LOGO Manual

Article · October 2004

Source: OAI

CITATION
1

READS
26

3 authors, including:

Hal Abelson
Massachusetts Institute of Technology
30 PUBLICATIONS **503** CITATIONS
SEE PROFILE

Lee Rudolph
Clark University
69 PUBLICATIONS **1,316** CITATIONS
SEE PROFILE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

December 1974

A.I. Memo 313 LOGO Memo 7

LOGO MANUAL

by

Hal Abelson
Nat Goodman
Lee Rudolph

ABSTRACT

This document describes the LOGO system implemented for the PDP 11/45 at the M.I.T. Artificial Intelligence Laboratory. The "system" includes not only the LOGO evaluator, but also a dedicated time-sharing system which services about a dozen users. There are also various special devices such as robot turtles, tone generators, and CRT displays.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's education research is provided in part by the National Science Foundation under grant GJ-1049.

1. INTRODUCTION

This document describes the LOGO system implemented for the PDP 11/45 at the M.I.T. Artificial Intelligence Laboratory. The "system" includes not only the LOGO evaluator, but also a dedicated time-sharing system which services about a dozen users. There are also various special devices such as robot turtles, tone generators, and CRT displays.

We feel obliged to begin with some disclaimers. This manual is intended only as a language description and not as a primer. We hope it will be useful as a reference for users of our system and as a source of comparison for users of other LOGO systems. Secondly, we believe that the syntactic details of a computer language are of insignificant importance compared to considerations of how the language is used. Anyone who reads this paper to find out "what LOGO is like" should not neglect the papers of Papert and Solomon. Finally, the LOGO language is part of an ongoing research project. It is to be fully expected that the language specifications will change with our experience.

While there have been numerous contributors to the development of the LOGO language, those specifically concerned with the PDP 11 implementation are:

Ron Lebel, time-sharing system; Wade Williams, LOGO evaluator; Joe Cohen, time-sharing input/output; Nat Goodman and Hal Abelson, display and music box; Ron Lebel and Roger Hale, filing system. As part of the PDP11 programming project we have developed a PDP 11 debugging program which is similar but more powerful than DEC's ODT program. This was done by Radia Perlman. Our display controller was designed and built by Tom Knight.

There is also an essentially compatible LOGO evaluator which has been implemented in (M.I.T.'s version of) LISP by Ira Goldstein and Henry Lieberman. We must also cite the contributions of Richard Greenblatt on matters of system design and, of course, Seymour Papert and Cynthia Solomon on language specification.

Hal Abelson
Nat Goodman
Lee Rudolph

2. BASICS

This section introduces some basic LOGO vocabulary. LOGO statements are typed in at the console. You type a line of instructions and the computer executes it (or types an error message). The computer indicates that it is ready for an instruction by typing a question mark. Your line of instructions will not be executed until you end the line by typing a carriage return.

The basic "built-in" words that the computer understands are called LOGO primitives. These are described in this manual. The most important primitive is TO, which allows you to teach the computer new words. These private words that you teach the computer are called procedures.

Primitives and procedures often take one or more inputs. For example, PRINT is a primitive which takes one input and prints it on the console.

PRINT is an example of a command. A command is an imperative. It tells the computer to do something.

Other primitives may output. They provide a value which can be used as an input for another LOGO word. SUM, for example, takes two numeric inputs and outputs their sum. Procedures which output are called operations.

All complete LOGO statements are imperatives, so that an operation cannot stand by itself. If you type:

```
SUM 17 26
```

LOGO will respond with the error message:

```
YOU DON'T SAY WHAT TO DO WITH 43.
```

in contrast:

```
PRINT SUM 17 26
```

is a complete statement. The computer will print 43.

If you make a typing mistake you can press the "delete" or "rubout" key. This causes LOGO to ignore the previous character. It indicates this by retyping the ignored character with slashes around it. Pressing rubout again will cause LOGO to ignore the character before that, and so on.

Another feature is the "panic button" ctl-G (control G). Typing ctl-G in the middle of a command line causes the entire line to be ignored. Typing ctl-G while LOGO is executing a command causes it to stop and returns control to the teletype. This is especially useful if you write procedures which do not stop by themselves.

> Note: The "ctl" key is like a shift key. In order to type ctl-G, hold down the "ctl" key and type G. In this manual we sometimes use ^ to indicate "control." Thus ^G stands for control G.

Some primitives have abbreviations. If you type the abbreviation as part of a procedure line, LOGO will expand it to the full form.

3. DEFINING AND EDITING LOGO PROCEDURES

3.1 TO and EDIT (abbreviated ED)

Both commands put you into editing mode. Their inputs are restricted as follows:

i) No LOGO primitives

ii) For EDIT--one input, the name of a procedure which is already defined

iii) For TO--variable number of inputs; all but the first are dummy variables which stand for eventual inputs to the new procedure; the first input may be any name which is not already the name of a procedure.

For more on inputs, see Section 3.7 below.

3.2 END

Takes you out of editing mode.

3.3 RESTRICTIONS IN EDITING MODE

While you are defining/editing a procedure you can still do most of the things you usually can do in the LOGO world: use a display, a turtle, a music box, or other device; evaluate procedures--even the one you are in the midst of defining/editing. However, you cannot define/edit any other procedure. That is, TO and EDIT are invalid commands once you are in editing mode.

3.4 THE EDIT BUFFER

Lines you type pass into the procedure definition via an edit buffer. A line is put into the edit buffer if the first word on the line is a line number (a whole number greater that 0 and less than 32768). If a line number is not present, what you type in will be regarded as standard LOGO input, with the above restrictions. Every character and space you type after a line number is put into the buffer until you type a carriage return. (A machine-executed carriage return--which happens automatically if you type too many characters on one line for the console, or whatever, to accommodate-- doesn't have this effect). The succeeding carriage return will then empty that buffer into the procedure definition.

Warning: The buffer can only accommodate 190 characters and spaces, a little over three lines on the teletype; putting in more than that will jam things up and leave you no recourse but ^G.

Three other ways to get things into the edit buffer are EDIT LINE, EDIT TITLE and ^Y. EDIT LINE (abbreviated EDL), a single command despite the space, takes one input, a line number, and if that line exists it is put in the buffer. EDIT TITLE (abbreviated EDT) takes no inputs, and puts the title of the procedure you are defining into the buffer. ^Y puts the previous line you typed into the buffer (handy if you forgot to type a line number, for example).

When you have something in the edit buffer, you can manipulate it with these special control characters:

^C prints out and stores the next character in the buffer

^N prints out and stores the next word

^S skips--i.e., deletes the next word

^R prints out and stores the rest; and at any time (not just in editing mode) you can use, besides the rubout or delete key which deletes the previous character,

^W which deletes the previous word.

3.5 ERASING

When you are not in editing mode, ERASE (abbreviated ER) takes one input, a procedure name, and removes that procedure from your workspace. When you are in editing mode, it does the same, but you are not allowed to ERASE the procedure you are defining/editing. In editing mode only, you can use the command ERASE LINE (abbreviated ERL). Its input is a line number, and it erases that line.

3.6 PRINTING OUT

When you are not in editing mode, PRINTOUT (abbreviated PO) takes as input any procedure name, and prints out the definition of that procedure. By default, if it is given no input it prints out the definition of the last procedure you defined or edited. When you are in editing mode, it works the same; its default input is the procedure you are defining/editing.

In editing mode only, the commands PRINTOUT LINE (abbreviated POL) and PRINTOUT TITLE (abbreviated POT) can be used; the first takes one input, a line number, and prints out that line; the second takes no inputs, and prints out the title of the procedure being edited. Neither command puts anything in the edit buffer.

3.7 DUMMY VARIABLES FOR INPUTS

After its first input, TO can take further inputs of the form :<word>. Each of these stands for an input which the defined procedure will have to be given. See examples throughout the manual.

3.8 PROCEDURES WHICH EDIT PROCEDURES

It is possible to use any of the commands discussed above within a LOGO procedure. They work the same, with a single exception: if the user, working at top level, types END to finish defining a procedure, LOGO replies FOO DEFINED (where FOO is the title). This is <u>not</u> printed out when the editing procedures are called by other procedures; a great convenience.

Note, however, that the restrictions of Section 3.3 always apply--you cannot be defining more than one procedure at a time. (See Section 18.2 for more details).

3.9 COMMENTS

Text enclosed between exclamation points or between an exclamation point and the end of the line is ignored and can be used for comments.

4. WORDS AND LISTS OF WORDS

4.1 WORDS

In LOGO strings of characters are called words. A word may be indicated by prefixing it with a quote, as in:

```
?PRINT "WHOOPIE
WHOOPIE
```

The word consists of all the characters between the quote and the following space. (A carriage return also terminates words.) Therefore a word may not contain a space as one of its characters.

The PRINT command (abbreviated PR) can be used to print words as indicated above. PRINT takes one input and prints it followed by a carriage return. A very similar command is TYPE, which acts just like PRINT except that it does not include the carriage return. A word may include any printing character except space, carriage return and left and right square brackets [and]. In particular, a word may contain quotation marks:

```
?PRINT "A"
A"
?PRINT ""
"
```

A word may also contain no characters. Such a word is called "the empty word" and is indicated by a quote followed by no characters:

```
?PRINT "
```

A percent sign in a word is printed as a space. This is useful in drawing patterns on the teletype, as

```
TO BOX
10 PRINT "XXXXX
20 PRINT "X   X
30 PRINT "X   X
40 PRINT "X   X
50 PRINT "XXXXX
END
?BOX
XXXXX
X   X
X   X
X   X
XXXXX
```

4.2 LISTS OF WORDS

Ordered collections of words are called lists. (LOGO also allows more general lists as described in Chapter 15.) A list may be indicated by giving the words in the list, separated by spaces and enclosed in square brackets.

```
?PRINT [I AM A LIST]
I AM A LIST
```

Notice that the words in the list are not quoted and that the surrounding brackets are not printed. The spaces between the words serve only to separate the words and, strictly speaking, are not part of the list. Extra spaces are ignored by LOGO:

```
?PRINT [EXTRA        SPACES]
EXTRA SPACES
```

A carriage return within a list is equivalent to a space:

```
?PRINT [MORE THAN
ONE
LINE]
MORE THAN ONE LINE
```

Going along with the empty word we have the empty list which contains no words:

```
?PRINT [ ]
```

Note that the empty word is a word and the empty list is a list and they are not the same. (See also Section 4.5.)

