P-LISP

version 3.2 for the Apple II / II+ / IIe

User's Guide

published by gnosis

Gnosis, Inc.
4005 Chestnut Street
Philadelphia, PA 19104

TABLE OF CONTENTS

Introduction 1-1
CHAPTER 1 -- USING P-LISP 1-2
1.1 Running P-Lisp 1-2
1.2 Memory Usage 1-2
1.3 Interpreter Overview 1-2
1.4 Special Characteristics of P-Lisp 1-3
1.5 Garbage Collection 1-4
1.6 Error Recovery 1-4
1.7 Apple IIe Owners 1-5
1.8 Using a Printer or 80-Column Card 1-5
1.9 Earlier Versions of P-Lisp 1-5
1.10 About This Manual 1-6
CHAPTER 2 -- THE P-LISP LANGUAGE 2-1
CHAPTER 3 -- P-LISP FUNCTION SUMMARY 3-1
3.1 Elementary Functions 3-1
3.1.1 QUOTE 3-1
3.1.2 CAR 3-1
3.1.3 CDR 3-1
3.2 Elementary Predicates 3-2
3.2.1 ATOM 3-2
3.2.2 EQUAL 3-2
3.2.3 NUMBER 3-3
3.2.4 NULL 3-3
3.2.5 MEMBER 3-3
3.2.6 ZERO 3-3
3.3 Atom/List Building Functions 3-4
3.3.1 CONS 3-4
3.3.2 LIST 3-4
3.3.3 EXPLODE 3-4
3.3.4 IMPLODE 3-5
3.3.5 COPY 3-5
3.3.6 CONC 3-6
3.3.7 APPEND 3-6
3.3.8 RPLACA 3-7
3.3.9 RPLACD 3-7
3.3.10 DELETE 3-7
3.3.11 ARRAY 3-7
3.3.12 CHR 3-8
3.4 Numeric Functions 3-9
3.4.1 ADD 3-9
3.4.2 SUB 3-9
3.4.3 MULT 3-9
3.4.4 DIV 3-9
3.4.5 GREATER 3-9
3.4.6 ASC 3-10
3.4.7 LENGTH 3-10
3.4.8 INT 3-10
3.5 Boolean Functions 3-10
3.5.1 AND 3-10
3.5.2 OR 3-11

3.5.3 NOT 3-11
3.6 Atom Value and Property List Functions 3-11
3.6.1 SETQ 3-11
3.6.2 SET 3-12
3.6.3 PUT 3-12
3.6.4 GET 3-12
3.6.5 REM 3-13
3.7 Input/Output Functions 3-13
3.7.1 READ 3-13
3.7.2 READLINE 3-13
3.7.3 Free Format Input 3-14
3.7.4 PRIN1 3-15
3.7.5 PRINT 3-15
3.7.6 QPRIN1 3-15
3.7.7 QPRINT 3-15
3.7.8 GETCHR 3-15
3.8 Apple Functions 3-16
3.8.1 CALL,PEEK,POKE 3-16
3.8.2 HTAB 3-16
3.8.3 VTAB 3-16
3.8.4 Graphics 3-16
3.8.5 ONERR 3-16
3.9 Function Definition, Flow of Control 3-17
3.9.1 DEFINE 3-17
3.9.2 LAMBDA 3-17
3.9.3 FLAMBDA 3-18
3.9.4 COND 3-19
3.9.5 PROG 3-20
3.9.6 RETURN 3-20
3.9.7 GO 3-20
3.9.8 EVAL 3-21
3.9.9 APPLY 3-22
3.9.10 MAPCAR 3-22
3.9.11 PROGN 3-22
3.10 Object List Functions 3-23
3.10.1 OBLIST 3-23
3.10.2 REMOB 3-23
3.11 Debugging Aids 3-23
3.11.1 TRACE 3-23
3.11.2 UNTRACE 3-24
3.11.3 BREAK 3-24
3.12 Garbage Collection 3-25
3.12.1 GC 3-25
3.12.2 GC 3-26
3.13 Loading and Saving Workspace 3-26
CHAPTER 4 -- Graphics, File I/O, and Math Functions . . 4-1
4.1 Lores Graphics Functions 4-1
4.1.1 GR 4-1
4.1.2 TEXT 4-1
4.1.3 COLOR 4-1
4.1.4 PLOT 4-1
4.2 HIRES Graphics 4-1

4.2.1 HGR2 4-2
4.2.2 HCOLOR 4-2
4.2.3 HPLOT 4-2
4.2.4 HTO 4-2
4.2.5 DRAW 4-2
4.2.6 XDRAW 4-2
4.3 File I/O Functions 4-2
4.3.1 OPENSEQ 4-3
4.3.2 APPENDSEQ 4-3
4.3.3 WRITESEQ 4-3
4.3.4 READSEQ 4-3
4.3.5 CLOSEFILE 4-3
4.3.6 CLOSE 4-3
4.3.7 OPENRND 4-3
4.3.8 WRITERND 4-3
4.3.9 READRND 4-4
4.3.10 WRITEFCN 4-4
4.3.11 APPENDFCN 4-5
4.4 Mathematical Functions 4-5
CHAPTER 5 -- ERROR MESSAGES 5-1
5.1 Numeric Overflow 5-2
5.2 Too Few Args 5-2
5.3 Too Many Args 5-2
5.4 Bad Atomic Arg 5-2
5.5 Bad List Arg 5-2
5.6 Bad Numeric Arg 5-3
5.7 Recursion Check 5-3
5.8 No Dominating Prog 5-3
5.9 Missing Label 5-3
5.10 Undefined Atom 5-3
5.11 No Space 5-3
CHAPTER 6 -- FUNCTION EDITOR AND PRETTY PRINTER . . . 6-1
CHAPTER 7 -- MEMORY MANAGEMENT 7-1
CHAPTER 8 -- MACHINE LANGUAGE INTERFACE 8-1
CHAPTER 9 -- SAMPLE PROGRAMS 9-1

INTRODUCTION

The following User's Guide describes how to use P-LISP, a LISP interpreter for the Apple II (Copyright 1982 by Steven Cherry, All Rights Reserved) microcomputer. The manual is not intended to teach the reader how to program in LISP; it is intended to be used as a reference manual and assumes that the reader has already been exposed to LISP. If you are new to LISP, we strongly recommend that you obtain a book on LISP, such as Learning Lisp, the P-LISP Tutorial published by Prentice-Hall, to use in conjunction with this manual.

P-LISP may be run on an Apple II+ or Apple IIe with disk (Applesoft is required, so you can run P-LISP on an Apple II if you have Applesoft on a language card). P-LISP has a special memory management scheme that allows you to take advantage of the extra memory on a language card; if you have more than 48K available RAM, you should reconfigure P-LISP to use this memory. This is explained in more detail in section 1.6.

The P-LISP disk supplied by Gnosis contains:

-- the P-LISP interpreter
-- a function editor
-- a pretty printer
-- disk file I/O functions
-- mathematical/trig functions
-- Apple IIE conversion program
-- several sample programs, including Towers of Hanoi, ELIZA, and others.

This manual is organized as follows:

Chapter 1 provides some general information about using the interpreter.

Chapter 2 describes the dialect of LISP spoken by P-LISP and can serve as a quick refresher course in LISP.

Chapter 3 is a summary of the P-LISP functions.

Chapter 4 contains a description of the graphics, file I/O, and mathematical functions.

Chapter 5 is a listing of the P-LISP error messages.

Chapter 6 explains how to use the function editor and pretty-printer.

Chapter 7 contains information on P-LISP memory management.

Chapter 8 describes how to interface P-LISP to machine-language routines and provides a list of interpreter entry points.

Chapter 9 explains how to use the sample programs supplied on the P-LISP master disk.

CHAPTER 1 -- USING P-LISP

1.1 RUNNING P-LISP

The interpreter may be run either by typing BRUN LISP or by BLOADing LISP and then typing CALL 2048. You must have Applesoft in ROM or a language card for P-LISP to run.

1.2 MEMORY USAGE

The interpreter resides in hex locations 800-3AFF. The default recursion stack resides in locations 3B00-3FFF. The default workspace resides in locations 4000-95FF. The recursion stack and workspace locations may be modified as desired; see Chapter 7 on memory management for details.

P-LISP makes extensive use of Page Zero for pointers, accumulators, flags, etc. Page zero should be considered off limits. See Chapter 8 for more details on page zero usage.

1.3 INTERPRETER OVERVIEW

The P-LISP interpreter operates in what is commonly known as a READ-EVAL-PRINT loop: it reads some input, evaluates it, prints a result, and waits for more input. The normal interpreter input prompt is the colon, ":". Evaluation may be suspended by typing ctrl-S and interrupted by typing ctrl-C. If an evaluation is interrupted, either by ctrl-C or as the result of an error condition, the interpreter enters "break" mode, and the input prompt changes to the plus sign, "+". Break mode is further explained in Chapter 5.

P-LISP s-expressions must always have an equal number of left and right parentheses. Since counting parentheses can sometimes be a real pain, typing a lot of extra right parens at the end of a s-expr is sufficient; the extra parens will be ignored. For example,

```
:(a (b (c (d
:)))))))))
```

is equivalent to

```
:(a (b (c (d))))
```

Anything following a semicolon is ignored by the interpreter. Thus semicolons can be used if you wish to document a function definition in a text file, for example,

```
(DEFINE (BLEAT (LAMBDA (X)  ;one argument function
   (MOO (CDR X))            ;call MOO with the CDR
   )))                    ;  and return
```

String atoms may be delimited on the right by either a double quote or a carriage return; thus

:"foo bar"

and

:"foo bar <carriage return>

are equivalent.

P-LISP uses Applesoft entry points for parsing numeric atoms and for numeric computations. Because of this, you should be aware of a particular P-LISP anomaly: as a result of DOS, Applesoft, and P-LISP competing for the same page zero space, a NUMERIC OVERFLOW error will occasionally occur if a numeric atom is entered after several disk I/O operations. In these cases, the error is not really an overflow but rather an OUT OF MEMORY error that Applesoft generates because it gets confused. Hitting RESET in such cases will usually clear things up.

If you enter the monitor from P-LISP, ctrl-Y can be used to re-enter the interpreter.

1.4 SPECIAL CHARACTERISTICS OF P-LISP

There is no restriction on the lengths of atom print names; however, shorter print names consume less memory.

There is no limit to the nesting level of lists; however, deep nesting may cause a RECURSION CHECK during garbage collection (see section 1.5 below).

All of the Apple DOS commands (as well as PR#n and IN#n) can be entered from LISP. Thus, it is possible to save function definitions in a text file and then add the functions to your workspace by simply EXECing the file. Note that this feature can sometimes produce unexpected results: in particular, you should not embed DOS commands in a function definition because DOS sees them before P-LISP does.

P-LISP is best used in conjunction with a text editor. The editor can be used to create functions, which can then be EXECed into a P-LISP workspace. There are also LISP functions available that write LISP function definitions to a text file.

P-LISP makes slight modifications to DOS. You should reboot DOS after exiting P-LISP (typing INT or FP will exit P-LISP).

P-LISP saves workspaces using the special file type S. Because of this, certain copy programs will not copy these files correctly.

1.5 GARBAGE COLLECTION

During the course of an evaluation, the interpreter consumes memory "cells" for temporary storage and for creating new lists and atoms. The Garbage Collector is invoked whenever the interpreter runs out of these cells. Whenever the Garbage Collector is invoked, the message "** GARBAGE COLLECTION **" is printed, followed by the number of cells collected (this message can be suppressed; see Chapter 3 for details). You are rapidly approaching the limits of the interpreter's capabilities if a garbage collection results in less than 300 cells. Note: you should NEVER hit RESET during a garbage collection. Doing so will destroy the contents of your workspace.

1.6 ERROR RECOVERY

The following procedure should be followed if your results are not as you expect, or if P-LISP gives you error message:

1) Check all user-defined functions. Make sure all parentheses are where they should be (this is a common source of errors!). Use the trace facilities to track down bugs. (see chapter 3)

2) Check recursion levels.

3) Be sure that functions that should not be traced are not being traced. See Chapter 3 Section 11 for details.

4) It is generally a good idea to save your work space before testing your functions, since your functions may damage the work space if they are written incorrectly. Good programming practice dictates that you test your "help" functions first on data which will give known results before testing your main functions.

5) Examine local atom bindings. This means that you should use the suspended execution mode to take a look at the values of your atoms WHILE STILL IN THE FUNCTION. See chapter 4 for the explanation of this mode.

1.7 APPLE IIe OWNERS

If you are using an Apple IIe, or an Apple II+ with a language card, you can reconfigure your copy of the interpreter to take advantage of the extra memory. The P-LISP disk contains a program called CONVERT which will make the appropriate changes to the interpreter's memory map to use memory locations D000-FFFF. To reconfigure the interpreter, do the following:

1) Put the P-LISP disk in a drive and BRUN CONVERT.
2) The program will ask you to put a disk containing a copy of the interpreter into the same drive and then hit RETURN.
3) The program will then update the memory map. As a precautionary measure, you should be sure you update only a BACKUP COPY of the interpreter, NOT your original copy.

