Inter-LISP

for the Atari® Computer

Copyright © 1981 by Special Software Systems

All Rights Reserved

Published & Distributed Exclusively By

Datasoft Inc.
PERSONAL COMPUTER SOFTWARE

9421 Winnetka Avenue
Chatsworth, CA 91311

*Atari is a registered trademark of Atari Computer, Inc.

Inter-LISP

for the Atari® Computer

Copyright © 1981 by Special Software Systems

All Rights Reserved

Published & Distributed Exclusively By

Datasoft Inc.®
PERSONAL COMPUTER SOFTWARE

9421 Winnetka Avenue
Chatsworth, CA 91311

Atari is a registered trademark of Atari Computer, Inc.

LIMITED WARRANTY

This software product and the attached instructional materials are sold "AS IS," without warranty as to their performance. The entire risk as to the quality and performance of the computer software program is assumed by the user. The user, and not the manufacturer, distributor or retailer assumes the entire cost of all necessary service or repair to the computer software program.

However, to the original purchaser only, DATASOFT warrants that the medium on which the program is recorded will be free from defects in materials and faulty workmanship under normal use and service for a period of ninety (90) days from the date of purchase. If during this period a defect in the medium should occur, the medium may be returned to DATASOFT or to an authorized DATASOFT dealer, and DATASOFT will replace or repair the medium at DATASOFT'S option without charge to you. Your sole and exclusive remedy in the event of a defect is expressly limited to replacement or repair of the medium as provided above. To provide proof that you are the original purchaser, please complete and mail the enclosed Owner Warranty Card to DATASOFT.

If failure of the medium, in the judgment of DATASOFT, resulted from accident, abuse or misapplication of the medium, then DATASOFT shall have no responsibility to replace or repair the medium under the terms of this warranty.

The above warranties for goods are in lieu of all other express warranties and no implied warranties or merchantability and fitness for a particular purpose or any other warranty obligation on the part of DATASOFT shall last longer than ninety (90) days. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation may not apply to you. In no event shall DATASOFT or anyone else who has been involved in the creation and production of this computer software program be liable for indirect, special, or consequential damages, such as, but not limited to, loss of anticipated profits or benefits resulting from the use of this program, or arising out of any breach of this warranty. Some states do not allow the exclusion or limitation of incidental or consequential damages so the above limitation may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

The user of this product shall be entitled to use the product for his/her own use, but shall not be entitled to sell or transfer reproductions of the product or instructional materials to other parties in any way.

NOTICE:

Special Software Systems and Datasoft Inc. reserve the right to make improvements in the product described in this manual at any time and without notice. Licensing rights for *Inter-LISP* may be obtained through the publisher.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the publisher. Printed in the United States.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

1.0 INTRODUCTION

1.1 OVERVIEW OF INTER-LISP/65

Welcome to the world of LISP programming. INTER-LISP/65 is an implementation of a subset of the standard "INTERLISP" dialect of LISP. This version includes many enhancements to LISP which will allow the LISP programmer to take advantage of the graphic and sound capabilities of the ATARI computer. Over seventy pre-defined functions are included.

SOME FEATURES OF INTER-LISP/65 ARE:

- eight digit floating point arithmetic, including multiplication, division, exponential, and logarithms
- PEEK, POKE and XIO functions for access to monitor and hardware functions.
- STICK and STRIG for game controller input.
- Sequential disk file functions: including OPEN,CLOSE, PR#, IN#, NOTE and POINT.
- PAGE and TAB screen control functions.
- Debugging facilities including: BREAK and BAKTRACE.

System requirements for INTER-LISP/65 consist of an ATARI 800* computer, equipped with 48K of RAM memory and at least one disk drive. The package you have purchased should contain:

One (1) 5 1/4" diskette containing the following files:

* DOS SYS
* AUTORUN SYS
* EDIT
* DOCTOR
* HANOI
* CLISP
* MACLISP
* LIGHTS
* CALC

(NOTE: AUTORUN.SYS IS THE LISP PROGRAM)

One (1) INTER-LISP/65 reference manual

One (1) LISP programming text by Winston and Horn, published by Addison Wesley.

1.2 HOW TO USE THIS MANUAL

This manual assumes that you have some knowledge of the LISP language. It is not intended to provide an exhaustive treatment of LISP. Instead it gives synoptic descriptions of the commands available in INTER-LISP/65. These descriptions should be sufficient to explain the differences between INTER-LISP/65 and the dialect of MACLISP discussed in Winston and Horn. This should allow you to translate easily between the two dialects. For those who do not wish to perform the translations while working the exercises in Winston and Horn, a MACLISP emulator has been written in INTER-LISP/65. To use it, load the file MACLISP from your diskette.