4.3 MANIPULATING WORDS AND LISTS

There are a number of operations for manipulating words and lists:

COUNT

Takes one input. If the input is a word it outputs the number of letters in the word. If the input is a list it outputs the number of words in the list.

FIRST (abbreviated F)

Takes one input. If the input is a word it outputs the first letter of the word. If the input is a list it outputs the first word of the list.

LAST (L)

Similar to FIRST. Outputs the last letter (resp. word) or a word (resp. list)

BUTFIRST (BF)

Outputs all but the first letter of a word; all but the first word of a list.

BUTLAST (BL)

Similar to BUTFIRST. FIRST, LAST, BUTFIRST, and BUTLAST may not be applied to the empty word or the empty list.

These operations take words and lists apart. For putting them

together we have:

WORD

Takes two inputs, both of which must be words and puts them together to make a longer word:

```
?PRINT WORD "NOW "HERE
NOWHERE
```

SENTENCE (abbreviated SE)

Takes two inputs. If both are lists it puts them together to make a longer list. If one is a word and one is a list it adds the word to the list. If both are words it makes a list out of them:

```
SENTENCE [THIS IS] [A LIST] outputs [THIS IS A LIST]
SENTENCE "MANGO "CHUTNEY outputs [MANGO CHUTNEY]
SENTENCE [MATH IS] "YECCH outputs [MATH IS YECCH]
```

Example:

```
TO REPLACE :LET :L1 :L2
10 IF :LET = :L1 OUTPUT :L2 ELSE OUTPUT :LET
END

TO LISP :W
10 IF :W = " OUTPUT "
20 OUTPUT WORD REPLACE FIRST :W "S "TH LISP BUTFIRST :W
END

TO MULTILISP :S
10 IF :S = [ ] OUTPUT [ ]
20 OUTPUT SENTENCE LISP FIRST :S MULTILISP BUTFIRST :S
END

?PRINT MULTILISP [THIS IS A RECURSIVE PROCESS]
THITH ITH A RECURTHIVE PROCETHTH
```

4.4 REQUEST AND TYPEIN; MULTIPLE INPUTS

The LOGO operation REQUEST waits for the user to type in a list and then outputs that list.

```
TO AGREE
10 PRINT [TYPE SOMETHING YOU LIKE]
20 PRINT (SENTENCE [I LIKE] REQUEST [TOO])
END
?AGREE
TYPE SOMETHING YOU LIKE
>PICKLE JELLO WITH PEANUT BUTTER
I LIKE PICKLE JELLO WITH PEANUT BUTTER TOO
```

In the above example the prompt character > indicates that LOGO is waiting for a REQUEST to be typed in. The list typed into a REQUEST is not enclosed in brackets, and is terminated by a carriage-return.

The above example also illustrates that the SENTENCE operation can be made to take more than two inputs by enclosing the word SENTENCE and all the inputs in parentheses. SENTENCE then combines all the inputs into one list. WORD, PRINT and TYPE can also take multiple inputs in this way.

Note: Do not confuse parentheses and square brackets. Parentheses indicate grouping of inputs. Square brackets are more like quotation marks. They indicate that something is to be taken literally as a list.

REQUEST always outputs a list, even if it is a list containing one (or no) words. The operation TYPEIN is like REQUEST except that it outputs a word. TYPEIN is equivalent to FIRST of REQUEST.

4.5 FPRINT

Under the LOGO printing conventions the empty word and the empty list both print as a blank line. Likewise a word and a one-word list print the same:

```
?PRINT "MUMBLE
MUMBLE
?PRINT [MUMBLE]
MUMBLE
```

While this is convenient for writing conversational programs it can also be misleading, especially when tracking down bugs which may come from confusing words and lists. To help here, LOGO provides the command FPRINT ("Full PRINT") which is just like PRINT except that it prints the brackets surrounding lists.

```
?FPRINT [MUMBLE]
[MUMBLE]
```

5. NUMBERS AND ARITHMETIC

5.1 INTEGERS

The largest integer that is accepted by LOGO arithmetic is 2,147,483,647. The smallest number is -2,147,483,647. (Note, however, that numbers are written in LOGO without commas.)

5.2 INFIX FORMS

Each of the basic arithmetic operations has an infix form (infix because the symbol goes between the operands):

+ for addition 1+2

- for subtraction, 1-2

* for multiplication, 1*2

/ for the quotient of integer division, 3/4 outputs 0

\ for the remainder of integer division, 3\4 outputs 3

- for unary minus. (This is not technically an infix operator, but its use is effectively the same as subtracting the operand from zero.)

5.3 PRECEDENCE

*,/,and \ have a higher precedence than + and -; that means that *,/, and \ are evaluated before + and -:

```
?PRINT 4*5-4*5
   is equivalent to
?PRINT (4*5)-(4*5)
       20  - (4*5)
       20  -  20
           0
0
```

As opposed to

```
?PRINT 4*(5-4)*5
      4*  1  *5
        4 * 5
          20
20
```

All these infix operators have higher precedence than prefix ones which is why all the arithmetic gets done before any PRINTing happens.

5.4 NUMERICAL CONDITIONALS

< , > , and = are infix forms of operations that compare two numbers. < means "less". < outputs TRUE if the first number is less than the second, FALSE otherwise. = means "equal". > means "greater". These work in the same manner as < :

```
?PRINT 100 = 100
TRUE
?PRINT 100 > 100
FALSE
```

< , > , and = have lower precedence than the other arithmetic operations, so when performing a comparison the arithmetic is done first. These operations are usually used with the LOGO conditionals, IF and TEST (see Chapter 9).

< and > require that their inputs be numbers.

```
"APPLE > "PEAR
```

is considered nonsense and will generate an error.
= , however, can be used to compare any two LOGO objects. For example,

```
"FOOEY = "FOOEY
```

does make sense and will work.

5.5 PREFIX FORMS

Each of the arithmetic operations has a prefix form (prefix because it comes before the operands):

SUM	+
DIFFERENCE	-
PRODUCT	*
QUOTIENT	/
REMAINDER	\
GREATER	>
LESS	<
EQUAL	=

SUM and PRODUCT make use of the variable number of inputs feature (see Section 4.4).

```
?PRINT (SUM 96)
96
?PRINT (SUM 1 2 3 4 5 6 7 )
28
```

5.6 RANDOM

Takes no inputs, outputs a single digit random integer.

5.7 FLOATING POINT NUMBERS

In addition to the integer numeric form already mentioned, LOGO also accepts floating point numbers. These numbers can be expressed in two different formats for both input and output. These forms are standard decimal notation and exponential notation. Exponential notation is of the following form:

<number><E or N><exponent>

where <number> (called the mantissa) is either a floating point number in standard decimal form or an integer, and <exponent> is an integer

(representing a power of 10). If the exponent is positive, the letter E is used to separate the number from its exponent; if the exponent is negative, the letter N is used. There should be no spaces between these three elements of exponential notation. For decimal notation, a number is written in almost the same form as an integer, but there must be a decimal point at some position in the number. If this decimal point is not present, the number will be regarded as an integer. Here are some examples of valid floating point numbers in exponential form:

```
3E4             5N2
3.4E4           4.563N3
-4E4            -4N5
-2.5925E9        -259.259N2
```

The following are not in correct exponential form:

```
3E5.1           (The exponent is not an integer)
2.7E-1                  (The letter should be N)
10 E1           (There should be no space)
```

The following numbers are examples of floating point numbers in decimal notation:

```
4.
4.56
259.259
.00000000259
```

The output of a floating point number will always be in one of these two formats. Usually the output will be in decimal notation with non-significant and trailing zeros removed. If the number is too large or too small to be represented with 7 digits and a decimal point, it will be automatically converted to exponential notation (with one digit of the mantissa to the left of the decimal point). The maximum magnitude of a floating point number in LOGO is approximately 1.7014E38. The minimum magnitude is approximately 2.9387N39 (however, this cannot be entered from

a console-- the smallest magnitude that can be given as input to LOGO is 1.N38). These limits are the same for both positive and negative numbers.

All arithmetic operations on floating point numbers yield floating point numbers. In order to force conversion of a floating point number to an integer, the INTEGER operation is used. INTEGER takes one input, a floating point number in decimal or exponential form, and outputs this number converted to an integer. (Remember, however, that INTEGER is subject to the standard LOGO limits on integer size, and conversion of floating point numbers exceeding this size will fail.) In order to force the conversion of an integer to a floating point number, simply multiply by 1..

5.8 NUMBERS AS WORDS

Numbers are considered to be LOGO words (see Chapter 4); all word-manipulating operations work on numbers, too.

```
?PRINT FIRST 45678
4
?PRINT BUTFIRST 45678
5678
?PRINT (WORD 12 34) + (WORD 56 78)
6912
```

Integers or floating point numbers exceeding their respective size limits are still treated as words but cannot be used in arithmetic. Even though numbers are words they do not have to be quoted when typed in.

5.9 NUMBERP

Takes one input. If the input is a number in the allowable range for its respective type, NUMBERP outputs TRUE. If the input is a number outside that range, or isn't a number at all, NUMBERP outputs FALSE.

6. ARRAYS

Arrays in LOGO are of one, two or three dimensions. There are three types of arrays-- integer, floating point, and pointer.

6.1 DEFINEARRAY

To create an array, the command DEFINEARRAY (abbreviated DEFAR) is used. Depending on the number of dimensions, DEFINEARRAY takes up to five inputs: the name of the array, the dimensions of the array, and the array type (0 for integer, 1 for floating point, 2 for pointer). If the array has more than one dimension, the command DEFINEARRAY and its inputs must be enclosed in parentheses.

6.2 ARRAY SIZE and SPACE ALLOCATION

The maximum size of arrays is dependent upon the current amount of array space. Array space is normally allocated to allow approximately 2600 elements in pointer arrays, and half this amount in integer and floating point arrays. The total amount of space used is approximately equal to the sum of the number of elements in each array, with pointer array elements only using half as much space as integer and floating point. (However, it should be noted that every time an array is created some extra space is used to store the description of that array, making it more efficient regarding space to use single large arrays whenever possible.)