You can also do this reconfiguration manually if you so desire. See Chapter 7 for details.

1.8 USING A PRINTER OR 80-COLUMN CARD

P-LISP is designed to work with any display device of arbitrary width. By default, the interpreter assumes that the display device has a width of 40 columns. This can be changed by simply POKEing the desired display width into Page Zero location 240 (hex location $F0). (See Chapter 3 for details on the POKE function). Note that if you are using a printer you must do this in addition to sending the usual control sequences to the printer so it will print more than 40 columns.

1.9 EARLIER VERSIONS OF P-LISP

The current release version of P-LISP is version 3.2. This version contains several enhancements over earlier versions of the interpreter, including the addition of new functions, modifications to existing functions, and performance improvements. If you are a current user of a version of P-LISP earlier than this release, you should make note of the following changes:

1) new functions MEMBER, READLINE, ASC, QPRIN1, and QPRINT have been added.

2) The functions APPLY, ARRAY, CHR, GETCHR, GO, and REMOB have been modified.

3) FREAD and FREADLINE have been de-implemented, although their effect can still be achieved. See Chapter 3 under READ for details.

4) QP has been de-implemented. The effect of QP can now be obtained by QPRIN1 and QPRINT. See Chapter 3 for details.

5) *UNDEF has been deimplemented. Taking the CAR of an atom will now cause an error.

6) String atoms are now implemented differently. A string atom is now considered a constant (like a numeric atom) and does not have a property list. Chapter 2 contains more details.

7) Applesoft in ROM is now required. All mathematical calculations are done in floating point.

8) Error-checking for some SUBRs has been relaxed.

9) The interpreter is 15-20% faster than older versions.

10) Some I/O anomalies that occured when using a printer or an 80-column card have been corrected (see Section 1.7).

11) P-LISP no longer does any special output formatting for trace or error messages. Output is no longer indented.

12) The function editor and pretty-printer have been improved.

13) The nesting level of PROGs is now limited to the size of the recursion stack.

1.10 ABOUT THIS MANUAL

Throughout this manual, any input that should be typed by the users appears in lower case, and anything that the computer responds with will be in upper case.

Ex:

```
:(print '"hello there computer")
  HELLO THERE COMPUTER
```

NOTE: At certain times in the text, there will be special, indented passages. These mean that there is some special thing to take note of, such as a programming tip, or something that is commonly done wrong, and you should watch out for it. The face you see on the side is named GNORMAN (from GNOSIS, where else?), and he is there to remind you to take special note.

CHAPTER 2 -- THE P-LISP LANGUAGE

This chapter will briefly discuss the basic structures and mechanisms of LISP in general and P-LISP in particular. It is not intended to teach the reader LISP programming; for this we direct the reader to the P-LISP Tutorial. However, the chapter does introduce the novice user to LISP and can also serve as a quick refresher course for those already well versed in LISP.

LISP stands for "LISt Processor." It was developed by John McCarthy at MIT in the late 1950's, originally as a tool for mathematical research. Because of its unique features, LISP quickly caught on as the language of choice for any application requiring the symbolic manipulation of data. Unlike many other programming languages, like FORTRAN, PASCAL, COBOL, and others, there is no LISP "standard"; there are as many dialects of LISP as there are implementations. P-LISP is loosely based on MTSLISP, a dialect developed at the University of Michigan.

Some of the major strengths and features of LISP can be outlined as follows:

--LISP uses the same data structure to represent both functions and data. Since programs and data are indistinguishable (as far as the LISP interpreter is concerned), it is relatively easy to write LISP programs that construct and execute other LISP programs.

--LISP is interpreted, providing the user with immediate response. The interactive nature of LISP provides a friendly environment for program development.

--LISP has a simple and uniform syntax; there are only a few rules to remember, and they quickly become second nature.

--LISP is ideal for applications requiring symbolic manipulation because the mechanisms needed for such an application are built into the language. Thus, for example, a program to differentiate polynomials might be a chore to write in BASIC or PASCAL, but is relatively simple and straightforward in LISP.

--LISP is an heirarchical language; it is possible to build LISP-like languages (such as Smalltalk, Prolog, Logo, etc) out of a LISP "toolkit".

The basic unit of information in LISP is the "atom." There are three kinds of atoms: "literal" atoms, "numeric" atoms, and "string" atoms. A literal atom is represented by a string of characters of arbitrary length. For example, the following are literal atoms:

```
APPLE
THISISAVERYLONGATOM
```

```
ABC83KR
#%&?
```

A numeric atom is simply a number; it can be either an integer or a floating-point number. Thus, the following are numeric atoms:

```
61
-37
3.14159
1.7e+09
```

String atoms are strings of characters surrounded by quotes. The following are examples of string atoms:

```
"good morning"
"what's all this then"
"282 is a number"
```

Atoms can be combined to form the basic data structure of LISP, the "list." A list is a sequence of "symbolic expressions", or "s-exprs", bound by a pair of parentheses, where a s-expr is defined to be an atom or a list. So,

```
(A B C)
```

is a list comprised of three s-exprs, the atoms A, B, and C. Similarly,

```
(HAIL AND (WELL MET))
```

is a list comprised of three s-exprs, namely the atom HAIL, the atom AND, and the list (WELL MET), which itself is comprised of the two atoms WELL and MET. A list may contain any number of atoms or lists as its elements.

One list important to the LISP system is the object list, or OBLIST. The OBLIST is a list of all literal atoms that the interpreter knows about. Whenever a literal atom is typed into LISP, the atom is added to the OBLIST if it is not already there; there can only be one instance of a given literal atom. Numeric and string atoms are handled differently; these atoms are not stored on the OBLIST, but rather, a separate instance of such atoms is created whenever they are typed in.

At the heart of LISP is the Evaluator. Whenever you type something into LISP, the interpreter tries to evaluate what you typed in and return the result (this is known as a READ-EVAL-PRINT loop). The following rule is used for evaluating lists: LISP treats the first element of the list as the name of a function, and the remaining elements (if any) as the arguments to the function. So, if you type (A B C) into LISP, the interpreter will try to apply some function named A to the arguments B and C. For example, to add two numbers

together, the ADD function is used; typing the list (ADD 1 2) will cause the interpreter to apply the ADD function to the arguments 1 and 2, returning the result 3.

In LISP, all s-exprs are considered to have values. The value of the s-expr (ADD 1 2) is the numeric atom 3. What about atoms? Numeric and string atoms are treated as "constants": they have themselves as values. So, if you type the numeric atom 25 into LISP, its value is returned, namely 25. Similarly, giving the evaluator a string atom causes the evaluator to return the atom.

Literal atoms can be assigned arbitrary values. The value of a literal atom can be any s-expr. The LISP function SETQ is used to assign values to atoms; the first argument to SETQ is the atom to be assigned, and the second argument is the value. So, typing

```
(SETQ A 1)
```

sets the value of A to 1. Now, whenever the atom A is typed into LISP, the value 1 will be returned.

Two literal atoms have pre-defined values. One of these is the atom NIL; the value of NIL is NIL. NIL is also used to represent a list containing zero elements, i.e., (). NIL is also used to represent the truth value "false". Another literal atom, T, is used to represent the truth value "true"; the value of T is T.

For most functions, LISP evaluates the function's arguments and then applies the function to the argument values. Thus, if A is SETQed to 1, and B is SETQed to 2, then typing

```
(ADD A B)
```

will result in the value 3 (the arguments A and B are evaluated, resulting in 1 and 2, to which the ADD function is then applied).

QUOTE is a function which is used to prevent evaluation of a s-expr. The apostrophe ' is used as a shorthand for QUOTE. Thus, typing

```
(F 'A 'B)
```

causes the function F to be applied to the atoms A and B, RATHER than the VALUES of the atoms A and B. Typing

```
'A
```

is the same as typing

```
(QUOTE A).
```

Two of the fundamental LISP functions are called CAR and CDR.

The function CAR takes a list and returns the first element of the list. So, typing

```
(CAR '(ONE TWO THREE))
```

returns the first element of the list (ONE TWO THREE), namely the atom ONE. The function CDR is the complement of CAR; given a list, CDR returns the list minus the first element. So,

```
(CDR '(ONE TWO THREE))
```

returns the list (TWO THREE). Note that taking the CDR of a one-element list, like

```
(CDR '(FOUR))
```

results in the empty list, NIL.

Lists can be put together as well as be taken apart. The function CONS takes two arguments and returns a list which has the first argument as its CAR and the second argument as its CDR. So,

```
(CONS 'THIS '(IS FUN))
```

returns the list (THIS IS FUN).

One area where the pre-defined atoms T and NIL come into play is with "predicates". Predicates are functions that perform a certain test on their arguments and return T if the argument passes the test and NIL if it fails. One such predicate is ATOM, which returns T if its argument is an atom and NIL otherwise. So,

```
(ATOM '(IM A LIST))
```

returns NIL, whereas

```
(ATOM 'BOMB)
```

returns T.

LISP wouldn't be much fun or useful unless you were able to create your own functions. The LISP function DEFINE is used to define functions. The general form of a function definition is as follows:

```
(DEFINE (function-name (LAMBDA (formal arguments)
         function-body
        )))
```

The function name must be a literal atom. A "LAMBDA-expression" follows the function name; all user-defined functions must be a form of LAMBDA-expression. The LAMBDA is followed by a list of the function's "formal arguments"; this list tells LISP the number of

actual arguments the function takes, and how these arguments are referred to in the body of the function. The formal argument list is followed by the function body, which is a s-expr whose value is returned when the function is invoked.

As an example, suppose you want a function that returns the second element of a list; that is, if you gave the function the list (A B C), you want the function to return B. Call this function SECOND:

```
(DEFINE (SECOND (LAMBDA (X)
          (CAR (CDR X))
        )))
```

X is the formal argument to SECOND. When SECOND is invoked, X is bound (SETQed) to the value of the actual argument. Then, the body of the function is evaluated and the value is returned. When SECOND is exited, X is restored to whatever value it had before SECOND is entered. So, if you type

```
(SECOND '(A B C))
```

the argument is evaluated, returning (A B C) (because of the quote), which is assigned to the formal argument X. The body of the function is evaluated, returning the CAR of the CDR of X, which is B. The function is then exited, with X restored to its previous value.

Function definitions are stored on an atom's "property list". A property list is a list of properties and property values that may be associated with a literal atom. For example, you may want to assign the property COLOR to the atom BALL with the property value RED. This is accomplished via the LISP function PUT; typing

```
(PUT 'BALL 'COLOR 'RED)
```

stores the property COLOR with value RED on the property list of atom BALL. The function GET returns property values; typing

```
(GET 'BALL 'COLOR)
```

will now return the property value RED.

Functions defined by the user are called EXPRs; thus, for the function SECOND defined above, SECOND has the property EXPR on its property list, with the property value being the function definition. Built-in functions are called SUBRs; the property value for the SUBR property is a pointer to the machine-language routine in the interpreter that evaluates the function.

P-LISP uses the following method for evaluating s-exprs; if the CAR of the s-expr is an EXPR, that EXPR is evaluated. Otherwise, if the CAR is a SUBR, that SUBR is evaluated. If the CAR is neither an

EXPR nor a SUBR, the interpreter will evaluate the CAR, then check if the value is an EXPR, a SUBR, etc. The interpeter will continue this evaluation until an EXPR or SUBR is reached; if an atom is reached that is neither an EXPR nor a SUBR and has no value, the interpreter will complain with an error. For example, using the function SECOND defined above, you can type

```
:((car '(second)) '(d e f))
```

The CAR of the s-expr will be evaluated, returning SECOND, which will then be applied to the argument, returning the atom E.

There is much more to LISP than has been described here. The following chapters, in conjunction with Learning Lisp, the P-Lisp Tutorial, should help the novice gain more familiarity with the language than this brief introduction has been able to provide. Learning Lisp is a very helpful book containing many clear examples using P-Lisp, and is available from Gnosis, Prentice-Hall Publishers, most bookstores or an authorized dealer.

CHAPTER 3 -- P-LISP FUNCTION SUMMARY

NOTE: In the following descriptions, <s> represents a s-expr, <a> an atom, <L> a list, and <na> a numeric atom. It is understood that all arguments are evaluated before the function is applied unless stated otherwise.

In the examples below, user input is presented in lower-case, and the interpreter's responses are in upper-case.

3.1 ELEMENTARY FUNCTIONS

3.1.1 (QUOTE <s>)

QUOTE is a pseudo-function used to prevent evaluation of an argument. The argument <s> is NOT evaluated. The value of (QUOTE <s>) is <s>. The punctuation mark ' is used as a shorthand for QUOTE for input purposes; thus 'A and (QUOTE A) are equivalent.