If you are a novice LISP programmer, we suggest that you read Chapter 2.0 of this manual and then try the examples in Winston and Horn on your ATARI. If you encounter errors while trying the exercises, refer to Chapter 13.0 of this manual.

2.0 USING INTER-LISP/65

2.1 HOW TO RUN LISP

Remove all cartridges from your ATARI and power up your ATARI computer. Insert the LISP system diskette, and "boot" your system. (Note: If you forget to remove the cartridges the program will ask you to remove them and reboot the disk.) After a few moments LISP will respond by displaying the copyright statements and the message:

```
>>> INTER-LISP/65 V2.0 July 1981 <<<
```

which indicates the version of LISP you are running, (2.0 in the current example). LISP will then display the prompt:

```
LISP
```

which indicates LISP is ready to accept input.

2.2 LISP SUPERVISOR AND LISP DATA TYPES

Unlike BASIC which has immediate and deferred execution commands, LISP accepts inputs called s-expressions (short for symbolic expression). It then evaluates them and prints the result. The program which performs this loop is called the "LISP SUPERVISOR".

Try typing a 1 followed by RETURN. LISP responds by printing

```
1
```

This is the decimal value, (or evaluation), of the input which was the number 1. Now type:

```
X
```

followed by RETURN.

LISP will respond with

```
NIL
```

This is LISP's way of saying that the "atom" X is currently undefined, (i.e. it has no value).

LISP has two types of data — atoms and lists. Collectively these data types are the symbolic expressions, (or s-expressions), referred to above. Numbers are atoms as are alpha-numeric strings like A, ABC and AB123. Some pre-defined atoms are T, which has the value of logical TRUE, and NIL, which has the value of logical FALSE. Blanks (spaces), are used to delineate the end of an atom.

Lists are constructed from atoms or from other lists. All lists must start with a left parenthesis and end with a right parenthesis. For example:

(A B C)

is a list whose elements are the atoms, A, B and C.

The expression,

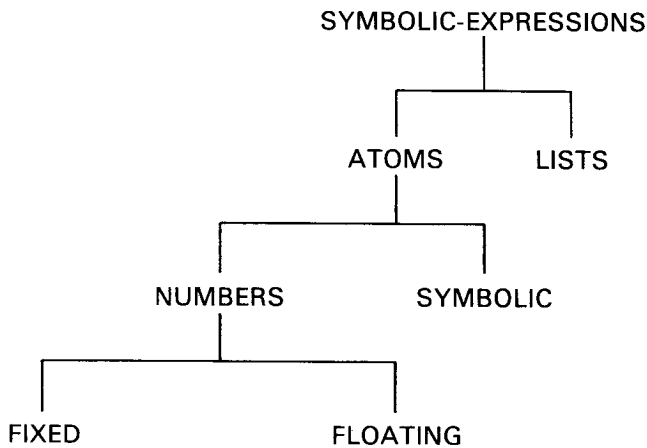
((A B) C)

is a list whose elements are the list

(A B) and the atom C

The list (A B) in this example is sometimes referred to as a sub-list.

LISP data types can be summarized by the following diagram:



Atom values can be altered by use of the SETQ function. You may type:

(SETQ LANGUAGES '(LISP PASCAL BASIC FORTRAN))

This establishes the list

(LISP PASCAL BASIC FORTRAN)

as the value of the atom LANGUAGES. If you now type

LANGUAGES<CR>

LISP responds with:

(LISP PASCAL BASIC FORTRAN)

LISP

The single quote mark in the above example is used to indicate that the s-expression which follows should not be evaluated, (i.e. the expression should be taken literally). This is why LISP responds with:

(LISP PASCAL BASIC FORTRAN)

when you enter LANGUAGES, but

'LANGUAGES

yields

LANGUAGES

LISP

In fact, the single quote mark is a short hand for the QUOTE function which returns its argument (not the value of the argument).

(QUOTE X)

X

LISP

Atoms may also be used to represent functions. Function definitions are established by using the DEFINE and DEFINEQ functions. These will be discussed in Chapter 7.0.

2.3 LISP KEYBOARD FUNCTIONS

INTER-LISP/65 allows you to use all of the standard ATARI cursor control sequences as described in the ATARI reference manual. Additionally a LISP print, or evaluation, may be aborted by typing a Control-B. (This has the same effect as the BREAK function described in Chapter 9.0).