It is possible to allocate extra space for arrays by using the SETASIZE command. SETASIZE takes one argument, the amount of space to be allocated, in words (every word stores one pointer array element, every two

words stores one integer or floating point array element). The maximum input to SETASIZE is approximately 11000, which makes it possible to expand array space by a factor of three. One important note about SETASIZE-- the use of this command involves a re-initialization of the size of the user's workspace, which means that a new "WELCOME TO 11LOGO" message will be printed upon completion of the command. Thus, any use of SETASIZE should be invoked before using the filing system in order to save the trouble of having to read files again, etc.

Arrays are not saved in the filing system (see Section 11.1). They are automatically erased when you type HELLO or GOODBYE to LOGO, and so they must be re-created each time you use LOGO. However, they can be erased at any time by using the command ERASE ARRAY (abbreviated ER ARRAY). This command takes as input an array name and erases it from your workspace (see Section 10.2). The command ERASE ARRAYS (ER ARRAYS) takes no inputs and erases all arrays from your workspace.

6.3 GET and STORE

When an array is created, all the elements are set to zero. In order to assign values to specific elements of the array, the STORE command is used. GET is an operation used to reference the value of a specific element. STORE takes the same number of inputs as the corresponding DEFINEARRAY command; its inputs in order are: the array name, the coordinates of the desired element, and the value to be assigned to the element. GET takes one less input than the corresponding STORE command; its inputs are: the array name, and the coordinates of the desired element.

Unlike STORE and DEFINEARRAY, GET outputs a value. As in DEFINEARRAY, GET and STORE and their respective inputs must be enclosed in parentheses if the array is of more than one dimension.

It is important to remember in using the GET and STORE operations that LOGO uses 0-origin indexing, i.e. for any dimension of length n, the elements are numbered 0 to n-1. For example, the fourth element in a single dimension array is given coordinate number 3; the element in the fourth row, third column of a two- dimensional array FOO is addressed "FOO 3 2 ; and so on.

6.4 PRINTING OUT ARRAYS

ASIZE is an operation which takes one input, an array name (quoted), and outputs the dimensions of the array. The command PRINTOUT ARRAY (PO ARRAY) takes as input an array name and prints out its type and dimensions. PRINTOUT ARRAYS (PO ARRAYS) takes no inputs and performs PO ARRAY for all arrays curently defined in your workspace.

Examples of array operations:

```
?(DEFAR "B 2 2 0)
?PR (GET "B 1 1)
0
?(STORE "B 1 1 7)
PR (GET "B 1 1)
7
?PR (GET "B 1 2)
INDEX REFERENCE OUT OF BOUNDS
?DEFAR "A 2000 0
NOT ENOUGH ARRAY SPACE
?SETASIZE 6000
```

```
WELCOME TO 11LOGO
?DEFAR "A 2000 0
?PO ARRAYS
A SIZE 2000 TYPE INTEGER
```

7. NAMING

7.1 MAKE

The LOGO command MAKE is used for naming. MAKE takes two inputs; the first is the name and the second is the thing being named.

```
MAKE "X 27
```

will assign the name X to 27. A name must be a LOGO word (see Section 4.1 about words). 27 is the value or thing, and may be a number or a word or a list or anything else.

The use of the symbol : (pronounced "dots") as a prefix to a word retrieves the thing or value of a word.

```
?PRINT :X
27
?MAKE   "XYZ          27
        (The NAME)    (The THING)
?PRINT :XYZ
27
```

THING is an explicit LOGO primitive that does what : does, i.e., it extracts the thing from a word.

```
?PRINT THING "XYZ
27
?MAKE "RABBIT "HARE
?PRINT :RABBIT
HARE
?PRINT THING :RABBIT
```

will generate an error, because the word HARE has no thing.

Let's give HARE a value:

```
?MAKE "HARE [A B C]
?PRINT :HARE
[A B C]
```

Now,

```
?PRINT THING :RABBIT
[A B C]
```

The backarrow symbol (←) is an infix form of MAKE

```
?"NAME ← "REALNAME
?:NAME ← 27
?PRINT :REALNAME
27
?(WORD "REAL "NAME) ← 27
```

works, too. So does:

```
?"NUM ← 0
?(WORD "ARRAY :NUM) ← "ROSE
?PRINT THING (WORD "ARRAY :NUM)
ROSE
```

It is also possible to use the MAKE command to give multiple names to the same value. This is done in the following manner: to give two names A and B the same value, we say

```
"A ←"B ←<value>
```

This can be done for an arbitrary number of names (subject only to the limitation on the length of one line in LOGO). Multiple MAKE can be performed only with the infix form of the MAKE command (←).

7.2 LOCAL AND GLOBAL NAMES

The inputs to a procedure are local names, that is, the name is the procedure's own private name. For example:

```
?TO INC :A
>10 MAKE "A :A+1
>20 PRINT :A
>END
INC DEFINED
?MAKE "A 3
?INC 3
4
?PRINT :A
3
```

In the above example, the name A in the procedure has nothing to do with the name A at the top level.

In contrast, names which are not inputs to a procedure are global.

Compare with the previous example.

```
?TO INC
>10 MAKE "A :A+1
>20 PRINT :A
>END
INC DEFINED
?MAKE "A 3
?INC
4
?PRINT :A
4
```

Both uses of A refer to the same global variable.

It is possible to cause a name to be local to a procedure even if it is not an input. This is done with the LOCAL command. LOCAL takes one input, the name to be declared local.

Example:

```
TO COUNTSQUARES :X
10 IF :X =0 STOP
20 MAKE "XSQ :X * :X
30 COUNTSQUARES :X-1
40 PRINT :XSQ
END
?COUNTSQUARES 4
1
1
1
1
```

Here XSQ is a local variable. Compare with

```
TO COUNTSQUARES :X
5 LOCAL "XSQ
10 IF :X=0 STOP
20 MAKE "XSQ :X * :X
30 COUNTSQUARES :X-1
40 PRINT :XSQ
END
```

```
?COUNTSQUARES 4
1
4
9
16
```

Here each invocation of countsquares has its own XSQ.

Names local to a procedure are defined in that procedure and in all subprocedures (unless the subprocedure has its own local version of the same name).

8. CONTROL

8.1 GO

This command must be part of a procedure. It takes one input, a line number in that procedure, and transfers control to that line.

8.2 STOP

This also belongs in a procedure. It terminates execution of a procedure and returns control to the calling procedure.

8.3 TOPLEVEL

This returns control in a procedure immediately to the top level.

8.4 OUTPUT

This command can only be used in a procedure; it returns control to the calling procedure, and outputs its argument. By using OUTPUT the user can define procedures which are operations.

9. CONDITIONALS AND RELATED COMMANDS

9.1 Conditionals are operations which output either TRUE or FALSE. You can make your own, if you want. LOGO has three classes of readymade conditionals:

a) Numerical conditionals, e.g. LESS. (See chapter 5.) But note that EQUAL and = can also take non-numeric inputs.

b) Logical conditionals. These take inputs which evaluate either to TRUE or to FALSE, and perform logical operations on them; thus they are used in conjunction with other conditionals. They are:

BOTH-- takes two inputs: outputs TRUE if both evaluate to TRUE, outputs FALSE if one or both evaluate to FALSE.

EITHER-- takes two inputs; outputs TRUE if one or both evaluate to TRUE, outputs FALSE if both evaluate to FALSE.

NOT-- takes one input; outputs TRUE if the input evaluates to FALSE, and vice versa.

c) Predicative conditionals. These see whether or not their inputs evaluate to a specified kind of thing. They are:

WORDP-- outputs TRUE if the input evaluates to a word, FALSE otherwise.

LISTP-- outputs TRUE if the input evaluates to a list, FALSE otherwise.

EMPTYP-- outputs TRUE if the input evaluates either to the empty word or to the empty list, FALSE otherwise.

NUMBERP-- outputs TRUE if the input evaluates to a number, FALSE otherwise.

Examples:

```
MAKE "X [ ]  MAKE "Z 37

?PRINT WORDP :X
FALSE

?PRINT LISTP :X
TRUE

?PRINT LISTP :Z
FALSE

?PRINT EMPTYP :X
TRUE

?PRINT NUMBERP COUNT :X
TRUE
```

9.2 Conditionals can be iterated, that is, their inputs can be conditionals:

```
TO ANYOF :A :B :C
10 OUTPUT EITHER :A EITHER :B :C
END

TO STROKE :A :B
10 OUTPUT EITHER BOTH :A NOT :B BOTH :B NOT :A
```

Parentheses can be inserted, if desired. They might clean up STROKE:

```
10 OUTPUT EITHER (BOTH :A NOT :B) (BOTH :B NOT :A)
END
```

There are also a number of standard commands which are used in conjunction with conditionals: they all demand that their inputs evaluate to TRUE or FALSE.

a) IF and ELSE. IF takes one input; if the input evaluates to the word TRUE, the rest of the LOGO line is evaluated; if the input evaluates to FALSE, the rest of the line is skipped. However, if ELSE appears in the line, this behavior is modified in the obvious way.

```
TO CHECK :W
10 IF WORDP :W PRINT "WORD ELSE PRINT "NOWORD
END

?CHECK "FOO
WORD
?CHECK [THIS LIST]
NOWORD
```

IFs (or IF--ELSE pairs) can be stacked within themselves.

```
TO BETTERCHECK :W
10 IF NOT EMPTYP :W THEN (IF WORDP :W THEN PRINT "WORD
   ELSE PRINT "NOWORD) ELSE PRINT "EMPTY
END
```

Here, THEN is a "noise word" which does not in itself affect the evaluation of the LOGO expression in which it appears, but helps to make the syntax of the expression more "natural". The parentheses are also optional, but help to make the expression easier to read.

b) TEST, IFTRUE, and IFFALSE. TEST evaluates its argument (to TRUE or FALSE) and puts the result in a "test box". Until the next TEST, IFTRUE (abbreviated IFT) and IFFALSE (abbreviated IFF) will look into the box and cause conditional execution of the rest of the line in which they appear. There is no device analogous to ELSE which allows conditional execution of only a part of the rest of the line. Some non-obvious properties of the test box are: first, if no TEST has been made, the box contains FALSE by default; second, the box is unique-- a new TEST, although it may have been executed conditionally, changes the contents of the box absolutely.

Each procedure has its own "test box", which is strictly local to that procedure. TESTs which are made in a subprocedure do not affect the test box of the calling procedure, and conversely.