Ex:

```
:'a
  A

:'(this is a list)
  (THIS IS A LIST)

:(quote (b c))
   (B C)
```

3.1.2 (CAR <L>)

CAR returns the the first element of a list. If <L> is NIL then CAR returns NIL.

Ex:

```
:(car '(a b))
 A

:(car '((this is) a test))
 (THIS IS)
```

3.1.3 (CDR <s>)

The argument to CDR may be either a list or a literal atom. IF <s> is a list, CDR returns the list without its first element. If <s> is a literal atom, CDR returns the atom's property list.

Ex:

```
:(cdr '(a b c))
 (B C)

:(cdr 'car)
 (SUBR *)

:(cdr '(coulder))
 NIL
```

3.2 ELEMENTARY PREDICATES

3.2.1 (ATOM <s>)

ATOM returns T if <s> is an atom; otherwise, NIL.

Ex:

```
:(atom 'bomb)
 T

:(atom '(im a list))
 NIL

:(atom "deoxyribonucleic acid")
 T
```

3.2.2 (EQUAL <s1> <s2>)

EQUAL returns T if <s1> and <s2> are equal and NIL otherwise. Equality is defined as follows: if the arguments are lists, the lists are EQUAL if they have the same list structure; if the arguments are atoms, they are EQUAL if they are the same literal atom, if they are numeric atoms with the same numeric value, or if they are string atoms consisting of the same sequence of characters.

Ex:

```
:(equal 'a 'a)
 T

:(equal '(hi there) '(hi there))
 T

:(equal '(hi there) '(hi where))
 NIL

:(equal 25 (mult 12.5 2))
 T

:(equal "neutron" "neutron")
 T
```

3.2.3 (NUMBER <s>)

NUMBER returns T if <s> is a numeric atom; otherwise, NIL.

Ex:

```
:(number 3.493)
 T

:(number 'bleat)
 NIL

:(number -1)
 T
```

3.2.4 (NULL <s>)

NULL returns T if <s> is NIL; otherwise, NIL.

Ex:

```
:(null ())
 T

:(null '(not null))
 NIL
```

3.2.5 (MEMBER <s> <L>)

MEMBER returns T if <s> is EQUAL to an element of <L>, and NIL otherwise.

Ex:

```
:(member 'what '(which what where))
 T

:(member 12 '(11 (12) 13))
 NIL

:(member '(goo) '((gee) (gaa) (goo))
 T
```

3.2.6 (ZERO <na>)

ZERO returns T if <na> is zero; otherwise, NIL.

Ex:

```
:(zero (sub 2 2))
 T

:(zero (add 3 2))
 NIL
```

3.3 ATOM/LIST BUILDING FUNCTIONS

3.3.1 (CONS <s1> <s2>)

CONS is used to construct new lists. CONS returns a list whose CAR is <s1> and whose CDR is <s2>. If <s2> is an atom, a special form of list, called a "dotted pair", is formed. The dotted pair construct is more or less a holdover from the early days of LISP and is not used much.

Ex:

```
:(cons 'hi '(there))
 (HI THERE)

:(cons 'thisatom ())
 (THISATOM)

:(cons 24 91)
 (24 . 91)

:(cons '(meow) 'arf)
 ((MEOW) . ARF)
```

3.3.2 (LIST <s1> <s2> . . . <sn>)

LIST takes any number of arguments and returns a list whose elements are the given arguments. If no arguments are supplied, LIST returns NIL.

Ex:

```
:(list 'this 'is '(a list))
 (THIS IS (A LIST))

:(list 'n)
 (N)

:(list 'thisatom ())
 (THISATOM NIL)

:(list)
 NIL
```

3.3.3 (EXPLODE <a>)

EXPLODE takes as its argument an atom and returns a list of atoms whose print names are the individual characters of the atom's print name. The atom may be literal, numeric or string. For a numeric atom, EXPLODE will return a list of numeric atoms representing the number's digits. EXPLODEing a string atom has the same effect as exploding a literal atom.

Ex:

```
:(explode 'boom)
 (B O O M)

:(explode 54721)
 (5 4 7 2 1)

:(explode "meow")
 (M E O W)
```

3.3.4 (IMPLODE <L>)

IMPLODE does the reverse of EXPLODE. IMPLODE takes a list of atoms and returns an atom whose print name is a concatenation of the print names of the atoms in the list. If the first element of the list is a literal atom, IMPLODE will create a literal atom from the list. If the first element is a numeric atom, IMPLODE will create a numeric atom. Note that in this case any non-numeric atoms in the list will cause the remainder of the list to be ignored (see example). If the first element of the list is a double quote, IMPLODE will return a string atom (this can be achieved by CONSing a (chr 34), which returns a double quote, onto the desired list of atoms).

Ex:

```
:(implode '(a p p l e p i e))
 APPLEPIE

:(implode '(1 2 9 4 1))
 12941

:(implode '(3 2 a q 5))
 32

:(implode (cons (chr 34) '(a e i o u)))
 "aeiou"
```

Note that a string atom can be converted to a literal or numeric atom by EXPLODEing it and then IMPLODEing the result. For example,

Ex:

```
:(implode (explode "garp"))
 GARP

:(implode (explode "234"))
 234
```

3.5 (COPY <s>)

COPY returns a copy of its argument. If <s> is atomic, COPY returns the atom but does NOT make a copy of the atom. If <s> is a

list, COPY returns a copy to the list.

```
Ex:
     :(copy 'cat)
       CAT

     :(copy '(this that those))
        (THIS THAT THOSE)
```

3.3.6 (CONC <L1> <L2> ...<Ln>)

All the arguments to CONC must be lists. CONC returns a concatented list of copies of the arguments <L1> through <Ln>. That is, CONC makes a copy of each list and then connects the end of each list to the beginning of the next list. If NIL is one of the arguments, it is ignored.

```
Ex:
     :(conc ' (a)  '(b))
       (A B)

     :(conc '(take) '(me) nil '(to your leader))
       (TAKE ME TO YOUR LEADER)
```

3.3.7 (APPEND <L> <s1> <s2>...<sn>)

APPEND returns a copy of the list <L> with s-exprs <s1> through <sn> appended to the end of the list as new elements. If <L> is NIL, APPEND simply returns the LIST of the remaining arguments. APPEND is the same as CONCatenating the first argument with the LIST of the remaining arguments.

```
Ex:
     :(append '(a b)  '(c d)  'e)
       (A B (C D) E)

     :(append ' (take) '(me) nil '(to your leader))
       (TAKE (ME) NIL (TO YOUR LEADER))
```

In both CONC and APPEND, the list arguments are copied because the functions modify their structure. For example, if the value of L is (A LIST), then after

```
:(append L '(and another))
   (A LIST (AND ANOTHER))
```

the value of L is still (A LIST), since the APPEND made a copy of L before appending the second argument.

3.3.8 (RPLACA <L> <s>)

RPLACA replaces the CAR of <L> with <s> and returns the result. REPLACA is considered a destructive function, in that it directly modifies the list <L> rather than a copy of <L>. Thus all s-exprs that reference <L> will reference the new <L>. Contrast this to APPEND and CONC, both of which make copies of their arguments first.

Ex:

```
:(rplaca '(this is a list) 'where)
 (WHERE IS A LIST)

:(rplaca '(write this carefully) 'read)
 (READ THIS CAREFULLY)
```

3.3.9 (RPLACD <L> <s>

RPLACD replaces the CDR of <L> with <s> and returns the result. Like RPLACA, RPLACD is destructive.

Ex:

```
:(rplacd '(change this) '(to that))
 (CHANGE TO THAT)
```

WARNING: Both RPLACA and RPLACD can be very dangerous. Use them with caution.

3.3.10 (DELETE <S> <L>)

DELETE deletes all occurrences of <s> from <L> and returns the result. DELETE deletes those elements of <L> that are EQUAL to <s>. Like APPEND and CONC, DELETE makes a copy of <L> before doing the deletion.

Ex:

```
:(delete 'a '(d a c a b a))
 (D C B)

:(delete '(howdy) '(this (howdy) is (howdy) it))
 (THIS IS IT)
```

3.3.11 (ARRAY <L1> <L2>)

ARRAY is used to select specific elements or sub-elements of a list. <L1> is the list of indices into the "array"; <L2> is the "array", the list to be indexed. ARRAY returns the s-expr in <L2> which resides at the "address" indicated by <L1>; if <L1> does not select a valid element (by indexing a non-existent element or trying to index an atom), ARRAY returns NIL. Note that the elements of <L1>

are evaluated before ARRAY is applied.

Ex:
Assume that LIST is the list

((A B C) (D E F) (G H I))

```
:(array '(1) list)
 (A B C)

:(array '(2) list)
 (D E F)

:(array '(3 2) list)
 H

:(array '(4) list)
 NIL
```

If I is SETQed to 1, and J is SETQed to 2, then

```
:(array '(i j) list)
 B
```

3.3.12 (CHR <na>)

CHR returns a one-character string atom which is the ASCII character corresponding to the value of <na>. This is analogous to the CHR$ function of Applesoft.

Ex:

```
:(chr 65)
 "A"

:(chr 40)
 "("
```

3.4 NUMERIC FUNCTIONS

3.4.1 (ADD <na1> <na2>)

The arguments to ADD must be numeric atoms. ADD returns the sum of the arguments. There must be two arguments.

```
Ex:
     :(add 3 4 )
      7
```

3.4.2 (SUB <na1> <na2>)

SUB subtracts <na2> from <na1> and returns the result. The arguments must be numeric atoms.

```
Ex:
     :(sub 5 8)
      -3
```

3.4.3 (MULT <na1> <na2>)

MULT returns the product of <na1> times <na2>.

```
Ex:
     :(mult 5 7)
      35
```

3.4.4 (DIV <na1> <na2>)

DIV returns the quotient of <na1> divided by <na2>.

```
Ex:
     :(div 12 2)
      6

     :(div 9 4)
      2.25
```

3.4.5 (GREATER <na1> <na2>)

GREATER returns T if <na1> is strictly greater than <na2>; otherwise, NIL.

```
Ex:
     :(greater 1 -1)
       T

     :(greater 3 3)
       NIL
```

3.4.6 (ASC <a>)

ASC returns the ASCII value of the first character of the print name of <a>. <a> can be either a literal or a string atom.

Ex:

```
:(asc 'apple)
 65

:(asc "lisp")
 76
```

3.4.7 (LENGTH <L>)

LENGTH returns the number of elements of a list.

Ex:

```
:(length ())
 0

:(length '(how long is (this list)))
 4
```

3.4.8 (INT <na>)

INT returns the integer part of a floating point numeric atom.

Ex:

```
:(int 371.63)
 371
```

3.5 BOOLEAN FUNCTIONS

3.5.1 (AND <s1> <s2> . . . <sn>)

AND performs a boolean AND on its arguments. AND evaluates each argument in succession; if one of the arguments evaluates to NIL, AND returns NIL and the remaining arguments are NOT evaluated. If none of the arguments evaluate to NIL, AND returns the value of the last argument.

Ex:

```
:(and (null ()) (add 3 7))
 10

:(and (zero 3) (list t))
 NIL
```

and the LIST s-expr is not evaluated.

3.5.2 (OR <s1> <s2> . . . <sn>)

OR performs a boolean OR on its arguments. Each argument is evaluated in succession; if one of the arguments evaluates to a non-NIL value, OR returns that value and the remaining arguments are NOT evaluated. If all arguments evaluate to NIL, OR returns NIL.

Ex:

```
:(or (null t) (cons 'a '(b)))
 (A B)

:(or (greater 4 2) (car '(hi there)))
 T
```

and the CAR s-expr is not evaluated.

3.5.3 (NOT <s>)

NOT returns NIL if <s> is non-NIL: otherwise, T.

Ex:

```
:(not (and (null ()) (car nil)))
 T

:(not (or (zero 0) (car '(a b))))
 NIL
```

3.6. ATOM VALUE AND PROPERTY LIST FUNCTIONS

3.6.1 (SETQ <a1> <s1>...<an> <sn>)

SETQ is used to set the value of an atom. The value of each <ai> is set to the corresponding <si>, and the value of the last <si> is returned as the value of the SETQ. Note that each <ai> is NOT evaluated, but each <si> IS evaluated. In addition, each <ai> must be a literal atom.

Ex:

```
:(setq a (cdr '(this is)) b '(a list))
  (A LIST)
```

sets the value of A to (IS) and the value of B to (A LIST).

3.6.2 (SET <a1> <s1> . . . <an> <sn>)

SET works like SETQ, except that each <ai> IS evaluated before the SET is applied, and must evaluate to an atom.

Ex:

```
:(set 'a '(a list))
 (A LIST)
```

sets the value of A to (A LIST).

```
:(set 'b 'c b '(another list))
  (ANOTHER LIST)
```

sets the value of B to C and the value of the value of B (which is now C) to (ANOTHER LIST).

