DOCUMENT RESUME

ED 118 368 SE 019 898

AUTHOR Goldstein, Ira; And Others
TITLE LLOGO: An Implementation of LOGO in LISP. Artificial Intelligence Memo Number 307.
INSTITUTION Massachusetts Inst. of Tech., Cambridge: Artificial Intelligence Lab.
SPONS AGENCY Advanced Research Projects Agency (DOD), Washington, D.C.; National Science Foundation, Washington, D.C.
REPORT NO LOGO-11
PUB DATE 27 Jun 74
NOTE 77p.; For related documents, see ED 077 236, 240-243, SE 019 893-894, and 896-900

EDRS PRICE MF-$0.83 HC-$4.67 Plus Postage
DESCRIPTORS Artificial Intelligence; *Computer Programs; *Computers; *Computer Science Education; Instruction; *Manuals; Mathematics Education; Problem Solving; *Programing Languages

ABSTRACT

LISP LOGO is a computer language invented for the beginning student of man-machine interaction. The language has the advantages of simplicity and naturalness as well as that of emphasizing the difference between programs and data. The language is based on the LOGO language and uses mnemonic syllables as commands. It can be used in conjunction with character-oriented display terminals, graphic display systems, and music generation. This document provides a discussion of the merits of LISP LOGO, as well as a user's manual for the language. (SD)

****
* Documents acquired by ERIC include many informal unpublished *
* materials not available from other sources. ERIC makes every effort *
* to obtain the best copy available. Nevertheless, items of marginal *
* reproducibility are often encountered and this affects the quality *
* of the microfiche and hardcopy reproductions ERIC makes available *
* via the ERIC Document Reproduction Service (EDRS). EDRS is not *
* responsible for the quality of the original document. Reproductions *
* supplied by EDRS are the best that can be made from the original. *
****

U.S. DEPARTMENT OF HEALTH, EDUCATION & WELFARE
NATIONAL INSTITUTE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

ED118368

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

June 27, 1974

A. I. MEMO 307 LOGO MEMO 11

L L O G O:

An Implementation of LOGO in LISP

Ira Goldstein

Henry Lieberman

Harry Bochner

Mark Miller

Abstract:

This paper describes LLOGO, an implementation of the LOGO language written in MACLISP for the ITS, TEN50 and TENEX PDP-10 systems, and MULTICS. The relative merits of LOGO and LISP as educational languages are discussed. Design decisions in the LISP implementation of LOGO are contrasted with those of two other implementations: CLOGO for the PDP-10 and 11LOGO for the PDP-11, both written in assembler language. LLOGO's special facilities for character-oriented display terminals, graphic display "turtles", and music generation are also described.

This work was supported in part by the National Science Foundation under grant number GJ-1049 and conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0005. Reproduction of this document in whole or in part is permitted for any purpose of the United States Government.

PERMISSION TO REPRODUCE THIS COPYRIGHTED MATERIAL HAS BEEN GRANTED BY
MIT's Artificial Intelligence Lab's LOGO Project
TO ERIC AND ORGANIZATIONS OPERATING UNDER AGREEMENTS WITH THE NATIONAL INSTITUTE OF EDUCATION FURTHER REPRODUCTION OUTSIDE THE ERIC SYSTEM REQUIRES PERMISSION OF THE COPYRIGHT OWNER

19 898

TABLE OF CONTENTS

Section 1. Why Implement LOGO in LISP

LISP has proved itself to be a powerful language for representing complex information processing tasks. This power stems from:

1. The uniform representation of programs and data.

2. The ability to build arbitrarily complex data structures in the form of s-expressions.

3. Recursion.

Power, however, is not necessarily good pedagogy. LOGO is a computer language designed especially for the beginner. Its purpose is to introduce the fundamental ideas of computation as clearly as possible.

LISP LOGO is an implementation of LOGO in LISP. It has been designed for several reasons. The first is that these two languages share a fundamental core in common. Both are time shared, interpretive languages capable of full recursion. Variable and procedure names may be any string of letters and digits. Sub-procedure definitions are independent of super-procedures. Both numerical and list-structured information can be manipulated with equal facility. Thus, the LOGO systems programmer is freed of the necessity of re-developing various facilities already available in LISP (lists, recursion, garbage collection, error service traps, interrupts). He can concentrate on additions (better error analysis) and modifications (pedagogical simplifications) to LISP. LLOGO unifies language development across a broad spectrum ranging from PLANNER and CONNIVER through LISP to LOGO.

A second reason for this implementation is to provide a natural transition to the more powerful computational world of LISP as the student grows more sophisticated. When desired, the student has access to all of the capabilities of LISP including:

Arrays
Functions of arbitrary number of inputs
Functions that do not evaluate their inputs
MICRO-PLANNER and CONNIVER
Interrupts
LISP compiler
Property lists
Floating point numbers
Character display cursor manipulation
Infinite precision fixed point arithmetic

Section 2. Differences between LOGO and LISP

The differences between LOGO and LISP can be described on the basis of three educational goals:

Simplicity	of both the computational and explanatory kind.
Naturalness	wherein the overhead for a naive user is minimized by following standard English conventions.
Disparity	which emphasizes the distinction between various modes such as defining versus running programs.

It should be noted, however, that there can be no one unique solution to the "best" educational language: These three goals can conflict. Furthermore, they cannot be so emphasized that important ideas of computation are completely eliminated from the language. For students of different backgrounds, simplicity and naturalness may have very different meanings. Hence, alternatives to the particular choices made in designing CLOGO and 11LOGO are also described. This section may be viewed as presenting a spectrum of possibilites from which a teacher can build a computational world tailored to his own pedagogical purposes.

2.1 Simplicity

Lists versus Sentences

Lists have a simple recursive definition. A list is either

1. *NIL*, the empty list
2. (word1 word2 . . .) , a sequence of words (= atoms)
3. A list of lists.

This definition is confusing when the student is still having trouble with the concept of recursion. CLOGO limits itself to lists built from only the first two of these three clauses. Such lists are called "sentences".

> Alternative view: the concept of recursion is too important to be eliminated from LOGO. Recursive programs are allowed. Educationally, the more examples of recursion available, the easier it is to understand. Hence, lists should be allowed.

Computational power is not always in conflict with educational simplicity. In addition to the standard list operations of *FIRST* (*CAR*) and *BUTFIRST* (*CDR*), LOGO provides *LAST* and *BUTLAST*. Furthermore, all four of these operations work on words as well as sentences. The fact that word manipulation is more costly than list manipulation for LISP, or that taking the *LAST* of a list is more expensive than computing its *FIRST* is not of interest to the beginner. The natural symmetry of having all of these operations is to be preferred.

> Alternative view: LOGO introduces two data types - words and sentences.

There is both an empty word and an empty sentence. LISP's world is easier to understand. There is only one type of data, s-expressions. Primitives like *CAR* are list operations only; they do not operate on words by manipulating the word's print name, as LOGO's *FIRST* does.

Repeatedly *BUTFIRST*ing a sentence in LOGO always terminates in the empty list. In LISP, with its more general list structure built from "dotted pairs" and *CONS*ing, this is not always so. The result is the possibility of "slip-through" bugs for *EMPTYP* endtests of recursive procedures. Thus, LOGO eliminates a common source of error without significantly limiting computational power.

Alternative view: Allowing an atom to be the *CDR* of an s-expression sometimes allows for economy of storage. Also, the symmetry of *CAR* and *CDR* in LISP make the data structure easier to explain, although they are symmetric as list operations only for the particular representation of lists used in LISP.

Rigid program form

LISP allows programs to be lists of any form. Editing and debugging consequently become awkward due to the difficulty in naming parts of the program. LOGO simplifies program structure by requiring that a program be a series of numbered lines. The locations of bugs and intended edits are then far easier to describe.

Criticism: LOGO violates this assumption by allowing the user to create lines of unlimited complexity. It would be preferable to limit a line to a single top level call. This does not prohibit nesting, a fundamental idea in computation. But it does prohibit defeating the entire point of line numbers with such code as:

>10 FD 100 RT 90 FD 100 RT 90 . . .

An alternative scheme might be to adopt a "DDT" like convention. Lines are identified by offsets from user-defined location symbols. This has the advantage of encouraging the use of mnemonic names for portions of the user's program, rather than line numbers, which have no mnemonic value, while retaining the virtue of having a name for every part of the program. The user would not have to renumber lines if he wanted to insert more lines between two lines of code than the differenee between their line numbers.

Integer Arithmetic

The initial CLOGO world limits the user to integer arithmetic. The rationale behind this is to avoid the complexity of decimal fractions. This is clearly a simplification whose value depends on the background of the students.

Criticism: even for elementary school children, this simplification may cause confusion. Most beginners are troubled with

$$\frac{1}{2} = 0$$

Proponents of fixed point arithmetic might reply that this is no worse than

$$\frac{1}{1} = .999999$$

However, a decimal printer can be clever in performing roundoff.

Other alternatives are to limit arithmetic to rational numbers, or to use the following LISP convention: Numbers are fixed point unless ending in a decimal fraction. Operations only return fixed point if both operands are fixed point.

Another virtue of LISP is that fixed point numbers can be infinitely large. Arbitrary limitations due to the finite size of the computer's word do not exist to confuse the beginner.

Conditionals

LOGO allows the following type of branching:

```
>10 TEST <predicate>
>20 IFTRUE ....
>30 IFFALSE ....
```

TEST sets a flag which subsequent *IFTRUE*'s and *IFFALSE*'s access. This avoids the necessity of the entire conditional appearing on a single line of the procedure. The student has explicit names in the form of line numbers for each branch.

Criticism: This prevents nesting of conditionals. A second conditional wipes out the results of the first. In LLOGO, the flag set by *TEST* is simply a LISP variable. Since it is global, *TEST*'s in sub-procedures can affect *IFTRUE*'s in the super-procedure. This introduces a non-local nature to control structure.

LOGO's lack of canned loops such as *DO* and *MAPCAR* can be criticized as encouraging bad programming practice, such as excessive use of *GO*. This obscures the logical structure of programs. Also, it may be significantly confusing to the beginner, and the source of many bugs. A child might understand quite well a control structure concept like "do this part of the program three times", or "do this part of the program for each element of the list", but may be unable to open-code that control structure in terms of jumps and conditionals. LOGO programs can't be "pretty printed" to reveal their logical structure as can programs written in LISP or a block structured language.

2.2 Naturalness

Mnemonic Names

An obvious virtue of any computer language is to use procedure names whose English meaning suggests their purpose. Consequently, LISP's primitives *CAR* and *CDR* are renamed *FIRST* and *BUTFIRST*.

Note: Everyone remembers how un-mnemonic *CAR* and *CDR* are. However, most LISP primitives are named after their English counterparts.

CLOGO syntax allows the use of certain "noise words", words which appear in the user's code, but have no effect beyond making the code read more like English sentences. For example, in the following lines of LOGO code, the *AND*, *OR*, *THEN*, and *TO* are permitted but serve no computational purpose. They do not designate procedures, as is the usual case with words not beginning with a colon.

?BOTH <predicate 1> AND <predicate 2>
?EITHER <predicate 1> OR <predicate 2>
?IF <predicate 1> THEN ...
?GO TO .....

However, as the student gains more insight into LOGO, noise words become a burden. They complicate the task of the parser, preventing the student from feeling that he really understands the language. Most of the noise words have been eliminated in both PDP11 LOGO and LISP LOGO. [LLOGO will tolerate *THEN* in conditionals, and *TO* in transfers, however, because they are so commonly used.]

Matching English vocabulary to computer functions can be difficult. English words rarely have a single meaning. Following are some examples where CLOGO may have made the wrong choice.

1. CLOGO uses *IS* instead of *EQUAL* for its equality predicate. The rationale is that *IS* will be more familiar to a non-mathematical beginner. However, the omnipresent nature of this English verb results in such LOGO code as:

?TEST IS :THIS.NUMBER GREATERP :THAT.NUMBER

thus, it might be better for LOGO to use *EQUAL*.

2. Another example where LOGO may have chosen the wrong word is in defining procedures. This is done via:

?TO PROCEDURE.NAME :INPUT1 :INPUT2 ...

The English word "to" can imply execution. For example, "he is to run his program". A better choice would be "define".

Parsing

LISP avoids the necessity of parsing through the use of parentheses. This might be considered well worth emulating in LOGO for its explanatory simplicity. However, simplicity must be contrasted with naturalness. A beginner is used to using English where verbs and modifiers are connected by grammar, context and meaning rather than explicit parenthesizing. This naturalness can be preserved for procedures that take a fixed number of inputs. This allows such lines of code to be understood by anyone without any special programming knowledge.

?FORWARD 100 RIGHT 90

Thus, a beginner can express himself with no extra burden of parenthesizing when his programs are still very simple.

Parsing can be used to permit infix notation. Again it is simpler to demand that all functional calls be in prefix notation. However, a beginner is far more familar with *FORWARD :SIDE+10* than with *(FORWARD (SUM :SIDE 10))*.

Eventually, as one's code becomes more complex, parentheses become a simplifying tool. One does not have to guess how the parser will work. LLOGO allows this. If desired, parentheses are permitted and interpreted in the standard way.

Criticism: LOGO complicates its parsing algorithm in several ways, making it difficult to explain to a student. For example, the language does not insist that all primitives take a fixed number of inputs. In some cases such as the title lines of definitions, this is reasonable. On the other hand, it is somewhat confusing to limit such primitives as *SUM* to only 2 inputs if not parenthesized but any number of inputs if parenthesized. Equally bad is the fact that primitives like 11LOGO's *PRINTOUT* for printing definitions do not evaluate their inputs. It would be more consistent for

?PRINTOUT "PROGRAM"

to be required.

