; j<i; j++)
{
printf("movf *(r4)+,fr0\n");
printf("movf *,(r4)+\n");
}
return;

case 37:
printf("movf *(r4)+,fr0\n");
printf("movf *,(r4)+\n");
printf("rts pc\n\n\n\n",i);
printf("rts pc\n\n\n\n",i);
k=i-symtable[i].dimen;
for (j=k; j<i; j++)
symtable[i].symbol[0]=1;'
return;

case 38:
printf("movf *(r4)+,fr0\n");
printf("movf *,(r4)+\n");
return;

case 39:
j=forctr++;
printf("movf *(r4)+,fr0\n");
printf("movf $F1%-%(r4)-\n");
printf("movf fr0,*(r4)+\n");
return;
printf("movf fr1,\*(r4)+\nbr 1f\n");
printf("\nF6%a:\n",j);
printf("mov $S%a,-(r4)\n",i);
printf("mov $F1%a,-(r4)\n",j);
printf("movf \*(r4),fr1\n");
printf("movf \*(r4),fr0\n");
kl=looknf(1.);
if (kl == -1) { kl=insertnr(1,1.);
numbers[kl].luse=1; }
printf("addf N%a,fr0\n");
printf("movf r0,\*(r4)+\n",kl);
printf("cmpf fr0,fr1\n");
printf("cfcc \njat FS%a\n",j);
return;

case 40: 
j=forctr++;
printf("movf \*(r4)+,fr2\n");
printf("mov $F1%a,-(r4)\n",j);
printf("movf fr2,\*(r4)+\n");
printf("movf \*(r4),fr1\n");
printf("mov $F1%a,-(r4)\n",j);
printf("movf fr1,\*(r4)+\n");
printf("mov $S%a,-(r4)\n",i);
printf("movf \*(r4)+,fr0\n");
printf("mov $F1%a,-(r4)\n",j); 
printf("movf \*(r4)+,fr2\n");
printf("movf \*(r4),fr1\n");
printf("addf fr2,fr0\n");
printf("movf fr0,\*(r4)+\n");
kl=looknf(0.);
if (kl == -1) { kl=insertnr(0,0.);
numbers[kl].luse=1; }
printf("\n1:\ncmpf \%a,fr2\n");
printf("cfcc\njat F5%a\n",x1);
printf("cmpf fr0,fr1\nmcfcc\n");
printf("\n\n1:\ncmpf fr0,fr1\nmcfcc\n");
printf("\n\n1:\ncmpf fr0,fr1\nmcfcc\n");
printf("cfcc\njlt F5%a\n",j);
printf("\n3:\n\n");
return;

case 41: 
printf("\n\n1:\ncmpf F6%a\nF5%a\n\n",i,i);
return;

case 42: 
printf("\n\n1:\ncmpf Fr\n\n",i,i);
printf("\n\n1:\ncmpf fr\n\n",i,i);
printf("\n\n1:\ncmpf fr\n\n",i,i);
printf("\n\n1:\ncmpf fr\n\n",i,i);
return;

case 43: 
printf("\n\n1:\ncmpf fr\n\n",i,i);
printf("\n\n1:\ncmpf fr\n\n",i,i);
printf("\n\n1:\ncmpf fr\n\n",i,i);
printf("\n\n1:\ncmpf fr\n\n",i,i);
return;
case 44: printf("\t.globl lindmo\n");
    printf("\t.jsr pc,lindmo\n");
    return;

case 45: printf("\t.movf *(r4)+,fr0\n");
    printf("\t.movfi fr0,-(r4)\n");
    return;

case 46: printf("\t.mov $%a,-(r4)\n",i);
    k=symtable[i].length;
    printf("\t.mov $%a,-(r4)\n",k);
    return;

case 48: printf("\t.globl nbrror,atof\n");
    printf("\t.jsr pc,nbrror\n");
    printf("\t.jsr pc,atof\n");
    printf("\t.movf fr0,*(r4)+\n");
    return;

case 49: printf("\t.globl strror\n");
    printf("\t.jsr pc,strror\n");
    return;

case 50: printf("\t.mov $%a,-(r4)\n",
    numbers[i].number*2);
    return;

case 51:
    printf("\t.mov $1f,-(r4)\nor 2f\n");
    printf("\t.1: %s\n,even\n", stdin);
    printf("\t.2: mov $%a,-(r4)\n",
    numbers[i].number*2);
    printf("\t.globl OPEN\n jsr pc,OPEN\n");
    return;

case 52: printf("\t.mov $%a, -(r4)\n",
    numbers[i].number*2);
    printf("\t.globl CLOSE\n");
    printf("\t.jsr pc,CLOSE\n");
    return;

case 53: switch(symtable[i].length) {
    case 0: // std calling fru and jsr pc, x
    if(symtable[i].amt == 0)
        { printf("\t.globl %s\n",
            symtable[i].symbol);
            symtable[i].amt++;
            }
    printf("\t.movf *(r4)+,fr0\n");
    printf("\t.jsr pc,%s\n",
            symtable[i].symbol);
    j=tempcnt++ % 20;
    printf("\t.mov $%a, -(r4)\n",j);
    printf("\t.movf fr0, *(r4)\n");
    return;

case 1: // ABS
    printf("\t.movf *(r4),fr0\n");
    printf("\t.ansf fr0\n");
    printf("\t.movf fr0, *(r4)\n");
    return;

case 2:
if(symtable[i].amt == 0)
  { printf(".globl %s,atof\n", symtable[i].symbol);
    symtable[i].amt++; }
printf("jsr oc,%s\n", symtable[i].symbol);
printf("jsr oc,atof\n");
printf("mov $T%d,-(r4)\n",j);
printf("movf fr0,* (r4)\n");
return;
case 3:
if(symtable[i].amt == 0)
  { printf(".globl %s\n", symtable[i].symbol);
    symtable[i].amt++; }
printf("jsr oc,%s\n", symtable[i].symbol);
printf("jsr oc,atof\n");
j=tempcnt++ % 20;
printf("mov $T%d,-(r4)\n",j);
printf("movf fr0,* (r4)\n");
return;
case 5:
  // chr§ unique because $ not valid
  // in as
if(symtable[i].amt == 0)
  { printf(".globl %s\n","chr");
    symtable[i].amt++; }
printf("jsr oc,chr\n");
return;
case 8:
if(symtable[i].amt == 0)
  { printf(".globl %s\n", symtable[i].symbol);
    symtable[i].amt++; }
printf("jsr oc,%s\n", symtable[i].symbol);
    symtable[i].symbol);
return;
default: return;
}
case 54:
    printf(".globl datadr\n");
    printf("jsr oc,datadr\n");
return;
case 55:
  if (i == -1)
    printf("mov $1301, r0\n");
else
    printf("movf *(r4)+,fr0\n");
    printf("movfi fr0,r0\n");
    printf(".globl srand\n");
    printf("jsr oc,srand\n");
    return;
case 56:
    printf(".globl XOR\n");
133
printf("jsr oc,OR\n");
return;
case 57:
printf(".globl OR\n\njsr oc,OR\n");
return;
case 58:
printf(".globl AND\n");
printf("\nsr oc,AND\n");
return;
case 59:
printf(".globl NOT\n");
printf("jsr oc,NOT\n");
return;
case 60:
printf(".globl SDCAL\n");
printf("jsr oc,SDCAL\n");
return;
case 61:
printf(".globl SUBSTR\n");
printf("jsr oc,SUBSTR\n");
return;
case 62:
printf("mov %e, -(r4)\n\nbyte 012,0;\n");
printf("\neven\n2:n\n");
printf(".globl strmp\n");
printf("\njsr oc,strmp\n");
return;
case 63:
printf("mov $%o, -(r4)\n", symtable[i].length);
return;
case 64:
printf("clr -(r4)\n");
return;
case 65:
if (ocnt != symtable[i].dimen)
  error(symtable[i].symbol, memo);
if (symtable[i].amt == 0)
  ( symtable[i].amt = 1;
  printf(".globl +%s\n", symtable[i].symbol);
  printf("\nmov %s, -(r4)\n", symtable[i].dimen);
  printf(".globl CSET\n");
  printf("\nsr oc,CSET\n");
  printf("\njsr oc,\n", symtable[i].symbol);
  printf(".globl CRET\n");
  printf("\njsr oc,CRE\n");
  return;
return;
case 66:
printf("\nmov (r4)+,r3\n");
printf("\nmov (r4)+,r2\n");
printf("\nmov r2, -(r4)\n");
printf("mov r3, -(r4)\n");
return;
case 68:
printf("mov \$%o, -(r4)\n", 
numbers[i].numbi*2);
printf("globl READF\n");
printf("jsr oc,READF\n");
return;
case 69:
printf("globl READFN,atof\n");
printf("jsr oc,READFN\n");
printf("jsr oc,atof\n");
printf("movf fr0,*(r4)+\n");
return;
case 70:
printf("globl READFS\n");
printf("jsr oc,READFS\n");
return;
case 71:
printf("globl READFE\n");
printf("jsr oc,READFE\n");
return;
case 72:
printf("globl WRITFN\n");
printf("jsr oc,WRITFN\n");
return;
case 73:
printf("globl WRITFS\n");
printf("jsr oc,WRITFS\n");
return;
case 74:
printf("globl WRITFE\n");
printf("jsr oc,WRITFE\n");
return;
case 75:
printf("mov \$%o, -(r4)\n", 
numbers[i].numbi*2);
printf("globl WRIT\n");
printf("jsr oc,WRIT\n");
return;
}
caldope(i,j,k) int i,j,k;
{int il; dope[dopeot]=1;
symtable[i].amt=1;
for (il=dopeot-1; il < k; il--)
{ symtable[il].amt *= (dope[il]+1);
  dope[il]=* dope[il+1];
}
symtable[il].amt *= (dope[k]+1);
dope[k]=j;}