3.6.3 (PUT <a> <prop> <pval>)

PUT is used to put properties and values on an atom's property list (p-list). PUT puts the property <prop> with property value <pval> on the p-list of atom <a>. Both <a> and <prop> must be literal atoms; <pval> can be any s-expr. PUT returns the assigned property value.

Ex:

```
:(put 'apple 'color 'red)
 RED
```

puts the property COLOR with value RED on the p-list of APPLE. Thus (CDR 'APPLE) returns (COLOR RED). If we now enter

```
:(put 'apple 'fruit T)
 T
```

then (CDR 'APPLE) is now (COLOR RED FRUIT T).

If <prop> is already on <a>'s p-list, <pval> replaces the old property value. Thus, if we enter

```
:(put 'apple 'color 'green)
  GREEN
```

then (CDR 'APPLE) is now (COLOR GREEN FRUIT T).

3.6.4 (GET <a> <prop>)

GET returns the property value of property <prop> from the p-list of atom <a>. If <prop> is not on <a>'s p-list, GET returns NIL. <a> and <prop> must be literal atoms.

Ex: Using the above example for PUT,

```
:(get 'apple 'color)
 RED

:(get 'apple 'size)
 NIL
```

3.6.5 (REM <a> <prop>)

REM removes the property <prop> and the associated value from the p-list of atom <a>. REM always returns NIL. Note the property <prop> does not necessarily have to be on the p-list of <a>, in which case REM has no effect.

Ex: Using the above examples,

```
:(rem 'apple 'color)
 NIL
```

and (CDR 'APPLE) is now (FRUIT T).

```
:(rem 'apple 'size)
 NIL
```

and the p-list of APPLE is unchanged.

3.7 INPUT/OUTPUT FUNCTIONS

3.7.1 (READ)

READ causes the interpreter to read input from the current input device. READ will return when a s-expr has been read, returning the s-expr as its value. The question mark '?' is the input prompt for READ.

Ex:

```
:(setq X (read))
```

will set the value of X to be whatever is input from the current input device. Note that the input is NOT evaluated.

3.7.2 (READLINE)

Like READ, READLINE causes the interpreter to read input from the current input device. READLINE will read a complete line of text, up to a delimiting carriage return, and form a list of the all the s-exprs read from the line of text. The list is returned as the value of READLINE.

Ex: After typing in

```
:(readline)
```

and then entering a line of text,

```
?this is a test
```

READLINE will return the list

```
(THIS IS A TEST).
```

3.7.3 "Free-format" Input

It is possible to cause READ and READLINE to behave as though the input consisted entirely of literal atoms. Use of this feature may be desirable, for example, if it is necessary to enter streams of atoms which contain LISP punctuation marks in their print names. To get this effect, the value 255 should be POKEd into page zero location 253 ($FD hex). After doing this, all subsequent READs will be "free-format" READs. Note that you should only do this from within a function and NOT from the keyboard; EVERYTHING you type in after the POKE will be treated as though it were a literal atom. The interpreter can be restored to doing normal READs by POKEing a value of 0 into location 253. The interpeter is also restored by a RESET or an error.

Ex: Suppose you define the following function:

```
:(define (freadline (lambda ()
:  (prog (r)
:    (poke 253 255)
:    (setq r (readline))
:    (poke 253 0)
:    (return r)
:  )
:)))
```

Now, if you invoke this function and then type

```
?i don't (know) .
```

the function will return the list

```
(I DON'T (KNOW) .)
```

This list consists of four atoms: the atom I, the atom DON'T, the atom (KNOW), and the atom '.'. Be aware that although (KNOW) LOOKS like a list, it is actually an atom whose print name consists of the characters (, K, N, O, W, and), this being a result of the free-format READ.

3.7.4 (PRIN1 <s>)

PRIN1 prints the s-expr <s> on the current output device and returns <s>. PRIN1 begins printing at the current cursor position and does NOT execute a carriage return after printing. If <s> is omitted, PRIN1 executes a single carriage return and returns NIL. Note that string atoms are printed with the delimiting quotes. See the description of QPRIN1 below to avoid this.

Ex:

```
:(progn (prinl 'this) (prinl "that"))
 THIS "THAT"
```

3.7.5 (PRINT <s>)

PRINT works like PRIN1 except that PRINT DOES execute a carriage return after printing the argument <s>. If <s> is omitted, PRINT executes TWO carriage returns and returns NIL.

3.7.6 (QPRIN1 <s>)

QPRIN1 works like PRIN1 with the exception that string atoms are printed WITHOUT their delimiting quotes. All other s-exprs are treated the same way as for PRIN1.

Ex:

```
:(progn (qprinl "lisp is fun") (qprinl 24))
 LISP IS FUN 24
```

3.7.7 (QPRINT <s>)

QPRINT is identical to PRINT with the exception that string atoms are printed without their delimiting quotes.

3.7.8 (GETCHR)

GETCHR is similar to the Applesoft GET command. GETCHR waits for a keypress and returns an atom that corresponds to the pressed key. If the key is a digit, a numeric atom is returned; otherwise, a string atom is returned.

3.8 APPLE FUNCTIONS

3.8.1 (CALL <na>)
(PEEK <na>)
(POKE <nal> <na2>)

These function are completely analogous to their Basic counterparts. All arguments must be numeric atoms. CALL returns the location being CALLed after returning from the call; PEEK returns the contents of the PEEKed location; POKE returns the value being POKEd.

It is not advisable to POKE anywhere between locations 2048 decimal (800 hex) and HIMEM, or anywhere on page zero except where specified in the manual.

3.8.2 (HTAB <na>)

HTAB' causes a horizontal tab to column <na>. <na> is returned.

3.8.3 (VTAB <na>)

VTAB causes a vertical tab to row <na>. <na> is returned.

3.8.4 Graphics

P-LISP supports both hi-res and lo-res graphics. Please see chapter 4 for full definitions of these functions.

3.8.5 (ONERR)
(ONERR <s>)

ONERR is analogous to the ONERR statement in Applesoft. (ONERR <s>) sets the error flag, so that if an error occurs during the evaluation, the s-expr <s> will be evaluated and its value returned. ONERR with no arguments resets the error flag so that normal error messages will be printed. If ONERR is active, the error code is stored in location 222 before <s> is evaluated. ONERR may also be used to trap DOS errors, provided the error occurs during an evaluation. See chapter on error messages for the correct codes.

Ex:

```
:(onerr '(that is wrong))
 T
:(add 3)
  (THAT IS WRONG)
:(onerr)
 NIL
:(add 3)
  **ERROR: TOO FEW ARGS**
```

3.9 FUNCTION DEFINITION, FLOW OF CONTROL

3.9.1 (DEFINE (<fcn-name> <fcn-def>))

DEFINE is used to define user functions. <fcn-name> must be a literal atom; <fcn-def> should be anything LISP can evaluate (usually a LAMBDA-expression). Neither <fcn-name> nor <fcn-def> are evaluated before DEFINE is applied. DEFINE puts the property EXPR and the value <fcn-def> on the atom <fcn-name>'s property list. The value returned by DEFINE is the function name.

Functions can also be defined by PUTting the property EXPR and the function definition on the function name's property list.

Ex: To define the function CADR, which takes the CAR of the CDR of its argument, then:

```
:(define (cadr (lambda  (x)
:   (car (cdr x)))))
 CADR

:(cadr '(apple sauce))
 SAUCE
```

See below for details on Lambda-expressions.

3.9.2 (LAMBDA <L> <s1> <s2> ...<sn>)

User-defined LISP functions are typically LAMBDA-expressions. <L> is a list of the formal parameter to the function (possibly NIL). Each formal parameter must be a literal atom and cannot be NIL. The formal parameters correspond to the arguments that are to be supplied to the function. Upon entrance into a LAMBDA-expression, LISP SETQs the formal parameters to the values of the arguments, evaluates the s-exprs <s1> through <sn>, and resets the formal parameters back to their previous values, returning the value of the last evaluated <si> as the value of the function.

Ex: When we enter (CADR '(APPLE SAUCE)) for the example above, the following sequence occurs:

1) X is SETQed to (APPLE SAUCE);
2) (CAR (CDR X)) is evaluated;
3) X is un-SETQed back to its previous value;
4) The atom SAUCE is returned as the value of the function.

Note that LAMBDA-expressions can be applied directly without first putting them in a function definition, i.e., we can type

```
:((lambda (x)  (car (cdr x))) '(apple sauce))
 SAUCE
```

with the same effect. Note the syntax for the above expression: the LAMBDA-expression and the arguments must be the CAR and CDR respectively of the s-expr. Of course, it is more convenient to define functions in terms of LAMBDA expressions than to use this method.

3.9.3 (FLAMBDA <L> <s1> <s2> ...<sn>)

FLAMBDA is used for functions that should not evaluate their arguments. The formal parameter list <L> must consist of a single literal atom other than NIL. Upon entering an FLAMBDA, LISP puts the CDR of the s-expr that invoked the FLAMBDA in an UNEVALUATED argument list and SETQs the formal parameter in <L> to that list. Like LAMBDA, the <si>'s are evaluated in succession, the formal parameter is un-SETQed, and the value of the last evaluated <si> is returned.

Ex: If we don't want CADR in the example above to evaluate its argument, we can define it as follows:

```
:(define (cadr (flambda (x)
    (car (cdr x)))))
 CADR
```

To apply CADR, we can type

```
:(cadr apple sauce)
 SAUCE
```

Note that the following sequence occurred:

1) X is SETQed to the unevaluated argument list, which is the CDR of the s-expr that invoked CADR, in this case (APPLE SAUCE);
2) (CAR (CDR X)) is evaluated;
3) X is un-SETQed to its previous value;
4) The value SAUCE is returned.

Like LAMBDA, FLAMBDA can also be applied directly:

```
:((flambda (x) (car (cdr x))) apple sauce)
  SAUCE
```

FLAMBDA is also useful for functions that take an indeterminate number of arguments. For example, suppose you wanted a function that added up an arbitrary list of numbers. First, define a function that adds up a list of numbers:

```
:(define (addlist (lambda (x)
:   (cond
:      ((null x) 0)
:      (t (add (car x) (addlist (cdr x))))
:   )
:)))
```

You can now define a function that hands to ADDLIST a list of numbers of arbitrary length.

```
:(define (add-em-up (flambda (x)
:   (addlist x)
:)))
```

Now, if you type in the s-expr

```
:(add-em-up 5 4 3 2 1)
```

you will get the result 15.

3.9.4 (COND <L1> <L2> . . . <Ln>)

COND evaluates the CAR of each <Li> until a non-NIL value is reached; COND then evaluates the remaining s-exprs in the <Li>, returning the value of the last s-expr as the value of the COND. If the CDR of the <Li> is NIL, COND returns the value of the CAR. If the CAR of each <Li> evaluates to NIL, COND returns NIL. The elements of each <Li> may be any s-expr that LISP can evaluate; it is perfectly valid to embed a COND within a COND, a PROG within a COND, etc.

Ex:

```
:(cond ((and (null L) (atom z))
:            (htab 5)(print 'yes))
:       (t (print 'no)))
```

This will move to column 5 and print YES if the value of L is NIL and the value of Z is an atom. Otherwise, NO will be printed and returned.

If we envision the form of the COND as

```
(COND (<c1> <e1>)
      (<c2> <e2>)
       .    .
      (<cn> <en>))
```

then the evaluation of COND can be thought of as

```
    IF <c1> THEN <e1>
ELSE IF <c2> THEN <e2>
  .      .      .
ELSE IF <cn> THEN <en>
ELSE NIL
```

3.9.5 (PROG <L> <s1> <s2> . . . <sn>)

PROG is used primarily for iterative programming. <L> is a list of literal atoms (possibly NIL) which are used as local atoms to the PROG. NIL cannot be used as a local atom. These atoms are initially SETQed to NIL. The <si>'s are evaluated in succession, with the value of the last evaluated <si> returned as the value of the PROG. Any <si> that is an atom is considered a label and is NOT evaluated. The local atoms are un-SETQed to their previous values after exiting the PROG. Numeric and string atoms should not be used as labels in a PROG, since GO can only branch to literal atoms.

3.9.6 (RETURN <s>)

RETURN is used to exit a PROG. RETURN evaluates its argument and returns the value as the value of the PROG. Evaluation proceeds with the first s-expr following the s-expr that invoked the PROG. The RETURN may be used from anywhere within the PROG, e.g., it may be embedded in a COND (See example below).

3.9.7 (GO <a>)

GO is used in a PROG as a GOTO. <a> must be a literal atom and is not evaluated. When a GO is encountered in a PROG, evaluation will proceed at the first s-expr following the first occurrence of the label in the PROG. If the label is not in the immediately dominating PROG, an error will result (it is not possible to branch out of a PROG). Like RETURN, GO may be used from anywhere within the PROG.