2.3 Disparity

Program Versus Data

Both programs and data are information structures. The difference between the two is solely a matter of use. LISP preserves this elegant view by allowing programs to be passed as input and, indeed, to even redefine themselves. This power, for all its simplicity, can confuse the beginner. For the novice, the difference between defining and running a procedure is unclear. LOGO provides clarification by forcing a complete distinction between the processes of defining and of evaluation.

Criticism: LOGO violates this idea. A program can be executed inside a definition if not preceded by a line number. This is a mistake. The typical case is for the user to have intended to type the line number. In its wistful desire for more computational power, LOGO has forgotten its epistemological foundations.

Homonyms

LISP has the ability for a word to be the name of both a procedure and a variable. The position of the word in a list then determines how it is used. Homonyms, however, can be confusing. How should a word which is both a procedure and a variable be treated when it is the first element in a list? The choice is arbitrary.

LOGO prevents such homonyms. Words evaluate as variables only when preceded by ":".

... *X* .. causes X to evaluate as a procedure call.
... *:X* .. returns the value of the variable X.

thus, LOGO and LISP share the power of allowing any string of letters to be either a procedure or a variable name. But LOGO insists on an unambiguous "local" distinction, independent of position, between these two uses.

Another example of the clever ways LISP takes advantage of homonyms is *NIL*. LISP uses this word to name both the empty list and the logical truth value *FALSE*. This can result in more economical procedures. The convenience, however, has no conceptual basis. Hence, it can confuse the user who does not yet understand either list manipulation or logical analysis well. This is similar to the situation in APL, where the

logical constants are the integers 0 and 1, and conditionals are accomplished by numerical manipulation. It can lead to obscuring the purpose of a given piece of code.

Line oriented input

LISP evaluates an expression when parentheses balance. Thus it cannot catch errors caused by typing too many right parentheses. LOGO waits for a carriage return. Hence it is capable of recognizing this problem. Furthermore, a user can write several calls on a line. Execution is delayed until a carriage return is typed. This has the virtue of separating the tasks of forming grammatical expressions from executing programs.

Section 3. Overview of the Implementation

LISP LOGO is designed so that the user need never know that he is communicating with other than a standard LOGO. However, if desired, he can insert parenthesized LISP code anywhere in his LOGO program.

LISP LOGO is basically a compiler. It converts LOGO input to LISP programs. The result is that running most procedures takes less time since the code need not be repeatedly interned and parsed.

The following pages provide an overview of the major parts of the system. These are its reader, parser, evaluator, printer, and error handler. More detailed explanations of these will follow in later sections of this memo. For implementation details, LISP LOGO is available in well-commented interpretive code.

Code for the LOGO display turtle is discussed in Section 15, and code for the music box in Section 16. The "LOGO project" is concerned with more than the development of a computer language. Of major interest is the design of various computer-driven devices which provide a rich problem solving environment for the student. However, special purpose primitives for driving these devices are independent of LOGO versus LISP issues and must be added individually. A LISP-based implementation does have one special virtue. For those devices like the music box which are driven by ASCII characters, the primitives can be written in LISP or LOGO and then compiled. It is not necessary to create code at the machine level.

3.1 Reader

The LOGO reader is basically a line-oriented LISP reader. It returns a list of atoms read between carriage returns. The fundamental tasks of interning atoms and building list structure are handled by LISP. Conflicts in character syntax and identifiers between LISP and LOGO present the only subtleties.

Certain characters such as the infix operators +, -, *, and / do not require spaces to be set off as atoms. This is equivalent to being a "single character object" in LISP. Other characters such as "." in dotted pairs are special in LISP but not in LOGO. The solution to these conflicts is found in using separate "readtable"s for LOGO and LISP.

Conflicts in names also occur. The LOGO user has access to all the ordinary LISP procedures, but must be prevented from accessing LISP procedures which are internal to LLOGO. This is accomplished by using two "obarrays". When the user types in an identifier with the same name as an internal procedure, he accesses a different atom.

MACLISP allows any number of separate "readtable"s and "obarray"s. This permits multiple worlds - PLANNER, CONNIVER, LISP, LOGO - to co-exist with no conflict. Switching worlds is computationally fast. All that is necessary is to rebind the ***READTABLE*** and ***OBARRAY*** variables to the desired world. On the other hand, the naive user is protected completely from other environments and need not even know of their existence.

3.2 Parser

The parser converts a LOGO line to list-structured form. This requires that information on the number of inputs used by a procedure be available. Inserting parentheses is a trivial computation for procedures with a fixed number of inputs. However, complexities are introduced into the LOGO parser by:

1. Having infix as well as prefix operators.

2. Changing the number of inputs depending upon whether the user embedded the form in parentheses (*SUM, SENTENCE,* ...).

3. Primitives like *TO* that do not parse their input.

4. Homonyms: Functions which have the same name in LISP and LOGO, but have different meanings. These are handled by having the parser detect the names of LOGO primitives which conflict with LISP, and convert them to functions with different names that do not conflict.

This makes the parser the most complicated part of the simulation.

Parsing information is stored on the property list of a function. The major subprocedures are concerned with prefix, infix, and user-typed parentheses. Special primitives are parsed by storing a procedure as the parsing property.

3.3 Evaluation

The basic LOGO functions that do the user's computation - i.e. the arithmetic, list, and logical primitives - are the simplest part of the simulation. These functions all occur in LISP, usually in a somewhat more general form. Hence, this part of the implementation is little more than renaming. For many primitives, LLOGO provides more argument type checking and informative error messages than are supplied by their LISP counterparts.

Parsed code is executed directly by the LISP evaluator. Indeed, a user-defined program in parsed form is simply a LISP *PROG*. The line numbers are tags in the *PROG*.

3.4 Printing

LOGO procedures could be represented as lists of unparsed lines internally. In this case, a line must be interned and parsed each time it is run. However, the problems of printing the definition and editing a function are simplified. The internal format is identical to the format in which the user originally typed the expression.

An alternative solution is to represent LOGO programs in parsed, i.e. LISP form. A LOGO program internally is a LISP program. This maximizes run time speed and simplifies building program understanders. It has the disadvantage of complicating the parser and the printer.

1. The parser must handle functions that have not yet been defined. This can be accomplished, however, by reverting to the solution of parsing at run time those lines which contain unknown functions. This run-time parsing can alter the program's definition as well so it only need occur once.

2. Printing definitions and editing lines requires an inverse parser or "unparser" which returns the LISP-ified code to its original form. This is possible providing there is no information lost in parsing. Such is the case if the parser makes special provision for distinguishing user-typed parentheses from parser-generated parentheses. One way to accomplish this is by beginning user-lists with a do-nothing function ***USER-PAREN*** defined as:

(DEFUN USER-PAREN (X) X)

3. Editing title lines is made more complex. The editor must reparse the lines of super-procedures in which the edited function appears. This can be accomplished by maintaining a super-procedure tree, although LLOGO does not currently do this.

These complications can be avoided by storing both representations of the procedure. This is an excellent example of a space versus complexity trade-off. LISP LOGO currently does not store both representations.

3.5 Error Analysis

Since LOGO is a language which is designed to be used by beginning programmers, it is important to provide informative error messages. Consequently, all LOGO primitives do extensive type checking on their inputs. LLOGO will try to print out the form which caused the error, and give the line number if the error occurred inside a procedure. After a simple mistyping error which can be detected by the parser, the user is given an immediate opportunity to correct the line. For run time errors, he is given the option of causing breakpoints. Facilities for exploring the stack from inside a breakpoint loop are available. Since LOGO procedures are represented internally as LISP procedures, the standard LISP ***TRACE*** package can be used.

These facilities are implemented using LISP error interrupt handlers and ***EVALFRAME***. If the sophisticated user wishes to provide his own error handlers, he can access the LISP facilities directly.

Section 4. Performance

4.1 Size

LISP	26 Blocks (1024 36 bit words)
LLOGO (compiled)	7 Binary program
	5 List structure
	2 Numbers, Atomic symbols, etc.
Total space	40

These figures do not include space for user programs, or loading the display turtle, music, or GERMLAND packages. Between 5 and 10K beyond the amount of storage mentioned above would provide a reasonable amount of workspace for user programs and data; this would correspond roughly to programs of perhaps a few pages. In the current MACLISP, storage expands as needed. LLOGO takes advantage of this feature -- if programs grow beyond a certain size the user is asked whether he wishes the allocation to be increased. Storage is expanded automatically on loading special packages such as the display turtle. Of the 14 blocks which comprise the LLOGO system, all but 3 are pure, and can be shared among users.

4.2 Computation Time

For most processing, LLOGO enjoys a speed-up over CLOGO and 11LOGO due to the fact that parsing and interning occur only once at define time. Further, LLOGO makes it possible to compile LOGO source programs into machine code using the MACLISP compiler for increased efficiency [See Section 9]. Workspaces can be stored on the disk in internal LISP format. [See Section 10.1] Consequently, re-reading files has no overhead. CLOGO has an advantage, however, in manipulating words, as its internal data structure is string rather than list oriented.

4.3 Use

Almost all of the primitives of CLOGO and 11LOGO, (including the music box, display turtle for the PDP-6 and GT40) are implemented. Hence, LISP LOGO is capable of reading, parsing and running most files saved under CLOGO or 11LOGO [perhaps necessitating minor modification].

It can also be used real-time by an individual familiar only with LOGO: no knowledge of LISP is required. On the other hand, all of LISP's facilities are available. Programs can be written in LISP, or in machine language using LAP, and can be made callable from LOGO. The special packages for the display turtle, music box and GERMLAND can also be used from an ordinary LISP. Some other facilities of LLOGO, such as the breakpoint and stack manipulating functions, are also available for use in LISP. LISP users can take advantage of these facilities without interaction with LOGO simply by loading the appropriate files of LISP functions.

4.4 Availability

The implementation is written completely in interpretive code. It runs compiled under the MACLISP currently in use at the Artificial Intelligence Laboratory. LLOGO has also been implemented on standard DEC PDP-10's under the TEN50 and TENEX systems, and on MULTICS. These implementations are discussed in Section 13 and Section 12 of this memo.

Section 5. Getting Started

In the following sections, we will go into more detail concerning the implementation of LISP LOGO, and provide some practical information for using it. We will not attempt to provide the reader with an introduction to the LOGO language; several excellent sources for this already exist, such as the LOGO Primer, and the 11LOGO User's manual [LOGO memo 7]. We will assume that the reader has read these, or is already familiar with CLOGO or 11LOGO, the other implementations of the LOGO language available at the AI lab. Instead, we will concentrate on pointing out differences between LLOGO and other implementations of LOGO, and describing features unique to our implementation. It is not necessary to know LISP to understand most of what follows, although some knowledge of LISP would be helpful in gaining insight into the implementation. For more information on LISP, see the MACLISP Reference Manual by Dave Moon, and the Interim LISP User's Guide [AI memo 190].

Notational conventions: Throughout this memo, ***USER TYPEIN and LOGO CODE will appear in a font like this.*** SYSTEM TYPEOUT will appear in a font like this. Control characters are denoted by ^ followed by the character. You type a control character by holding down the control key while you are typing the character. $ means escape or altmode, not dollar-sign, except where otherwise noted. Angle brackets < > mean something of the appropriate type suggested within the brackets; for instance, if your user name is HENRY, <user name> means your user name, e.g. HENRY. Except for control characters, which usually take immediate effect, and except where otherwise noted, end all lines of typein with a carriage return.

The following procedure is intended to help very naive users of ITS to get logged in, and to obtain LISP LOGO. See AI memo 215, How To Get On the System, for more details.

1. Find a free console. A console which is free shows,

   AI ITS <version> CONSOLE <number> FREE, <time>.

2. A console which is free understands only one command. It is ^Z. The computer will respond with the following messages:

   AI ITS <version>. DDT <version>.
   <number> USERS.
   <news>

3. When it stops printing, login as follows: type

   :LOGIN <user name>

If there are any messages for you,

   --MAIL--

will be printed. You can type a space to receive it or any other character to postpone it. A * will be typed at the end.

4. Now you have completed logging in to the AI system. LLOGO is a subsystem of AI ITS. To get LLOGO started,

5. Decide which version of LISP LOGO you want. Choose from:

LLOGO - Standard version of LISP LOGO. Vocabulary is compatible with CLOGO.

11LOGO - A version which uses a vocabulary which is compatible with PDP11 LOGO. See Section 11 for details.

NLLOGO -The very latest version of LISP LOGO. This is experimental, so we make no promises.

When you decide which you want, type

:<name of program>

for example, *:LLOGO* .

6. Then LLOGO will print out some initial messages, including its version number and LISP's and will ask you some questions.

DO YOU WANT TO USE THE DISPLAY TURTLE? /

If you want to define and edit a procedure which contains turtle display commands, you should answer *YES* to this question. It is not necessary that you have the 340 display scope, or the GT40 display, to do just defining and editing. You can even run the procedure if you do not mind not being able to see what the procedure does. See Section 15 for more information.

GERMLAND?

If you want to play with GERMLAND, the display turtle for character displays such as DATAPOINT terminals, answer *YES*. This has a prompter which will run some demonstrations and provide help if you need it. Again, if you intend to define or edit procedures designed to run in GERMLAND, you must answer *YES*. See Section 14.

MUSIC BOX?