IFTRUE and IFFALSE can appear anywhere in a procedure, and they do

not have to be on the same line (indeed a command such as IFTRUE "FOO IFFALSE "BAR can never cause BAR to be evaluated.)

9.3 TRUE and FALSE are LOGO words. It is perfectly valid to say

```
TEST "TRUE
or
TEST WORD "TR "UE
```

Remember that all words which are to be taken literally must be quoted on typein (except for numbers):

```
TEST TRUE
```

will expect you to have a procedure named TRUE. Also

```
TEST [TRUE]
```

is an error. (The input to TEST must be a word.)

10. WORKSPACE

When you define a procedure or create a name by using MAKE, it becomes a part of your workspace. You can think of your workspace as a chalkboard or scratchpad containing all the procedures you are using. The LOGO file system allows you to store everything that is in your workspace and retrieve it at some later time (see Chapter 11). Before a procedure or name can be used it must be in your workspace. To get something into the workspace you must either type it in at the console or read it in from a LOGO file. There are also commands for examining and getting rid of various parts of the workspace.

10.1 WHAT'S IN A WORKSPACE?

PRINTOUT (abbreviated PO) is a command which prints out various parts of your workspace. To see the text of a procedure, type PRINTOUT followed by the procedure name. (The procedure name is not quoted.)

PRINTOUT TITLES (abbreviated PO TITLES or POTS) prints out the titles of all procedures in your workspace.

PRINTOUT NAMES (abbreviated PO NAMES) prints out all the names in your workspace.

PRINTOUT PROCEDURES (PO PROCEDURES) prints out the text of all defined procedures.

PRINTOUT ALL (PO ALL) prints both names and procedures.

(The PRINTOUT LINE and PRINTOUT TITLE commands are discussed under editing. PRINTOUT FILE and PRINTOUT INDEX are discussed under filing. The PRINTOUT ARRAY and PRINTOUT ARRAYS commands are discussed under arrays.)

The operation CONTENTS outputs a list containing the titles of all procedures in your workspace. Note the distinction between CONTENTS and PRINTOUT TITLES. The former is an operation, while the latter is a command.

A similar distinction exists between the command PRINTOUT and the operation TEXT. TEXT takes one input, a procedure name (not quoted), and outputs the text of the procedure as a list. The lines of the procedure appear as sublists. The END statement is not included.

10.2 GETTING RID OF PARTS OF THE WORKSPACE

The basic command is ERASE (abbreviated ER). ERASE followed by a procedure name (not quoted) removes the procedure from the workspace.

There are also:

ERASE ALL	Gets rid of all names and all procedures.
ERASE PROCEDURES	Gets rid of the procedures-- leaves the names.
ERASE NAMES	Gets rid of the names-- leaves the procedures.
ERASE NAME	Takes one input, a name (quoted), and gets rid of that particular name.

(ERASE LINE is discussed under editing. ERASE FILE and ERASE INDEX are discussed under filing. ERASE TRACE and ERASE STEP are discussed under debugging. ERASE ARRAY and ERASE ARRAYS are discussed under arrays.)

10.3 BURY

The BURY command effectively "hides" a procedure and protects it from accidental deletion. BURY takes one input, the name of the procedure to be buried (the command BURY ALL can also be used). Once a procedure has been buried, it will not appear in any PRINTOUT command unless it is named explicitly. For example,a file FOO which has been been buried will not appear under the commands POTS and PO ALL, but the command PO "FOO will print out the contents of FOO. This property also holds for ERASE commands. ERASE ALL will not erase buried procedures, but if the name of a buried procedure is specified in an ERASE command it will be erased. To unbury a procedure, the command ERASE BURY followed by the procedure name (or ALL) is used.

11. THE FILE SYSTEM

The LOGO file system allows you to save (on disk) what is in your workspace and read it back at a later time. A user may have many files at once. The files are distinguished by the fact that they are named. All files belonging to a single user are grouped under that user's index.

11.1 USE, READ, and WRITE

The command USE specifies the index under which LOGO should reference files. All subsequent READ or WRITE commands will refer to that index, until the next USE. USE takes one input, the quoted name of an index, which is a word of up to ten characters. For example, to reference the files under the index ELOISE, type

```
USE "ELOISE
```

The WRITE command creates files. It takes one input, the name of the file to be created (a word of up to ten characters). Into the file goes everything that is currently in the user's workspace.

This is a common source of confusion. Many users often think that if they have, for example, a procedure named BEETLE, then

```
WRITE "BEETLE
```

will somehow save only that one procedure. This is not the case. All the procedures in the workspace will be saved. The file will merely have the same name as one of the procedures in it.

If the user already has a file with the same name as the one to be created, LOGO will first ask if the old one should be erased and then wait for a response (Y or N) to be typed in. The old file must be erased before

a new file with the same name can be created.

The READ command takes a file name as input and reads the contents of the file into the workspace. All procedures and names saved in the file will then become defined in the workspace. If the workspace already contains a procedure with the same name as one of the procedures in the file, the procedure will not be redefined and the definition in the file will be skipped.

SNAPS (see Section 13.3) cannot be saved in files. If PIC, for example, is the name of a snap, then writing out the workspace and reading it back will cause :PIC to become the empty word.

11.2 POI and POF

These are commands which allow you to examine files without reading them into the workspace.

PRINTOUT INDEX (abbreviated POI) takes no inputs and prints the names of all the files in the index. (The index is specified by the previous USE command.) PRINTOUT FILE (PO FILE) takes a file name as input and types the contents of the file. It does not read the file into the workspace, i.e., procedures in the file will not become defined in the workspace.

ERASE FILE (ER FILE) takes a file name as input and gets rid of the file.

11.3 FILE SUBTLETIES

The description of the filing system so far, though accurate and

adequate for ordinary use, is incomplete. In fact, the LOGO file system has a tree structure. The commands USE, READ, WRITE, etc., as described so far, make use of that structure only implicitly. But they can be made to use the filing tree much more explicitly, and there are other commands not yet discussed which allow a user to manipulate the tree -- pruning it, adding branches, etc.

Each user's index can be the root of a general tree structure of files. This means that the index may not only contain files, but also sub-indices which contain files and sub-indices and so on. For example, the index ELOISE might contain the files DAY1, MARBLE, and the sub-index DISPLAY which contains files dealing with Eloise's display projects. In this case, the file structure would be:

```
I ELOISE
    DAY1
    MARBLE
  I DISPLAY
      WALK
      DRAW
      POLY
etc.
```

The I's specify indices. (Normally a number will appear after each file name. This is just the number of storage blocks that the file occupies.) This tree structure is printed out by the PO TREE command. (Note that the tree is actually upside down, with the root at the top. For this reason, the root of a tree is often referred to as the "top level" of the file system, while the branches are called the "lower levels", where each sub-index represents a new level.) POI does not print out the contents of subindices. In the above case, it would print:

```
ELOISE
  DAY1
  MARBLE
I DISPLAY
```

(The root index does not have an I printed before it.)

Each file in this tree structure is specified by a list starting with the user name. This list gives the "path" down the tree from the user name to the file. For example, to read the file DRAW in Eloise's sub-index DISPLAY we could say:

```
USE "ELOISE
READ [DISPLAY DRAW]
```

or we could say,

```
USE [ELOISE DISPLAY]
READ "DRAW
```

In general, the input to the READ command is added onto the input of the previous USE command and the whole string becomes the file specification. (What happens is that the USE command establishes the level from which all following commands are to be carried out. Thus, if the USE command refers only to the user name, the path to a file contained under a sub-index will be longer than if the USE command refers to the the desired sub-index. We will see later how this level can be changed without repeating the USE command.) The inputs to WRITE, POF, and ERASE FILE all work in the same way. While POI prints out the index specified by the previous USE command, PO TREE prints out a more complete version, consisting of all of the tree contained below this index, including sub-indices.

To create a sub-index, the CRINDEX command is used. This adds on

to the branch specified by the previous USE. For example, ELOISE'S sub-index DISPLAY was created by typing:

```
USE "ELOISE
CRINDEX "DISPLAY
```

To then create a sub-index of DISPLAY named PLOTTER, we could simply say:

```
CRINDEX [DISPLAY PLOTTER]
```

To get rid of an index use the ERASE INDEX command. This takes one input which specifies, in the same manner as the above commands, the index to be erased. An index may not be erased unless it is empty, i.e., contains no files or sub-indices.

Once an index has been created, it is possible to adjust the root of the file tree so that all filing commands refer only to those files and indices contained in the tree below a specified index. The command which does this is SETINDEX. SETINDEX (abbreviated SETI) takes one input specifying an index which will be the root of a new tree forming some part of the whole file tree of the workspace. As before, the input specifies the path taken down or up the tree from the current root to reach the desired index. If more than one level of indices is involved in the path, a list is necessary to specify the whole path. However, if the desired index is just one level above or below the current root, a single word is sufficient to specify the path.

Once an index below the user name has been established as the current root, all commands refer only to that part of the tree below this new index. For example, let's consider Eloise's file system again, only this time with a few more branches added:

```
ELOISE
I DISPLAY
  WALK
  DRAW
  POLY
I ALGEBRA
    FACTOR
  I MATRIX
      INVERSE
```

In order to make ALGEBRA the new root index, the command SETI "ALGEBRA is used. The new tree (printed out by PO TREE) looks like this:

```
ALGEBRA
  FACTOR
I MATRIX
    INVERSE
```

Notice also that files not in this part of the tree cannot be read without going back up the "branch" and down another until the appropriate index under which the file is contained is reached. For example, typing READ "POLY will return an error message, "FILE NOT FOUND", since the path name has not been fully specified.

In order to move the current root back up the tree towards the main directory name, the input "^ is used. The up-arrow can be used in lists just like index names, so that a list containing n ^'s used as input to SETI moves the current root up n levels. Thus, in the above example, in order to read POLY, the command READ [^ POLY] is used.

11.4 OPEN FILE MANIPULATION

.OPENW takes one input, a desired file name (quoted). (This file does not have to already exist in the file system.) .OPENW opens the selected file (or creates one if it does not already exist) and allows it

to be written into, starting at the beginning of the file. If the file already exists in the filing system, the user will be asked to delete it before writing, since the old information contained in the file will be written over by the new input to the command .FILEP.