135
# define TRUE 1
# define FALSE 0
# define ERRORFILE 2
# define SYMSIZE 200
# define NUMSIZE 200
# define NAMELENGTH 14
# define SIMLEN 10
# define CDISP 10
# define MAXFOP 10

char *filein[518];
char *filn = filein;
extern int fout;
int fileout;
char *filnam [NAMELENGTH+1];

/* this array defines the function of each character in
the ASCII character set for use in yylex

-2  =  eof and end token for yacc
-1  =  illegal characters to be deleted
  0  =  blanks and tabs to be discarded
  1  =  newline -- used to update line counter
  2  =  legal special characters
  3  =  all letters and the dollar sign '$'
  4  =  digits and the decimal point '.'
  5  =  quote -- used to delimit strings and
       deleted ""'
  6  =  "" -- continuation
*/

int chartype [128] {  
  -2,-1,-1,-1,-1,-1,-1,-1, -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, -1,-1,-1, -1,-1,0,2,5,-1,3,-1,  
  2,-1,2,2,2,2,2,2,4,2,2,4,4,4,4,4,4,4,4,4,4,4,4,4,  
  -1,2,2,2,-1,-1,  
  2,-1,2,2,-1,-1,  
  -1,2,-1,2,-1);  

int yylex 1,errnoCnt 0; // glob line cntr for yacc
int conflag 0,exflag 1; // cont and external ref flag
int ci; // the global next character
char stig [256]; // the global string collector
int stigl; // the length of the string literal

136
```c
int dimv 0, eofflag 0; int defy 0;
int funcnt 0; // global flags and counter
int j;     // general temp
int fds[15];     // fds for READ/WRITE FILE

struct {
    double numberf;
    int numberi;
    int use;
    int luse;
    int dec;    } numbers[NUMSIZE];

/* numbers is a structure used to hold literal numbers */
dec = -1 floating point declaration
    0 integer declaration
luse = 1 used as number
use = 1 used only as label
    2 used as statement label
numberf = floating point values
numberi = integer value
*/

int numbernt 0;     // index of numbers
int dope[200];      // vector used to hold dope values
int dopeot;        // next available dope position

struct {
    char symbol[SIMLEN];
    int type;
    int dimen;
    int length;
    int amt;
    int dopev;    } symtable [SYMSIZE];

/* symtable is a structure used as a symbol table */
symbol = identifier value
    type = -1 null oars of extern variables
    = 0 numeric id
    1 numeric array
    2 string id
    3 string array
    4 function
    5 numeric bif
    6 string bif
    7 simple format
    8 numeric format
```
9 numeric string bif
10 external variables

dimen = dimension of array
length = length of a string
dopev = index of the first element of
the arrays dope vector in dope
amt = use for bif's 1=used 0=unused
number of elements in numeric array
number of bytes in a string array

*/

int sympt SYMSIZE-1; // pointer into the symbol table
int tnum, tsym; // temporary is structures
int RWBASE; // base of reserve words in symbol table

int forcnt 0; int tempcnt 0, maxtemp -1;
int maxfor -1; int oncnt 0;
int forctr 0;

*/
forcnt = current depth of nested for loops
tempcnt = value used to manage the temporary
pool used as tempcnt%20
maxfor = maximum number of for loops nested
to this point in the program --
used to determine number of for loop
variables needed
oncnt = count of ON statement label
forctr = current count of all FOR's used --
used for label definition

*/

int dfunar[20]; int defunar 0;

int datact 0; // pointer to the next data value
double data[100]; // data list to be used as data to READ

char *msa [] {
  "**ERROR** attempt to redefine a numeric id as array ",
  "**ERROR** attempt to redefine an array id ",
  "**ERROR** attempt to redefine a string ",
  "**ERROR** attempt to redefine function ",
  "**ERROR** attempt to redefine built in function ",
  "**ERROR** attempted use of numeric id as array ",
  "**ERROR** incorrect number of parameters ",
};
"**ERROR** illegal use of external name",
"**ERROR** illegal use of string id as string array",
0
);

extern char *bifs[];
int biftype[] {
  5,5,5,5,5,5,5,5,5,6,6,9,5,5,5,5,5,8,8,7,7,1
};
int bifact [] {
  0,0,3,0,3,0,0,3,3,0,3,0,3,0,3,0,8,8,-1,1,-1
};

/* variables for data string collection */

char datastring[400];
char *datastart &datastring[0];

#define SYMSIZE 200
#define NUMSIZE 200
#define SIMLEN 10

struct numbers {
  double numberf;
  int numberi;
  int use;
  int luse;
  int dec;
};
extern struct numbers numbers[];

/* numbers is a structure used to hold literal numbers */

dec = -1 floating point declaration
  0 integer declaration
luse = 1 used as number
use = 1 used only as label
  2 used as statement label
numberf = floating point values
numberi = integer value

*/

struct symtable {
  char symbol[SIMLEN];
  int type;
  int dimen;
  int length;
  int amt;
  int dopv;
};
extern struct symtable symtable[];
/* symtable is a structure used as a symbol table */

symbol = identifier value
type = -1 null parms of external variables
      = 0 numeric id
      = 1 numeric array
      = 2 string id
      = 3 string array
      = 4 function
      = 5 numeric bif
      = 6 string bif
      = 7 simple format
      = 8 numeric format
      = 9 numeric string bif
      = 10 external variables
dimen = dimension of array
length = number of parameters for function
dopev = index of the first element of
      the arrays dope vector in dope
amt = use for bif's 1=used 0=unused
      number of elements in numeric array
      number of bytes in a string array

/*

the following definitions are the reserve words of BASIC
reservewords = capital spellings
lreservewords = lower case spellings
note the '+' inserted to all texts
  to allow "C" to process the values
  of its own reserve words.

*/

char *reservewords [] =
"STEP",
"GO",
"IF",
"ON",
"TO",
"DEF",
"DIM",
"END",
"FOR",
"LET",
"REM",
"DATA",
"ELSE",
"GOTO",
"FILE",
"NEXT",

140
"READ",
"OPEN",
"STOP",
"THEN",
"GOSUB",
"INPUT",
"PRINT",
"CLOSE",
"WRITE",
"RETURN",
"RESTORE",
"SUB",
"RANDOMIZE",
"EQ",
"LT",
"GT",
"GE",
"LE",
"NE",
"REMARK",
"CALL",
"EXTERN",
"INTEGER",
"FLOAT",
"DOUBLE",
"CHAR",
"ADDR",
"OR",
"XOR",
"NOT",
"AND",
0

char *1 reservewords [] { 
    "step",
    "go",
    "if",
    "on",
    "to",
    "def",
    "dim",
    "end",
    "for",
    "let",
    "rem",
    "data",
    "else",
    "goto",
    "file",
    "next",
    "read",
    "open",
    "ston";
}
"+then",
"+gosub",
"+input",
"+print",
"+close",
"+write",
"+return",
"+restore",
"+sub",
"+randomize",
"+eq",
"+lt",
"+gt",
"+ge",
"+le",
"+ne",
"+remark",
"+call",
"+extern",
"+integer",
"+float",
"+double",
"+char",
"+addr",
"+or",
"+xor",
"+not",
"+and",
0
}

char *nifs[] { "atan",
"exp",
"mod",
"log",
"rand",
"sin",
"cos",
"sort",
"tan",
"len",
"asc",
"chr5",
"cosh",
"int",
"sinh",
"val",
"rad",
"deg",
"tan",
"col",
SCANNER