If you want to use LOGO music box primitives, answer *YES*. This will inquire further, as to which music box, etc. See Section 16. In case you have answered *YES* to any of these questions you have to wait for a while, because it takes some time to load in the files. If you want to interrupt loading in type ^*X* , not ^*C*. If you have a file named *LLOGO (INIT)* on your directory or there is a file named *<user name> .LLOGO.* on the *(INIT)* directory, LLOGO will read it as an initialization file, executing LOGO code contained therein. When all this is finished, LLOGO will indicate its readiness with

```
LLOGO LISTENING
?
```

7. If you find yourself in the unfortunate situation of meeting a bug in LISP-LOGO, you may report it by using the function *BUG*. The input to *BUG* should be a message describing the difficulty, enclosed in dollar signs. For example,

```
?BUG $
THE TURTLE ESCAPED FROM THE
DISPLAY SCREEN...
$
;THANK YOU.
?
```

8. You can logout when you are finished by typing *GOODBYE* to LOGO. The terminal should then say,

```
AND A PLEASANT DAY TO YOU!
AI ITS <version> CONSOLE <number> FREE <time>
```

9. Have fun!

Section 6. Parsing LOGO

This section will discuss a few of the more complex issues in parsing LOGO into LISP, and discuss how they are handled by LLOGO. LISP is trivial to parse, as its syntax is totally unambiguous. The application of a function to its inputs always happens in prefix notation, and the precise syntactic extent of a form is always clearly delineated by parentheses. LOGO syntax affords the beginning programmer some conveniences over LISP syntax, while retaining much of the expressive power of LISP. Parentheses can be omitted surrounding every form, and the more customary infix notation for arithmetic expressions can be arbitrarily intermingled with prefix notation. These conveniences are bought at the cost of complicating the parser, and introducing some cases where ambiguity results regarding the user's intent for some of the language's syntactic constructs.

6.1 Infix Expressions

LLOGO allows infix notation to be used as well as prefix functions in arithmetic expressions. Most LOGO arithmetic functions exists in both prefix and infix flavors, and the user is free to use whichever he desires.

*?PRINT 3*4+:A^SUM FIRST :X DIFFERENCE :C*17 2*

is the same as

?PRINT (TIMES 3 4)+(EXPT :A ((FIRST :X)+(TIMES :C 17)-2))

LLOGO observes the usual precedence and associativity of arithmetic operators.

Note that a complication of the LOGO syntax is that all functions, not just infix operators, are required to have precedence levels. Is

*?FIRST :A * 17*

the same as

?TIMES (FIRST :A) 17 or *?FIRST (TIMES :A 17) ?*

The situation is further complicated by the user's probable expectation that functions which manipulate logical values have lower precedence than comparison operators like <, > and =. So,

?TEST :NUMBER < :PI

is taken to mean,

?TEST (LESSP :NUMBER :PI) and not *?LESSP (TEST :NUMBER) :PI*

CLOGO gives all arithmetic operators the same precedence on the grounds that precedence would be difficult to explain clearly to children. However, this has the drawback of deviating from the customary mathematical convention. Since the motivation for introducing infix notation into LOGO syntax is so that arithmetic expressions can be written in the infix form in common use, LLOGO has been designed to obey the usual precedence conventions.

LLOGO tries to please everybody. If you feel that the precedence scheme which has been implemented does not agree with your intuition, you are free to redefine the precedence levels as you wish. LLOGO also provides the capability of defining new infix operators.

The initial default precedences are identical to those of 11LOGO and are as follows:

700: ^ [exponentiation]
600: + - [prefix]
500: * / \
400: + - [infix]
300: [default precedence for system and user functions]
200: < > =
100: *IF NOT BOTH EITHER AND OR TEST*
50: _ [*MAKE*]

Initially, operators of levels 50 and 700 are right associative, and the rest are left associative, which is the default. Logical functions should have precedence lower than comparison operators, so if the user defines a logical function he should set the precedence himself, otherwise it will receive the default precedence. The user can change things by using the following functions:

PRECEDENCE *<op>*

Returns the precedence level of <op>.

PRECEDENCE *<op> <level>*

Sets <op>'s precedence level to the specified <level>, which may either be a number, or another operator, which means that <op> is to be given the same precedence as that operator.

PRECEDENCE NIL *<level>*

Sets the default precedence for functions to <level>. All functions which are not in the above list of infix functions, or have not been assigned a precedence by the user, receive the default precedence.

ASSOCIATE *<number> <which-way>*

Declares that all functions of precedence level <number> will associate <which-way>, which is either *'LEFT* or *'RIGHT*.

INFIX *<op> <level>*

Defines <op> to be an infix operator of precedence <level>. Specifying a precedence is optional.

NOPRECEDENCE

Forces all infix operators to the same precedence level [this will be higher than the default precedence]. Makes LISP LOGO look like CLOGO [well, almost...].

:INFIX

This variable contains a list of all current infix operators. Look, but don't touch. Use ***INFIX*** to add new infix operators.

6.2 Minus Sign

There is some ambiguity in the handling of minus sign. For example, consider

?(SENTENCE 3 -:A)

If the minus sign is interpreted as an infix difference operator, this will result in a list of one element. If the minus sign is interpreted as prefix negation, it will result in a list of two elements. CLOGO uses the spaces in the line to disambiguate this case. If there is a space between the minus sign and the ***:A***, it is interpreted as infix, Otherwise, it is interpreted as prefix. In 11LOGO, spaces are not semantically significant except to delimit words, so this is interpreted as ***(SENTENCE (DIFFERENCE 3 :A))*** regardless of the occurrence of spaces. LLOGO treats minus sign as does 11LOGO. One would obtain the result of the other interpretation by using

?(SENTENCE 3 (-:A))

The preceding discussion applies only to the parsing of infix expressions. So, ***[-4]*** is a list of one element, a negative number, but ***[- 4]*** is a list of two elements, minus sign and 4.

6.3 Homonyms

LLOGO makes all the functions of LISP directly accessible to the LOGO user, in exactly the same way as LOGO primitives. This runs into difficulty when a LISP function and a LOGO function have the same name but different meanings. These are currently handled by the parser, which converts them into innocuous atoms which do not conflict with LISP, and are reconverted upon unparsing. Currently the following functions are homonyms:

PRINT, RANDOM, LAST, EDIT, SAVE [in MULTICS only]

When the user types in one of these, it is converted by the parser to an internal representation consisting of a different function name [***LOGO-PRINT, LOGO-LAST LOGO-EDIT LOGO-RANDOM*** or ***LOGO-SAVE***, as appropriate]. When the user requests that the line be printed out or edited the unparser converts it back to the way it was originally typed in. In the CLOGO-compatible version of LLOGO, when ***:CAREFUL*** is set to non-***NIL*** the following primitives which conflict with CLOGO are also changed by the

parser: *LIST* is changed to *PRINTOUT*, *DISPLAY* to *STARTDISPLAY*, *GET* and *READ* to *READFILE*, and *DO* to *RUN*. Warning messages are also printed in these cases.

There is one pitfall in the current method of handling homonyms: sometimes, as with passing functional arguments, the parser does not get a chance to do its thing, so the user may find an unexpected function called; *APPLY 'PRINT* . . . . calls LISP's *PRINT* function, not LOGO's.

6.4 Abbreviations

Abbreviations are accomplished in LLOGO by putting the name of the function which is abbreviated on the property list of the abbreviation as an EXPR or FEXPR property, as appropriate. Abbreviations are expanded into their full form on parsing, and are left that way. The user has the capability of creating his own abbreviations by

?ABBREVIATE <new name> <old name>

and erasing them by

?ERASE ABBREVIATION <name>

ABBREVIATE evaluates its inputs, but *ERASE* doesn't. A complete listing of abbreviations can be obtained by doing

?PRINTOUT ABBREVIATIONS

Section 7. Defining and Editing Functions

In LOGO, when the user defines a procedure using *TO*, or *EDIT*s a procedure he has previously defined, LOGO enters an "edit mode", where lines beginning with a number are inserted into the procedure under modification. LOGO prompts with ">" rather than "?" to indicate this. The intent of having a separate mode for editing procedures is to stress the distinction between defining procedures and executing them. This distinction is not strictly maintained; if the line does not begin with a number, the commands are executed as they would be ordinarily, with a few exceptions [the user is prevented from doing another *TO* or *EDIT* for instance]. Occasionally, this leads to errors, for instance if the user forgot to type the line number at the beginning of a line intended for insertion.

The default state of LLOGO is to retain the separation of edit mode from ordinary mode as in 11LOGO and CLOGO. The slightly more sophisticated user, however, might find himself in an unnecessary loop of continually typing *EDIT*'s and *END*'s while working on the same procedure. Since the lines typed by the user for insertion into a procedure are inserted immediately when the user finishes typing the line, *END* does not cause anything to happen other than the termination of edit mode. The system always remembers the name of the last function mentioned by *TO*, *EDIT*, *PRINTOUT*, etc. as a default for these functions, so when working on a single function, *EDIT* serves only to enter edit mode. The user has an option of turning off the separate edit mode by setting the variable *:EDITMODE* to *NIL*. This will cause lines beginning with a number to be inserted into the default procedure at any time. In this mode, it is never necessary to use *END*, and *EDIT* will only change the name of the default procedure if given an input. The prompter will not be changed.

In LLOGO, it is not necessary to be in edit mode to use *EDITLINE* or *EDITTITLE* on a line of the default procedure, and the editing control characters are available even when not in edit mode.

7.1 Control Character Editing

LLOGO has a control-character line editor similar to those in CLOGO and 11LOGO. This makes it particularly convenient to correct minor typing errors, by providing a means of recycling portions of the line typed previously, instead of requiring retyping of the entire line. The editor keeps track of two lines: an old line which you are editing, and a new line, which LLOGO is to use as the next line of input. The old line is always the last line you typed at LLOGO, except immediately after a parsing error, when the offending line will be typed out at you, and it may be edited. You can also set the old line yourself to be a line in the current default procedure by doing *EDITLINE* <*line number*>, or the title of a procedure by calling *EDITTITLE*. Everything you type after the prompter, or cause to appear using the control characters, is included in the new line, until you type carriage return, which terminates editing for that line. You may use parts of the old line in constructing the new line by using the following editor commands:

∧*E* - Get the next word from the front of the old line, and put it on the end of the new line.

∧*R* - Put the rest of the old line at the end of the new line. This is like doing ∧*E*'s until there is nothing left in the old line.

^S - Delete a word from the front of the old line.

^P - Delete a word from the end of the new line. Like rubout, except rubs out a word instead of a character.

LLOGO uses different characters than 11LOGO and CLOGO do because LISP uses most of the control characters for interrupts and i/o.

7.2 Printing Function Definitions

The function *PRINTOUT* can be used to look at definitions of user procedures. In addition, it has other options for examining the state of your LLOGO. *PRINTOUT* doesn't evaluate its inputs.

PRINTOUT <procedure-name> [Abbreviation PO]

Will print out the definition of the specified procedure. If the name is omitted, it will assume the last function that was defined, edited, or printed.

PRINTOUT LINE <number> [POL]

Prints out only the specified line in the default procedure.

PRINTOUT TITLE <procedure> [POT]

Prints the just the title of the procedure given. If the input is omitted, prints the title of the current default procedure. This is useful if you forget which procedure is the default.

PRINTOUT TITLES [POTS]

Prints the titles of all current user procedures. Ignores buried procedures [see Section 10.1].

PRINTOUT PROCEDURES [POPR]

Prints out the definitions of all currently defined user procedures. Will not print the definitions of procedures that are buried [see Section 10.1].

PRINTOUT NAMES [PON]

Prints the names and values of all user variables.

PRINTOUT ALL [POA]

Does ***PRINTOUT PROCEDURES*** and ***PRINTOUT NAMES.***

PRINTOUT SNAPS

Prints a list of saved display turtle scenes. See Section 15.11.

PRINTOUT FILE, PRINTOUT INDEX

See Section 10.2.

PRINTOUT ABBREVIATIONS

Prints a list of all current abbreviations, and the names of the procedures which each abbreviates.

PRINTOUT PRIMITIVES

Prints a complete list of all LLOGO primitives.

Another useful command is ***LINEPRINT***, which causes a listing, similar to the output of ***PRINTOUT ALL***, to appear on the line printer. It takes an optional input, a word to be used as a title to name the listing generated.

7.3 Erasing

The command ***ERASE*** will remove unwanted portions of your LOGO. The inputs to ***ERASE*** are not evaluated. The options available are:

ERASE *<procedure, variable or snap name>*

Cause the definition of the specified object to vanish. Note: When you define a function using ***TO***, it checks to see if there already exists a procedure of the same name, and if so, inquires whether you want the old definition ***ERASE***d. This is to prevent you from accidentally overwriting definitions of functions.

ERASE PRIMITIVE *<primitive name>*

The LLOGO primitive given as input will be erased. You might use this, for example, if you wanted to use a name used by LOGO for one of your own functions. If you define a name using ***TO*** which conflicts with a LOGO primitive, it will inquire if you want the definition of the primitive to be erased.

ERASE LINE *<number>* ***[ERL]***

Erases line <number> of the default procedure.

ERASE NAMES [ERN]

Unbinds all user variables.

ERASE PROCEDURES [ERP]

Erases all interpretive user functions. Does not affect compiled or buried procedures.

ERASE COMPILED

Erases all compiled user functions.

ERASE ALL [ERA]

Like doing *ERASE PROCEDURES*, *ERASE COMPILED* and *ERASE NAMES*.

ERASE ABBREVIATION <abbreviation>

Erases the abbreviation given as input. Does not affect the procedure that it abbreviates.