Ex: This example uses the above functions to illustrate their use. Assume that LISP was not supplied with a multiplication function. We could write one using the PROG construct as follows:

```
:(define (mult (lambda (x y)
:   (prog (z result)
:         (setq z y result 0)
:         loop
:         (cond ((zero z) (return result))
:               (t (setq result (add result x)
:                        z (sub z 1))))
```

```
:       (go loop)))))
  MULT

:(mult 4 5)
  20
```

In the above example, invoking MULT SETQs X to 4 and Y to 5. Entering the PROG SETQs local atoms Z and RESULT to NIL. However, the next s-expr SETQs Z to 5 and RESULT to 0. Evaluation then loops, adding Z to RESULT and decrementing Z by 1. When Z is 0, the PROG is exited and the value of RESULT, 20, is returned.

GO and RETURN can also be used to re-start evaluation that has been suspended (for example, by a ctrl-C), provided that the suspension occurred from within a PROG. For example, suppose you typed ctrl-C while the interpreter was evaluating (MULT 500 500), which would take a while (using the MULT function described above). The interpreter would then enter break mode, with the input prompt a '+'. If you then typed

```
+(go loop)
```

the interpreter would re-enter the MULT function and continue evaluating. Note that this can be dangerous: for example, if you hit ctrl-C in the middle of the SETQ in the MULT loop above, after RESULT was incremented but before Z was decremented, then re-entering the function will produce an incorrect result.

You could also have typed something like

```
+(return 12)
```

after entering break mode; in this case, the PROG will be exited returning 12. Using RETURN from break mode can be useful while debugging, for example, if you want to force a function to return a specific value to another function.

3.9.8 (EVAL <s>)

EVAL evaluates the value of its argument and returns the result.

Ex: If the value of A is B, then

```
:(eval 'a)
 B
```

If the value of X is (A B), then

```
:(eval (cons 'car '(x)))
 A
```

The argument is evaluated, CAR is CONSed onto (X), and the resulting value, (CAR X) is evaluated by EVAL, returning A.

3.9.9 (APPLY <a> <L>)

APPLY applies the function <a> to the list of arguments <L> and returns the result. Note that <a> treats the elements of <L> as though they are quoted.

Ex:

```
: (apply 'cons '(a (b)))
  (A B)

: (apply 'mult '(3 34))
  102

:'(apply 'conc '((a) (one) (anda) (two)))
  (A ONE ANDA TWO)
```

3.9.10 (MAPCAR <a> <L>) (MAPCAR <a> <L1> <L2>)

MAPCAR applies the function <a> to each element of list <L> (or each pair of arguments from the lists <L1> and <L2>) and returns a list of the results after the function has been applied to all the list elements. MAPCAR should only be used with functions that take one or two arguments. Note that MAPCAR behaves like APPLY in that the arguments to <a> are treated as though they are quoted.

Ex:

```
:(mapcar 'length '(nil (a) (b c) (d e f)))
 (0 1 2 3)

:(mapcar 'cons '(a b) '((c) (d)))
 ((A C) (B D))

:(mapcar 'atom '(3 (a) b 14 (d c) "qwerty"))
 (T NIL T T NIL T)
```

3.9.11 (PROGN <s1> <s2> . . . <sn>)

PROGN evaluates each <si> in succession and returns the value of the last evaluated <si>.

Ex:

```
:(progn (setq z '(hello)) (print 'done))
 DONE
```

and Z is set to (HELLO).

3.10 OBJECT LIST FUNCTIONS

3.10.1 (OBLIST)

OBLIST returns the current object list, i.e., the list of all the currently active atoms.

3.10.2 (REMOB <s>)

REMOB removes atoms from the object list. Typically, REMOB is necessary to free space that would be wasted otherwise by useless atoms. <s> must evaluate to the literal atom to be REMOBed. WARNING: if you REMOB an atom that is referenced by other s-exprs, or if you REMOB an important atom like T or NIL, the results will be disastrous. USE THIS FUNCTION WITH CARE!

Ex:

```
:(remob (car '(peach pie)))
```

will remove the atom PEACH from the object list. The space occupied by PEACH will now be reused after the next garbage collection.

3.11 DEBUGGING AIDS

3.11.1 (TRACE <a1> <a2> . . . <an>)

TRACE is used to turn trace on for particular functions. The <ai> are NOT evaluated and should be the names of the functions to be traced (both SUBRs and EXPRs can be traced). Attempting to trace a non-existent function has no effect. A maximum of 10 functions can be traced at any one time; attempting to trace above this limit has no effect and does not produce an error message. TRACE with no arguments also has no effect.

TRACE is normally used for debugging purposes. When a function is traced, the interpreter will print the list of (evaluated) arguments to the functions when it is invoked, and the value returned by the function after evaluation. TRACE returns T.

Ex: To turn trace on for CONS,

```
:(trace cons)
 T

:(cons 'a '(b))

 ->>CONS :: (A (B))
 <<-CONS :: (A B)

 (A B)
```

The ->> arrow indicates the function is being entered, with the list of (evaluated) arguments following the ::. The <<- arrow indicates the function is being exited, with the s-expr following the :: being the value returned.

Higher-level functions are always traced before lower-level functions. This means that if, for example, we are tracing both CAR and CONS, then

```
:(car  (cons 'a '(b)))

->>CAR :: ((A B))
->>CONS :: (A (B))
<<-CONS :: (A B)
<<-CAR :: A
```

CAR is traced before CONS despite the fact that CONS is evaluated first. In fact, the CONS s-expr is evaluated twice: once for the trace of CAR, and then again for the trace of CONS.

NOTE: Never trace any function unless it evaluates its arguments. In other words never trace functions such as GO, SETQ, TRACE, COND and PROG.

3.11.2 (UNTRACE <a1> <a2> . . . <an>)

UNTRACE turns trace off for the functions supplied in the argument list. UNTRACE with no arguments turns trace off for all traced functions. Untracing a function that is not being traced has no effect. Like TRACE, the <ai> are not evaluated. UNTRACE returns NIL.

Ex: To turn trace off for CONS,

```
:(untrace cons)
 NIL
```

3.11.3 (BREAK <S>)

BREAK is used for setting breakpoints in a function. The <s> is an optional (unevaluated) s-expr that is used to label the breakpoint. When a BREAK is encountered, a break message and the <s> if one is supplied, is printed, and the interpreter enters break mode. The user may then examine local atom bindings or execute any other LISP command, as in normal break mode. Execution of the BREAKed function may be resumed at the point after the BREAK by typing GO.

Ex: Suppose we have the following function:

```
(TEST (LAMBDA (X)
   (PROG (Z)
      (PRINT '(A TEST FOR BREAK))
      (BREAK HERE)
      (PRINT '(THAT WAS FUN))
   )
)
```

We type:

```
:(test 'thisatom)
```

and P-Lisp responds

```
(A TEST FOR BREAK)
>>BRK: HERE
+
```

We can now examine atom bindings, for example,

```
+x
  THISATOM
+z
  NIL
```

To resume execution, we type,

```
+go
(THAT WAS FUN)
NIL
```

The atom BREAK is used as a switch to enable or disable breakpoints. By default, BREAK is SETQed to T, which enables all breakpoints. To disable breakpoints, SETQ the atom BREAK to NIL; all breakpoints will then be ignored. SETQing BREAK to T (or any other non-NIL value) will re-enable the breakpoints.

Remember that ctrl-C is NOT the same as a BREAK, even though both put you in break mode. Specifically, you cannot resume evaluation after a ctrl-C by typing GO; you can only do so after a BREAK.

3.12 GARBAGE COLLECTION

3.12.1 (GC <s>)

This form of the GC function is used to enable or disable the garbage collection message that appears whenever the GC is invoked. If <s> is NIL, the message is disabled; otherwise, the message is enabled. This command has no other effect.

3.12.2 (GC)

GC with no arguments invokes the Garbage Collector. This is a convenient way of determining how much free memory is available. GC will return the number of cells collected during the garbage collection.

3.13 LOADING AND SAVING WORKSPACES

P-LISP workspaces are saved to and loaded from disk via the DOS commands SAVE and LOAD. The syntax for these commands is the same as that described in the DOS Reference Manual. Be aware that P-LISP makes patches to DOS to re-define the SAVE and LOAD commands; thus, you should reboot DOS after exiting P-LISP.

For example, to save a workspace on the disk in drive 2 with the name TEST1, simply type

```
:save test1,d2
```

The workspace will be saved and T will be returned. To reload the workspace, type

```
:load test1,d2
```

The workspace TEST1 will then be reloaded. As you would expect, LOADing a workspace overwrites the contents of the current workspace.

Workspaces are saved on disk with the special filetype S. The file containing the workspace actually consists of a "core dump" of the workspace in RAM. Only the memory pages marked as "free" in the workspace memory map (see Chapter 7) are saved in the file. A copy of the memory map is also saved in the file and reloaded when the file is LOADed.

Note that if, for some reason, you have modified the property lists of the P-LISP SUBRS, you will have to save a separate copy of the interpreter as well. You will then have to use this interpreter in conjunction with the workspace. Normally there is no reason to ever touch the p-lists of the SUBRs, so you should be able to ignore this note for almost all cases.

Functions can also be saved in text file format. WRITEFCN and APPENDFCN, in conjunction with the pretty-printer, can be used to write function definitions to a text file (see Chapter 4). Function definitions (or, in fact, any s-exprs) can be loaded from a text file by simply EXECing the file; the contents of the file are added to your current workspace.

It is generally a good idea to save your workspace often when you are developing functions. Because functions and data share the same environment, a simple mistake can be especially volatile. Any number of errors can destroy your workspace, probably the worst being a RECURSION CHECK during a garbage collection. SAVE does not take particularly long and is well worth the wait.

(If you feel like you're in a destructive mood, here's an easy way to wipe out your workspace: simply type

```
:(rplaca L L)
```

for some list L and then invoke the garbage collector.)

Chapter 4 - Graphics, File I/O, and Mathematical Functions

4.1 LORES Graphics functions (see Apple BASIC manuals for details on these functions)

4.1.1 (GR)

GR takes no arguments. GR puts the Apple in LORES graphics mode (with four lines of text at the bottom) and returns NIL.

4.1.2 (TEXT)

TEXT returns the Apple from graphics to text mode. TEXT returns T.

4.1.3 (COLOR <na>)

COLOR sets the color for following PLOTs. <na> must be a numeric atom between 0 and 15 inclusive. COLOR returns <na>.

```
Ex:
    :(color 15)
     15
```

sets all succeeding plots to color 15 (white).

4.1.4 (PLOT <na1> <na2>)

PLOT plots a small square on the screen at location row <na2>, column <na1>. The color of the square will be whatever was set by the most recent COLOR command. The arguments must be numeric atoms between 0 and 39 inclusive. PLOT returns T. Note that PLOT does NOT check to make sure that the arguments are valid square coordinates.

```
Ex:
    :(plot 4 7)
     T
```

plots a square at row 7, column 4.

4.2 HIRES Graphics

You should be familiar with hires graphics as explained in the Applesoft reference manual before using these functions. You must protect the hires display in the memory map before using these functions. The workspace HILBERT on the P-LISP Master disk contains a memory map configured for hires graphics (it does NOT make use of language card RAM).

4.2.1 (HGR2)

HGR2 puts the Apple in hires graphics mode (display page 2).

4.2.2 (HCOLOR <na>)

HCOLOR sets the plotting color to <na>. <na> must be between 0 and 7 inclusive.

4.2.3 (HPLOT <na1> <na2>)

HPLOT plots a hires point at column <na1>, row <na2>. P-LISP does not check to make sure these coordinates are in bounds.

4.2.4 (HTO <na1> <na2>)

HTO draws a line from the most recently plotted point to <na1>, <na2>. The color used is the most recently plotted color, regardless of the setting of HCOLOR.

4.2.5 (DRAW <na>)

DRAW draws the <na>th shape in the shape table, starting at the most recently plotted point. The address of the shape table should be in page 0 locations 232 and 233 ($E8-$E9 hex). The shape table should reside in a protected area of memory. The default scale and rotation factors are 1 and 0 respectively. The scale factor may be changed by POKEing the desired value into location 231 ($E7 hex). The rotation factor may be changed by POKEing the desired value into location 249 ($F9 hex).

4.2.6 (XDRAW <na>)

XDRAW is the same as DRAW, except that the color of each dot drawn is complemented.

4.3 File I/O Functions

The file FILEIO.TEXT on the P-LISP Master disk contains the functions described below. The functions can be loaded by EXECing the file into your workspace.

Note that if you wish to use the functions WRITEFCN or APPENDFCN to write function definitions to a text file, you should also have the pretty-printer loaded. The pretty-printer is stored in the file PPRINTER.TEXT on the Master disk.

In the function descriptions below, atoms representing file names can be either literal or string atoms.

4.3.1 (OPENSEQ <a>)

This function opens a sequential text file with filename <a>.