.FILEP is used, once the file has been opened for writing, to perform the actual writing. .FILEP takes as input a list containing the desired information to be written into the file. Each time .FILEP is used, a new line (with a maximum length of 190 characters) is created in the file. The input to .FILEP can also be a procedure name (unquoted), in which case the procedure is executed and its output is written into the file. If there is no output, nothing is written.

.OPENA opens a file for writing much like .OPENW, but all information written is added (appended) to the end of the file. None of the original contents of the file are altered. .FILEP works in the same manner as before.

.OPENR is used to open a file for reading. It takes one input, the desired file name (quoted). The command used with .OPENR to do the actual reading is .FILER. Unlike .FILEP, .FILER takes no inputs but returns an output. This output is the current line of the file being read. Every time a file is opened for reading, an internal pointer is set to the first line of the file. Each time .FILER is used, this pointer is incremented so that the next line will be read with the next use of .FILER. When the end of the file is encountered, one blank line will be printed out by .FILER, and then the file will be automatically closed. In order to read the file again, .OPENR must be used again. If the current line of the file is a

procedure name, that procedure will be executed. (This fact enables a user to execute procedures "implicitly" while reading a file. It is utilized by the INIT file; see the LOGIN command.) Any word encountered which is not part of a procedure definition or is not a LOGO command is assumed to be a procedure name. If such a procedure does not exist, an error will be returned.

.CLOSEF is used to close any open file. .CLOSEF takes no inputs and closes the file currently open (only one file at a time can be open). If no file is currently open, the command is ignored.

11.5 LOGIN

LOGIN takes one input, your user name (quoted). This "tells the system who you are", and this information is used by the PEEK command. LOGIN also performs an automatic USE. However, LOGIN is most useful for initializing certain conditions in the workspace and for printing out mail.

When LOGIN is used, the system looks in the user's file system to see if there is a file named INIT in the top level of the file tree. If this file exists, the system reads the file and performs all commands written directly into the file (through the use of the commands .OPENW, .OPENA, and .FILEP). This facility allows the user to do many useful things. For instance, the INIT file can be used to read all the files in the user's file system "automatically" when logging in, or to print out messages upon logging in.

The LOGIN command also causes the system to search for the file named MAIL. If this file is found, the contents are printed, and the user

is then asked if the mail should be deleted.

11.6 PAPER TAPE

LOGO procedures may also be stored on paper tape. The user may pass information between paper tape and the workspace using the following primitives.

WRITEPTP punches out the contents of the workspace onto paper tape. It does not involve the file system.

READPTR reads into the workspace from the paper tape reader. Neither READPTR nor WRITEPTR take any inputs.

You should have someone show you the mechanics of using the reader and punch before you use the paper tape commands.

12. TURTLES

This section is about the physical turtles that run around on the floor. (See Chapter 13 for display turtles.)

12.1 You must tell the system which turtle you want to use before issuing turtle commands.

TURTLE

Takes one input which should be the number of the turtle you want to use. The numbers are marked above the plugs on the controller box, and are 1, 2, 3 or 4. Normally only 1 and 2 have turtles plugged into them.

NOTURTLE

Takes no inputs. NOTURTLE releases the turtle from your control so someone else can use it.

12.2 BASIC TURTLE COMMANDS

FORWARD (FD)

BACK (BK)

These commands take one input which must be a number between -32768 and 32767 inclusive. They command the turtle to move <input> units forward (in the direction it is pointing) or backward.

RIGHT (RT)

LEFT (LT)

These take one input which must be a number between -32768 and 32767 inclusive. They command the turtle to rotate <input> degrees clockwise or counterclockwise, respectively.

PENUP (PU)

PENDOWN (PD)

Raises or lowers the pen holding mechanism of the turtle. When the pen is down the turtle will draw a line as it moves.

12.3 SOME TURTLES HAVE A HEADLIGHT

LAMPON

LAMPOFF

Turns the headlight on and off.

12.4 SOME HAVE A WHISTLE

TOOT

Takes one input which must be a number between -32768 and 32767 inclusive. It blows the whistle that number of times.

12.5 TOUCH TURTLES

There is a variety of turtles that have sensors that can tell when the turtle is bumping against an object. The following primitives are used to test whether sensors are on or off.

FTOUCH

Outputs TRUE if the front of the turtle is touching something; FALSE otherwise.

BTOUCH means back touch

RTOUCH means right touch

LTOUCH means left touch

Here's a procedure to bounce the turtle between two walls:

```
TO BOUNCE
10 TEST FTOUCH
20 IFTRUE LEFT 180    ! TURN AROUND
30 FORWARD 10
40 BOUNCE
END
```

If the turtle is hitting an object at an oblique angle two touch sensors may be TRUE at the same time.

12.6 SEEING EYE TURTLE

We have a prototype turtle-with-an-eye. The eye is extremely simple - it tells the amount of light it is seeing in a very narrow field.

LIGHT

Outputs the light level. This is a number between 0 and 63.

LAMPON

LAMPOFF

Seeing eye turtles don't have headlights (yet). These commands currently enable and disable a mode whereby the eye can rotate. LAMPON enables this mode, LAMPOFF disables it.

FORWARD, BACK, RIGHT, LEFT

When in the mode described above these commands change the angle of elevation of the eye. When not in that mode these commands perform their normal functions.

12.7 THE LIGHT BOX

It is convenient to use the light box in conjunction with the seeing-eye turtle. The light box can supply current to any subset of up to

six light bulbs (at the moment only four light bulbs are plugged in). The light box is controlled by sending it a number via the .TYO command (see Chapter 19). The light box is specified by the input "LIGHT , so to send the number 20 to the box type

```
.TYO "LIGHT 20
```

What the box does when it gets a number is determined as follows:

Write the number in binary. Then the 1's place specifies whether light number 1 is to be on or off, the 2's place light number 2, the 4's place light number 3, etc.

Note: The controller to which the light box is attached traps 17 (octal 21) as its reset character. Thus 17 sent to the light box will not be seen. If you desperately want to send 17 (to turn on lights five and one together) you can use the fact that the light box ignores more than six bits and send it 64+17=81 (octal 131).

12.8 THE SWITCH PANEL

The switch panel is a "blue box" which has on its front 16 small light bulbs. Each of these lights represents the status of one of 16 hardware switches. When a switch is set, the corresponding light is turned on; when it is cleared, the light is turned off. The following LOGO primitives can be used to test and manipulate the status of these switches.

SWITCH

Takes one argument, the number of a switch (0-15) and outputs TRUE if the switch is set, otherwise FALSE.

RELAY

Takes two inputs, the number of a switch and a value to be assigned

to it (0 or 1). If 0, the switch is turned off, if 1, it is turned on.

BITOUT

Takes one input, a number which is converted into a 16-digit binary value. Each digit of this value is then used to determine the condition of the corresponding switch (effectively the same as performing a RELAY for each switch).

13. DISPLAY TURTLES

The display screen is 400 turtle units high by 400 wide. The center of the screen has coordinates (0,0); the bottom left hand corner is (-200,-200); and the top right hand corner is (200,200). When the turtle is pointing straight up its heading is 0. Heading is measured clockwise (i.e. RIGHT) from that position.

13.1 INITIALIZING THE DISPLAYS

CLEARSCREEN (CS)

Erases everything on the display screen and places the turtle at the center of the screen pointing up. You should do CS first, before issuing any other commands to the turtle.

WIPECLEAN (WC)

Erases everything on the display screen but leaves the turtle where it is.

NODISPLAY

Turns off the display.

STARTDISPLAY

Takes one input, which must be 0 or 1. 0 starts up a half-size display, 1 starts a full size display. STARTDISPLAY 1 is the same as CLEARSCREEN.

13.2 BASIC COMMANDS

FORWARD (FD)

BACK (BK)

Takes one input which must be either an integer or floating point number. Moves the turtle <input> units forward (in the direction it is pointing) or backward.

RIGHT (RT)

LEFT (LT)

Takes one input which must be either an integer or floating point number. Rotates the turtle <input> number of degrees clockwise or counterclockwise, respectively.

PENDOWN (PD)

PENUP (PU)

Lowers or raises the turtle's metaphorical pen. If the pen is down the turtle draws a line as it moves.

PENP

Outputs TRUE if the pen is down. FALSE if it is up.

WRAP

NOWRAP

WRAP enables a mode so that if you try to move the turtle beyond the edge of the display it wraps around to the other edge. NOWRAP turns off this mode; if you try to move beyond the edge you get an error.

HIDETURTLE (HT)

SHOWTURTLE (ST)

HIDETURTLE tells the computer to stop showing the little triangle that represents the display turtle. The turtle will still draw lines (if the pen is down) even though you cannot see him. SHOWTURTLE brings the turtle out of hiding.

13.3 SNAPS

SNAP

Takes no inputs. Outputs a reference to "the stuff on the display screen." For example:

```
MAKE "PIC SNAP
```

causes :PIC to refer to whatever is currently on the screen. Later you can cause another copy to appear by saying:

```
DISPLAY :PIC
```

or erase it by saying:

```
WIPE :PIC
```

Each SNAP has associated with it a "starting location" which is normally the center of the screen (but see also NEWSNAP). SNAPS cannot be saved with the WRITE command (see Section 11.1).

DISPLAY

Takes one input which must be a SNAP (i.e. a reference created by SNAP). The SNAP is displayed at the current turtle position and the turtle is then moved to the end of the SNAP, as if the turtle had just drawn the SNAP explicitly. The turtle's heading has no effect on the displayed SNAP. The SNAP always appears in the orientation in which it was originally drawn.

WIPE

Takes one input which must be a SNAP. It erases all appearances of that SNAP from the display screen.

NEWSNAP

Takes no inputs. It causes the image currently on the screen not

to be part of subsequent SNAPS. Also sets the starting location of subsequent SNAPS to the current position of the turtle rather than the center of the screen.

13.4 LOCATING THE TURTLE

HERE

SETTURTLE (SETT)

HERE outputs a list of the x-coordinate, y-coordinate and heading of the turtle. You can use HERE to name a place on the display. SETTURTLE takes one input, a list of three numbers between -32768 and 32767 inclusive. The first is assigned to the x-coordinate, the second to the y-coordinate and the third to the heading. You can use SETTURTLE to move the turtle to a place that was named using HERE. It will draw a line if the pen is down.