ERASE FILE <file spec> [ERF]

See Section 10.2.

ERASE TRACE <function> [ERTR]

Removes trace from <function>. See Section 8.6.

ERASE BURY <functions> [ERB]

The functions will no longer be buried. For a discussion on buried procedures, see Section 10.1.

Section 8. Error Handling and Debugging

The philosophy of the LISP LOGO error handling system is to try to be as forgiving as possible; the system will give you an opportunity to recover from almost any type of error [except a bug in LLOGO!]. There are two types of errors which can occur:

8.1 Parsing Errors

If for some reason, LLOGO cannot parse the line you typed [for example, you may have typed mismatched parentheses], this causes a parsing error. When this happens, LLOGO will print a message telling you why it was unhappy, retype the offending line at you, and type the editor prompt character. You now have a chance to redeem yourself by correcting the line -- you may use any of the editing control characters [see Section 7.1]. When you are satisfied that the line is correct, type carriage return, and LLOGO will resume evaluation, using the corrected line in place of the one which was in error.

8.2 Run Time Errors

When a run time error occurs, a message will be printed. If the error occurs inside a LOGO user defined function, the message will say something like:

```
;ERROR IN LINE <number> OF <procedure>
;LINE <number> IS : ....
;<reason for error>
```

If the error occured inside a LOGO primitive, the message will look like:

```
;COULDN'T EVALUATE <bad form>
;BECAUSE
;<reason for error>
```

where <bad form> is what LLOGO was trying to evaluate when the error occurred. Usually, this will give you enough information to figure out where the error occurred, although <bad form> is sometimes uninformative. Usually, LLOGO will simply return to the top level loop when such an error occurs. However, if you *SETQ* the variable *:ERRBREAK* to something other than *NIL*, [or *MAKE 'ERRBREAK . . .*] a run time error will cause a LOGO break loop to be entered after the message is printed. Setting the variable *:LISPBREAK* to non-*NIL* will cause a LISP style breakpoint to occur when an error happens. [For a detailed discussion of breakpoints, see below, Section 8.3.] You can resume execution of your program from the point at which the error occurred, by *CONTINUE*ing with something to be used in place of the piece of data which caused the error. If the error was an undefined function, you may *CONTINUE* with the name of a function which has a definition. If the error was an unbound variable, *CONTINUE* with a value for that variable. If the error was a wrong type of input to a LOGO primitive, *CONTINUE* with some appropriate value for an input to that function, etc. Usually it will be obvious from the context what sort of item is required. Computation will be resumed from where the error occurred, with the returned item substituted for the one which caused the error. [Note: the usual LISP interrupt handler functions expect a list of the new item to be returned, while LLOGO's expect simply the item]. The LISP LOGO run-time

error handling works by utilizing the LISP error interrupt facility. If you don't like the way LLOGO handles any of the error conditions, you are free to design your own error interrupt handlers, either in LISP or in LOGO.

8.3 Breakpoints

A powerful debugging aid is the ability to cause breakpoints. Stopping a program in the process of being evaluated allows the user to examine and modify its state, and explore the history of evaluation which led up to the breakpoint. LISP provides excellent facilities for doing this, including automatic generation of breakpoints when an error occurs. Whenever LISP starts to evaluate a form, it first pushes the form on a stack; from a breakpoint one can examine the stack to determine what forms were in the process of being evaluated, and perform evaluations relative to a particular stack frame. LISP LOGO attempts to make these features easily available to the user, from either LISP or LOGO. Versions of these breakpoint functions are also available which can run in an ordinary LISP, without the rest of the LOGO environment. The following facilities are available for causing breakpoints:

***LOGOBREAK** <message> <condition> <return-value> [Abbreviation PAUSE]*

The inputs are all optional, and are not evaluated. Unless <condition> is given and evaluates to ***NIL***, ***LOGOBREAK*** causes the user to enter a loop where LOGO commands can be typed and the results printed. This is similar to the top level loop except that % is printed as a prompter rather than ?; it is very much like repeatedly evaluating ***PRINT RUN REQUEST***. If <message> is present, it will be printed out upon entry to the break point. It also prints the form in the current stack frame, which will be the call to ***LOGOBREAK*** if called explicitly by the user. If the breakpoint happened because of an error, the initial stack frame will be the one containing the form which caused the error. ***LOGOBREAK*** tries wherever possible to print out the current form as LOGO code before it enters a LOGO break point. However, the current version is not always smart enough to distinguish between LISP and LOGO frames on the stack, so you might occasionally see what looks like internal LISP garbage there. If you go up far enough, you are sure to find the LOGO code. A smarter version could recognize the LISP frames and ignore them. The third input is a default value for ***LOGOBREAK*** to return if it is ***CONTINUE***d. [See description of ***CONTINUE***, below] Caution: the breakpoint functions described in this section use LISP's ***CATCH*** and ***THROW***. Unlabelled ***THROW***s from inside a breakpoint loop are highly discouraged.

^A

If control-A is typed at any time, even while a program is running, it will cause an interrupt and a LOGO break point will be entered.

***LISPBREAK** <message> <condition> <return value> [Abb. BREAK]*

This is like ***LOGOBREAK***, except that the loop is a LISP ***(PRINT (EVAL (READ)))*** loop. This is especially useful when debugging a set of LISP functions designed to run in LOGO. To access your LOGO variables and user functions from inside a LISP break loop, prefix them with a sharp sign ["#"]. LISP users note: you can interact with this break loop as with the standard LISP ***BREAK*** function, except that it is set up to allow use of the stack hacking functions described below. If ***$P*** is typed, or ***(CONTINUE)*** invoked, the <return value> will be the value of the call to ***LISPBREAK***.

^H

As in LISP, *^H* typed at any time will interrupt and cause a LISP breakpoint to be entered.

:ERRBREAK

If this variable is not *NIL*, when a run time error happens, *LOGOBREAK* will be called automatically. This gives you a chance to find out what went wrong, and recover by *CONTINUE*ing with a new piece of data to replace the one that caused the error. It is initially set to *NIL*.

:LISPBREAK

Like *:ERRBREAK*, except that if set to something other than *NIL*, when an error happens, *LISPBREAK* rather than *LOGOBREAK* will be called. Initially set to *NIL*.

The following functions can be called from inside a breakpoint to examine and manipulate the stack:

UP

Moves the breakpoint up one frame in the stack, printing out the form which was about to be evaluated in that frame. This will be the form which called the one which was last typed out by any of the functions mentioned in this section. Evaluation now takes place in the new stack frame. This means that all local and input variables will have the values they did when that form was about to be evaluated. However, side effects such as assignment of global variables are not undone. Frames are numbered for the user's convenience, from 0 increasing up to top level.

UP <number>

Goes <number> frames up the stack. Like doing *UP*, <number> times. The <number> may be negative, in which case, the breakpoint is moved down the stack rather than up.

UP <atom>

Goes up the stack until a call to the function whose name is <atom> is found.

UP <atom> <number>

Goes up the stack until the <number>th call to <atom> is found. Searches downward for the <number>th call to the specified function if <number> is negative.

DOWN <atom> <number>

Like *UP*, except that it proceeds down the stack instead of up. Both inputs are optional, and default as for *UP*, except that <number> defaults to -1 instead of +1. If <number> is given it is equivalent to *UP .... (-<number>)*.

PRINTUP <atom> <number>

Accepts inputs as does *UP*, but instead of moving the breakpoint up the stack to the desired frame, all frames between the current one and the one specified are printed out. This function is good for getting a quick view of the stack in the immediate vicinity of the breakpoint. The breakpoint remains in the same frame as before. The two inputs are optional, and default as for *UP*.

PRINTDOWN *<atom> <number>*

Like ***PRINTUP***, except that the inputs are interpreted as for ***DOWN*** rather than as for *UP*, that is, it prints frames going down the stack.

EXIT *<return-value>*

Causes the current stack frame to return with the value <return-value>. That is, the computation continues as if the form in the current frame had returned with <return-value>. The input is optional, and defaults to ***NIL***.

CONTINUE *<return-value> [Abbreviations CO, $P]*

Causes the frame of the originally invoked breakpoint to return with the specified value. The input is optional. Use ***CONTINUE*** to return a new item of data from inside an error breakpoint; for instance a new function name to use in place of one which was undefined. Note that in many situations, for example from a user-invoked breakpoint or from an error breakpoint which expects an item to be returned as the value of the form which caused the error, if you haven't moved the breakpoint around the stack, ***CONTINUE*** will be identical to ***EXIT***. If the input to ***CONTINUE*** is omitted, the default return value specified by a third input to ***LISPBREAK*** or ***LOGOBREAK*** will be returned as the value of the breakpoint. If no such default return value was given, ***NIL*** will be returned.

Here's an example:

```
?MAKE 'ERRBREAK T                    !Assure LOGO break happens!
                                     !when an error occurs!
;CHANGING A SYSTEM NAME
T
?TO SCREWUP :N                       !Define our losing procedure.!
>1 IF :N=0 THEN OUTPUT :UNBOUND
                                     !Count down to 0, then!
>2 OUTPUT SCREWUP :N-1
                                     !eval variable which  has no value!
>END
;SCREWUP DEFINED
?SCREWUP 3
;ERROR IN LINE 1 OF SCREWUP
;LINE 1 IS: IF :N=8 THEN OUTPUT :UNBOUND
;:UNBOUND IS AN UNBOUND VARIABLE
;BREAKPOINT FRAME 8: :UNBOUND
?:N                                  !Frame 0 is the variable. Eval was!
8                                    !working on this when we bombed!
                                     !We can do any command!
                                     !while in the breakpoint.!
?UP                                  !Going up a frame. :UNBOUND!
```

```
;BREAKPOINT FRAME 1: OUTPUT :UNBOUND
                                        !was the input to OUTPUT!
?DOWN                                   !going down a frame!
;BREAKPOINT FRAME 8: :UNBOUND
?UP 'SCREWUP
;BREAKPOINT FRAME 4: SCREWUP :N-1
                                        !we arrive at recursive invocation!
?:N                                     !where :N had the value 1!
1
?UP 'SCREWUP 2                          !If we rise past 2 calls to SCREWUP,!
;BREAKPOINT FRAME 10: SCREWUP :N-1
?:N                                     !:N was 3.!
3
?EXIT 'SCREWED                          !We decide for some reason!
SCREWED                                 !that SCREWUP of 2 is !
?                                       !to return the value 'SCREWED!
                                        !and all the previous invocations !
                                        !of SCREWUP return with the value!
                                        !'SCREWED and we are at top level!
                                        !Wasn't that fun?!
```

8.4 Wrong Number of Inputs Errors

Since LOGO syntax requires that the parser know how many inputs a function requires, and LLOGO parses your input as you type it in, errors may be generated if you change the number of inputs a function takes by redefining the function, or by calling *EDITTITLE*. Calls to that function which you typed previously are now incorrectly parsed. LLOGO will catch most occurrences of this when the function is called, and print a message like:

;REPARSING LINE <number> OF <procedure> AS: <new parse>

and attempt to recover. LLOGO always attempts to reparse a line which caused a wrong number of inputs error. It is not always possible to win, however, as side effects may have occurred before the error was detected.

8.5 Garbage Collector Errors

Versions of LLOGO running in BIBOP LISP [LISPs with the capability of dynamically allocating storage] have special handlers for garbage collector interrupts. If it decides you have used too much storage space of a particular type, or too much stack space, it will stop and politely ask if you wish more to be added. If you see these questions repeated many times in a short span of time while running one program you should give serious consideration to the possibility that your program is doing infinite *CONS*ing or recursing infinitely.

8.6 Tracing

The standard LISP *TRACE* package may be used to trace LLOGO primitives or user functions. The tracer is not normally resident, but is loaded in when you first reference it. See the LISP manual for details on the syntax of its use and the various options available.

8.7 Interaction with LISP

In debugging functions written in LISP for use in LLOGO, it is often useful to be able to switch back and forth between LOGO and LISP top level loops. You can leave the LOGO top level loop and enter a LISP *READ-EVAL-PRINT* loop by using the LLOGO function *LISP*. From this mode, executing *(LOGO)* [remember to type the parentheses, you're in LISP!] will return to LOGO. Typing control-atsign [^@] at any time will cause an interrupt and switch worlds; you will enter LISP if you typed ^@ from LOGO, or enter LOGO if you typed it from LISP. The LISP loop gives you access to all internal LLOGO functions and global variables, which are normally inaccessible from LOGO since they are on a different obarray. LLOGO primitives and system variables are on both obarrays, so they will be accessible from both LISP and LOGO, but LOGO user functions and variables are on the LOGO obarray only. The character sharp sign ["#"] is an obarray-switching macro; to access LOGO user functions and variables from the LISP loop, prefix them with a sharp sign.

Section 9. Compiling LLOGO User Procedures

LISP LOGO compiles a LOGO source program into LISP and it is stored internally only as LISP code. Since this is the case, the LOGO user has the capability of using the LISP compiler directly on his LOGO programs, and obtain a substantial gain in efficiency, once his programs are thoroughly debugged. LISP LOGO provides an interface to the LISP compiler which should make it unnecessary for the user to worry about interacting with a separate program.

To compile all of the functions currently in the workspace, the function ***COMPILE*** is available. [This does not include buried procedures -- see Section 10.1.] It expects one word as input, to name the file which will contain the compiled code. The names of the functions which are being compiled will be printed out. A temporary output file [named *.LOGO. OUTPUT*] will be written on the current directory and deleted after the compilation is complete. The output file will have as first name the input to ***COMPILE***, and second file name ***FASL***. [In the MULTICS implementation, the temporary file will be named *logo_output* and placed in the current directory. The output file will appear in the working directory, with one name, the input to ***COMPILE***.] Since the LISP compiler must be called up as a separate program, be careful about interrupting the compilation before it is finished [for instance, by ^G] as you will not find yourself in LLOGO anymore.