/* the following are the user defined functions required to provide scanning */
yylex()
{} extern int yyval; // this value is used to return values to yacc
double atof();
char id[10], numstr[50]; int i, k, l; double d, b;
while (TRUE) // do forever or until return
{
switch(chartype(c))
{
  /* eof and the end
   token for yacc -- case -2
   illegal characters -- case -1
   blanks -- case 0
   newline -- case 1
   legal specials -- case 2
   letters -- case 3
   digits and decimal -- case 4
   strings -- case 5
   continuation -- case 6
  */
default:
case -2: return (c); // if we get here
          // we'd better be done
  case -1: id[0]=c; id[1]=\0';
           error(id, "illegal character deleted");
           c=getc(filein); break;
           // throw away illegal characters
  case 0: c=getc(filein); break;
          // blanks thrown away
  /* just update the line counter and return newline to yacc */
  case 1:
    yyline++; i=c; c=getc(filein);
    if (eofflag) c=0;
    if (! conflag)
      { exflag=1; dimv=0; defv=0;
        if (tempcnt > maxtemp)
          { maxtemp = tempcnt;
            tempcnt=0; return(i); }
        conflag=0; break;
          // continuation on next line
      case 2: i=c; c=getc(filein);
            // return the legal character as is
            return(i);
      case 3: i=0; id[i++] = c;

    "}
/collect id's and reserved words

```c
int i;
int char_type;
char id[6];
c = getc(filein);
while (!c && i < 9) {
    id[++i] = c;
c = getc(filein);
}
id[i] = '\0';
// pad null to end string

switch (lookrs(id)) {
    case 0: return (STEP);
    case 1: return (GO);
    case 2: return (IF);
    case 3: return (ON);
    case 4: return (TO);
    case 5: defv=1; return (DEF);
    case 6: dimv=1; return (DIM);
    case 7: eofflag=0; return (END);
    case 8: return (FOR);
    case 9: return (LET);
    case 10:
    case 11: return (DATA);
    case 12: return (ELSE);
    case 13: return (GOTO);
    case 14: return (FILE);
    case 15: return (NEXT);
    case 16: return (READ);
    case 17: return (OPEN);
    case 18: return (STOP);
    case 19: return (THEN);
    case 20: return (GOSUB);
    case 21: return (INPUT);
    case 22: return (PRINT);
    case 23: return (CLOSE);
    case 24: return (WRITE);
    case 25: return (RETURN);
    case 26: return (RESTORE);
    case 27: return (SUB);
    case 28: return (RANDOMIZE);
    case 29: yylval=0; return (relspec);
    case 30: yylval=12; return (relspec);
    case 31: yylval=8; return (relspec);
    case 32: yylval=20; return (relspec);
    case 33: yylval=16; return (relspec);
    case 34: yylval=4; return (relspec);
    case 35: return (LET);
    case 36: oncnt=0; eflag=0;
}
```
return(CALL);
case 37: oncnt=0; exflag=0;
    return(EXTERN);
case 38:
case 39:
case 40:
case 41:
case 42:
    zy1val=1-36; return(TYPE);
case 43: return(OR);
case 44: return(XOR);
case 45: return(NOT);
case 46: return(AND);
}

/* not a reserve word look for an ID if length ok */
if (j >= 5 && exflag) // 4 char limit on std ids
    error(id,
    "illegal ID name -- numeric ID used");
    id[i] = '\0'; i = lookup(id);
else i = lookup(id);

/*
Any ID which conforms to normal BASIC ID definitions is acceptable -- thus the following forms are recommended

numeric id's    -- letter
    letter digit

string id's     -- letter '3'
    letter digit '3'

function id's   -- FN letter
    FN letter digit

These forms are recommended however the following are the restrictions which are enforced.

1) length 1-4 characters
2) the id must begin with a letter, upper or lower case
3) rules for the recognition of types

    numeric id's    wxyz
        w = F,f
        x = N,n
        y = $  

    string id's     wxyz
        w = F,f
        x = N,n

146
\[ x \text{ or } y \text{ must } = \frac{3}{n} \]

function id's \( \text{wxvz} \)

\( w \text{ must } = F, f \)
\( x \text{ must } = N,n \)

4) id's may mix upper and lower case freely

*************************************************************************

NOTE

reserve words are acceptable as entirely UPPERCASE or LOWERCASE, however they may not be MIXED!

*/

if (i != -1)  // return type if predeclared
  if (dimv==1 || defv==-1)
    switch(symtable[i].type)
      {  
      case 0:
        case 10: if (defv == 1)
          { yyval=insert(id);
            return(numeric+id);}
        error(id,msg[0]);
            return(numeric+id);
        case 1: error(id,msg[1]);
            return(array+id);
        case 2: case 3: error(id,msg[2]);
            return(string+id);
        case 4: error(id,msg[3]);
            return(function+id);
        case 5: case 6:
        case 7: case 8: case 9: error(id,msg[4]);
        default: error(id,msg[7]);
      }
      yyval=i;
      switch (symtable[i].type)
      {  
      case 0:
        case 10: return (numeric+id);
        case 1: return (array+id);
        case 2: return (string+id);
        case 3: return (string+id);
        case 4: return (function+id);
        case 5: return (numeric+if);
        case 6: return (string+if);
        case 7: return (simple+format);
        case 8: return (numeric+format);
        case 9: return (str+num+if);
      }

147
/* check for a function definition FN, Fn, fN, fn */

    if ((id[0] == 'F' || id[0] == 'f') &&
        (id[1] == 'N' || id[1] == 'n'))
    {i=insert(id); yyval=i;
     symtable[i].type=4;
     return (function+id);}

/* not a function -- a string id?? xs, xy$ */

    if (id[1] == 'S' || id[2] == 's')
    {i=insert(id); yyval=i;
     if (dimv != 1) {
       error(id,
       "**WARNING** undefined string id");
       error(id,"assigned default length 1n");
       errorcnt = errorcnt+2;
       // back out error on warning
     symtable[i].length=16; }
     symtable[i].type=2;
     return (string+id);}

/* not function or string must be numeric */

    i=insert(id); yyval=i;
    return (numeric+id);

    case 4: d=0; b=.1; j=0; i=0;
    // numbers fall here

/* does the number begin with a decimal point ?? */

    numstr[j++]=c; if (c=='.') i=1;
    c=getc(filein);

while(chartype[c] == 4 & 3 < j<49)
    if (c == '.')
    {numstr[j++]=c; c=getc(filein); }
    else if (i==1) { break; } 
    else { i=1; numstr[j++]=c;
     c=getc(filein); }

if (i == 5) numstr[j] = '\0';
else if (j>5 && numstr[0] >= '3')
    {i=1; numstr[j++]=.'; 
     numstr[j] = '\0'; }
else { numstr[j] = '\0'; }

    if (i == 0) { j = atoi(numstr); k=lookup(j);
    // declared as integer lookup
    if (k == -1)
    { d=j; k=insertnr(j,d);
     numbers[k].dec=0; }
    }
else { d=atof(numstr); 
148
/declared as real lookup
j=d; k=looknf(d);
if (k == -1)
  { k=insertnr(j,d);
      numbers[k].dec= -1;  }
/* return index in number table
in yylval and return number */
yylval=k; return (number);

//  case 5: stigl=0;  // strings fall here
while (((c=getc(filein)) != '"
  && stigl < 256)
stig[stigl++]=c;
  // collect the string in stig
c=getc(filein); stig[stigl]='$0';
  // put in the null for string
return (string);
  case 6: // continuation
  conflag=1; c=getc(filein); break;
  // flag on nextchar
default: return (0)
  /* end of yylex */
}}

vyinit (argc,argv) int argc; char **argv; { int ij;
  j=0;
  ij=0;
  while(argc[i][j] != '/0' && ij < NAMELENGTH - 2) { ij=0; j++;}
  // set filename back
  else fnam[i++]=argv[i+];

  if( ! (filnam[i-1] == 'b'
     && filnam[i-2] == '.') || i<3)
    ( error("file type??",0); exit(1); )
  filnam[i-1]='s';
  filnam[i]='/0';
  fout = creat(filnam,0666);
  if (fout == -1)
    ( error(filnam,"can not open??"); exit(1); )
  printf(".global +main\n\n.text\n\n+main:\n\n\n");
  printf("setd\nmov $STACK\n\n");
  semant(32,-1);
  semant(55,-1);
  for(tnum=0; bifs[tnum] != 0; tnum++)
    { j=insert(bifs[tnum]);
      symtable[j].tvoe = bitype[tnum];
      symtable[j].dimen = 1;
      symtable[j].length = bifact[tnum];
    }
#include "mod"; symtable[j].dimen=2;
j=lookup("mod"); symtable[j].dimen=2;

if (fopen(filename) == -1)
{ error("can not open arq1",0);
  unlink(filename); exit(1);
}
c=getc(filename);