HOME

XCOR

YCOR

HOME is equivalent to SETI [0 0 0]. XCOR outputs a number which is the current x-coordinate of the turtle. YCOR outputs a number which is the current y-coordinate of the turtle.

SETX

SETY

SETXY

SETX takes one numerical input and moves the turtle horizontally to that X-coordinate. SETY takes one numerical input and moves the turtle

vertically to that Y-coordinate. SETXY takes two numerical inputs and moves the turtle to the designated position. Each of these commands will draw a line if the pen is down.

HEADING

SETHEADING

HEADING outputs a number which is the heading (in degrees) of the turtle, i.e., the direction in which it is pointing. SETHEADING takes one number as input and points the turtle in that direction.

13.5 PLOTTER

The plotter is often used in conjunction with displays. To use the plotter, type PLOTTER. When you are through using the plotter, type NOPLOTTER. All other plotter commands are the same as display commands. However, certain display commands do not apply to the plotter and will be ignored if typed to the plotter. These are all fairly obvious; they include: CLEARSCREEN, WIPECLEAN, WRAP, HIDETURTLE, SHOWTURTLE, SNAP, WIPE, and DISPLAY. The plotter also has a more restricted argument range than the displays.

13.6 MULTIPLE DEVICE CONTROL

It is possible to control more than one device at at time on LOGO. This can be done through the use of the commands ALSO and CNTRL. If you are already using one device and wish to use another without giving up the first, type

ALSO <device>

where <device> is the device you wish to use. ALSO gives you access to additional devices, but it is necessary to specify to which device a specific command is directed. This is done with the CNTRL command. CNTRL takes one input, the name of a device which you currently "own", and specifies that device as the object of the next command you type. For example, suppose you are using a display and you decide you want to use a floor turtle also. First type

ALSO TURTLE <number>

However, if you now type

TOOT

LOGO will respond

TOOT ONLY VALID IF YOU HAVE A TURTLE

Before you can direct commands to the turtle, it is necessary to type

CNTRL TURTLE <number>.

All valid floor turtle commands will now be accepted. If you want to use the display again, you must say

CNTRL DISPLAY

LOGO now will accept display commands, but not floor turtle commands. Each time you want to change between devices, the CNTRL command referring to the device that you want to use must be typed. However, subsequent commands can only refer to one device at a time.

14. MUSIC BOX

LOGO has primitives which supply output for the music box. A LOGO user can specify parts for up to 4 simultaneous voices, each voice having a range of five chromatic octaves. In order to avoid timing problems the music is compiled into temporary storage and then output to the box at a constant rate, rather than played in "real time".

14.1 NOTE

The NOTE command generates one note of music. NOTE takes two numeric inputs, the first specifies the pitch and the second the duration. Pitches are numbered chromatically from -24 to 36 with 0 being middle C. There are also three special "pitches":

-28 is a silence

-27, -26 are the percussion sounds "boom" and "ssh"

-25 is not a valid pitch

Durations must be between 0 and 127 units. Each unit is normally about 1/8 second (but see NVOICES below).

The actual output of NOTE :P :D is determined as follows:

If :D = 0 NOTE generates nothing.

If :D = 1 NOTE generates a pitch 1 unit long

If :D > 1 NOTE generates a pitch :D-1 units long followed by 1 unit of rest. Therefore, music will not sound "slurred".

If :P is -26 or -27 NOTE generates a sound for one unit followed by :D-1 units of rest. This makes it convenient to use the percussion sounds to "generate a beat".

NOTE can also take multiple inputs . The format is (NOTE :P1 :D1 :P2 :D2 etc.) where each pair specifies the pitch and duration for one note. (The total number of inputs must be even.)

14.2 PM

The command PM (stands for PLAY MUSIC) takes no inputs. It causes the output of previous NOTE commands to be played on the music box. As the music is played it is erased from temporary storage and must be regenerated if you wish to hear it again. Typing a ctl-G while music is being played causes an immediate break and also clears out temporary storage. MCLEAR can be used to clear out the music buffer. (Temporary storage is allocated in the same area as the user's procedures, variables, etc. The amount of music that may be compiled at one time depends therefore on how much other stuff is in the user workspace.)

14.3 VOICE

The music system can generate up to four simultaneous voices. The VOICE command directs the output of subsequent NOTE commands to the various voices. PM then plays the voices simultaneously.

Example:

```
TO SEVENTH :TONIC :D
10 VOICE 1 NOTE :TONIC :D
20 VOICE 2 NOTE :TONIC+3 :D
30 VOICE 3 NOTE :TONIC+6 :D
40 VOICE NOTE :TONIC+9 :D
50 PM
END
```

will play a chord.

VOICE takes one input which must be a number between 1 and 4. This becomes the current voice until the next voice command is given. If no VOICE command is given, the system outputs to voice number 1.

14.4 VLEN and MLEN

These are operations which take no inputs. VLEN outputs the total length of the music compiled for the current voice. MLEN outputs the length for the largest voice.

For instance, if we have generated a melody in voice number 1 we can provide it with an accompaniment:

```
TO OOMPAH
10 VOICE 2
20 (NOTE 0 4 (-5) 4)
30 IF VLEN = MLEN STOP ELSE GO 20
END
```

This will generate oompahs for as long as the melody lasts.

14.5 NVOICES

The music system normally multiplexes output among four voices. Voices for which no output has been generated are fed silences. It is also possible to send output to only one or two voices. This is done with the NVOICES command:

NVOICES 1 output only to voice 1

NVOICES 2 output to voices 1 and 2

NVOICES 4 output to all four voices (the normal mode)

Since the music box is fed at a constant rate NVOICES 1 (resp. NVOICES 2) causes the basic unit of duration to be one-fourth (resp. half)

as long as with NVOICES 4. NVOICES also clears out temporary storage and resets the current voice to voice 1.

14.6 NOMUSIC

Only one user at a time may have access to the music box. When a user gives a music command, LOGO assigns the music box to him if no one else if using it. A user may release the music system by giving the NOMUSIC command.

14.7 MUTYO AND MUCTRL

These commands are a contibution toward real-time music generation. They enable the user to bypass the muusic buffer, and the PM command, and hence have a chance at least to generate real-time music.

MUTYO takes 2 inputs, each one a music box pitch, and makes the music box play the pitches. Pitches to MUTYO are the same numbers as pitches in the regular music system -- middle C is 0, rest is -28, and so forth. MUTYO automatically converts those pitch numbers into the numbers the music box hardware requires.

MUCTRL stands for music control. Its main purpose is to specify to the music box hardware how many voices you wish to load. This is similar to the NVOICES command in the regular music system.

14.8 SYSTEM CONSTRAINTS

The music system has been designed with specific uses in mind. (See, e.g, the papers of Jeanne Bamberger.) Users who are hampered by this

should bear in mind that the .TYO command can be used to output arbitrary characters to the music box. (See Chapter 19.)

15. LISTS

LOGO is equipped to handle general lists, i.e. lists whose elements may themselves be lists. For example:

```
[[THIS IS] A [LIST STRUCTURE]]
```

is a list of three elements, two of which are lists.

PRINT does not print the outer brackets around a list.

```
?PRINT [[THIS IS] A [LIST STRUCTURE]]
[THIS IS] A [LIST STRUCTURE]
```

All of the list operations described in Section 4 work with general lists, e.g.:

```
MAKE "A [[THIS IS] A [LIST STRUCTURE]]
COUNT :A output 3
FIRST :A outputs [THIS IS]
BUTFIRST :A outputs [A [LIST STRUCTURE]]
LAST :A outputs [LIST STRUCTURE]
BUTLAST :A outputs [[THIS IS] A]
```

SENTENCE generalizes as follows:

If all of its inputs are lists, it puts their elements together to make one big list:

```
(SENTENCE [PIECES OF] [A BIG] [LIST])
outputs [PIECES OF A BIG LIST]
```

If any of the arguments to SENTENCE are words it first converts each word to a one-word list and applies the above rule:

```
(SENTENCE "PIECES [OF] "A "BIG [LIST])
outputs [PIECES OF A BIG LIST]
```

LOGO has some other list operations. LIST takes two inputs and outputs a 2-element list:

```
LIST [THIS IS] [A LIST]
outputs [[THIS IS] [A LIST]]
```

LIST can also take multiple inputs. It outputs a list whose

elements are the inputs:

```
(LIST [LOTS] "OF [LISTS])
outputs [[LOTS] OF [LISTS]]
```

Notice that if all inputs to LIST are words, then LIST is equivalent to SENTENCE.

FPUT is another list operation. It takes two inputs of which the second must be a list. It sticks the first input onto the front of the second to make a new list:

```
FPUT "THIS [IS HOW FPUT WORKS]
outputs [THIS IS HOW FPUT WORKS]

FPUT [THIS IS] [HOW FPUT WORKS]
outputs [[THIS IS] HOW FPUT WORKS]
```

LPUT is similar. It sticks its first input onto the end of the list.

```
LPUT "THIS [IS HOW LPUT WORKS]
outputs [IS HOW LPUT WORKS THIS]

LPUT [THIS IS] [HOW LPUT WORKS]
outputs [HOW LPUT WORKS [THIS IS]]
```

FPUT and LPUT can also take multiple inputs. The last input must always be a list:

```
(FPUT "MAKE [THIS] [A LIST])
outputs [MAKE [THIS] A LIST]

(LPUT "MAKE [THIS] [A LIST])
outputs [A LIST MAKE [THIS]]
```

Lists may also contain SNAPS as elements, e.g.:

```
MAKE "A SENTENCE "WOW SNAP
```

then it is perfectly valid to say:

```
PRINT FIRST :A
```

or

```
DISPLAY LAST :A
```

but

```
PRINT :A
```

will give an error.

16. DEBUGGING FEATURES

LOGO includes features which aid users in debugging their programs. The simplest such feature is ctl-G. Pressing ctl-G will stop execution of any program. If it is not much help as a debugging feature, it at least can serve as a panic button.