To load a compiled file into LLOGO, say ***READFILE*** *<name>* ***FASL***. This will load all the compiled functions which were compiled by ***COMPILE*** *<name>*, and also restore the values of variables that were defined at that time. The names of compiled functions will be kept on a list called ***:COMPILED*** and not on ***:CONTENTS***. For debugging purposes, you might want to read in both the compiled and interpreted definitions of the same functions, and you can use the functions ***FLUSHICOMPILED*** and ***FLUSHINTERPRETED*** to switch back and forth between compiled and interpreted definitions.

The LOGO ***COMPILE*** function supplies declarations for LOGO primitives. Some of the declarations include LISP macros which replace calls to LOGO primitives with calls to their faster LISP counterparts for efficiency, and some optimization is done. For safety's sake, all variables are automatically declared ***SPECIAL***. However, the sophisticated user is free to include in his program ***DECLARE***s to ***UNSPECIAL*** input or local variables which he knows will not be referenced globally, or provide declarations which will make use of the fast-arithmetic LISP compiler.

A few warnings about compiling LOGO procedures: First, remember that LOGO syntax requires that it be known how many inputs a function expects, before a decision can be made as to how to parse a line of LOGO code. If, when defining a procedure, you include a call to a procedure which is not yet defined, parsing is delayed until run time [see Section 6 and Section 3.2 of this memo for more details]. The compiler, of course, cannot do anything reasonable with an unparsed line of LOGO code, so all parsing must be completed by the time the definition of any procedure is compiled. The ***COMPILE*** function attempts to make sure this is the case. Therefore, it is an error to attempt to compile a procedure which contains a call to a procedure which is not a LOGO primitive and has not yet been defined.

Also, it must be remembered that compilation of LOGO procedures, like those of LISP, is not "foolproof". It is not always the case that a procedure which runs correctly

when interpreted, will be guaranteed to run correctly when compiled. Self-modifying procedures, weird control structures, and in general procedures which depend heavily on maintaining the dynamic environment of the interpreter may fail to compile correctly without modification.

Section 10. Using Files in LLOGO

A file specification on ITS has four components. Each file is named by two words, of up to six characters each, a device [almost always DSK], and a directory name [usually the same as the user's name]. You can refer to a file in LOGO by using anywhere from 0 to 4 words. If you leave out the name altogether, it will be assumed that you are referring to the last file name mentioned. One word will be taken as the first file name, and the second will default to >, which means the highest numbered second file name which currently exists if you are reading, or one higher if you are writing. Two words will be taken as the two file names, and the directory and device will be defaulted. If three names are given, the third will be assumed as the directory name, and the device will be DSK. If four words are given, the third is device and fourth is the directory. Here are some examples:

[Assume that the current user name is ESG, and FOO 3 is the highest numbered file with FOO as its first filename]

LOGO	ITS [<fn1> <fn2> <dev>:<dir>;]
?READFILE FOO	*FOO > DSK:ESG; [FOO 3]*
?SAVE FOO	*FOO > DSK:ESG; [FOO 4]*
?READFILE FOO BAR	*FOO BAR DSK:ESG;*
?READFILE FOO BAR HENRY	*FOO BAR DSK:HENRY;*
?READFILE FOO BAR DSK HENRY	*FOO BAR DSK:HENRY;*

See Section 12.2 and Section 13 for information about file specifications on the MULTICS and TEN50 implementations. File specifications are accepted by LOGO in the same format as on ITS, so it may not be necessary to change any code to run on other implementations.

10.1 Saving and Reading Files

There are two ways of storing LOGO programs on the disk for later use. To store the contents of the current workspace [all user functions and variables currently defined] on the disk in the form of LOGO source code, use *SAVE*. It expects as input a file specification, as discussed above. The file created will contain the contents of the user's workspace, function definitions and *MAKE*s for variables, exactly in the form that he would see if he did a *PRINTOUT ALL*.

Workspaces can also be saved in LISP format, as they are represented internally by LOGO. This is accomplished by the function *WRITE* which takes its inputs as does *SAVE*. Although the file created will not be so pretty to look at if you print it, using *WRITE* produces files which are considerably faster to reload, since the program does not have to be reparsed. For long-term storage of programs, however, it is recommended that you use *SAVE* rather than *WRITE*. Changes in the implementation of LISP LOGO may result in changing the internal format of LOGO programs, in which case, files created by *WRITE* would not remain compatible, but files created by *SAVE* would remain so.

To reload a file from the disk, use the function *READFILE*. This accepts a standard file specification, and reads it in, printing the name of the file. *READFILE* does not care whether the file is in *SAVE*d or *WRIT*ten form. If the file was created by *SAVE*, lines of code will be printed out as they come in from the disk. For written files, only the names of functions and values of variables will appear. If you get annoyed at all this output, you can shut it up with ^*W*. LOGO will return with a question mark when the loading is complete.

It is often convenient to treat a set of functions as a "package" or "subsystem". For instance, you may have a set of your favorite functions which you place in your initialization file, or a set of functions designed for a specific purpose. When this is the case, it is inconvenient to have all these functions written out when you are working on additional procedures, or have to see their definitions when you do a *PRINTOUT ALL*. That is, one would like a method of having the package of functions available, but not considered as part of the workspace by certain commands. You can do this by using the function *BURY*. It takes unevaluated procedure names as input, and will assure that the function is ignored by the following commands: *PRINTOUT PROCEDURES*, *PRINTOUT ALL*, *PRINTOUT TITLES*, *ERASE PROCEDURES*, *ERASE ALL*, *SAVE*, *WRITE* and *COMPILE*. Otherwise the function is unaffected, and can be invoked, printed, edited, etc. A list of the names of buried procedures is kept as the value of the variable *:BURIED*. *BURY ALL* will *BURY* all currently defined procedures, and *ERASE BURY* will undo the effect of a *BURY*.

10.2 Other File Commands

PRINTOUT FILE [abbreviated *POF*] will print out the contents of a file. *ERASE FILE* will cause the specified file to vanish [This has a safety check to make sure you don't do anything you'll be sorry about]. These take file names as above, except that if only one input is given to *ERASE* it defaults to <, the least numbered second file name, again for safety reasons. *PRINTOUT INDEX [POI]* will print out all the file names in the directory specified by one word. *USE* will change the name of the default directory.

Section 11. Differences between 11LOGO and LLOGO

LISP LOGO was originally written to be compatible with CLOGO, a version of LOGO written in PDP10 assembler language. There now exists a version of LLOGO which we believe to be "semantically compatible" with the PDP11 version. By this we mean that the vocabulary is the same -- any primitive in 11LOGO also exists in LLOGO and will (hopefully) have the same meaning. LLOGO in fact has many primitives which do not exist in 11LOGO, as well as offering the user access to the full capabilities of LISP. There are substantial differences between LLOGO and 11LOGO with regard to file systems and error handling, and somewhat less substantial differences in the editor, turtle and music packages. These are described in detail in other sections of this document. There are also are several less substantial differences, not mentioned in the preceding discussions, and what follows is an attempt to provide a reasonably complete list of the knowledge that an experienced 11LOGO user would need to use LLOGO.

In 11LOGO, the double quote character " is used to specify that the atom following it is not to be evaluated-

```
?PRINT "FOO
FOO
```

It is like LISP's single quote, except that it also affects the LOGO reader's decision about when to stop including successive characters in forming the name of an atom. In ,

```
?PRINT :FOO+3
```

the plus sign is a separator character; it signals the end of the atom *:FOO* just as if there was a space following *:FOO*. However, following a double quote, the only separator characters recognized are space, carriage return, and square brackets. Thus, in 11LOGO,

```
?PRINT "FOO+3
FOO+3
```

In LLOGO, the user may use the LISP single quote to specify that an atom or parenthesized list following the single quote is not to be evaluated. The presence of the single quote does not change the way LLOGO decides when an atom ends. In LLOGO,

```
?PRINT 'FOO+3
;THE INPUT 'FOO TO + IS OF THE WRONG TYPE
```

because the plus sign is still a separator character. LLOGO uses the double quotes as CLOGO does; they are always matched. If one s-expression (atom or list) occurs in between double quotes, it is quoted. If more than one occurs, the list containing them is quoted. The correspondence between LLOGO double quoted expressions and LISP s-expressions is as follows:

```
"" ==> NIL
"<atom>" ==> (QUOTE <atom>)
"<s1> . . . .<sN>" ==> (QUOTE (<s1> . . . . <sN>))
"(<s1> . . <sN>)" ==> (QUOTE (<s1> . . . <sN>))
```

Square brackets in 11LOGO specify quoted lists. Parentheses are never used around lists as in LISP, but are only used to delimit forms. LLOGO recognizes square brackets as well as LISP's parentheses in denoting lists. The difference between brackets and parentheses in LLOGO is that the brackets always denote list constants, and not forms, and that the outer level of brackets is implicitly quoted:

[[FOO BAR]] ==> (QUOTE ((FOO BAR)))

There is a minor pitfall in the current implementation: note that top level parentheses implicitly quote the list; interior ones do not. This does not always work, for instance when using *RUN* one may expect interior lists also to remain unevaluated:

?PRINT [PRINT [FOO BAR]] ==> (PRINT '(PRINT (FOO BAR)))
PRINT (FOO BAR)
?RUN [PRINT [FOO BAR]] ==> (RUN '(PRINT (FOO BAR)))

prints the value of the function *FOO* applied to input *BAR*.

Square brackets in 11LOGO also share with double quotes the property described above of affecting the LOGO reader's decision on ending the names of atoms. Within a square bracketed list in 11LOGO, an atom is terminated only by a space, carriage return or bracket. This property is not true of square brackets in LLOGO. In LLOGO, *[FOO 3]* is a list containing three elements, but in 11LOGO, it contains only one element.

String quoting in LLOGO is accomplished using the dollar sign character, *$*. LLOGO will treat anything appearing between dollar signs literally, with special characters devoid of any special meaning. Within such a string, two consecutive dollar signs will be interpreted as a single dollar sign. So, *$$$$* would be the word whose name is a single dollar sign. *$$* is the empty word. Rubout, editing and interrupt characters cannot be quoted in this manner. Use the *ASCII* function of LISP if you really need them.

The character sharp sign ["#"] in 11LOGO is used as a prefix macro character which takes one input which must be a word, and executes it as a procedure. It is used where one wants to use a weird name for a procedure, or a name already used by the system. Sharp sign is used as an escape to call that procedure. Thus, a procedure defined in 11LOGO by *TO "PRINT* . . . would be called by *#"PRINT*, *TO "3* . . . would be called by *#"3*, etc. In LLOGO, sharp sign is used as a macro character which causes the next s-expression to be interned as if it were read in LISP if you are in LOGO, or as if it was read by LOGO if you are in LISP. If you are in the LISP mode of LLOGO and want to access your LOGO variables, you can say *#:FOO*, etc. The conflict may be changed in the near future by altering LISP LOGO's macro character to one that does not conflict with 11LOGO. Suggestions welcome.

The Boolean [logical] constants in 11LOGO are *TRUE* and *FALSE*, while in LLOGO, they are *T* and *NIL*, as in LISP.

The 11LOGO function *LEVEL*, which returns the current procedure depth, is not implemented.

11LOGO forms are divided into two categories: those that output [return a value] and those which do not. In LLOGO, as in LISP, every form returns a value. To simulate 11LOGO and CLOGO in this respect, as a special hack, forms which return a question mark do not have their values printed by LLOGO's top level function. However, LLOGO cannot

catch the error of such a form hiding inside parentheses, as can 11LOGO. Most of the primitives which do not return a value in 11LOGO return ? in LLOGO.

The character : in 11LOGO is treated as a macro "the value of ...". if A is bound to B and B is bound to C, then ::A is C. In LLOGO, variables set by *MAKE* are just LISP atoms beginning with the character :, so ::A will be the value of the variable set by *MAKE* ":A" <*whatever*>, etc. We are seriously considering changing this, eliminating the incompatibility. The present setup requires *MAKE* to do an expensive *EXPLODE* on the variable name, in order to create the word which begins with a colon.

LLOGO expects to find only one form inside parentheses: constructs like

?(FD 100 FD 50 SUM 4 5)

are prohibited. 11LOGO allows more than one form inside parentheses under certain restrictions.

The 11LOGO procedure *TEXT*, which returns a list of lists which are the lines of a procedure whose name is given as input, is not implemented in LLOGO. However, you can access the definition of a function in its parsed LISP form on the property list [*CDR*] of the atom.

Comments: LLOGO understands two comment conventions: LISP's convention of treating as a comment anything between a semicolon and the next carriage return, and LOGO's of treating as a comment anything in between exclamation points. [The exclamation points must be matched, and comments can be continued past the end of the line]. Anything after exclamation points on a line is ignored.

The top level loop in LISP LOGO is a *READ-EVAL-PRINT* loop whereas PDP11 LOGO is a *READ-EVAL* loop. This means that 11LOGO prints out only when you ask it to print unlike LLOGO which prints out values after every evaluation of a LOGO form.

In 11LOGO:
?SUM 4 8
YOU DONT SAY WHAT TO DO WITH 12

In LLOGO:
?SUM 4 8
12

Line numbers can be any integer inside the INUM limit. Floating point, negative numbers and zero are allowed also.