Ex:
```
:(openseq "tmp1,d2")
```

opens text file TMP1 on the disk in drive 2.

4.3.2 (APPENDSEQ <a>)

This function works the same as OPENSEQ, except that subsequent output is appended to the end of the file.

4.3.3 (WRITESEQ <a>)

This function sends a WRITE command to DOS for sequential file <a>. Subsequent LISP PRINT statements will send output to file <a>.

4.3.4 (READSEQ <a>)

This function sends a READ command to DOS. A subsequent LISP READ statement will get the next line of input from file <a>.

4.3.5 (CLOSEFILE <a>)

This function closes file <a>.

4.3.6 (CLOSE)

This function close all open files.

4.3.7 (OPENRND <a> <na>)

This function opens a random access file with filename <a>. <na> is the record length of the file.

Ex:
```
:(openrnd 'sesame 20)
```

opens random access file SESAME with record length 20.

4.3.8 (WRITERND <a> <na>)

This function sends a random access WRITE command to DOS. <na> is the record number to be written to. Subsequent LISP PRINT statements will send output to record <na> of file <a>.

4.3.9 (READRND <a> <na>)

This function sends a random access READ command to DOS. <na> is the record number to be read from. Subsequent LISP READ statements will read from record <na> of file <a>.

The following example will illustrate how to use some of the above functions.

```
:(progn
:   (openrnd "meow" 30)
:   (writernd "meow" 14)
:   (qprint "this is a test")
:   (close "meow")
:)
```

The above s-expr opens random file MEOW with record length 30, then writes the string "THIS IS A TEST" (without quotes) to record number 14. The following s-expr,

```
:(progn
:  (openrnd "meow" 30)
:  (readrnd "meow" 14)
:  (setq x (readline))
:  (close)
:)
```

will open file MEOW, and read record 14, SETQing the atom X to the list (THIS IS A TEST).

4.3.10 (WRITEFCN <a> <s>)

This function may be used to store function definitions in a text file. YOU MUST HAVE THE PRETTY PRINTER IN YOUR WORKSPACE TO USE THIS FUNCTION! <a> must be the name of the file in which the functions will be stored. <s> may be either the name of a single function or a list of functions to be saved (see examples). For each function in the list, WRITEFCN writes the function definition to the text file. The functions may be reloaded from the file by EXECing the file.

Ex:

```
:(writefcn 'work 'apple)
```

writes the function APPLE to the file WORK.

```
:(writefcn 'work '(peach plum pear))
```

writes the three functions PEACH, PLUM, and PEAR to the file WORK.

4.3.11 (APPENDFCN <a> <s>)

This function works the same way as WRITEFCN except the functions are appended to the end of the file.

4.4 MATHEMATICAL FUNCTIONS

Supplied on the P-LISP master disk are the files which provide P-LISP the ability to access Applesoft's intrinsic math functions. The functions are listed below, with the appropriate syntax; you should consult the Applesoft reference manual for more details.

(SIN <na>) - sine
(COS <na>) - cosine
(TAN <na>) - tangent
(ATN <na>) - arctangent
(LOG <na>) - natural logarithm
(EXP <na>) - exponential
(RND <na>) - pseudo-random number
(SQRT <na>) - square root

To utilize the above functions, you should do the following:

1) Deallocate page $6000 (bit 7 of byte $81F in the memory map). See Chapter 6 for details on memory allocation.

2) Type 800G to restart P-LISP (remember that you must restart the interpreter after modifying the memory map).

3) From P-LISP, type BLOAD TRIG.CODE to load the machine language code for the functions.

4) Type EXEC TRIG.TEXT to load the LISP code for the functions.

5) The math functions are now available. Note that if you SAVE your workspace, the machine code for the functions is NOT saved as part of the workspace.

Chapter 5 - Error Messages

A LISP error occurs whenever the interpreter encounters a s-expr which, for some reason, it cannot evaluate. This will typically result from giving a function the wrong number of arguments or the wrong types of arguments. When an error occurs, the interpreter will print an error message describing the error, followed by the name of the function being evaluated and the contents of the argument list at the time of the evaluation (this assumes that ONERR is not active). For example, attempting a CONS with too many arguments will cause the following:

```
:(cons 'this '(that) '(those))
**ERROR: TOO MANY ARGS **
CONS :: ((QUOTE THIS) (QUOTE (THAT)) (QUOTE (THOSE)))
```

Attempting an ADD with non-numeric arguments will give the following error:

```
:(add 3 nil)
** ERROR: BAD NUMERIC ARG **
ADD :: (3 NIL)
```

A summary of the P-LISP error messages appears below.

When an error occurs, the interpreter enters a special evaluation mode called "break" mode. (The interpeter will also enter this mode as the result of a ctrl-C, a BREAK, or a RESET). In this mode, it is possible to examine the current local environment (if any). If break mode was entered as the result of a BREAK, then typing GO will resume evaluation (GO will NOT be effective if break mode was entered as the result of an error).

The input prompt in break mode is the plus sign '+'. To return back to normal LISP mode, simply type () or NIL. The normal input prompt ':' will reappear, and the environment stack will be cleared.

For example, if you have defined the following function:

```
:(define (a (lambda (x)
:    (cons x))))
```

and then type

```
:(a 19.7)
```

you will get

```
** ERROR: TOO FEW ARGUMENTS **
CONS :: (X)
```

You can then examine the current binding for X:

```
+x
 19.7
```

If you now type

```
+()
 NIL
```

you get the input prompt ':'. The environment stack has been cleared, so if you now type

```
:x
```

you will get

```
** ERROR: UNDEFINED ATOM **
EVAL :: X
```

5.1 NUMERIC OVERFLOW (ONERR code 17)

A numeric atom greater than the maximum number the computer could handle was entered from the keyboard or calulated in an evaluation.

5.2 TOO FEW ARGS (ONERR code 22)

A function was invoked and not enough arguments were supplied.

5.3 TOO MANY ARGS (ONERR code 21)

A function was invoked and too many arguments were supplied.

5.4 BAD ATOMIC ARG (ONERR code 20)

A function was expecting an atom, and something else was supplied. This error will also occur when a function is expecting a literal atom and a numeric atom is supplied (e.g., SETQ).

5.5 BAD LIST ARG (ONERR code 24)

A function was expecting a list, and something else was supplied. This error will also occur if a list is supplied to a function with illegal elements, for example, trying to define a function with a numeric atom as the function name.

5.6 BAD NUMERIC ARG (ONERR code 23)

A function was expecting a numeric atom, and something else was supplied. This error will also occur if a function is expecting an integer (e.g. CALL) and a floating-point atom is supplied.

5.7 RECURSION CHECK (ONERR code 18)

This error will occur if you have recursive functions that overflow the recursion stack, or if a s-expr with a very deep nesting level is encountered during garbage collection, or if garbage collection occurs during deep recursion. If this error occurs, you may have to adjust the amount of space allocated to the recursion stack (see Chapter 7). Be aware that, if this error occurs in the middle of a garbage collection, the contents of your workspace will in all likelihood be lost.

5.8 NO DOMINATING PROG (ONERR code 25)

A GO or RETURN was attempted, and there was no dominating PROG to return from or branch in.

5.9 MISSING LABEL (ONERR code 26)

A GO was attempted to a non-existent label.

5.10 UNDEFINED ATOM (ONERR code 19)

An attempt was made to evaluate an atom that has no value. This will also occur if an attempt is made to evaluate a non-existent function.

5.11 NO SPACE (ONERR code 27)

The Garbage Collector was unable to collect any free space. The problem may alleviate itself after this error occurs, depending on what was going on at the time. Otherwise, the object list may be too crowded, or there might be extremely long lists floating around. REMOB may be able to remedy this situation.

CHAPTER 6 -- FUNCTION EDITOR & PRETTY PRINTER

The P-LISP interpreter is supplied with a function editor and pretty-printer, both written in LISP. The editor provides an easy way to modify function definitions without having to retype them entirely and without having to leave the LISP environment. The pretty-printer prints the function definitions on the output device in an easy-to-read format.

The editor is supplied in text file format in the file ED.TEXT. The pretty-printer resides in the text file PPRINTER.TEXT. You can load the editor and pretty-printer into your workspace by EXECing them in. (Note: you do not need to have the pretty-printer loaded in order to use the editor. However, if you wish to use them both, you must EXEC the editor BEFORE EXECing the pretty-printer).

Both the editor and pretty-printer are also supplied in a workspace format (file type S) file named EDITOR. To load this workspace, simply type LOAD EDITOR. Be aware that this workspace is configured with the default memory map. If you are creating a new workspace and wish to use the default map, you should LOAD this file before you begin; that way you will always have the editor available while you develop your functions (you can REMOB the editor functions from your workspace when you are finished to make extra memory available). If you wish to create a workspace with a different memory map, and wish to have the editor present, you must modify the map (see Chapter 7) and then EXEC the editor files into your new workspace.

To edit a function, type (ED <fcn>), where <fcn> is the name of the function to be edited. The editor responds by typing:

```
TOP: (LAMBDA & &)
```

The (LAMBDA & &) is called your Point-of-View (POV), and the TOP: is your level indicator. There are any number of commands that you can give that will allow movement through your function to edit it. You will note that your POV is displayed in a compressed format. This is to allow rapid and easy manipulation of long and complicated lists. If you view the function as a tree, the Lambda and the two &'s are nodes. Any node which is represented by an & is a list, while atoms are explicitly spelled out (as in LAMBDA) If you wish to take a look at a particular node, you may move to it, by saying 3 (to move to node three). The editor will respond with a new point of view which might look something like:

```
TOP: 3: (COND & & &)
```

This would tell you that you were looking at a COND statement, with 3 items below it. If you wish to see a full printout of the current POV, type P (for pretty-print), and the entire expanded POV is printed out.

The table below summarizes all the possible commands to the editor: (# means any number)

#	Move to son # (i.e. if # is 3, move to the third element in the POV.
-#	Work from the reverse. If you type a -2, and the length of the POV is 7, this will move you to the 5th position of the POV.
0	Cancel the last level you are working on and move to the father of the tree.
NX	Moves you to the next level. In other words, if your POV is (3 2 2 1), this will move you to (3 2 2 2) if possible. If you would end up on an atom, it will skip it. If you would fall off the edge, it will not allow you to go.
BK	Analagous to NX, but moves you backwards. If you are at 3 2 2 2, you would go to 3 2 2 1 if possible.
GO	(new pov) This command lets you set a new POV. The new POV must be legimitate (it must point to a list) and should be supplied as a list of numeric atoms. If a null list is supplied, e.g. GO (), the POV will be set to the TOP of the function.
EXIT	Normal exit, puts modified function into your workspace.
ABORT	Aborts this edit, does not save function.
I	<sexpr> F/A/B # This is the insert/replace command. Any valid <sexpr> can be inserted anywhere in the function. If you use the F option, this stands for FOR, and means replace whatever was at # with <sexpr>. If you use the A option, this means to insert <sexpr> AFTER whatever is at #. If you use the B option, insert <sexpr> BEFORE whatever is at #. The example below should clarify these options.
D #	Deletes whatever was at #.
P	Pretty-print the entire POV. This requires that the pretty-printer be resident; otherwise, the POV will be printed in normal list form.

DP # Delete the pair of parentheses around the s-expr at #. The command has no effect if the s-expr is an atom.

IP #1 #2 This command inserts a pair of parentheses around the #1th and #2th s-expr in the POV.

Working through an example is the best way to see how the editor works. Suppose we have the following definition for REVERSE:

```
(REVERSE (LAMBDA ((L))
  (CIND
    ((NULL L) A NIL)
    ((APPEND (REVERSE CDR L) (CAR L)))
  )
)
```

We want to make the following changes:

-- Delete the extra pair of parens around the formal argument L.
-- Change the CIND to a COND.
-- Delete the A before the NIL.
-- Insert a T before the APPEND s-expr.
-- Insert a pair of parentheses around the CDR L.