// called first by yacc
c getopt();
int k,l,m,n; char *dataptr; double d; d=0;
printf(".globl DATCNT,DATA,DATAMN"");
printf("DATCNT: 0\n");
printf("DATA: \n");
for (j=0; j < datalength; j++)
  numrecv(&data[j]);
printf("DATAMN: 0; 0; 0; 0\n");
printf("STRMEXT: .+,2\nSTRMEND: \n");
if (dataptr != datastring(0)) printf("\n");
else printf("\n");
for (dataptr = &datastring(0);
dataptr < &data; dataptr++)
  if (*dataptr != '0') printf("0
");
else putchar(*dataptr);

printf("strnc: byte 0; byte 0; .even\n");
for (j=0; j < numberof; j++)
  if (numbers[j].dec != 0 || numbers[j].luse == 1)
    printf("%d: ",j);
  numrecv(&numbers[j].numberf);
for (j=symptable[j].type; j < RWBASE; j++)
  switch (k)
  {
    case 0:
    case 10:
      printf("%d: 0; 0; 0\t/ %s\n",
          numbers[j].symbol);
      break;
    case 1:
      l=symtable[j].number;
      m=symtable[j].dimen;
      printf("30%0: %0\t/ %s\n",
          numbers[j].symbol);
      for (n=l+1; n < l+m; n++)
        if (k==1) dopp[n]=0;
        printf("%0\n",
            dopp[n]);
      break;
    case 2:
      l=symtable[j].length-1;
      printf("%d: 0; .+,%c\n",j,1);
      printf(".even\t/ %s\n",
          symtable[j].symbol); break;
    case 4:
      printf("%d: 0; 0; 0\t/ %s",j);

    case 4:
      printf("%d: 0; 0; 0\t/ %s",j);
}
printf("/%s\n", symtable[j].symbol);
    break;
}

j=0;
for (k=0; k<15; k++)
    if (fdisk[k] != 0)
        if (fdisk[k] == 1) j++;
        else
            error("more files referenced than opened", 0);
    if (j!=0)
    {
        printf("BUF: =.+%s\n", j*518);
        printf(".global FD,FD0\n\n");
        for(k=0; k<15; k++)
            if (fdisk[k] != 0)
                printf("\tBF: =.+%s\n", j*518*(1++));
            else
                printf("\tBF: =.+\n\n");
        printf("FD0: =.+30..\n");
        printf(".text\n.globl FCLOSE\n\n");
        printf(".jsr rc,FCLOSE\n");
        printf(".sys exit\n");
    }
    else
        printf(".text\n\n");
printf(".bss\n");
printf("STACKTOP: =.+50..\nSTACK: =.+2..\n");
for (j=0; j < forcnt; j++)
    printf("FM%d: =.+%s\n", j, j);
if (maxtemp < 20) k=maxtemp; else k=20;
for (j=0; j<50; j++)
    printf("T%d: =.+%s\n", j);
for (j=sym+1; j < R*N; j++)
    { k=symtable[j].type;
      l=symtable[j].amt;
      switch(k)
      {
      case 1: l=i*8; printf("S%o: =.+%o\t/t/%s\n", 
                       j, l, symtable[j].symbol);
              break;
      case 3: printf("S%d: =.+%o/.even\t/t/%s\n", 
                       j, l, symtable[j].symbol);
                }
      }

151
# define ERRORFILE 2
# include ".\bstruc.h"
extern int errorcnt, fileout, fout, yyline, base, 
numberpt, sympt, j;
extern char filnam[];
main (argc, argv)
int argc;
char *argv[];
{ 
yyinit(argc, argv);
if(yyparse() || errorcnt > 0)
{ unlink(filnam); exit(1); }
yyaccot();
flush();
exit(0);
}
compar(s1, s2) // compares two strings returns 0 if n.e.
char *s1, *s2;
{ 
while (*s1++ == *s2)
if (*s2++ == '0') return (1);

return (0);
}
strcopy(s,t) // this procedure copies strings
char *s, *t; { 

while(*t++ = *s++);
}
numbrcv(st) int *st [];
{ int i;
for (i=0; i<3; i++)
printf("%d ", st[i]);
printf("%d 0, st[3]);
}
error(x, y) char *x, *y;
{ 
flush();
fileout=fout;
fout = ERRORFILE;
if (v == 0)
printf("153s0,x");
else
  printf("153d: s: %s, line %d, yline, x, y");
flush();
fout = fileout;
errorcnt++;}

yyerror(s) char *s; {
extern int yvchar;
extern char *yystrlen[];
flush();
fileout=fout;
fout=ERRORFILE;
printf("153s", s);
if(yvline) printf(" line %d", yvline);
printf(" on input: ");
if(yvchar>=0400)
  printf("%s0, yystrlen[yvchar-0400] ");
else switch (yvchar) {
  case ' ': printf("\t0"); break;
  case '0': printf("\n0"); break;
  case '0': printf("\send0"); break;
  default: printf("%c0, yvchar"); break;
}
errorcnt++;flush();
fout=fileout;
}

lookup(s) char *s; { // this procedure validates id's int i; // returning -1 or symboltable index
  for (i=sympt+1; i<RWBASEx; i++)
    if (compare(s,symtable[i].symbol) > 0) return (i);
/* handle upper and lower case reserved words */
  for (i=RWBASEx; i<SYMSIZE; i++)
    if (compare(s,symtable[i].symbol) > 0)
      bifcompare(s,i) > 0) return(i);
return (-1);
}

bifcompare(s,i) char *s; int i;
{ // check bifs by translating all lowercase to uppercase
  // returns index or -1 if no match
  int k,l,k;
  char t [SIMLEN];
  k1='a'-'A'; // differences between uppercase and lowercase
  for (k=0; s[k] != '0'; k++)
    t[k] = s[k] + k1;

t[k] = '0';

return(compar(t, symtable[i].symbol));

lookrs(str) /* reserved word lookup -1 is not found */
char *str; {
int i;

for (i=0; reservewords[i] != 0; i++)
    if (compar(str, reservewords[i]) ==
        compar(str, &reservewords[i][1])) return(i);
return (-1);
}

looknf(nf) //locates numbers declared as real
double nf; {
int i;

for (i=0; i < numberot; i++)
    if (numbers[i].numberf == nf) return (i);
return (-1); // return -1 for not found
}

lookni(ni) // this procedure locates numbers
            // declared as integer
int ni; {
int i;

for (i=0; i < numberot; i++)
    if (numbers[i].numberi == ni) return (i);
return (-1); // return -1 for not found
}

insert(cc) // this procedure inserts new id's and
char *cc;  // zeros all entries --
j=symot--;  // returns index in table
if (j<0)
    ( error("fatal error -- symbol table overflow",
            "compilation terminated");
        unlink(filnam); exit(1);
)
strcopy(cc, symtable[j].symbol);

symtable[j].type=0;
symtable[j].dimen=0;
symtable[j].length=0;
symtable[j].dopv=0;
return (j);
insertnr(j1, d) // this procedure adds new numbers to the
double d; int j1; // number table -- zeroing all entries
{ int i; // returns the index in the table

  i = numberpt++; if (i >= NUMSIZE)
    { error("fatal error -- number table overflow",
              "compilation terminated");
      unlink(fnam); exit(1);
    }

  numbers[i].numberf = d;
  numbers[i].numberi = j1;
  numbers[i].use = 0;
  numbers[i].luse = 0;
  numbers[i].dec = 0;
  return (i);
}
LIBRARY

.globl COMPAR, AND, OR, XOR, NOT
.text
COMPAR:
    jmp 3f(r3)

3:
    beq 4f
    br 5f
    bne 4f
    br 5f
    bgt 4f
    br 5f
    blt 4f
    br 5f
    ble 4f
    br 5f
    bge 4f

5:
    clr -$r4) // FALSE into stack
    rts ac

4:
    mov $1,-(r4) // TRUE into stack
    rts ac

NOT:
    tst r4) // TRUE or FALSE in stack?
    beq 1f // TRUE before set FALSE
    clr (r4) // TRUE before set FALSE
    rts ac