Percent sign (%) does not echo as a space. Carriage returns within square-bracketed lists print out as such, not as spaces, as in 11LOGO.

:EMPTY is the empty list, which is LISP's *NIL*. *:EMPTYW* is the empty word, which is the LISP atom whose print name is *(ASCII 0)*.

The character control-T [^T] is converted to double quote ["] when it is read in. This is for compatibility with CLOGO. I haven't the faintest idea of why CLOGO does it.

LISP LOGO and 11LOGO differ on the syntax for arrays. LISP LOGO uses the LISP array facility; to define an array use:

?ARRAY <name> T <dimension 1> . . . <dimension N>

Values can be stored by

?STORE <array name> <subscript 1> . . <subscript N> <value>

Values are accessed as if the array were a function, which expected the same number of inputs as the number of dimensions in the array.

The LLOGO function *RANDOM*, of no inputs, returns a random floating point number, which is between zero and one. If given two arguments, it returns a random number between its first and second argument, inclusive. If both its inputs are fixed point, it returns a fixed point number, otherwise it returns a floating point number. *(RANDOM 0 9)* behaves as 11LOGO *RANDOM*.

LLOGO has only one global test box. When a subprocedure performs a *TEST* the result replaces the result produced by any *TEST*'s prior to the subprocedure call in its superprocedure. *IFTRUE*'s and *IFFALSE*'s after the subprocedure call in the superprocedure will be conditional on the last *TEST* which was performed, regardless of what procedure it was in.

ROUNDOFF in LLOGO takes either one or two inputs. If given one input, the number is rounded to an integer, otherwise it is rounded to as many places to the right of the decimal point as specified by the second input.

LOCAL variables are handled differently in LLOGO than in 11LOGO. Regardless of where a *LOCAL* statement is placed in a procedure, the variables declared will be local to the entire procedure. This corresponds to a *PROG* variable in LISP. *LOCAL* accepts any number of variable names as input.

Inserting lines into procedures under program control should be done using the function *INSERTLINE*. In 11LOGO, the following will insert a line into *BLETCH* when *MUNG* is executed:

```
?TO MUNG
>10 EDIT BLETCH
>20 10 PRINT [NEW LINE ADDED TO BLETCH]
>END
```

This will not work in LLOGO. Instead replace line 20 with:

>20 INSERTLINE 10 PRINT [NEW LINE ADDED TO BLETCH]

There is a memo by Wade Williams which explains some of the finer points of 11LOGO syntax, and should be consulted for further information. The 11LOGO User's Manual should also be of assistance.

Section 12. Using LLOGO on MULTICS

LISP LOGO has now been implemented on MULTICS, and this is the only version of LOGO available for that system. Below are instructions for using it, and a list of differences between the MULTICS and ITS versions. Except for the differences in file naming conventions, and limitations imposed by the operating system, source language programs should be entirely compatible. For more information on MULTICS LISP, see the MACLISP Reference Manual by Dave Moon.

The LISP LOGO music package is available for use on MULTICS. See Section 16 for more details. The display turtle and GERMLAND packages are not available in the MULTICS implementation. MULTICS does not have adequate facilities for using displays such as the 340 and the GT40. It probably would be possible to implement a rudimentary turtle package for the storage type displays on MULTICS such as the ARDS and TEKTRONIX terminals, but we have no plans to do so at present. We do hope to have available soon, however, facilities for using the mechanical floor turtles [controlled by the Thornton Box] on both ITS and MULTICS.

12.1 Where To Find It

To obtain LISP LOGO, you must first create a link to the necessary files. After you log in, type

link >udd>ap>lib>logo

This needs to be done only once for each user. Subsequently, you can get LLOGO simply by typing

logo

You should then get a message indicating the version numbers of LISP and LOGO, as on ITS, and the allocator will ask you if you want to use the music box. If you have a file in your directory named *start_up.logo* it will be read in as an initialization file.

12.2 File Naming Conventions

An ITS file specification consists of two file names of up to six characters each, a device and directory name. A file specification on MULTICS is called a "pathname", and consists of arbitrarily many components each naming a node in a tree structure of directories and segments [files]. The components of a MULTICS pathname are separated by ">" characters. Any pathname beginning with ">" is considered to be a full pathname, i.e. start at the root of the tree, otherwise, it is considered to be relative to the directory which is currently the default. This will usually be something like ">udd>your-project-name>your-user-name". File names are assumed also to have two components as on ITS and you type them into to LOGO the same way, as two words, except that each word is not limited to six characters. The default second file name is "logo", not ">", to be consistent with MULTICS conventions. In your directory, the two file names will appear separated by a ".". Files whose second names are "fasl" are assumed to contain object code produced by the LISP compiler. This will correspond to the file with only the first

name [no second component] in your directory. Here are some examples: [assume your name is "person" and your project is "project"]

LOGO file name	MULTICS file name
?readfile foo	*>udd>project>person>foo.logo*
?readfile foo bar	*>udd>project>person>foo.bar*
?readfile foo fasl	*>udd>project>person>foo*
?readfile foo bar mumble	*>udd>project>person>mumble>foo.bar*
?readfile foo bar >udd>llogo	*>udd>llogo>foo.bar*

12.3 Terminalogy

On MULTICS, control characters are entered to LISP by first hitting the *break* or *attn* key [if you have one] and LISP should type CTRL/, then typing the ordinary non-control character, then a carriage return. MULTICS has no other way of acknowledging your existence before you hit a return, which is the reason for this kludge. Because of this the control-character line oriented editor which exists in the ITS implementation, does not exist in the MULTICS implementation. MULTICS uses # to rub out the previous character, and @ to rub out the entire line. To enter these characters to LLOGO, precede them with \.

If you should have to use an IBM 2741 terminal, remember that certain characters must be escaped. The worst offenders are [and] (type *<cent-sign> <less-than>* for [and *<cent-sign> <greater-than>* for]), type *<not-sign>* for ^, *<cent-sign> <cent-sign>* for \, and type a *<cent-sign>* before # and @. Upper and lower cases are distinguished on MULTICS, and all of the system functions, both MULTICS's and LLOGO's, have lower case names.

To use LISP LOGO on MULTICS over the ARPANET from ITS, it is recommended that Dave Moon's program TN6 be used rather than TELNET. See DSK:INFO;TN6 INFO for more details.

Section 13. Using LLOGO on TEN50 and TENEX systems

The version of LLOGO for TEN50 runs in a version of MACLISP that is nearly compatible with that used at MIT-AI. The TEN50 version can also be used on TENEX systems. Most of the incompatibilities are those necessitated by the difference in operating systems. Specifically, the following commands are not implemented:

PRINTOUT INDEX (alias POI, LIST FILES)
LOGOUT (BYE)
COMPILE
LINEPRINT
BUG

Also, the special packages for LLOGO (the turtle primitives, the music primitives, and GERMLAND) are unavailable.

Another difference between TEN50 LLOGO and LLOGO on ITS is in the typing of control characters (such as ∧*G*, ∧*H*, and all the editing characters - ∧*R* ∧*E* etc.). on ITS these characters may be typed at any time. Those specifying an interrupt action (∧*G*, ∧*H*) will always take effect immediately. Unfortunately, this is not true in the TEN50 implementation, because TEN50 allows a running program to be interrupted only by the character ∧C. As a result of this, if the user wants to interrupt the LLOGO system while it is running (e. g. executing a user defined function), he must first type ∧C. This will interrupt the program, and cause it to print ?∧, indicating that it is waiting to read a control-character. The user may then type the desired control-character, and it will be acted upon. Note that typing ∧C is not necessary if the LLOGO system is not running, but rather waiting for input. Therefore the editing characters may be used without difficulty, even on the TEN50 system.

Another minor difference between the two operating systems is in the notation for file names. This difference is minimized by the syntax used by the LLOGO file commands. For instance, the command

?READFILE PROGRM LGO DSK USER

will read the file *DSK:USER; PROGRM LGO* on ITS, while on TEN50 the file read will be *DSK:PROGRM.LGO[USER]*. Thus most user programs will be able to run with little or no modification to their input/output operations. (Note that the default second file name is > on ITS, while on TEN50 it is *LGO*.) If you want to use a LLOGO initialization file with the TEN50 implementation, the name of the file should be *INIT.LGO* on your user directory

A version of TEN50 LLOGO is currently available at Carnegie-Mellon (CMU-10B). It may be loaded there by means of the following command:

.RUN DSK:LOGO[A480LG99]

Section 14. GERMLAND

The GERMLAND package is designed to provide the user with a display environment in which interesting nontrivial questions can easily be investigated, without the need for sophisticated display equipment. The current implementation runs on any of the character display consoles in use at the A.I. laboratory.

Conceptually, GERMLAND consists of a square grid, on which may "live" as many as 10 "germs". Each germ may have an arbitrary LOGO program associated with it; this program determines the germ's movements, as well as whether it eats any of the "food" present at its position of the grid. For a discussion of some of the problems that can be investigated in this environment, see LOGO working paper 7.

14.1 Starting Up

The GERMLAND package may be loaded automatically at the start of an LLOGO run. When started, LLOGO will ask which of the special packages you want. Simply type *YES*, followed by a carriage return, when it asks whether you want GERMLAND. The GERMLAND package will then be loaded, and give you instructions for further help. Note that if the grid becomes garbled, because of a transmission error for instance, you can at any time cause it to be redisplayed by typing the character ^\ [control-backslash].

14.2 Toplevel Primitives

RUNGERM

Invokes prompter. Asks questions necessary to get started and offers help.

GERMDEMOS

Runs a series of demos, leaving the demo programs available for the user to play with.

TOPGERM

Starts up a GERMLAND *READ-EVAL-PRINT* loop, using the grid set up by the most recent call to *RUNGERM*.

UNGRID

Exits from *TOPGERM*, back to LLOGO.

REPEAT *<program1> <program2> ...*

Each program defines one creature. A round consists of executing each program in turn. After each round, the program waits for input. If the user types a space, one round is performed; if the user types a number, that many rounds are done. This is repeated indefinitely until an error occurs. *REPEAT* is not subtle with respect to parallel processing. No effort is made to try each program and see whether any conflicts

occur. However, eventually a more elaborate version could be designed that was sensitive to synchronizing the lives of the germs. If no programs are passed to ***REPEAT***, it attempts to use the programs associated with each germ by ***RUNGERM***.

14.3 Grid Primitives

GRID <number>

Initializes GERMLAND. A square grid is created with <number> squares in a side.

PRINTGRID

Clear screen and redisplay GERMLAND grid. Typing ^\ also causes this to happen. If there is a germ on the square, the character which represents that germ is printed in the square's position. If the square is an obstacle, an "X" is printed. If there is food on the square, the number of particles is printed. If the square is empty, a "." is printed.

GRIDP <position>

A predicate which outputs *T* iff the position is a legitimate grid square.

WRAP

Go into "wraparound" mode, in which germs are allowed to go across the boundaries of the grid.

NOWRAP

Leave "wraparound" mode.

Note that ***WRAP*** and ***NOWRAP*** affect the variable ***:WRAPAROUND***. See Page 46.

MOVE <position>

The germ is moved to the specified grid square. <position> is a sentence of the x and y coordinates of the square. Typical use is: ***MOVE NORTH***. If the germ moves to a square which is already inhabited, the former inhabitant is killed. ***MOVE*** prints an error message if an attempt is made to ***MOVE*** to a square with an obstacle on it, or a square outside the grid. The <position> does not have to be adjacent to the current location of the germ. Hence, ***MOVE*** allows non-local movement to any grid square.

STEP <direction>

<direction> is interpreted as a heading. It must be either 0, 90, 180 or 270 (mod 360). ***STEP*** allows more elegance in the description of a germ program. If the same structure is used for all directions, then the program can call a subprocedure whose input is cycled through the four directions.

14.4 Property Primitives

PUTSQUARE <*position*> <*information*> <*property*>

For the specified grid square, the data stored under the given property is set to <information>.

GETSQUARE <*position*> <*property*>

The information stored under the <property> is returned. Typical uses are:

(GETSQUARE <*position*> *'FOOD)* returns food at <position>.

(GETSQUARE <*position*> *'INHABITANT)* returns the number of the germ currently living there, *NIL* if unoccupied.

(GETSQUARE <*position*> *'OBSTACLE)* returns *T* iff the square is an obstacle.

REMSQUARE <*position*> <*property*>

Removes information stored under <property>.

WHAT <*position*>

Outputs all of the information stored for the given position.

FOOD <*position*>

Outputs the number of food particles at the given position. *FOOD* returns 0, not *NIL*, when there is no food.

FOODP <*position*>

Predicate which returns number of food particles if any at the given position; *NIL* if none.

FILLFOOD <*n*>

Puts <n> morsels of food on each square of GERMLAND.

EAT <*number*>

Subtracts <number> of food particles from the current square. Generates an error if <number> is larger than the total food available. There are two types of germs -- those that are hungry and those that are not. Each hungry germ has a food supply associated with it. The food supply is increased every time he eats by that number of particles, and decreased by one for each generation. If it ever reaches zero, the germ dies. So, if he eats only one particle of food on a turn, he must eat again on the next turn; if he eats 2, he can skip a turn without eating, etc.

14.5 Multiple Germ Primitives

WHERE *<:germ>*

Returns the coordinates of the square that :germ is currently inhabiting.

NORTHP *<:germ>*

Returns true only if the x coordinate of :germ is greater than the X coordinate of the germ whose program is currently being executed by ***REPEAT***.

SOUTHP, WESTP, EASTP

Analogous to ***NORTHP***.

KILL *<:germ>*

Assassinates <:germ> and prints eulogy.