Here's how we'd make the changes. Comments in the dialogue below are preceded by a semicolon. User input is typed in lower case, and the computer response is in UPPER case. First we start by invoking the editor:

```
:(ed reverse)
;the editor responds with a top-level POV
TOP: (LAMBDA & &)

?p
;Let's first pretty-print the function.
;Note that it won't be particularly pretty because of
;the mess the function is in.
(LAMBDA ((L))
  (CIND ((NULL L) A NIL) ((APPEND (
   REVERSE CDR L)(CAR L))))
)

TOP: (LAMBDA & &)
```

```
;first we want to get rid of the extra parens around the L.

?2
TOP:2: (&)

;let's print this to make sure it's what we want.

?p
((L))

TOP:2: (&)

;ok -- delete the parens

?dp 1
TOP:2: (L)

;we want to edit the next s-expr. NX will take us there.

?nx
TOP:3: (CIND & &)

;first change the CIND to a COND. INSERT FOR will do this.

?i cond f 1
TOP:3: (COND & &)

;now to get rid of the A.

?2
TOP:3:2 (& A NIL)

;DELETE will do the job for us.

?d 2
TOP:3:2 (& NIL)

;lets print it to make sure it's what we want.

?p
((NULL L) NIL)

TOP:3:2 (& NIL)

;let's do a NX to get to the next s-expr.

?nx
TOP:3:3 (&)

;(note we could also have gotten here by typing 0 to get to the
;COND s-expr and then a 3).
;This doesn't tell much, so let's print it.
```

```
?p
((APPEND (REVERSE CDR L) (CAR L)))

TOP:3:3  (&)
;first we want to make T the first s-expr of this list. we can do
;that with an INSERT BEFORE.

?i t b 1
TOP:3:3 (T &)

;the last thing to do is surround the CDR L with a pair of
;parens. We have to go there first.

?2
TOP:3:3:2 (APPEND & &)

?2
TOP:3:3:2:2 (REVERSE CDR L)

;let's use the INSERT PARENS (IP) command.

?ip 2 3
TOP:3:3:2:2 (REVERSE &)

;and make sure it's right

?p
(REVERSE (CDR L))

TOP:3:3:2:2 (REVERSE &)

;let's go back to the top and pretty-print the function to see
;the results of our efforts.

?go ()
TOP: (LAMBDA & &)

?p
(LAMBDA (L)
  (COND
    ((NULL L)
      NIL
    )
    (T
     (APPEND (REVERSE (CDR L)) (CAR
      L))
    )
  )
)

TOP: (LAMBDA & &)
```

```
;looks good. Save the function by typing EXIT.

?exit

(LAMBDA (L) (COND ((NULL L) NIL) (T (
APPEND (REVERSE (CDR L)) (CAR L)))))
```

We can now try it out.

```
:(reverse '(a b c d e))
 (E D C B A)
```

A job well done.

THE PRETTY-PRINTER

You can invoke the pretty-printer from outside the editor by simply typing (PPRINT <fcn>), where <fcn> is the name of the function to be printed. Remember that, by default, P-LISP formats output for a 40-column display. To change this, you should POKE the desired column width into page zero location 240 ($F0 hex). If you are using a printer, you must do this in addition to whatever control sequence is necessary to get the printer to print more than 40 columns.

CHAPTER 7 -- MEMORY MANAGEMENT

The memory allocation scheme implemented in P-Lisp allows the user to select the size and location of the recursion stack, to protect certain areas of memory from P-Lisp (e.g. the Hires display area), and to take advantage of extra RAM provided on the language card. The following instructions explain how to configure the interpreter's memory to suit your needs. Follow the instructions carefully, and be sure to double-check everything before saving the map on disk. To be safe, you should always work with a copy of the interpreter, never the master.

Note that the memory map is an integral part of a workspace. Once a workspace has a given memory map, the map cannot be changed for that workspace. The map is saved as part of the workspace on disk; once the map is modified, you must tell the interpreter about it by restarting P-LISP, which always results in clearing the workspace.

If you wish to create a workspace with a memory map different from the default map, you should follow the following procedure:

1) Enter the monitor via CALL -151. You should have a copy of the interpreter loaded (if you are already in LISP, you do not have to reload the interpreter; just type (CALL -151)).

2) Make the appropriate changes to the map.

3) Type 800G to restart the interpreter. You MUST do this in order to tell the interpreter about the changes you just made.

4) You can now create your new functions, EXEC them from a text file, or whatever. When you SAVE your workspace, a copy of the map is saved with it, so when you reload the workspace, everything will be as you left it when you did the SAVE.

If you wish, you can modify the map and make the new map a permanent part of the interpreter, so that every time you BRUN LISP you get the new map. To do this, simply BLOAD the interpreter, enter the monitor to make your changes, then type BSAVE LISP, A$800, L$3300 to save a new copy of the interpreter. To be safe, be sure you do not overwrite your only copy of the interpreter: ALWAYS USE A BACKUP COPY.

The default memory map is as follows:

Recursion stack: $3B00-3FFF.
Workspace: $4000-95FF.

The rest of memory is protected. If you have an Apple IIE, or an Apple II+ with a language card, you may want to run the CONVERT program supplied on the P-LISP Master disk. This changes the default workspace to $4000-95FF, $D000-$FFFF. Chapter 1 describes how to do this (you should NOT run CONVERT if you are running an Apple II with Applesoft on a language card!)

1) Setting the Recursion Stack

The default recursion stack provides roughly 124 recursion/nesting levels. The stack can be increased or decreased as desired, or moved to another section of memory. The start and end addresses of the stack are stored in locations $80D-80E and $80F-810 respectively, low-order byte first. Keep in mind the following restrictions when changing the recursion stack:

a) The end address should always be greater than the start address (naturally).

b) The start address should NEVER be less than $3B00. Otherwise the stack will overwrite the interpreter, making things very ugly and unpleasant.

c) The stack should not overlap the workspace area (see below).

d) You must have at least 1 page (256 bytes) of stack for P-LISP to work.

e) Only the high order byte is significant for the end-of-stack boundary. Thus, for example, to allocate $3B00-$3DFF to the stack, you should set the end-of-stack boundary to $3E00, ** NOT ** $3DFF.

2) Protecting the hires display area.

You should never use hires graphics unless the hires display area is protected. To do this, store a zero in locations 81B through 81E. If you wish to free the hires area, store an FF in each of these locations.

3) Setting the Workspace Area

The memory map allows the user to protect any page in the free space area from being used by P-LISP. By default, the free space area is locations 4000-95FF. Each page is represented by a bit in the map; if the bit is 1, the page is free for use; if the bit is a 0, the page is protected. Thus, pages can be protected for hires graphics, shape tables, or any other desired use.

To protect a page in the map, the page's corresponding bit should be st to 0; to allocate the page, the bit should be set to 1. The table on page 7-3 indicates the map locations for each page (the table entries are the pages addresses). The X'd locations are those that should NEVER be changed. See the example below for more details.

To change the memory allocation to the workspace area, do the following:

1) Set the appropriate bits in the memory map.

2) If you change the starting page of the workspace, you must indicate this in the map by storing the address of the first page in locations $811-812, low-order byte first.

Be aware of the following restrictions:

a) Do not allocate the language card areas unless you intend to use a language card.

b) Do not change any of the bytes that do not appear in the table.

c) The free space area and recursion stack should not overlap.

Addresses:

80D: Start of stack
80F: End of stack
811: Start of workspace

Workspace map:

BYTE ADDRESS	BIT# 7	6	5	4	3	2	1	0
81A	X	X	X	3B00	3C00	3D00	3E00	3F00
81B	4000	4100	4200	4300	4400	4500	4600	4700
81C	4800	4900	4A00	4B00	4C00	4D00	4E00	4F00
81D	5000	5100	5200	5300	5400	5500	5600	5700
81E	5800	5900	5A00	5B00	5C00	5D00	5E00	5F00
81F	6000	6100	6200	6300	6400	6500	6600	6700
820	6800	6900	6A00	6B00	6C00	6D00	6E00	6F00
821	7000	7100	7200	7300	7400	7500	7600	7700
822	7800	7900	7A00	7B00	7C00	7D00	7E00	7F00
823	8000	8100	8200	8300	8400	8500	8600	8700
824	8800	8900	8A00	8B00	8C00	8D00	8E00	8F00
825	9000	9100	9200	9300	9400	9500	X	X
82D	D000	D100	D200	D300	D400	D500	D600	D700
82E	D800	D900	DA00	DB00	DC00	DD00	DE00	DF00
82F	E000	E100	E200	E300	E400	E500	D600	E700
830	E800	E900	EA00	EB00	EC00	ED00	EE00	EF00
831	F000	F100	F200	F300	F400	F500	F600	F700
832	F800	F900	FA00	FB00	FC00	FD00	FE00	FF00

Apple II with language card users: if you wish to allocate the language card RAM, you may only use locations $D000-DFFF, as the remainder of the card must be occupied by Applesoft. Apple II+ or IIE owners may allocate the entire language card area.

Example: Suppose I only need a small (2-page) stack, I want to use hires graphics, and I want to allocate the remainder of memory to the workspace. Suppose further that I am starting with a default workspace as described above.

First, I load the interpreter and enter the monitor. I want the stack to range from $3B00-3CFF, so I have to change the ending address of the stack:

```
*80F: 00 3D
```

I want the area normally used by the stack ($3D00-3FFF) to be made part of my workspace. To do this, I type

```
*81A: E7
```

(The three high-order bits of byte 81A must ALWAYS be set, hence the E in the first half of the byte. The 7 in the second half allocates pages $3D00, 3E00, and 3F00 to the workspace.)

Since I changed the starting location of the workspace, I must also type the following:

```
*811: 00 3D
```

Now I want to protect the hires display area:

```
*81B: 00 00 00 00
```

Finally, I want to allocate the language card RAM:

```
*82D: FF FF FF FF FF FF
```

If I dump the map on the monitor (by typing 80D.832), I should see the following:

```
80D - 00 3B 00
810 - 3D 00 3D 80 00 00 00 00
818 - 00 7F E7 00 00 00 00 FF
820 - FF FF FF FF FF FC 00 00
828 - 00 00 00 00 00 FF FF FF
830 - FF FF FF
```

NOTE: Due to bank switching methods used by Apple to access the language card, it is possible that the Apple will "hang" if an I/O error occurs during a LOAD or SAVE.

I can now type 800G to start the interpreter with the newly-configured workspace. If i'm going to use this map several times for different workspaces, I should immediately SAVE the workspace to create a permanent copy of the map on disk. I could also save the map with the interpreter by BSAVEing the interpreter, although I should do this BEFORE typing 800G.

Another example: Suppose I want to put the recursion stack on the language card and allocate everything else to the workspace. To do this, I do the following:

First, load the interpreter and enter the monitor. To put the stack on the language card, I type:

```
80D: 00 D0
80F: 00 00
```

(Recall that only the hi-order byte is significant. So, If I want to set the stack to range from $D000-FFFF, I have to set the end address to $0000).

Now I allocate the memory normally used by the stack to the workspace:

```
81A: FF
```

and I adjust the start-of-workspace address:

```
811: 00 3B
```

I have to make sure the stack area is protected:

```
82D: 00 00 00 00 00 00
```

I should now type 800G to start the interpreter. I can then SAVE the workspace to keep a permanent copy of this map.

CHAPTER 8 -- MACHINE LANGUAGE INTERFACE

After obtaining some experience with LISP programming, you may wish to implement some functions, particularly those that are called frequently, in machine language. This chapter will provide some of the information necessary for interfacing P-LISP with machine language subroutines. Be aware that you should use this information at your own risk. All entry points apply to version 3.2 of the interpreter (these entry points are NOT valid for earlier versions).

All of the P-LISP workspace is divided into 4-byte "cells". The first two bytes are the CAR of the cell, and the last two bytes are the cell's CDR. The CAR and CDR of a cell typically contain pointers (addresses) to other cells. Cells are aligned on 4-byte boundaries, so the 2 low-order bits of a cell's address are 0. Pointers to atoms have bit 1 set to indicate the pointer is atomic, so for example, an atom that lives at hex location $4000 would be referenced in a cell as $4002. Bit 0 of a cell is used to mark active cells during garbage collection.

Atoms are stored with the following formats. They are divided into four types: literal, numeric-integer, numeric-floating point, and string.

1) Literal Atom
   Cell 1:
     CAR: pointer to value (0 if no value)
     CDR: pointer to cell 2
   Cell 2:
     CAR: pointer to print-name cell
     CDR: pointer to property list
   Print-name cell:
     CAR: 1st 2 characters of print-name
     CDR: pointer to next print-name cell

Property list:
The property list is stored like any other list, as a chain of cells, with the CAR of each cell pointing to a list element, and the CDR pointing to the next cell in the chain. Lists are terminated with a NIL in the CDR ($0072).

2) Numeric-integer
   Cell 1:
     CAR: pointer to cell 2
     CDR: NIL
   Cell 2:
     CAR: 16-bit value (2's complement)
     CDR: NIL

3) Numeric-floating point
  Cell 1:
    CAR: pointer to cell 2
    CDR: NIL
  Cell 2:
    CAR: exponent and 1st byte of mantissa
    CDR: pointer to cell 3
  Cell 3:
    Bytes 1-3: last 3 bytes of mantissa
    Byte 4: sign byte

4) String Atom
  Cell 1:
    CAR: pointer to print-name of string
    CDR: 0

Because P-LISP is normally looking at language-card RAM (even if you do not have a language card), you must do the appropriate bank-switching in your machine language routine (depending on what you want to do). If you wish to call a monitor ROM routine, you must select ROM in your routine by referencing ROMSEL (if you don't know what any of this means, you should consult your language card reference manual). If you wish to select Applesoft, you should CALL the routine FPSEL. If your routine is going to reference workspace memory, you MUST first select RAM: this is done by making TWO references to RAMSEL. The addresses for these labels are listed below. Note that P-LISP does a ROMSEL before executing a CALL; therefore, you cannot put your machine language routine on the language card.