1:
    mov $1,(r4) // FALSE before set TRUE
    rts ac

AND:
    cmp (r4)+(r4)
    bne 1f
    rts ac // both the same so AND is correct

1:
    clr (r4) // different AND => FALSE
    rts ac

OR:
    cmp (r4)+(r4)
    bne 1f
    rts ac // both the same so OR is correct

1:
    mov $1,(r4) // different OR => TRUE
rts pc

XOR:
cmp (r4)+(r4)
beq 1f
mov $1,(r4) // different XOR => TRUE
rts oc

1:
clr (r4) // same XOR => FALSE
rts oc
.globl CSET,CRET
.globl endhalt,ERROR,strmv
.text
CSET:
    mov (r4)+,r3
    mov (sp)+,r0
    mov sp,stacksave
    / save old stack pointer for chop
    cmp $0,r3
    beq 3f
    mov $here,r5
1:
    mov (r4)+,r2
    jmp *2f(r2)
here:
    sub r3,1b
3:
    jmp *r0
2:     / table of actions
    intval
    floatval
    dbival
    charval
    special
    intarray
    floatarray
    dbarray
    charstring
    special
    intval:
        movf *(r4)+,fr0
        movfi fr0,r2
        mov r2,-(sp)
        jmp *r5
    floatval:
        movf *(r4)+,fr0
        setf
        movf fr0,-(sp)
        setd
        jmp *r5
    dbival:
        movf *(r4)+,fr0
        movfr0,-(sp)
        jmp *r5
    dbarray:
        tst (r4)+
        /throwaway dopedvector info
        mov (r4)+,-(sp)
        / put address in stack
        jmp *r5
    floatarray:
    intarray:
       jsr r5,ERROR

158
charval:
tst (r4)+  / throw away length
movb *(r4)+,-(sp)
jmp *r5

charstring:
tst (r4)+  / throw away length
mov (r4)+,-(sp)
jmp *r5

special:
mov *(r4)+,-(sp)
jmp *r5

CRET:
mov (sp)+,r1
mov stacksave,sp
setd
mov *2(r4)
beq 1f
mov (r4)+,r2
jmp *2f(r2)

2:
intret
floatret
dblret
charret
secret
intotr
floatotr
dblotr
charptr
secret

intret:
tst (r4)+  / throwaway dummy
movf r0,fr0
movf fr0,*(r4)+
jmp *r1

floatret:
tst (r4)+  / throwaway dummy
setf
clrf 4(r4)
movf fr0,*(r4)+
setd
jmp *r1

dblret:
tst (r4)+  / throwaway dummy
movf fr0,*(r4)+
jmp *r1

charret:
tst (r4)+ / throw away dummy
movb r0,(r4)
movb $'\0',1(r4)
cmp (r4)+,(r4)+ /throw away old length and addr
jmp *r1

charot:
tst (r4)+ / throw away dummy
mov (r4)+,r3 / get address
mov (r4)+,r2 / get old length
mov r3,-(r4) / restore address on bottom
mov r2,-(r4) / restore length on top
mov r0,-(r4) / new string address
mov $777777,-(r4) /dummy len to force use of old len
jsr oc,strmv
jmp *r1

secret:
tst (r4)+ / throw away dummy
mov r0,* (r4)+ / move pointer into place
jmp *r1

inptr:
floatptr:
dblptr:
jsr r5,ERROR

***ERROR** unimplemented call option\0> ; even

1: / procedure calls come here clean stack

charouse = 0
charouse = 16

cmp (r4),$charouse / check for char call with 4 parms
beq 2;
cmp (r4)+,$charouse / check for char call with 4 parms
beq 2f;
cmp (r4)+,(r4)+ / throwaway unneeded function adrs
jmp *r1

2:
cmp (r4)+,(r4)+ / throwaway unneeded function adrs
tst (r4)+ / throwaway unneeded function adrs
jmp *r1

stacksave: .=.+2
DOPCAL.s

.globl DOPCAL
.text
DOPCAL:

    mov $0,dctmp
    mov *(r4),r0       /get number of subscripts
    mov (r4)+,r1       /move address of dope vector into r1
    inc r1
    inc r1
    mov $1,r2          /the first displacement is 1

TOPC:

    mul (r4)+,r2
    add r3,dctmp
    mov (r1)+,r2
    sub r0,TOPC
    mov dctmp,r3
    add (r4),r3        /add in the case of the array
    mov r3,(r4)        /leave the address in the stack
    rts pc

.data
dctmp:  0       / temporary for calculation
.globl ERROR
ERROR:
1:
  mov  $2, r0
  movb (r5)+, erch
  beq  2f
  sys  write;  erch:  1
  br  1b
2:
  sys  exit
erch:  .=.+2
OPEN:
  mov  (r4)+,r1   /mode
  mov  (r4)+,r0   /address of name
  mov  r0,r2
  mov  $FDO,r3   /open flags
  add  (r4),r3   /select correct flag
  tst  (r3)      /open or closed
  beq  1f
  jsr  r5,SERROR
  <attempted to reopen: \0>  .even

1:
  inc  (r3)
  mov  $F0,r3    /buffer base
  add  (r4)+,r3  /select correct buffer
  jmp  *3f(r1)   /select mode

3:
  /table of modes

RANDO
ROPEN
CREATE
APPEND

RANDO:
  jsr  r5,SERROR
  <unimplemented random access: \0> .even

ROPEN:
  mov  (r3),3f
  jsr  r5,fopen; 3: 0
  beq  FILEERROR
  rts  pc

CREATE:
  mov  (r3),3f
  jsr  r5,fcreat; 3: 0
  beq  FILEERROR
  rts  pc

APPEND:
  mov  r0,3f
  sys  open; 3: 0; 1
  beq  FILEAPPR0R
  mov  r0,*r3
  mov  $512,*2(r3)
  mov  r3,r2
  add  $5,r2
  mov  r2,*4(r3)
  mov  *(r3),r0
  sys  seek; 0; 2
rts dc

FILEERROR:
    jsr r5, SERROR
    <file open error: \\

FILEERROR:
    jsr r5, ERROR
    <error on open for append\\n
SERROR:
1:
    mov $2, r0
    movb (r5)+, sech
    beq 2f
    sys write; sech; 1
    br 1b
2:
    mov $2, r0
    movb (r2)+, sech
    beq 3f
    sys write; sech; 1
    br 2b
3:
    mov $2, r0
    mov $'\n, sech
    sys write; sech; 1
    sys exit

CLOSE:
    mov $FD, r3
    add (r4), r3
    mov (r3), 2f
    jsr r5, flush; 2: 0
    mov *(r3), r0
    sys close
    mov $FD0, r3
    add (r4)+, r3
    clr (r3)
    rts dc

FCLOSE:
    mov $14, ., r1
    mov $FD, r2
    mov $FD0, r3
1:
    tst (r3)
    beq 2f
    mov (r2), 3f
    jsr r5, flush; 3: 0
    mov *(r2), r0
    sys close
2:
    add $2, r2
    add $2, r3
sob rl, lb
rts oc
data
sech: 0
.globl READF,READFN,READFS,READFE
.globl FD,FD0,getc
.text

READF:
  mov $FD,r2
  add (r4)+,r2
  mov (r2),READFILE
  rts pc
.data

READFILE: 0
.globl ERROR
.text

READFN:
  mov $rnumbst,r3
  mov $23 ..r1 /length of number limited to 23 digits
  clr r2
  1:  
      mov READFILE,2f /standard input
      jsr r5,getc; 2: 0
      bes bbadread
      movb r0,rch
      cmpb $'9',rch
      blt 6f
      cmpb $'0',rch
      bgt 2f
      movb rch,(r3)+
      sbb r1,1b
      br 6f
  2: 
      cmpb $' ',rch
      beq 3f
      cmpb $'	',rch
      beq 3f
      cmpb $'\n',rch
      beq 3f
      cmpb $' ',rch
      beq 5f
      cmpb $'=',rch
      beq 4f
      cmpb $'+',rch
      beq 3f
      br 6f
  3: 
      cmp r3,$rnumbst
      beq 1b
      cmp r3,$rnumbst+1
      bne of
      tst r2
      bne of
      sbb r1,1b
  4: 
      cmp r3,$rnumbst
      bne of
movb rch,(r3)+
br 1b

5:
tst r2
bne of
movb rch,(r3)+
inc r2
sob r1,1b

b:
tst r2
bne 2f
movb $'(r3)+

2:

movb $'\0,(r3)
mov 3rnumbst,-(r4)

rts cc

badread:
jsr r5,ERROR
<ERROR bad system call READFS\n\0> .even

.data
rch: .=.+2
rnumbst: .=.+24.