GERM *<:germ> <square>*

Initializes :germ to start out located at <square>. :germ is an integer between 1 and 10.

FOODSUPPLY *<:germ>*

Returns the amount of food that the germ has.

ACCESSIBLE *<square> <:germ>*

True if and only if <:germ> can get to <square> on his next move.

14.6 Turtle Primitives

HEADING *<:germ>*

Returns the current heading of the germ.

FORWARD *<number>*

Move <number> spaces in the direction of the current heading. Abbreviates to ***FD*** <number>. <number> may be negative.

BACK *<number>*

Move <number> spaces opposite to the current heading. Abbreviates to ***BK*** <number>.

NEXT *<direction>*

Returns the coordinates of the next square in the current direction.

RIGHT *<number>*

Turn right <number> degrees--<number> should be a multiple of 90. This may be abbreviated as ***RT*** <number> .

LEFT *<number>*

Equivalent to ***RIGHT*** -<number>. Abbreviates as ***LT*** <number>.

FRONT

Returns coordinates of the square in front of the turtle.

RIGHTSIDE, REAR, LEFTSIDE

Analogous to ***FRONT***.

FSIDE, RSIDE, BSIDE, LSIDE

Abbreviations for ***FRONT*** etc.

14.7 Touch Primitives

TOUCH *<position>*

Outputs ***NIL*** if <position> does not contain something that can be touched. Otherwise it outputs an atom describing the touchable object, e.g. ***BORDER*** or ***OBSTACLE***. Typical use is: ***TOUCH FRONT***.

OBSTRUCT *<square>*

Puts an obstacle at <square>. Germs cannot move onto squares with obstacles.

DESTRUCT *<square>*

Removes obstacle at <square>.

14.8 Global Variables

:GERM

The number of the germ whose program is being executed by ***REPEAT***.

:GRIDSIZE

Size of the GERMLAND grid set by the ***GRID*** function.

:HUNGRY

T => Germs are killed if their foodsupply goes to 0.

NIL => A germ's foodsupply is ignored by ***REPEAT***.

:WRAPAROUND

T => Motion across borders is permitted.

NIL => Motion across borders is an error.

The user should never change *:WRAPAROUND* directly. Use *WRAP* and *NOWRAP* to change modes.

14.9 Implementation

GERMLAND uses an array to represent the grid, and additional arrays for easy access to information about a particular germ. The individual primitives are, for the most part, straightforwardly implementable, given this data representation. Some care is taken in interfacing with the standard LLOGO environment, so that all the usual debugging features of LLOGO may be used in the development of germ programs, without interference with the display of the grid.

Section 15. Display Turtle Primitives

The display turtle package for the 340 and GT40 is also usable from an ordinary LISP as well as from LLOGO. Do *(FASLOAD TURTLE FASL DSK LLOGO)* to get the simple display commands like *FORWARD, RIGHT*, etc. and *(FASLOAD DISPLAY FASL DSK LLOGO)* for the fancier snap-manipulating commands.

Abbreviations for the following primitives are noted in square brackets.

15.1 Starting The Display

STARTDISPLAY [SD]

Initializes the screen. The turtle is displayed at its home, the center of the screen. This command is also useful for restarting everything when things get fouled up, the PDP6 loses, etc. *STARTDISPLAY GT40* uses the GT40 display rather than the 340 display. If you are using the GT40 as a display for the LOGO turtle, it must not be logged in to ITS as a console.

NODISPLAY [ND]

Says you want to stop using the display. Flushes the display slave.

If the display slave for the PDP-6 dies, check that the run light is on. If not, stop, io reset, deposit 0 in 40 and 41 and then start.

LISP has three control characters for the display:

^N

Turns off display.

^Y

Display prints like tty.

^F

Turns on display for turtle, assuming a prior call to *STARTDISPLAY*.

15.2 The Turtle

HIDETURTLE [HT]

Makes the turtle disappear.

SHOWTURTLE [ST]

Brings the turtle back to life.

TURTLESTATE

Returns 0 if the turtle is not displayed, else returns the value of *:TURTLE*. *:TURTLE* is the number of the display item which is the current turtle.

MAKTURTLE <code>

The current turtle is replaced by the picture drawn by <code>. Provides capability to rotate pictures. Subsequent turtle commands, like *FORWARD*, *RIGHT*, etc. will make the picture drawn by <code> move as if it were the original turtle [triangle].

OLDTURTLE

Restores the original LLOGO turtle.

15.3 Moving the Turtle

FORWARD :steps [FD]

Moves the turtle :steps in the direction it is currently pointed.

BACK :steps [BK]

Moves the turtle :steps opposite to the direction in which it is pointed.

SETX :x

Moves the turtle to (:x, *YCOR*).

SETY :y

Moves the turtle to (*XCOR*, :y).

SETXY :x :y

Moves the turtle to (:x, :y).

DELX :dx

Moves turtle to (*XCOR*+:dx, *YCOR*).

DELY :dy

Moves turtle to (*XCOR*, *YCOR*+:dy).

DELXY :dx :dy

Moves turtle to (*XCOR*+:dx, *YCOR*+:dy).

HOME [H]

Moves turtle home to its starting state.

15.4 Erasing the Screen

WIPE

Erases the picture on the screen. Does not affect the turtle, or any snaps.

WIPECLEAN [WC]

Like ***WIPE***, except hides snaps also.

CLEARSCREEN [CS]

Equivalent to ***WIPE HOME***.

15.5 Turning the Turtle

RIGHT :angle [RT]

Turns the turtle clockwise :angle degrees.

LEFT :angle [LT]

Turns the turtle counter-clockwise :angle degrees.

SETHEAD :angle

The turtle is turned to a heading of :angle.

15.6 Examining the Turtle's State

Note: The turtle's home is (0, 0) and a heading of 0 corresponds to pointing straight up. The variables ***:XCOR***, ***:YCOR*** and ***:HEADING*** describe the state of the turtle in floating point. These variables should not be changed explicitly by the user. The following functions return components of the turtle's state rounded to the nearest integer.

XCOR

Outputs the X coordinate of the turtle.

YCOR

Outputs the Y coordinate of the turtle.

HEADING

Outputs the heading of the turtle.

XHOME

Outputs the X coordinate of the turtle's home in absolute scope coordinates (i.e. relative to lower left-hand corner of the screen)

YHOME

Outputs the Y coordinate of the turtle's home in absolute scope coordinates.

15.7 The Pen

PENDOWN [PD]

Pen lowered to paper. Turtle leaves a track when moved.

PENUP [PU]

Pen raised from paper. Turtle does not leave a track when moved.

PENSTATE

Returns +1 = penup or -1 = pendown

PENSTATE <1 or -1>

Sets the penstate. A common use for this primitive is to make a sub-procedure transparent to pen state.

PENP

T if pen is down, else *NIL*.

HERE

Outputs *(SENTENCE XCOR YCOR HEADING)*. Useful for remembering location via *MAKE "P" HERE*.

SETTURTLE :state [SETT]

Sets the state of the turtle to :state. :state is a sentence of X coordinate, Y coordinate, and heading. The heading may be omitted, in which case it is not affected.

RANGE :p

Distance from the turtle's current location to :p. :p is a point specified by a sentence of X and Y coordinates.

BEARING :p

Outputs absolute direction of :p from turtle.

TOWARDS :p

Outputs relative direction of :p from turtle.

15.8 Global Navigation

Note: These primitives return floating point if either of their inputs are floating point.

RANGE :x :y

Outputs distance of turtle from the point (:x, :y).

BEARING :x :y

Outputs absolute direction of (:x, :y) from turtle. *(SETHEAD (BEARING :x :y))* points the turtle in the direction of (:x,:y).

TOWARDS :x :y

Outputs relative direction of (:x, :y) from turtle. *(RIGHT (TOWARDS :x :y))* points the turtle in the direction of (:x, :y).

15.9 Trigonometry

COSINE :angle

Cosine of :angle degrees.

SINE :angle

Sine of :angle degrees.

ATANGENT :x :y

Angle whose tangent is :x/:y.

[*SIN*, *COS*, and *ATAN* are the corresponding functions which input or output in radians]

15.10 Text

SHOWTEXT

Subsequent printing is moved on the screen. Initially, printing begins in the upper left corner.

HIDETEXT

Subsequent printing is no longer displayed. Text currently on the screen remains.

REMTEXT

Any text on the screen is erased and subsequent printing is not displayed.

:SHOW

A variable which is *T* if printing is being displayed, *NIL* if not. Set by *SHOWTEXT*, *HIDETEXT*, and *REMTEXT*. Don't set it yourself.

:TEXT

Variable containing the number of the display item which is the text displayed by *SHOWTEXT*, etc.

MARK :x

(TYPE :X) is placed at the turtle's current location. *SNAP "title" MARK "text"* creates a snap of the word "text". This allows the word to be manipulated, i.e. Moved to any part of the screen, etc.

:TEXTXHOME, :TEXTYHOME

Variables containing coordinates of text to be displayed on the screen. Changeable by user. Initially *:TEXTXHOME*= 0., *:TEXTYHOME*= 1000. These are in absolute scope coordinates.

15.11 Manipulating Scenes

Note: *:PICTURE* is the name of the turtle's track. Does not include any snaps displayed via *SHOW*, *SHOWSNAP*, etc. *:TURTLE* is the name of the turtle. *:TEXT* is the name of any text displayed via *SHOWTEXT*.

SHOW :scene

:scene is moved to the current position of the turtle and displayed. It is not copied.

HIDE :scene

:scene is hidden but not destroyed.

PHOTO "scene" [SNAP]

The current picture is copied and named :scene. Any old snap of this name is destroyed.

PHOTO "scene" <line> [SNAP]

The picture drawn by <line> is named :scene.

ENTERSNAP "scene"

:PICTURE is rebound to a fresh display item. The initial state of this item hides the turtle. Subsequent commands refer to this new item.

ENDSNAP

The original *:PICTURE* is restored.

RESNAP "scene"

:scene is made the current picture. The only difference between this and *ENTERSNAP "scene"* is that a new display item is not created, and the turtle is not hidden, *ENDSNAP* also restores the original *:PICTURE*.

RESNAP "scene" <line>

The picture drawn by <line> is added to :scene. The <line> is executed, referring to the turtle residing in :scene. Subsequent commands will refer to the old turtle.

PICTURE <display commands>

:PICTURE is bound to a new display item while the commands are executed. The original *:PICTURE* is restored following execution of the commands. Similar to *SNAP "scene" <commands>* except that no name is given to the new item. Instead, the number of the item is returned. Thus, the same effect is achieved by:

SNAP "scene" <commands> or
MAKE "scene" PICTURE <commands>

Except that :scene is not added to the list of snaps.

SHOWSNAP :scene

A copy of :scene is displayed at the turtle's current position.

HIDESNAP :scene

All copies of :scene are hidden.

ERASE :scene

All copies of :scene are destroyed.

:SNAPS

A list containing all current snaps.

15.12 Plotter

PLOTTER

The display is plotted on a new plotter page. *PLOTTER* will ask if arrays from previous plot should be erased. The user should type *YES* if his preceding plot is complete.

PLOTTER 1

Display plotted on current plotter page.

NOPLOT

The plotter is released.

DISPAGE

Outline of 7x11 page displayed as *:PAGE*.

15.13 Pots

DIALS :x

Outputs the value of pot :x as a decimal fraction between 0 and 1. Careful: the numbers on the pots are marked in octal, but LLOGO normally expects decimal numbers as input.

15.14 Points

[Points are displayed whether or not the pen is down]

POINT

Displays a point at the turtle's current location.

POINT :p

Displays a point at :p.

POINT :snap :p

Displays a point in :snap at :p.

POINT :snap :x :y

Displays a point in :snap at (:x, :y)

15.15 Global State of the Turtle's World

For all of these functions, the first input ":scene" is optional. If left out, the command refers to *:PICTURE* by default.

SETHOME :scene

Resets turtle's home to current position.

SETHOME :scene :x :y

Resets the turtle's home to the absolute scope coordinates of (:x, :y). Takes effect immediately by moving the current *:PICTURE* to the new home. *(SETHOME :scene 512. 512.)* restores the home to the center of the screen.

MOTION :scene

Moves :scene under the control of space war console 1. Button terminates movement. The new home is returned, expressed in absolute scope coordinates. If the current home is returned immediately and the space war console is ignored, check that all switches on the color scope data switch extension are in the middle position.

BLINK :scene

Blinks :scene.

UNBLINK :scene

Terminates blinking.

BRIGHT :scene

Returns current brightness of :scene as a number from 1 (dimmest) to 8 (brightest). Ordinarily, *:TURTLE* and *:PICTURE* are at maximum brightness.

BRIGHT :scene :level

Sets brightness of :scene to :level, where :level is an integer from 1 to 8.

SCALE :scene

Returns current scale of :scene. Scale is an integer from 1 (standard scale) to 4 (16 times standard scale).

SCALE :scene :size

Sets scale of :scene to :size, where :size is an integer from 1 to 4. :size is a multiplicative scale factor. Hence, *SCALE 2* doubles the size of an ordinary picture, *SCALE 3* quadruples it and *SCALE 4* multiplies the size by 8. *SCALE 1* restores picture to standard size. This is a hardware scaling and affects the current display as well as future displayage.

DSCALE :scale

The length of a turtle step is reset to :scale. :scale may be any real number. Resetting the scale with *DSCALE* rather than *SCALE* has the advantage that the scale factor may be any real number. However, *DSCALE* applies only to future display and not the current picture.