INTER-LISP/65 uses a character type ahead buffer. Thus, many commands may be entered on one line and will be executed sequentially. An exception occurs with a LOAD command. A LOAD command causes the input buffer to be flushed. All input which follows the LOAD command and which lies before the next Carriage Return will be ignored. Therefore, entering:

(LOAD 'D:DOCTOR) (DIR)<CR>

will load the file D:DOCTOR , but the (DIR) will be ignored.

When reading s-expression from the keyboard the usual prompt is LISP; if you entered an "unbalanced" s-expression then the prompt character is changed as follows. If you have entered a return before all left parentheses are paired with right parentheses then the prompt becomes a single question mark (?). If you have entered a return before closing a quoted atom (one beginning with a ') then the prompt is changed to a double quote mark (").

IMPORTANT NOTE: DO NOT HIT THE SYSTEM RESET KEY during execution of a LISP program except in a dire emergency! INTER-LISP/65 V2.0 uses a modified version of the Schorr-Waite algorithm during "garbage collection". Therefore, an inopportune RESET could result in all of memory being destroyed. However, it is ok to use the reset key to abort a program provided it is "reading" from the keyboard.

3.0 LISP ARITHMETIC

3.1 NUMERIC ATOMS

The numeric atoms in INTER-LISP/65 are eight byte, binary coded decimal floating point numbers, (i.e. numbers in the range −9.99999999E−97 to 9.99999999E+97). If the result of any arithmetic expression results in a number exceeding these limits then an error message is displayed and control is passed to the error handler. (Refer to Chapter 13.0.)

3.2 ARITHMETIC EXPRESSIONS

LISP treats all expressions to be evaluated as functions. The same applies to arithmetic expressions. In BASIC you would write X + Y to add the numbers X and Y. LISP, however, treats + as a function which assigns the sum to two operands. Thus:

```
+ : (X, Y) ----> X+Y
```

and it is written as such in LISP, as follows:

```
(+ X Y).
```

This notation is sometimes referred to as Cambridge prefix notation.

The following table contrasts the corresponding BASIC and LISP expressions:

BASIC	LISP
X * Y	(* X Y)
X / Y	(/ X Y)
X + Y	(+ X Y)
X − Y	(SUB X Y)
EXP (X)	(EXP X)
LOG (X)	(LOG X)
INT (X)	(INT X)

More complex arithmetic expressions are constructed in the obvious way. For example, the expression:

(X - Y) / (X + Y)

would be written in LISP as:

(/ (SUB X Y) (+ X Y)).

These arithmetic expressions demonstrate the general syntax of LISP expressions. The EVAL function (which is discussed in the next section), expects lists in the form:

(functionname argument
argument ...)

If functionname refers to an undefined expression, then an error will result. (See Chapter 13.0 for a discussion of the LISP error handling procedure.)

NOTE: On input to LISP all numbers should be delimited by one of the LISP delimiting characters (i.e. a blank, parenthesis, single or double quote). For instance, if one were to enter the string:

123AB 45

then LISP would interpret this as a 123 followed by a 45. The AB falls into the "bit bucket".

4.0 BASIC LISP FUNCTIONS

This chapter discusses the basic LISP functions available in INTER-LISP/65.

4.1 THE SETQ FUNCTION

SETQ establishes values for atoms. SETQ is, in effect, the LISP assignment function.

```
LISP
(SETQ X 3.4) (SETQ Y 4.5)

3.4
4.5

LISP
( + X Y)

7.9
LISP
```

Note that the value returned by SETQ is the value assigned to the variable, (or atom).

4.2 THE QUOTE FUNCTION

The QUOTE function returns its single unevaluated argument as its value. On input the single quote mark is equivalent to QUOTE. For example:

```
LISP
(SETQ X 3)

3

LISP
X

3
```

```
LISP
(QUOTE X)

X

LISP
'X

X
```

4.3 THE EVAL FUNCTION

The EVAL function evalutes its single argument which is itself evaluated, so that in effect EVAL actually results in two evaluations. Thus, using X as in section 4.1 above, gives:

```
LISP
(EVAL (QUOTE X))

3
```

since the evaluation of (QUOTE X) is

```
X
```

which when evaluated yields

```
3
```

4.4 THE SET FUNCTION

SET is similar to SETQ except that the variable is evaluted. SET is somewhat like a "computed assignment" function.

```
LISP
(SETQ X 'A)

A

LISP
(SET X 100)
```

```
100

LISP
X

A

LISP
A

100
```

4.5 THE CAR FUNCTION

```
(CAR X)
```

The CAR function returns the first element of a list. If the argument is atomic, an error will result.