16.1 PAUSE, CONTINUE AND ctl-Z

When the PAUSE command is executed in a procedure, the procedure is temporarily halted and control is returned to the console. Instead of typing only ? as a prompt character, LOGO also indicates at what level it currently is. Level refers to "how many procedures deep" current execution is. (Commands given from the console are at level 0. Commands given in procedures called from the console are at level 1. Commands given in procedures called by level 1 procedures are at level 2. And so on.) While in a PAUSE the user can access any names which are local to the procedure. For example:

```
TO BLA :A
10 FORWARD :A
20 PAUSE
END
```

A is a local name. Running BLA will cause LOGO to pause. We can then access the value of A,

```
?BLA 20
PAUSE AT LEVEL 1 IN LINE 20 IN BLA
L1?PRINT :A
20
L1?
```

There are two ways to get out of a pause. Typing ctl-G will, as usual, return the user to the top level.

The CONTINUE command (abbreviated CO) will continue executing the procedure starting with the next line after the PAUSE. (Note that for this reason PAUSE should be the last command on a line if the user wishes to continue.)

While it is legal to give any LOGO command in a PAUSEd situation, the user may get into trouble if he or she erases or edits the procedure and then tries to continue.

Ctl-Z is similar to ctl-G except that it generates a pause rather than a break at the top level.

16.2 DEBUG

The DEBUG command changes what LOGO does when an error is encountered in a user's program. Normally, an error prints a message and terminates execution. The DEBUG facility causes errors to generate PAUSEs. The user can then examine local variables and CONTINUE with the next line of the procedure. DEBUG is a command which takes no inputs. Its use switches this feature on and off.

16.3 TRACE

The TRACE command takes one input, which is the name of a procedure. The procedure name is not quoted. Every time a TRACEd procedure is run, LOGO prints out a message to that effect and tells what the inputs to the procedure are. LOGO also indicates if the procedure outputs.

To get rid of a TRACE, use ERASE TRACE as in:

?ERASE TRACE <procedure name>

It is possible to trace all procedures by saying TRACE ALL. There is also ERASE TRACE ALL.

16.4 STEP

STEP is like a "super TRACE". Not only is the procedure TRACEd but before each line of the procedure is executed, LOGO types out the line and waits for the user to respond. There are three options:

(1) Typing a carriage return causes the line to be executed and goes on to the next line.
(2) Typing ctl-G terminates execution as always.
(3) Typing ctl-Z generates a PAUSE as described above. The user may then execute other commands and later CONTINUE.

The syntax for STEP is like TRACE. ERASE STEP, STEP ALL, and ERASE STEP ALL are also available.

17. ERROR HANDLING

The error handling facilities allow you to modify the way LOGO treats errors.

17.1 ERSET and ERCLR

The ERSET command takes one input, a procedure name. As with TO, EDIT, TRACE, etc., the procedure name is not quoted. ERSET causes the procedure given as input to be executed every time an error occurs. The procedure will also be run every time ctl-G or ctl-Z is hit. After the procedure has been executed, LOGO takes the following action:

(1) If the procedure does not output, LOGO prints the normal error message.

(2) If the procedure outputs, LOGO prints the output instead of the normal error message.

If an error occurs in the ERSET procedure itself the ERSET will not happen. The various SYSTEM BUG errors cannot be overridden by an ERSET. The ERCLR command, which takes no inputs, deactivates ERSET.

17.2 ERRET and ERLIN

As described above, an ERSET can provide personal error messages, but still terminates execution. Another use of ERSET is to allow the possibility of modifying the condition that caused the error and continuing execution. The command ERRET takes one input, a line number, and returns execution to that line of the procedure in which the error occurred. Useful in conjunction with this is the ERLIN operation which outputs the line number in which the error occurred. Thus, for example:

ERRET ERLIN

in an ERSET procedure will re-execute the line in which the error occurred.

Example:

Here is a way to move a ball back and forth acoss the screen. Supose :BALL is a SNAP of a ball and we move the ball by:

```
TO MOVEBALL
10 FORWARD 20
20 DISPLAY :BALL
30 WAIT 5
40 WIPE :BALL
50 GO 10
END
```

Run MOVEBALL but first ERSET the following procedure:

```
TO TURN
10 LEFT 180
20 ERRET 10
END
```

Now every time the "OUT OF BOUNDS" error occurs TURN will turn the turtle around and keep going.

But this simple scheme has a bad bug. TURN will be run whenever any error occurs. Even worse, it will be run when we hit ctl-G. There is no way to stop MOVEBALL! What we really want is to only execute the ERRET in the case of the particular error "OUT OF BOUNDS".

17.3 ERNAM, ERBRK, and ERPRO

To help overcome the above-mentioned bug, LOGO provides the ERNAM operation. The easiest way to find out the name of an error, is to generate the error and then PRINT ERNAM. We can fix the bug in TURN above:

```
TO TURN
10 TEST ERNAM = "OOB
20 IFFALSE STOP
30 LEFT 180
40 ERRET 10
END
```

("OUT OF BOUNDS" has error name OOB)

ERPRO outputs the name of the procedure in which the last error occurred, so if we would like TURN to only take effect during MOVEBALL we can TEST to see if ERPRO = "MOVEBALL.

ERBRK handles ctl-G and ctl-Z. It is an operation which outputs:

1 if the "error" was caused by pressing ctl-G

-1 if the "error" was caused by pressing ctl-Z

0 otherwise.

Sometimes it is useful to set things up so that pressing ctl-G during a REQUEST re-does the REQUEST rather than stopping the program. Here is an example of how that can be done:

```
TO SPECIALREQUEST
10 ERSET TRYAGAIN
20 MAKE "X REQUEST
30 ERCLR
40 OUTPUT :X
END

TO TRYAGAIN
10 IF ERBRK = 1 ERRET ERLIN
END
```

With TRYAGAIN we can still stop the show by hitting ctl-Z.

17.4 ERNUM, ERTOK, ERLOC, and ILINE

ERNUM Each type of error has a number as well as a name. ERNUM outputs the number of the error.

ERTOK outputs the "token number" at which the previous error occurred in the line. This gives some indication of exactly where in the line the error occurred, but it is hard to use unless you are initiated into the mysteries of the LOGO evaluator.

ERLOC outputs the location in the PDP11's core at which the error occurred. It is useful mainly to system programmers.

ILINE outputs the last line typed in at the console. It is useful for doing analysis of errors that occurred while the user was typing in "direct commands".

18. EVALUATING TEXT

18.1 RUN

It is often useful to evaluate commands that have been computed rather than typed in. The basic LOGO primitive which does this is RUN. RUN takes one input, a list, and evaluates it just as if the list were typed in at the console:

```
?RUN [PRINT "WOW]
WOW
```

If the input to RUN specifies an operation, then RUN outputs:

```
?PRINT RUN [SUM 18 5]
23
```

Of course, the input to RUN need not be typed in literally:

```
?PRINT RUN (SE "SUM 18 5)
23
```

For example:

```
?MAKE "X "PRINT
?RUN SE :X 5
5
```

Another example:

If we have procedures called, say, STRATEGY1, STRATEGY2, and STRATEGY3, one way to invoke the proper one is to say:

```
RUN (SE WORD "STRATEGY :N)
```

if :N is 1, 2, or 3. We use SE with one argument since RUN's input must be a list.

There is a tricky point here. RUN executes the list <u>just as if the list were typed in</u>. In the example

```
RUN [PRINT "WOW]
```

the first character in the second word of the input is a quote. To

generate the list we would say

```
RUN SE "PRINT ""WOW
```

In contrast the command

```
RUN SE "PRINT "WOW
```

is equivalent to

```
RUN [PRINT WOW]
```

which would be an error unless there were a procedure named WOW.

18.2 PROCEDURES DEFINING PROCEDURES

It is valid in LOGO to have a procedure which defines another procedure:

```
TO ORIB
10 TO ORAB
20 10 PRINT "WHOOPIE
30 END
END
```

Running ORIB defines the procedure:

```
TO ORAB
10 PRINT "WHOOPIE
END
```

Naturally, we can use RUN in this context:

```
TO DEFINE :PROC :N
10 RUN SE "TO :PROC
20 RUN SE [10 PRINT] :N
30 END
END
```

will cause

```
DEFINE "WHAMO 7
```

to create

```
TO WHAMO
10 PRINT 7
END
```

In a similar manner, procedures may edit procedures.

19. MISCELLANEOUS COMMANDS

19.1

BELL takes no inputs. Rings the bell on the console.

CLOCK takes no inputs. Outputs a number which is incremented every 1/60 seconds.

DATE outputs a 3-element list containing month, day, year.

TIME outputs a 3-element list containing hour, minute, second.

WAIT takes one input. Causes LOGO to wait for that many 1/30 second intervals.

HELLO clears out workspace; "restarts" LOGO for you.

GOODBYE same as HELLO.

LEVEL takes no inputs. Outputs a number which tells "how many procedures deep" current execution is.

PEEK prints out system status information.

MAIL The MAIL command enables users to send messages to other users on LOGO. MAIL takes one input- the user name (quoted) of the person to whom the mail is to be sent. The computer responds with a back-arrou (←) which indicates that anything typed in on the console is regarded as mail. To end the message, type a line with a single period followed by a carriage return. The completed message is then placed in the filing system of the user to whom it is sent, contained in a file named MAIL. The mail can be read either when logging in, or by printing the file using the PDF command. It is helpful to be logged in when sending mail, since the user's login name is used to identify the source of the mail to the person who receives it.

SEND takes two inputs. The first is the number of a console, and the second is a list which is printed out as a message directly on that console.

SYSPR takes as input a list which is printed out as a message on all consoles currently in use.

SIN takes one number (representing degrees) as input and returns the sine of that angle.

COS takes one input in degrees and outputs the cosine of that angle.

19.2 IMPLEMENTATION DEPENDENT PRIMITIVES

The following are primitives which refer specifically to this inplementation of LOGO.

.TYO takes two inputs. The first specifies a device. This may be either a device number or name. The names that LOGO understands for devices are:

"LIGHT	for the light box
"TUR1	for turtle 1
"TUR2	for turtle 2 when it is plugged in
"MUSIC	for the music box
"PLOTTER	for the plotter
"TTY	for you own console.

The second input specifies a number which is relayed to the given device as output.

The obvious use of this feature is with regard to the light box.

.TYOing to a turtle is slightly useful; certain commands that can be done by the turtle hardware have never been implemented in LOGO software (mainly because they are only very slightly useful). The turtle can, for instance, waddle instead of walk.