.globl ERROR
.text
READFS:

mov (r4)+,r1       /length to be read
mov (r4)+,r2       /address

1:
mov READFILE,2f     /default input
jsr r5,getc; 2: 0
bes badsread
movb r0,srch
cmpeq $'\n,srch
beq 2f
cmpeq $',srch
beq 2f

movb srch,(r2)+     /out character in place
sob r1,1b

2: movb $'\0,(r2)     /string full yet?

rts pc
badsread:
jsr r5,ERROR
<ERROR bad system call READFS\n\0>; .even

srch: .=.+2

READFE:

1:
mov READFILE, 2f
jsr r5, getc; 2: 0
bnez 2f
cmpeq $\backslash n, r0
bne 1b
2:
rts pc
.globl SDCAL
.text
SDCAL:
    movf *(r4)+,fr0
    movfi fr0,r2
    mov (r4),r1       // save length for later
    inc (r4)          // augment length by null on end
    mul (r4)+,r2     // multiply by length
    add r3,(r4)      // add displacement to base
    mov r1,-(r4)     // restore the length
    rts pc
.globl SUBSTR

.text

SUBSTR:

    movf *(r4)+,fr0  // length of substr
    movf *(r4)+,fr1  // starting offset
    mov (r4)+,r1    // length of string

    movfi fr1,r2
    dec r2
    cmp r2,r1
    bge 1f  // too long

    movfi fr0,r3
    add r3,r2
    cmp r2,r1
    bge 2f  // start+length too far

    movfi fr1,r2  // all OK
    dec r2
    add r2,(r4)   // alter address by starting byte

    movfi fr0,-(r4) // new length into stack
    rts pc

1:

    add r1,(r4)   // point to end of string('NULL')
    mov $1,-(r4)  // length now 1
    rts pc

2:

    sub r1,r2    // how much too big???
    movfi fr1,r3
    add r3,(r4)   // new starting address
    movfi fr0,r3  // get length again
    sub r2,r3
    mov r3,-(r4)  // new length
    rts pc
.globl WRITEF,WRTIFN,WRTIFS,WRTIFE
.globl FD,FDO,putc
.text

WRITEF:
    mov $FD,r2
    add (r4)+,r2
    mov (r2),WRTIFILE
    rts pc
.data
WRTIFILE:  0

.globl nodigit,floter,ERROR
.text

WRTIFN:
    mov $wnumbr,r3
    jsr pc,floter
1:
    mov $wnumbr,r3
    mov nodigit,r2
1:
    movb (r3)+,r0
    mov WRTIFILE,2f
    jsr r5,putc; 2: 0
    sob r2,lb
    movb $',r0
    mov WRTIFILE,2f
    jsr r5,putc; 2: 0
    rts pc

wnumbr: .=.+24.

.globl ERROR
.text

WRTIFS:
    mov (r4)+,r1
    mov (r4)+,r2
1:
    mov WRTIFILE,2f
    movb (r2)+,r0
    bea 5f
    jsr r5,putc; 2: 0
    sob r1,le
     rts pc
5:
movb $',r0
mov WRITFILE,2f
jsr r5,outc; 2: 0
sob r1,Sh
rts pc

WRITFE:

movb $'0r0
mov WRITFILE,2f
jsr r5,outc; 2: 0
rts pc
.globl asc
.text
asc:
    tst (r4)+ / pop stack
    mov (r4)+, r1 / address of string
    movb (r1), r2 / retrieve character
    movf r2, fr0 / convert into a floating pt number
    rts pc / for return
/ f = atof(p)
/ char *p;

ldfps = 170100
stfps = 170200

.globl atof

atof:
    stfps -(sp)
    ldfps $200
    movf fr1,-(sp)
    clr -(sp)
    clrf fr0
    clr r2
    mov (r4)+,r3
1:
    movb (r3)+,r0
    cmp $'0',r0
    beq 1b
    cmpb r0,$'0'
    bne 2f
    inc (sp)
1:
    movb (r3)+,r0
2:
    sub $'0',r0
    cmp r0,$9.
    phi 2f
    jsr pc,digitaf
    br 1b
    inc r2
    br 1b
2:
    cmpb r0,$'0','-0'
    bne 2f
1:
    movb (r3)+,r0
    sub $'0',r0
    cmp r0,$9.
    phi 2f
    jsr pc,digitaf
    dec r2
    br 1b
2:
    cmpb r0,$'E','-0'
    beq 3f
    cmpb r0,$'e','-0'
    bne 1f
3:
    clr r4
clr r1
cmpeq (r3),$'
bnne 3f
inc r4
inc r3
3:
movb (r3)+,r0
sub $'0,r0
cmpeq r0,$9.
bhi 5f
mul $10.,r1
add r0,r1
br 3b
3:
tst r4
bnne 3f
neg r1
3:
sub r1,r2
1:
movf $one,fr1
mov r2,-(sp)
beq 2f
bgt 1f
neg r2
1:
cmp r2,$38.
blos 1f
clrf fr0
tst (sp)+
bmi outaf
movf $huge,fr0
br outaf
1:
mulf $ten,fr1
sub r2,1b
2:
tst (sp)+
bge 1f
divf fr1,fr0
br 2f
1:
mulf fr1,fr0
cf cc
bvc 2f
movf $huge,fr0
2:
outaf:
tst (sp)+
beq 1f
negf fr0
1:
movf (sp)+,fr1
ldf os (sp)
rts pc
/
/
digitaf:
    cmof $big,fr0
    cfcc
    blt 1f
    mulf $ten,fr0
    movif r0,fr1
    addf fr1,fr0
    rts pc
1:
    add $32,(sp)
    rts pc
/
/
one = 40200
ten = 41040
big = 56200
huge = 77777
.globl chr
.text
chr:
  movf *(r4)+,fr0  // get number desired
  movf fr0,r2
  bit $0177,r2    / guarantee a valid character
  movb r2,chr+
  mov $chr+,-(r4)
  mov $1,-(r4)    / leave address and length 1 on stack

chr+: 0
.globl col
.globl nodigit
.text
col:
    movf *(r4)+,fr0
    movfi fr0,r1
    mov r1,nodigit
    rts pc
.globl  cosh
.globl  exp
cosh:    // cosh funct  \(0.5(e^{x} + e^{-x})\)
         movf  fr0,coshsave
         jsr  pc,exp
         movf  fr0,coshargl
         movf  coshsave,fr0
         negf  fr0
         jsr  pc,exp
         addf  coshargl,fr0
         mulf  onenalf+,fr0
         rts  pc
onensave:  .E+08
onenalf+:  040000; 0; 0; 0
.globl danrdr
.globl DATCNT, DATAEND, DATA
.text
danrdr:
    mov DATCNT, r2
    add $8, r2
    cmp r2, DATAEND
    bhi 1f
    mov $DATA, r2
    mov $2, r0
    mov $3f, r1
    sys write
    br 1f

3:
    <\n***RUN ERROR*** no num data num restore issued\n> 0
   even
1:
    movf *r2, f0
    movf f0, *(r4)+
    mov r2, DATCNT
    rts oc
.globl  dato_r
.globl  STRNEXT, STRDATA, STREND, strmv
.text
 dato_r:

  mov (r4), r3
  add STRNEXT, r3
  cmp $STREND-2, r3
  bge 1f
  mov $2, r0
  mov $3f, 4f
  svw write; 4: 0; 54,
  mov $STRDATA, STRNEXT
  br 1f

3: 


\\n***RUN ERROR*** no str data str restore issued\\n\\n       .even
1:

  mov (r4), r3       // save length
  mov STRNEXT,-(r4)  // move next data address into stack
  mov r3,-(r4)       // duplicate string length for strmov
  jsr oc, strmv
  dec r0
  tstb (r0)+        // did we read a while string??
  bne 1f
2:

  tstb (r0)+        // NO look for the end of this string
  bne 2b

1:

  mov r0, STRNEXT
  rts oc
ldfps = 170100ftst
stfps = 170200ftst
/ ftoa -- basic q to conversion
.globl nodigit

/ ecvt converts fr0 into decimal
/ the string of converted digits is pointed to by r0.
/ the number of digits are specified by nodigit
/ r2 contains the decimal point
/ r1 contains the sign

fcvt:
    clr eflag
    br 1f
ecvt:
    mov $1,eflag
1:
    stfps -(sp)
    ldfps $200
    movf fr0, -(sp)
    movf fr1, -(sp)
    mov r3, -(sp)
    mov $buf, r1
clr r2
clr sign
tstf fr0
cfcc
beq zer
bq 1f
inc sign
neqf fr0
1:
    modf $one, fr0
tstf fr1
cfcc
beq lss
gtr:
    movf fr0, -(sp)
    movf fr1, fr0
1:
    mov $buftop, r3
1:
    modf tenths, fr0
    movf fr0, fr2
    movf fr1, fr0
    addf $epsilon, fr2
    modf $ten, fr2
    movfi fr3, r0