Examples of CAR:

```
(CAR NIL)
```

returns NIL.

```
LISP
(SETQ A '((X Y) Z))

((X Y) Z)

LISP
(CAR A)

(X Y)

LISP
(CAR (CAR A))

X
```

The CAR function has retained its name from the first implementation of LISP on the IBM 7090* computer. The first element of a list was stored in the address register. Hence the term, "Contents of Address Register".

```
100

LISP
X

A

LISP
A

100
```

4.5 THE CAR FUNCTION

```
(CAR X)
```

The CAR function returns the first element of a list. If the argument is atomic, an error will result.

Examples of CAR:

```
(CAR NIL)
```

returns NIL.

```
LISP
(SETQ A '((X Y) Z))

((X Y) Z)

LISP
(CAR A)

(X Y)

LISP
(CAR (CAR A))

X
```

The CAR function has retained its name from the first implementation of LISP on the IBM 7090* computer. The first element of a list was stored in the address register. Hence the term, "Contents of Address Register".

4.6 THE CDR FUNCTION

(CDR X)

The CDR function returns the list which is obtained by deleting the first element of the list. If X is an atom then an error will result; however, (CDR NIL) returns NIL.

Examples of CDR:

```
LISP
(SETQ A '((X Y) Z))

((X Y) Z)

LISP
(CDR A)

(Z)

LISP
(SETQ B '(X . Y))

(X . Y)

LISP
(CDR B)

Y

LISP
(SETQ EXAMPLE '(THIS IS AN EXAMPLE OF CDR))

(THIS IS AN EXAMPLE OF CDR)

LISP
(CDR (CDR EXAMPLE))

(AN EXAMPLE OF CDR)
```

CDR gets its name from the first LISP implementation on the IBM 7090 where the tail of a list was stored in the decrement register, hence the acronym "Contents of Decrement Register".

4.7 THE CONS FUNCTION

(CONS X Y)

In brief, CONS reassembles the CAR and CDR of a list. In fact CONS is short for CONStructor.

For example,

```
LISP
(SETQ A '(X Y))

(X Y)

LISP
(CAR A)

X

LISP
(CDR A)

(Y)

LISP
(CONS 'X '(Y))

(X Y)

LISP
(SETQ A ' (X . Y))

(X . Y)

LISP
(CAR A)
```

```
X

LISP
(CDR A)

Y

LISP
(CONS 'X 'Y)

(X . Y)
```

4.8 THE LIST FUNCTION

(LIST X1 X2 ... XN) produces the list whose elements are the evaluated arguments X1 ... XN.

Thus:

```
LISP
(LIST 'A 'B 'C)

(A B C)

LISP
(LIST '(A) '(B) '(C))

((A) (B) (C))
```

4.9 THE APPEND FUNCTION

```
(APPEND X Y Z ...)
```

APPEND returns the list which is the "concatenation" of its arguments, (which are themselves lists). APPEND is used more frequently than CONS.

Examples:

```
LISP
(SETQ X '(A))
```

```
(A)
LISP
(SETQ Y '(B))

(B)
LISP
(APPEND X Y)

(A B)
```

Note that CONS would behave as follows:

```
(CONS X Y)

((A) B)
LISP
```

4.10 THE LENGTH FUNCTION

LENGTH returns the number of elements in a list. If the argument is not a list then a zero is returned.

```
LISP

(LENGTH '(A B C))

3

LISP
(LENGTH 'A)

0

LISP
(LENGTH '((A B) C (D) E))

4

LISP
```

4.11 THE LAST FUNCTION

LAST returns the last CDR of a list. If the single argument is an atom or NIL then NIL is returned.

```
LISP
(LAST '(THIS IS THE END))

(END)

LISP
```

5.0 OTHER LIST PROCESSING FUNCTIONS

5.1 THE SURGICAL FUNCTIONS: RPLACA & RPLACD

RPLACA and RPLACD are mnemonics for "REPLACE CAR" and "REPLACE CDR", respectively. They create new lists by actually modifying the list structure of the arguments.

(RPLACA X Y) replaces the CAR of X by Y and alters the list X.

```
LISP
(SETQ X '(BASIC IS GOOD))

(BASIC IS GOOD)

LISP
(SETQ Y X)

(BASIC IS GOOD)

LISP
(RPLACA X 'LISP)

(LISP IS GOOD)

LISP
X

(LISP IS GOOD)

LISP
Y

(LISP IS GOOD)
```

Note that altering X also modifies Y in this case since Y is identified with X via the SETQ.