Try this program:

```
TO WADDLE :STEPS
10 IF :STEPS = 0 STOP
20 .TYO "TUR1 94
30 .TYO "TUR1 95
40 WADDLE :STEPS - 1
END
```

The interested reader is referred to the TURTLE GENERAL Engineering Handbook, Appendix A for more details.

.TYI takes one input which specifies a device in the same manner as .TYO. The next piece of information (interpreted as a number) received from the device is given as output.

.CLOSE If you use .TYI or .TYO, LOGO assigns the device to you so that no one else can use it. To release the device the .CLOSE command is used, taking as its input the given device specification.

.CTYI and .CTYO are similar to .TYI and .TYO except that the specified device is the user's console. .CTYO takes one input, .CTYI takes none.

CTYOWAIT and TYOWAIT

These commands are a species of WAIT command -- like WAIT 60 -- but instead of waiting a fixed amount of time, they wait until a teletype has finished typing out.

A simple use of CTYOWAIT is to synchronize a procedure that draws

on the display and also prints stuff out. Suppose you wish to draw a POLY and print the turtle's heading at each corner. Normally in such a program the displaying gets further and further ahead of the printing. But, CTYOWAIT will force the procedure to wait for the printing to be completed before going on:

```
TO POLY :SIDE :ANG
10 FORWARD :SIDE
20 RIGHT :ANGLE
30 PRINT HEADING
40 CTYOWAIT 10 !The number is how much time to wait in
addition to waiting for the teletype.
50 POLY :SIDE :ANGLE
END
```

TYOWAIT is like CTYOWAIT, except that you specify which device to wait for. TYOWAIT understands the same device names that .TYO and company understand. It also accepts teletype numbers. A good use for TYOWAIT is to synchronize the floor turtle with the display turtle.

.GUN takes one input, a user number, and restarts that user's LOGO (equivalent to typing HELLO at his console). This is used when someone's consle gets hopelessly hung up.

.CASESW On consoles equipped to handle lower case, LOGO normally converts characters to upper case. .CASESW is a command which takes no inputs and turns this feature on and off.

.ECHOSW takes no inputs. Turns off and on a feature which inhibits echoing of characters typed at the console.

20. MAINLY FOR SYSTEM PROGRAMMERS

The following primitives deal with the maintenance and debugging of the LOGO system. They are useful mainly to system programmers.

.STF and .CTF — set and clear a trace of the evaluator.

.SGCF and .CGCF — set and clear a feature which causes a garbage collection each time a node is allocated.

.SPNF and .CPNF — set and clear printing the number of free nodes left when garbage collections are run.

.STATUS — unlock protected commands

.PWRCLR — do a reset of the PDP 11 and all devices. Not to be used casually.

.RUG — halt LOGO and start the debugger

.SETTV — set the system time variables

.GCOLL — run a garbage collection.

.EXAMINE — examine location in PDP11 core.

.DEPOSIT — deposit into core.

.NODES — output number of free nodes.

.VERSION — output which version of LOGO this is.

.VALUE — output the item on top of the S-PDL.

.USRTIME — output the amount of user time run by the system.

21. AN INDEX OF LOGO PRIMITIVES

The pages that follow are a listing of present LOGO primitives, with the exception of the special commands listed in Chapter 20. Along with each primitive is the following information:

ABB.-- Abbreviations and infix forms. Infix forms are indicated by (i).

NUMBER OF INPUTS-- The indication nV means the primitve may use the variable input feature (See Section 4.4), where n is the standard number of inputs.

OUTPUT-- Y means it does output, N means it does not.

PAGE(S)-- The primary page(s) on which the primitive is discussed, plus any important supplementary information.

PRIMITIVE	ABB	NUMBER OF INPUTS	OUTPUT	PAGE(S)
ALSO		1	N	55-56
ASIZE		1	Y	23
BACK	BK	1	N	46,48,51
BELL		0	N	75
BOTH		2	Y	30
BTOUCH		0	Y	47
BURY		1	N	36
" ALL		0	N	36
BUTFIRST	BF	1	Y	12,62
BUTLAST	BL	1	Y	12,62
.CASESW		1	N	77
CLEARSCREEN	CS	0	N	51
CLOCK		0	Y	75
.CLOSE		0	N	76
.CLOSEF		0	N	44
CNTRL		1	N	55-56
CONTENTS		0	N	35
CONTINUE	CO	0	N	66
COS		1	Y	76
COUNT		1	Y	12.2
CRINDEX		1	N	40-41
.CTYI		0	Y	77
.CTYO		1	N	77
CTYOWAIT		0,1	N	77
DATE		0	Y	75
DEBUG		0	N	66
DEFINEARRAY	DEFAR	3,4,5	N	21-24
DIFFERENCE	- (i)	2	Y	16-18
DISPLAY		1	N	53
.ECHOSW		1	N	78
EDIT	ED	1	N	8,73
" LINE	EDL	1	N	7
" TITLE	EDT	0	N	7
EITHER		2	Y	30
ELSE		0	N	31-32
EMPTYP		1	Y	30
END		0	N	6
EQUAL	=(i)	2	Y	17-18,30
ERASE	ER	1	N	8,35
" ALL		0	N	35
" ARRAY		1	N	22
" ARRAYS		0	N	22
" BURY		1	N	36
" BURY ALL		0	N	36
" FILE		1	N	38
" LINE	ERL	1	N	8
" NAME		1	N	35

" NAMES		0	N	35
" PROCEDURES		0	N	35
" STEP		1	N	67
" STEP ALL		0	N	67
" TRACE		1	N	66-67
" TRACE ALL		0	N	67
ERBRK		0	Y	70
ERCLR		0	N	68
ERLIN		0	Y	68-69
ERLOC		0	Y	71
ERNAM		0	Y	71
ERNUM		0	Y	70
ERPRO		0	Y	70
ERRET		1	N	68-69
ERSET		1	N	68
ERTOK		0	Y	70-71
.FILEP		1	N	43
.FILER		0	Y	43-44
FIRST	F	1	Y	12,62
FORWARD	FD	1	N	46,48,51
FPRINT		1V	N	14-15,40
FPUT		2V	Y	63
FTOUCH		0	Y	47
GET		2,3,4	Y	22,24
GO		1	N	29
GOODBYE		0	N	75
GREATER	> (i)	2	Y	17-18
.GUN		1	N	78
HEADING		0	Y	54-55
HELLO		0	N	75
HERE		0	Y	54
HIDETURTLE	HT	0	N	52
HOME		0	Y	54
IF		1	Y	31-32
IFFALSE	IFF	0	N	32
IFTRUE	IFT	0	N	32
ILINE		0	Y	71
INTEGER		1	Y	20
LAMPOFF		0	N	47,48
LAMPON		0	N	47,48
LAST	L	1	Y	12,62
LEFT	LT	1	Y	46,48,51
LESS	< (i)	2	Y	17-18
LEVEL		0	Y	75
LIGHT		0	Y	48
LIST		2V	Y	62-63
LISTP		1	Y	30
LOCAL		1	N	27
LOGIN		1	N	44
LPUT		2V	Y	61

LTOUCH		0	Y	48
MAIL		1	N	75
MAKE	← (i)	2V	sometimes	25-26
MCLEAR		0	N	58
MLEN		0	Y	59
MUCTRL		1	N	60
MUTYO		2	N	60
NEWSNAP		0	N	53
NODISPLAY		0	N	51
NOMUSIC		0	N	59-60
NOPLOTTER		0	N	55
NOT		1	Y	30
NOTE		2	N	57
NOTURTLE		0	N	46
NOWRAP		0	N	52
NUMBERP		1	Y	20,30
NVOICES		1	N	59
.OPENA		1	N	43
.OPENR		1	N	43-44
.OPENW		1	N	42-43
OUTPUT		1	N	29
PAUSE		0	N	65
PEEK		0	N	75
PENDOWN	PD	0	N	47,52
PENP		0	Y	52
PENUP	PU	0	N	47,52
PLOTTER		0	N	55
PM		0	N	58
PRINT	PR	1V	N	5,10,62
PRINTOUT	PO	1 or 0	N	8,34
" ALL		0	N	34
" ARRAY		1	N	24
" ARRAYS		0	N	24
" FILE		1	N	38
" INDEX	POI	0	N	38-40
" LINE	POL	1	N	8
" NAMES		0	N	34
" PROCEDURES		0	N	34
" TITLE	POT	0	N	8
" TITLES	POTS	0	N	24
" TREE		1	N	39-40
PRODUCT	* (i)	2V	Y	16-18
QUOTIENT	/ (i)	2	Y	16-18
RANDOM		0	Y	18
READ		1	N	38,40,42
READPTR		0	N	45
REMAINDER	\ (i)	2	Y	16-18
REQUEST		0	Y	13-14
RIGHT	RT	1	N	46,48,51
RTOUCH		0	Y	47

RUN		1	sometimes	72-74
SEND		2	N	76
SENTENCE?	SE	2V	Y	13,62
SETASIZE		1	N	21-22
SETHEADING		1	N	55
SETIINDEX?	SETI	1	N	41-42
SETTURTLE?	SETT	1	N	54
SETX		1	N	54
SETXY		2	N	54
SETY		1	N	54
SHOWTURTLE	ST	0	N	52
SIN		1	Y	76
SNAP		0	Y	52-53,38
STARTDISPLAY		1	N	51
STEP		1	N	67
" ALL		0	N	67
STOP		0	N	29
STORE		3,4,5	N	22-24
SUM	+ (i)	2V	Y	16-18
SYSPR		1	N	76
TEST		1	Y	32
TEXT		1	Y	35
THING		1	Y	25
TIME		0	Y	75
TO		1 or more	N	6,73
TOOT		1	N	47
TOPLEVEL		0	N	29
TRACE		1	N	66
" ALL		0	N	66
TURTLE		1	N	46
.TYI		1	Y	76-77
.TYO		2	N	77,49,61
TYOWAIT		1,2	N	78
TYPE		1V	N	10
TYPEIN		0	Y	14
USE		1	N	37,40
VLEN		0	N	59
VOICE		1	N	58
WAIT		1	N	75
WIPE		1	N	54
WIPECLEAN	WC	0	N	51
WORD		2V	Y	13
WORDP		1	Y	38
WRAP		0	N	52
WRITE		1	N	37
WRITEPTP		0	N	45
XCOR		0	Y	54
YCOR		0		