182
add $'0, r0
movb r0, -(r3)
inc r2
tstf fr0
cfcc
bne lb
/
mov $buf, r1
1:
movb (r3)+(r1)+
cmp r3, $buf+top
blo lb
/
movf (sp)+(r0)
br pad
zer:
inc r2
br pad
1ss:
dec r2
modf $ten, fr0
tstf fr1
cfcc
beq 1ss
inc r2
jsr pc, digit1
pad:
jsr pc, digit1
br out
br pad
digit:
cmp r1, $buf+top
bhi $1f
add $2, (sp)
modf $ten, fr0
digit1:
movf fr1, r0
add $'0, r0
movb r0, (r1)+
1:
rtsgcc
/
out:
mov $buf, r0
add nodigit, r0
tst eflag
bne $1f
add r2, r0
1:
cmp $r0, $buf
bli output
mov $r0, $r3
add $5, $r3
mov $r3, (r0)

1:
cmpb (r0), $'9
ble 1f
movb $'0, (r0)
cmp $r0, $buf
blos 2f
incb -(r0)
br 1b

2:
mov $'1, (r0)
inc $r2

1:
outout:
mov sign, $r1
mov nodigit, $r0
tst eflag
bne 1f
add $r2, $r0

1:
clrb buf(r0)
mov $buf, $r0
mov (sp)+, $r3
movf (sp)+, $fr1
movf (sp)+, $fr0
ldfos (sp)+

rts pc

epsilon = 037114
one = 40200
ten = 41040

.data
tenth: 037314; 146314; 146314; 146315
nodigit: 10.

.bss
buf: 
.sign: 
.eflag: 

.text
/ C library-- floating output
.globl floter

floter:
1:
movf *(r4)+, $fr0
jsr pc, fcvt
tst rl
beq 1f
movb $'-',(r3)+
1:
tst r2
bgt 1f
movb $'0',(r3)+
1:
cmp nodigit,r2
jle 6f
mov r2,r1
ble 1f
2:
movb (r0)+,(r3)+
sob r1,2b
1:
mov nodigit,r1
beq 1f
movb $'..',(r3)+
1:
neg r2
ble 1f
2:
dec r1
blt 1f
movb $'0',(r3)+
sob r2,2b
1:
tst r1
ble 2f
1:
movb (r0)+,(r3)+
sob r1,1b
2:
rts pc
6:
movb $'?',(r3)+
sob r2,bb
rts pc
psciend:
mov r0,nodigit
tst r2
bne 1f
mov $b,nodigit
1:
movf (r4)+,fr0
jsr pc,ecvt
tst r1
beq 1f
movb $'-',(r3)+
1:
movb (r0)+,(r3)+
movb $'..',(r3)+
mov nodigit, r1
dec r1
ble 1f

2:
   movb (r0)+, (r3)+
   sob r1, 2b

1:
   movb $'e, (r3)+
   dec r2
   mov r2, r1
   bge 1f
   movb $'-', (r3)+
   neg r1
   br 2f

1:
   movb $'+', (r3)+

2:
   clr r0
   div $10, r0
   add $'0', r0
   movb r0, (r3)+
   add $'0', r1
   movb r1, (r3)+

rts pc
.globl int
.text
one = 040200
int:
    movf *(r4)+,fr0
    modf $one,fr0
    movf fr1,fr0
    tstf fr0
    cfcc
    bae 1f
    sub $one,fr0
1:
    rts pc
.globl  len, length

.text
len:
length:

tst (r4)+  /pop off default length
mov (r4), r2  /copy address
c1r r0

1:
tsto (r2)+
beq 2f
inc r0
br 1b

2:
movf r0, fr0  // length now in fr0 for return
rts pc
.global lindmp
.globl stdout
.text

lindmp:
    mov $1,r0
    mov $'01ch
    sys write; lch; 1
    mov $@0,,stdout+2
    rts pc

lch:    .=.+2
.globl mod
one = 040200
.text
mod:
movf *2(r4),fr0
divf *(r4),fr0
modf $one,fr0
mulf *(r4),fr1
tst *(r4)+ // pop stack
movf *(r4),fr0
subf fr1,fr0
rts pc
.globl nbrrdr
.globl numbst,ERROR
.text
nbrrdr:
    mov $numbst,r3
    mov $23,,r1      ;length of number limited to 23 digits
    clr r2
1:
    mov $0,r0        ;/standard input
    sys read; rch; 1
    bes badread
    cmpb $'9',rch
    nlt of
    cmpb $'0',rch
    bgt 2f
    movb rch,(r3)+
    sob rl,lb
    br of
2:
    cmob $',',rch
    beq 3f
    cmob $'\t',rch
    beq 3f
    cmob $',',rch
    beq 5f
    cmob $'-',rch
    beq 5f
    cmob $'+',rch
    beq 5f
    br 6f
3:
    cmp r3,3numbst
    beq 1b
    cmp r3,3numbst+1
    beq of
    tst r2
    bne of
    sob rl,lb
4:
    cmp r3,3numbst
    bne 6f
    movb rch,(r3)+
    br 1b
5:
    tst r2
    bne 6f
    movb rch,(r3)+
    inc r2
    sob rl,lb
6:
    tst r2
    bne 2f
    movb $',',(r3)+
2:
    movb $'\000',(r3)
    mov $numbst,-(r4)
    rts cc
badread:
    jsr r5,ERROR
<ERROR bad system call nrdr
          even
          data
rch:    .=.+2
numbst: .=.+40.
.globl numptr
.globl numbr,nodigit,stdout,floter,linamp,ERROR
.text

numptr:
    mov $numbr,r3
    jsr pc,floter
    sub nodigit,stdout+2
    tst stdout+2
    bgt 1f
    jsr pc,linamp
    mov $80,,stdout+2
1:
    mov $numbr,r3
    mov nodigit,r2
2:
    mov $1,r0
    movb (r3)+,nch
    sys write; nch; 1
    sob r2,2b
    mov $1,r0
    movb $',nch
    sys write; nch; 1
    rts pc

nch: .=.+2
.globl rad,deg
.text
rad:
mulf pi+,fr0
rts pc
pi+  036610; 0175065; 011224; 0164706

deq:
mul rd+,fr0
rts pc
rd+  041545; 027340; 0151436; 07703

rad.s
.globl rnd
.globl rand
maxplusone = 044000
rnd:
    jsr pc, rand
    movif r0, fr0
    divf $maxplusone, fr0
    rts pc
.globl sinh
.globl exp
.text
onehalf = 040000

sinh:  // sinh function .5*(e**u-e**-u)
  movf fr0,sinhsave
  negf fr0
  jsr pc,exp
  movf fr0,sinharql
  movf sinhsave,fr0
  jsr pc,exp
  subf sinharql,fr0
  mulf onehalf,fr0
  rts pc

sinhsave: .=.+8.
sinharql: .=.+8.
.globl strcmp

.text

strcmp:
        mov (r4)+, r2
        mov (r4)+, r1
        mov (r4)+, r3
        mov (r4)+, r0
        clr r2
        clr r3

1:
        movb (r0)+, r2
        beq 2f
        movb (r1)+, r3
        beq 5f
        cmpb r2, r3
        bgt 5f
        blt 4f
        br 1b

2:
        movb (r1)+, r3
        / check to make sure not equal
        bne 4f
        mov $0, -(r4)
        / set flag to equal
        rts pc

4:
        mov $1, r3
        / set less than
        neg r3
        / -1 is less than
        mov r3, -(r4)
        rts pc

5:
        mov $1, -(r4)
        rts pc
.globl strdmo
.globl stdo ut,ch,numbr,lindmP,ERR OR
.text
strdmo:
mov (r4)+,r3
mov r3,r1
sub r3,stdout+2
tst stdout+2
bgt l
/ need a newline
jsr pc,lindmP
mov $80.,stdout+2
l:mov (r4)+,r2
2:
mov $1,r0
movb (r2)+,ch
beq 5f
 sys write: ch: 1
sob r1,2b

rts pc
5:
mov $1,r0
movb $',ch
 sys write ; ch: 1
sob r1,5b
rts pc

.data
numbr: .=.+20.
ch: .=.+2
stdout: i: 80 .; 0
```
.globl strdr
.globl ERROR
.text
strdr:
    mov (r4)+,r1  /length to be read
    mov (r4)+,r2  /address
1:
    mov $0,r0    /default input
    sys read ; srch: 1
    bes badread
    cmpb $'\n',srch
    beq 2f
    cmpb $'\n',srch
    beq 2f
    movb srch,(r2)+ /put character in place
    sob rl,lb      /string full yet?
2: movb $'\0',(r2) /all strings end in null
   rts pc
badread:
   jsr r5,ERROR
   <ERROR  bad system call strdr\n0> ;  .even
srch:  .r2+2
```
.globl tab
.globl stdout
.text
.tab:
  movf *(r4)+,fr0         // tab value
  movfi fr0,r3
1:
  cmp $80.,r3
  bae 2f
  sub $80.,r3
  br 1b
2:
  mov stdout+2,r2         // char left
  mov $80.,r1
  sub r3,r1                // char needed at end
  cmp r1,r2
  blt 3f                  // if ge or at already there or past
  mov r1,stdout+2         // new end
  sub r1,r2               // how many blanks?
  mov stdout,r0
4:
  movb $',tch
  sys write; tch; i
  sob r2,4b
3:
  rts pc
.data
  tch:     =.+2
.global tan
.globl cos, sin
.text