The LISP interface to your routine should be an EXPR that CALLs the routine. If you wish to pass arguments to the routine, this must be done via the environment list. For example, you can define a function like

```
(MYFUNCTION (LAMBDA (X)
  (CALL MYROUTINE))
```

and then access X in MYROUTINE via the environment list. The pointer to the environment list is in page zero location ENVPTR. After the call on MYROUTINE, X will be at the "front" of the list; a pointer to its value will be in the CAR of the SECOND cell on the list (the CAR of the first cell contains a pointer to X). If MYFUNCTION takes two arguments, say X and Y, the value of Y will be in the CAR of the FOURTH cell.

If your routine is to return a value, you should put a pointer to that value in RESPTR before the return; otherwise, CALL will simply return the address of the CALLed routine. You should return to P-LISP via an RTS.

Some important addresses and entry points are listed below. All addresses are given in hex. There is no guarentee that these routines will not have unexpected side effects when used by a caller other than the interpreter; you should be sure you think you know what you're doing before using them.

Page Zero locations:

RES	$50
CELPTR	$5A
ATMPTR	$5C
ATMTST	$69
RDPTR	$6A
RESPTR	$6C
STKCEL	$6E
OBLSTPTR	$7B
SIGN	$7F
FREPTR	$80
FAC	$9D
ARG	$A5
WK1	$CE
CARPTR	$EB
ARGPTR	$ED
PRINTFLG	$EF
WINDWDTH	$F0
ENVPTR	$FB
FREEFLG	$FD

Other locations:

RAMSEL	$C08B
ROMSEL	$C081

Location ATMTST contains the value $02 which should be used when testing the status of bit 1 of a pointer (see example below).

FREPTR points to the free-space list, a list of available cells.

OBLSTPTR points to the object list.

ROMSEL should be referenced to select the motherboard ROM.

RAMSEL should be referenced TWICE when selecting workspace RAM.

Entry points:

FPSEL - $A02 - select Applesoft

SCROLL - $A0A - send a carriage return to the output device

COUT - $A0C - send the character in the accumulator to the output device. RAM is selected after this call.

NEWNUMAT - $B37 - generates a new numeric atom from the contents of FAC and puts a pointer to the atom in ATMPTR. If the atom has a fractional part, a floating point atom is generated; otherwise an integer atom is generated. FAC should be in normalized exponential form, with the sign in FAC+5.

NEWINT - $B5C - generates a new integer atom. The integer value should be in RES, with the sign in SIGN ($0 for positive, $FF for negative). ATMPTR points to the new atom.

GETCEL - $C79 - moves contents of CELPTR to WK1 and puts a new cell from the free-space list in CELPTR. A garbage collection is performed if the free-space list is empty.

CARLINK2 - $BEF - links CELPTR to the CAR of WK1.

CDRLINK2 - $BCB - links CELPTR to the CDR of WK1.

WK1CAR - $DCE - replaces WK1 with its CAR

WK1CDR - $DC0 - replaces WK1 with its CDR

PUSH - $EB4 - pushes the contents of the X and Y registers onto the recursion stack, X register first.

POP - $F1A - pops the recursion stack, putting the top two bytes in the X and Y registers, Y register first.

CELPUSH - $EED - pushes CELPTR onto the recursion stack.

CELPOP - $EF4 - pops recursion stack, putting result in CELPTR.

RESPUSH - $EFC - pushes RESPTR onto recursion stack

RESPOP - $F03 - pops recursion stack, putting result in RESPTR.

WK1PUSH - $F0B - pushes WK1 onto recursion stack.

WK1POP - $F12 - pops recursion stack, putting result in WK1.

READ - $DFD - reads a sexpr from the input device. The sexpr is returned in RDPTR. For a normal READ, FREEFLG should contain a $0; for a free-format READ, it should contain a $FF. Be sure to reset FREEFLG to $0 if you change it!

PRINT - $F31 - prints the sexpr in WK1. String atoms will be printed with quotes if PRINTFLG is set to $FF and without quotes if PRINTFLG is set to $0.

FACGET - $1114 - loads FAC with a floating-point number. A pointer to cell 2 of the numeric atom should be in WK1.

INTGET - $10BC - loads RES with an integer. A pointer to cell 2 of the numeric atom should be in WK1.

EVLPUSH - $1253 - pushes EVLPTR onto recursion stack.

EVLPOP - $125A - pops recursion stack, putting result in EVLPTR.

EVLCAR - $128E - replaces EVLPTR with its CAR

EVLCDR - $1280 - replaces EVLPTR with its CDR.

ATEVAL - $1292 - evaluates an atom. The atom's pointer (with bit 1 CLEARED) should be in CARPTR. The value is returned in RESPTR. If the atom has no value, the carry flag is set.

LITCHK - $141B - checks if ARGPTR points to a literal atom. Carry is cleared if so, set if not. If the atom bit is set, LITCHK clears the bit.

ARGPUSH - $1389 - pushes ARGPTR onto recursion stack.

ARGPOP - $1390 - pops recursion stack, putting result in ARGPTR.

ARGCAR - $1377 - replaces ARGPTR with its CAR

ARGCDR - $1385 - replaces ARGPTR with its CDR

NILRET - $E9D - sets RESPTR to NIL.

TRET - $14F3 - sets RESPTR to T.

NUMCHK - $1510 - checks if ARGPTR points to a numeric atom. Carry is cleared if so, set if not.

ARGTOWK1 - $2134 - moves ARGPTR to WK1.

WK1ARG - $19FA - moves WK1 to ARGPTR.

ARGLD - $1D0D - loads ARG with a floating-point number. ARGPTR should point to cell 2 of the atom.

FACLD - $1D11 - same as ARGLD, except FAC is loaded.

INTALD - $1D2B - routine to float and load an integer value into ARG. ARGPTR should point to cell 2 of the integer atom.

INTFLD - $1D3B - same as INTALD, except FAC is loaded.

ADD2 - $1D69 - adds FAC and ARG and returns pointer to the resulting atom in RESPTR.

SUB2 - $1D9B - subtracts FAC from ARG. Result returned in RESPTR.

MULT2 - $1DC1 - multiplies FAC by ARG. Result returned in RESPTR.

DIV2 - $1DE7 - divides ARG by FAC. Result in RESPTR.

EQNUM - $1F98 - returns carry set if the atom at ARGPTR is equal to the atom at WK1, otherwise carry cleared. The atoms must be either numeric or string atoms.

ERROR - $24EA - error handler. Register X should contain the error code minus 8. Y should contain the function code ($42 for EXPRs). Part of the aerror message might not make sense; specifically, the argument list to the function in the error message will probably be the address of the machine language routine.

GC3 - $28B3 - garbage collector. RESPTR will contain a pointer to an atom equal to the number of cells collected.

Example:

Suppose I want to write a function, CDDR, which returns the CDR of the CDR of its argument, via a machine subroutine. I could do this as follows:

```
(DEFINE (CDDR LAMBDA (X)
(CALL 768))))
```

I would have the following subroutine at location 768 (hex location $300):

```
CDDR    LDA ENVPTR     ;move environment ptr to WK1
        STA WK1
        LDA ENVPTR+1
        STA WK1+1
        JSR WK1CDR     ;get second cell
        JSR WK1CAR     ;WK1 now points to the value of X
        LDA WK1        ;make sure WK1 doesn't point at an
atom
        BIT ATMTST
        BNE CDDRAT
        JSR WK1CDR     ;gives us (CDR X)
        LDA WK1+1      ;return if result is NIL
        BEQ CDDRFN
        JSR WK1CDR     ;gives us (CDR (CDR X))
CDDRFN  LDA WK1        ;return WK1 in RESPTR
        STA RESPTR
        LDA WK1+1
        STA RESPTR+1
        RTS
CDDRAT  LDX #$8        ;error code:Bad List Argument
        LDY #$3E       ;function code for EXPRS
        JMP ERROR      ;goto error routine
```

CHAPTER 9 -- SAMPLE PROGRAMS

P-LISP is supplied with four sample programs. They are meant to demonstrate sample LISP applications as well as the capabilities of the P-LISP system. By examining these programs, the novice reader may also learn a little bit about LISP programming.

The sample programs are:

-- Towers of Hanoi, a lo-res graphics demonstration of logical problem with a simple recursive solution.

-- Hilbert, a hires graphics program that draws a recursive design on the display screen.

-- Calc, a simple calculator program that demonstrates how LISP can be used in symbolic math applications.

-- Eliza, a classic Artifical Intelligence application of LISP.

All the above programs are supplied both in workspace format and in text file format.

1) Towers of Hanoi

You can load this program either by EXECing HANOI.TEXT or by LOADing HANOI. In brief, the Towers of Hanoi problem is as follows: assume there are three towers, labelled A, B, and C. On Tower A is a stack of disks, each of different size, the disks stacked in order of decreasing size from bottom to top. The problem is to move the stack of disks, one disk at a time, from Tower A to Tower B. The catch is that you cannot put a disk on top of a smaller disk.

This problem has a simple recursive solution: given N disks on tower A, you can move the disks to Tower B by:

a) moving N-1 disks from A to C
b) moving the bottom disk on A to B
c) moving N-1 disks from C to B.

The program HANOI demonstrates the solution to this problem. To invoke the program, type (HANOI). The program will ask you to enter the number of disks on Tower A (any number between 1 and 6 inclusive). After you enter the number, the program will demonstrate (in lores graphics) the moves necessary to solve the problem. When all the disks are on Tower B, hit any key to get back to the program.

2) Hilbert

HILBERT is a program that draws a design in hires graphics. The design has up to 7 levels; each level can be defined recursively in terms of the previous level. A text file version of the program is in the file HILBERT.TEXT; a workspace version (with the appropriate memory map for hires graphics) is in the file HILBERT. You should use this file to run the program.

To run the program, LOAD HILBERT and then type (HILBERT). The program will then ask you to enter a level between 1 and 7 inclusive. You should start with 1 and work your way up to see the recursion embedded in the higher levels. Once the design is finished, hit any key to get back to the program.

In both the Hilbert and Hanoi demos, you will notice the computer pause every now and then while drawing on the screen. This pause is when the interpreter is performing a garbage collection.

3) Calc

CALC is a simple calculator program. To use it, either type EXEC CALC.TEXT to EXEC the functions into your workspace or LOAD CALC to load the workspace version. To invoke CALC, type (CALC).

The program can perform addition, subtraction, multiplication, division, and assignment, via the operators +, -, *, /, and = respectively. Multiplication and division have precedence over additon and subtraction. Assignment has the lowest precedence. Expressions can also be nested in parentheses. If an expression cannot be evaluated, or if a numeric overflow or similar error occurs, the program will print the message SYNTAX ERROR.

To use the program, simply type an arithmetic expression into CALC; the program will return the result of the expression. Each token in the expression (atom, number, or operator) should be surrounded by spaces.

Ex:
With CALC running, I can enter the following:

```
?3 + (4 * 9) - 12 / 3
```

and CALC responds with

```
35
```

I can also assign values to atoms, for example,

```
?x = 9 - (8 + 2 * 3) + 10
 5
```

Now, X can be used in subsequent expressions:

```
?y = x + x * x
 30

?y - 2 * x
 20
```

If I try doing something illegal, like

```
?z = x / 0
```

the program responds with

```
SYNTAX ERROR
```

I would also get this error if I used an atom in an expression that I have not assigned a value.

4) Eliza

ELIZA is a classic example of the use of LISP in an Artificial Intelligence (AI) application. Originally written by Joseph Weizenbaum, ELIZA is a program that simulates a patient's discussion with a psychiatrist. While appearing on the surface to possess "intelligence", the program is actually a rather simple pattern matcher. Here is a sample dialogue with ELIZA; as usual, user input is in lower case, with the program's responses in upper case:

```
MAY I HELP YOU?

?i want you to help me

WHAT WOULD IT MEAN TO YOU IF YOU GOT ME TO HELP YOU?

?im afraid to tell you

YOURE NOT REALLY TALKING ABOUT ME, ARE YOU?

?what do you mean

WHY DO YOU ASK?

?are you asking me

WHY ARE YOU INTERESTED IN WHETHER OR NOT I AM ASKING YOU?

?why do you keep answering with questions
```

```
DOES THAT QUESTION INTEREST YOU?

?i dont know

DONT YOU REALLY KNOW

?no

ARE YOU SAYING NO JUST TO BE NEGATIVE?

?oh shut up

YOU ARE BEING UNFRIENDLY

?goodbye

GOODBYE
```

To run ELIZA, LOAD the ELIZA workspace and type (ELIZA). Your input to ELIZA should consist of words only (no lists!) with no punctuation (DON'T use quotes in contractions!). Type GOODBYE to exit ELIZA. ELIZA is also supplied in text file format in the file ELIZA.TEXT.