Section 16. The Music Box

The music box is a tone generator for from one to four simultaneous voices, having a range of five octaves. Because of the timesharing environment, music is compiled into a buffer, and then shipped to the music box all at once, for smooth timing. Wherever possible, these primitives have been made compatible with both those of PDP11 LOGO and PDP10 CLOGO. They made be used with the "old" (Minsky) music box, or the "new" (Thornton box compatible) music box.

16.1 Plugging In

To plug in the old music box, find an EXECUPORT terminal. Plug it into a 300 baud ITS line, using the phono type plug on the top right of the EXECUPORT back, or the acoustic coupler. Make sure the terminal is turned off, and plug the music box into the left back of the EXECUPORT. (Or find this all set up in the music room on the third floor.) Turn off the music box and attached percussion box, and put the EXECUPORT switches into the "line" and "uppercase" positions. Turn on the terminal, type ∧Z and log into ITS. The panic procedure for the old music box (symptom: keyboard dead but ITS not down) is to switch to local lowercase mode, turn off the music box, and type *b.*. Then type *$P*.

When using the music box from MULTICS, remember that both carriage return and line feed must be typed to end a line, when using an EXECUPORT. The terminal should be in "half duplex" and "lower case" modes. The panic procedure described above is not recommended, since putting the terminal into local mode will have the effect of logging you out of MULTICS.

Plugging in the new music box is a bit more of a problem due to limitations of present hardware. The critical item is a small piece of electronics known as the "terminal controller card", to be had from General Turtle in the basement of 545 Tech Square. This card is to be inserted in the correct orientation in port 4 of a Thornton box. (If you have never done this, ask! Putting it in backwards will burn out the card.) The music box should be plugged into port 1, 2, or 3, depending upon which has the music box card. (It should be labelled.) Then, plug the interface connector of the Thorton box into a 300 baud ITS line, a terminal into port 4, and log into ITS. The panic procedure for the new music box is to get your terminal to echo "∧*Q* " (control-Q space). Since the normal print routines will actually send <uparrow Q> for <control-Q>, this is most easily done with the "echo" gadget of the Thorton Box, a small connector which makes the Thorton Box look like a full duplex computer line. (If you want to make yourself one, see General Turtle or Mark Miller; you probably won't need it.)

16.2 Turning On

Assuming you are plugged in and logged into ITS, you may now run either music box in LISP or LLOGO. LLOGO will ask you if you want the music box; if so, it will ask you which one; if the new one, it will ask you which port it is plugged into. After answering all questions, type *STARTMUSIC*. It will tell you to turn on the music box (the old one will make a lot of noise), and then type *OK*. Then, the noise (if any) will stop, and you are ready to go. LLOGO behaves much like other LOGOs, and understands the primitives below.

The music box can also be run from a pure LISP. Type *(FASLOAD MUSIC FASL DSK LLOGO)*, and answer the questions. Type *(STARTMUSIC)* and the following primitives will behave like LISP SUBRS or FSUBRS. (If you do *ERRLIST* hacking, see Mark Miller.)

16.3 Music Primitives

A great deal of effort has gone into ensuring upward compatibility with CLOGO and 11LOGO. If you have programs for either of these which no longer work on LLOGO, please let me know. Notice that many "intermediate" level functions such as *CHORUS*, which had been written in LOGO code, are supplied as LISP primitives for efficiency. In addition, new facilities have been added which should be helpful. In the following, all such situations have been indicated. Occasionally, a single function replaces several LOGO functions; the others are still available, but may print a message recommending the newer function for future code. Since most music functions are executed for effect, unless otherwise indicated, the value of a function is the atom (word) *?*.

BOOM

Returns the number which corresponds to a drum beat. Using *DRUM* is more efficient. No inputs.

BRUSH <duration list>

Takes 1 input, a list of durations. Plays (i.e. stores in the music buffer for the current voice) a sequence of brush notes (see *CRITCH*) and rests. A duration of n means 1 brush followed by n-1 rests.

CHORUS <form 1> . . . . <form 4>

Takes from one to four inputs, which should be forms [procedures with arguments, or constants]. *CHORUS* evaluates each argument in turn, and then goes on to the next voice, in cyclic order, and evaluates the next argument. Example:

```
?CHORUS SING 1 10 SING 5 10 SING 8 10
?PM
```

If the number of inputs is the same as *:NVOICES*, sequential calls to *CHORUS* or *SING* will do the expected thing; if the number of voices used by the arguments is equal to *:NVOICES*, recursive calls will also work. For other situations, just remember that *:VOICE* is updated after evaluating each argument. For example, if *:NVOICES* = 3 and you *CHORUS* two calls to *SING*, the next call to *CHORUS* will affect voice 3.

CHORUS2 <form 1> <form 2>

Version of *CHORUS* which takes exactly two arguments. For upward compatibility only.

CHORUS3 <form 1> <form 2> <form 3>

Analogous to *CHORUS2*.

CHORUS4 <form 1> <form 2> <form 3> <form 4>

Analogous to *CHORUS3*.

DRUM <list of durations>

Analogous to *BRUSH* for drum notes (see *BOOM*).

CRITCH

Returns the number corresponding to the brush sound of the percussion speaker. More efficient to use *BRUSH*.

MAKETUNE <tune name>

Takes as input a name, like LOGO *MAKE* or LISP *SET*. It multiplexes the buffer and saves it as the "thing" of the name. That is, it stores the tune as data, as opposed to procedures. This allows faster playing (see *PLAYTUNE*) and easy storage (*SAVE*d with other LOGO variables.) Since *MAKETUNE* does not clear the buffer, allows saving and playing incrementally larger portions of a long piece. Tunes made on one music box can be played on the other, with the exception that tunes with exactly three voices can never be played on the new music box (see *NVOICES*). *MAKETUNE* did not exist in CLOGO or 11LOGO.

MBUFCLEAR

No inputs. Clears the music buffer, and starts at voice 1. This should be done for example, after typing ^*G* to kill an unpleasant song, or after *MAKETUNE*ing the final version of a song, before starting a new one.

MBUFCOUNT

Same as *VLEN*.

MBUFINIT

No-op. Prints message to let you know you tried to use this relic of the past.

MBUFNEXT

No-op. (See *MBUFINIT*)

MBUFPUT

No-op. (See *MBUFINIT*)

MBUFOUT

No inputs. Plays the music buffer. Does not clear it.

MCLEAR

Same as *MBUFCLEAR*.

MLEN

Returns the duration of the longest *:VOICE* created so far (since the last *MBUFCEAR*). Useful for building procedures such as percussion accompaniments for arbitrary length tunes. (see *VLEN, :MAX*)

MODMUSIC <T or NIL>

Takes one input, *NIL* or otherwise. If non-*NIL*, puts music in a mode where numbering is from 0 to 59., and note 60. is the same as note 0. (i.e., (note mod 60)), so that one need not worry about exceeding the range of the music box.

NEWMUSIC

No inputs. Informs system that you wish to use the new music box. Asks which port music box is plugged into. Normally user will not need to call *NEWMUSIC*, as the questionnaire at load time suffices. See: *OLDMUSIC*.

NOMUSIC

No-op. See *MBUFPUT*. This function may be reinstated as a way to excise the music package, for example, when one wants to load the turtle package instead.

NOTE <pitch> <duration>

Unfortunately, (through no fault of LLOGO), there are minor variations between 11LOGO and CLOGO. The difference between *NOTE* and *SING* is one such problem. According to LOGO memo 7 (8/10/73), *NOTE* is the basic 11LOGO music command. It takes two inputs, a pitch and a duration. It numbers pitches chromatically from -24. to 36. with 0 being middle C. There are also three special pitches, as follows:

-28. is a rest
-27. is a boom
-26. is a gritch
-25. is illegal.

11LOGO *NOTE* can also take multiple inputs. LLOGO music has implemented all of this for *NOTE*, except the multiple inputs. The numbering is slightly different from CLOGO *SING*, which is also implemented in LLOGO. (see: *SING*).

NVOICES <1, 2, 3, or 4>

Takes one input, hopefully a number between 1 and 4. Sets *:NVOICES* to that number, clears the buffer, and sets *:VOICE* to 1. Remember that 3 voices is illegal on the new music system, and will generate an error. It is generally better to use four voices, one blank, so that tunes will play on either music box. In *MODMUSIC T* mode, (normally not the case), calling *NVOICES* with a number outside of [1,4.] will not cause an error, but seems crazy. The 1+ input mod 4 will be used instead. *SET*ing *:NVOICES* or *MAKE*ing *"NVOICES"* cannot be prevented, but is considered a faux pas. Accessing *:NVOICES* is welcomed. Calls *MBUFCLEAR* and resets *:VOICE* to 1. See: *:NVOICES, :VOICE, VOICES, MODMUSIC*.

OLDMUSIC

No inputs. Puts system in mode for old music box. Normally not needed by user, as questionnaire at load time suffices. Might be used, for example, if you made a mistake answering the questions. See: *NEWMUSIC*.

PERFORM [Abbreviation PM]

No inputs. Outputs the music buffer, and then does an *MBUFCLEAR*. See: *MBUFOUT, MBUFCLEAR, PLAYTUNE*.

PLAYTUNE <tune>

Takes one input, which must evaluate to a tune created by *MAKETUNE*. It plays the tune. Does not clear or otherwise alter the current music buffer. *PLAYTUNE* is transparent to the current number of voices, even if the tune uses a different number. See: *MAKETUNE, PM*.

REST

No inputs. Returns the number of the note which generates silence on the music box. (Like *BOOM* and *GRITCH*, this will win independently of whether 11LOGO or CLOGO primitives are being used; likewise, it will be the correct number for *MODMUSIC T* or normal state, even for different scalebases.) Naturally this checking is less efficient than just calling *SING -25.* or *NOTE -28.* for the appropriate duration. See: *SING, NOTE, MODMUSIC, :SCALEBASE*.

RESTARTMUSIC

No inputs. Like *STARTMUSIC*, except re-initializes all system variables, and runs questionnaire as far back as asking which music box. Useful in situations of total loss after panic procedure. Usually tunes created by *MAKETUNE*, and user procedures will be intact. Buffer will be wiped out. In cases of peculiar behavior at login or load time, guarantees that everybody thinks they have the device you think they do. If this does not work, go to "PLUGGING IN".

SING <pitch> <duration>

Basic CLOGO and LLOGO music command. Takes two inputs, a pitch number, and a duration. It is highly recommended that durations be integers greater than 0! Very large durations (each unit corresponds to a character atom in LISP) are apt to slow down the system a lot, so small integers are highly advised. Pitches are from -25. to 39., with 0 being middle C. (But see the remarks about 11LOGO's variant, *NOTE*, and also *:SCALEBASE* and *MODMUSIC*.) Pitch -25. is a rest, -24. a boom, -23. a gritch, -22. ignored. (But see *REST, BOOM, GRITCH*) Durations are normally broken down into N-1 beats of pitch and 1 beat of rest, to avoid slurring the music. However, if the *SPECIAL* variable *:INSTRUMENT* is *"STACCATO"*, 1 beat of note followed by N-1 beats of rest will be sung. (i.e., stored in the music buffer under the current voice). If other phrasing is desired, it may be added later.

SONG <pitches> <durations>

Takes two inputs, a list of pitches and a list of durations. Calls *SING*, pairing pitches with durations until the shorter list is exhausted. In other LOGOs, this was not a primitive, but was written as a recursive LOGO procedure.

STARTMUSIC

No inputs. Should be called to turn on the music box. Unlike CLOGO, it pauses to let you turn on the box, to minimize the unpleasant noise generation on the old music box. (*PERFORM* alone will suffice). Clears the music buffer and sets *:VOICE* to 1. Probably unnecessary with new music box.

VLEN

No inputs. Returns duration of current buffer. See: *MBUFCOUNT*, *:MAX*, *MLEN*. Useful when chorusing a tune with an accompaniment. If the accompaniment is the last argument to *CHORUS* and contains a stop rule like,

IF VLEN = MLEN THEN STOP

the accompaniment can be used with arbitrarily long tunes.

VOICE <*voice*>

Sets *:VOICE* to its one input, provided that input is a positive integer less than 5. If greater than the current number of voices, *NVOICES* is called to increase the number. All music from now until the next call to *VOICE* (or a primitive like *CHORUS* which calls *VOICE*) will go into this voice. All the voices in use will be multiplexed prior to *PERFORM*ing the buffer. In *MODMUSIC T* mode inputs greater than 4. do not cause errors, but are simply cycled through the allowed voices. *MAKE*ing (LLOGO) or *SET*ing (LISP) *:VOICE* is not nice.

VOICES

No-op. See *NOMUSIC*. If anyone has a use for this which is reasonable, e.g., synonym for *NVOICES*, I will be glad to implement it.

:INSTRUMENT

Special system variable which is user settable. Its value determines the behavior of *NOTE* and *SING* as above. Current meaningful modes are *LEGATO* and *STACCATO*. Anything else is considered *STACCATO* for now.

:MAX

This pseudo variable is actually a call to *MLEN*, above. It exists for compatibility with CLOGO.

:NVOICES

Special system variable, not to be changed except by calling *NVOICES*. It tells you the number of voices being filled or played at present. Default is 2.

:VOICE

Special system variable, to be changed only by calling *VOICE*. Tells you the current voice that is being filled. *MBUFCLEAR* resets to 1. Always initialized to 1. Can be changed by call to *CHORUS*.

:SCALEBASE

Special system variable which may be changed by user. It tells the offset from middle C to be used in renumbering notes to ones taste. Default is zero.

Index

Index to LLOGO Primitives