tan:    // tan function sin/cos
.globl cos, sin
    movf fr0, tansave
    jsr pc, cos
    movf fr0, tancos
    movf tansave, fr0
    jsr pc, sin
    movf tancos, fr1  // test for div by 0 ans infinity
    tstf fr1
cfcc
    beq 1f
    divf fr1, fr0
    rts pc
1:
    movf hugeest, fr1  // plus or minus infinity??
tstf fr0
cfcc
    bge 2f
    negf fr1
2:
    movf fr1, fr0
    rts pc
tansave: .= .+8.
tancos: .= .+8.
hugeest: 077777; 177777; 177777; 177777
.globl val
.text
val:
    mov $numvst, r3     / pop stack
    tst (r4)+          / get starting address
    mov (r4)+, r0      / length of number limited to 22 digits
    clr r2
    movb $'0', (r3)+   / insure at least a zero
1:
    movb (r0)+, vch
    cmpb $'9', vch
    blt 6f
    cmpb $'0', vch
    bgt 2f
    movb vch, (r3)+
    sob r1, 1b
    br 6f
2:
    cmpb $', vch
    beq 3f
    cmpb $', vch
    beq 3f
    cmpb $', vch
    beq 5f
    cmpb $', vch
    beq 4f
    cmpb $', vch
    beq 3f
    br 6f
3:
    cmp r3, $numvst
    beq 1b
    cmp r3, $numvst+1
    bne 6f
    tst r2
    bne 6f
    sob r1, 1b
4:
    cmp r3, $numvst
    bne 6f
    movb vch, (r3)+
    br 1b
5:
    tst r2
    bne 6f
    movb vch, (r3)+
    inc r2
    sob r1, 1b
6:
    tst r2
    bne 2f
    movb $', (r3)+
2:
  movb $'0,(r3)
  mov $numvst,-(r4)
  rts pc

.data
vch:  .=.+2
numvst:  .=.+24.
int cflag;
int lflag;
int oflag;
int rflag;
in sflag;
in tflag;
int vflag;
char *av[50];
char *bprog;
char *llist[50];
char *q1 "/usr/graph/conie.o";
char *q2 "/usr/lib/libt.a";
char *q3 "/usr/graph/rmtksub.o";
char *q4 "/usr/graph/moresub.o";
char *q5 "/usr/graph/va.a";
char *pass0 "/usr/basic/haxcompS";
char *pass1 "/bin/as";
char *pass2 "/bin/ld";
char *pass3 "/bin/rm";
char ts[1000];
char *tsp ts;

main (argc, argv)
char *argv[ ]; {
   char *t;
   int i, j, bflag, nl, nxo;

   i=bflag=nl=nxo=0;
   while (++i < argc) {
      if (argv[i] [0] == '-')
         switch (argv[i] [1]) {
            default: goto passa;
            case 'S': //produce as-language file
                sflag++;
bflag++;
break;
            case 'o': //produce object file
                oflag++;
break;
            case 'C': //append C library for loader
                lflag++;
break;
            case 'c': //append conographic library
                cflag++;
lflag++;
}
```c
break;

case 't': // append Tektronics library
    tflag++;
    lflag++;
    break;

case 'r': // append ramtek library
    rflag++;
    lflag++;
    break;

case 'v': // append vq library
    vflag++;
    lflag++;
    break;
}
else {
    passa;
    t = argv[i];
    if (qetsuf(t) == '\n') { // is file.b an argument?
        bflag++;
        bproq = t;
        t = setsuf(t, '\0'); // if so, create file.o
    }
    if (nodup(llist, t)) { // does file.? exist as a
        llist[nl++] = t; // previous argument?
        if (qetsuf(t) == '\0') // is argument file.o?
            nxo++;
    }
}
if (!bflag) { // no file.b source program
    goto nocom;
    av[0] = "baxcomp"; // available for compilation
    av[1] = bproq;
    av[2] = 0;
    if (callsys(pass0, av) != 0) {
        printf("Procedure terminated at compilation state.\n");
        exit();
    }
    if (!(!bflag&&!oflag)) exit();
    t = setsuf(bproq, '\0');
    av[0] = "as";
    av[1] = "-";
    av[2] = t;
    av[3] = 0;
    callsys(pass1, av);
    if (oflag) {
        t = setsuf(bproq, '\0');
        unlink(t);
        if (link("a.out", t))
            printf("link fail %s\n", t);
        unlink("a.out");
        exit();
    }
    nocom:
}
```
i = 0;
av[0] = "ld";
av[1] = "-x";
if (!bflag)
    av[2] = t;
else
    av[2] = "a.out";
av[3] = "/usr/basic/basiclib.a";
j = 4;
while (j<n+1+3)
    av[i++] = list[++];
if (cflag)
    av[j++] = q1;
if (tflag) {  //three passes are needed due to
    av[j++] = q2;  //archiving of library
    av[j++] = q2;
    av[j++] = q2;
}
if (rflag) {
    av[j++] = q3;
    av[j++] = q4;
}
if (vflag)
    av[j++] = q5;
if (lflag)
    av[j++] = "-lc";
    av[j++] = "-la";
    av[j++] = 0;
if (callsys(pass2,av) != 0) {
    printf("Procedure terminated at load state.\n");
    exit( );
}
if (sflag) exit( );
t = setsuf(t,'s');
av[0] = "rm";
av[1] = t;
av[2] = 0;
callsys(pass3,av);  //remove file.s since not specified
exit( );
}
getsuf(as)
char as[];
{
    register int c;
    register char *s;
    register int t;

    s = as;
c = 0;
while(t = *s++)
    if (t=='/')
        c = 0;
    else
c++

s = 3;

if (c < 1 && c > 2)
return (ks);

setsuf(as, c);

register char *sl;

sl = copy(as);

while (*s++ == 'I')
s = s - 1;

return (sl);

call sys(f, v);

char 1(1, *v);

if ((t = fork()) == 0)
try again

if (t == -1)
exit();

if (t = 1)

while ((status & 0377) != 0 && t = 14)

exit();

print("Can't find %s", f);

return ((status >> 3) & 0377);

copy (as);

register char *otsp, *~~;

otsp = ~sp;

while (*otsp++)

return (otsp);

return (0);

setsuf(as, c);

register char *s, *sl;

s = sl = copy(as);

if (s == 0)

if (t — 1)

printf("Try again
");

return (1);

}
char **l, *os;
{
    register char *t, *s;
    register int c;

    s = os;
    if (getsup(s) != '0')
        return(1);
    while (t = *++l) {
        while (c = *++s)
            if (c != *t++)
                break;
        if (*t == '\0' && c == '\0')
            return(0);
        s = os;
    }
    return(1);
}
BIBLIOGRAPHY


<table>
<thead>
<tr>
<th>No.</th>
<th>Name and Address</th>
<th>No. Copies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Defense Documentation Center, Cameron Station, Alexandria, Virginia 22314</td>
<td>2</td>
</tr>
<tr>
<td>2.</td>
<td>Library, Code 0212, Naval Postgraduate School, Monterey, California 93940</td>
<td>2</td>
</tr>
<tr>
<td>3.</td>
<td>Department Chairman, Code 52, Naval Postgraduate School, Monterey, California 93940</td>
<td>1</td>
</tr>
<tr>
<td>4.</td>
<td>LT Lyle V. Rich, USN, Supply Officer, USS DUBUQUE (LPD 8), FPO San Francisco 96601</td>
<td>1</td>
</tr>
<tr>
<td>5.</td>
<td>LT Gary M. Raetz, USN, Code 52Rr, Naval Postgraduate School, Monterey, California 93940</td>
<td>1</td>
</tr>
<tr>
<td>6.</td>
<td>Computer Laboratory, Code 52, Naval Postgraduate School, Monterey, California 93940</td>
<td>2</td>
</tr>
<tr>
<td>7.</td>
<td>LT Michael D. Robertson, USN, Department Head Course, Newport, Rhode Island 02840</td>
<td>1</td>
</tr>
</tbody>
</table>