(RPLACD X Y) replaces the CDR of X by Y and returns the modified list as its value. Continuing with X and Y, retaining the values in the RPLACA example above, one has the following:

```
LISP
(RPLACD X '(IS BEST))

(LISP IS BEST)

LISP
X

(LISP IS BEST)

LISP

Y

(LISP IS BEST)

LISP
```

5.2.0 RETRIEVING LIST ELEMENTS

5.2.1 ASSOC PERFORMS KEYED LIST ACCESS

The ASSOC function performs keyed searches on lists which are referred to as association lists or a-lists for short. The structure of an a-list is:

```
(  (KEY1 DATA11 DATA12 ... DATA1j)
   (KEY2 DATA21 DATA22 ... DATA2k)
   .
   .
   .
   (KEYn DATAn1 DATAn2 ... DATAnl))
```

(ASSOC KEY ALIST) will search the argument ALIST for the sub-list whose CAR matches the key specified by KEY. If a match is found then the entire sub-list is returned as the value of ASSOC. Otherwise NIL is returned. If ALIST refers to the sample a-list shown above then the call:

```
(ASSOC 'KEY2 ALIST)
```

will return the list:

(KEY2 DATA21 DATA22 ... DATA2k)

As an example, suppose we have assigned the list

((SEX MALE) (HEIGHT 178) (AGE 32))

to the atom HENRY, then the following calls to ASSOC produce the indicated results:

```
LISP
(ASSOC 'SEX HENRY)

(SEX MALE)
LISP
(ASSOC 'HEIGHT HENRY)

(HEIGHT 178)
LISP
(ASSOC 'AGE HENRY)

(AGE 32)
```

5.2.2 THE @ FUNCTION PEFORMS SEQUENTIAL LIST ACCESS

(@ LIST I) returns the I th CDR of the list specified by LIST. This expression is equivalent to the expression:

```
(CDR (CDR (CDR ... (CDR LIST) ...)))
!<----- I TIMES ----->!
```

(@ LIST 0) or (@ LIST) just returns the value of LIST. Note that this expression is useful for array-like processing. If X is the list:

(1 2 3 4)

then (CAR (@ X I)) will return the number I + 1. If X is the list:

((1 2 3) (4 5 6) (7 8 9))

then (CAR (@ (CAR (@ X I)) J)) returns the (i,j) element of the 3x3 matrix represented by X.

5.3 THE PACKING FUNCTIONS: PACK AND UNPACK

PACK and UNPACK convert atoms to lists and vice versa.

(PACK X) converts a list of atoms to the atom which is the concatenation of the atoms in X. In this implementation any elements of X which are themselves lists are ignored.

```
LISP
(PACK '(S U I T C A S E))

SUITCASE

LISP
(PACK '(FRONT (THIS WILL BE SKIPPED) END))

FRONTEND
```

(UNPACK X)

The function converts the atom X to the list whose elements are the single character atoms in the atom X.

```
LISP
(UNPACK 'SUITCASE)

(S U I T C A S E)
LISP
```

UNPACK and PACK can be used to create elaborate string manipulation functions. The following examples return the length of a string (atom) and the right most characters of a string, respectively:

(DEFINEQ LEN$ (LAMBDA (STRING) (LENGTH (UNPACK STRING))))

(DEFINEQ RIGHT$ (LAMBDA (STRING START) (PACK (@ (UNPACK STRING) START))))

6.0 INPUT AND OUTPUT

In this chapter we describe the LISP input and output functions. Note that the LISP output functions described here differ slightly in effect from those described in Winston and Horn.

(READ)

The READ function accepts one entire s-expression from the current source of input. If the first character is not a left parenthesis, then the input characters are assumed to form an atom. READ then continues to accept characters until it encounters one of the LISP special characters, (namely a blank, "(", ")", carriage return), which is interpreted as a separator. If the first character is a left parenthesis, characters are read until all left parentheses are matched by right parentheses. (Note that if you don't count parentheses, this can produce the illusion that your computer is "hanging". Just type a few more parentheses). The value of (READ) is the s-expression entered.

(READA)

The READA function accepts one atomic s-expression from the current source of input. For READA, left parenthesis and right parenthesis are treated as atomic symbols provided they are the first characters encountered by READA. Otherwise they are treated as separators in the usual manner. The value of READA is the single atomic expression entered.

(READC)

READC accepts one character from the current source of input. The READC function does not wait for a terminating carriage return. The value of READC is the single character atom entered.

(PRINT X)

The PRINT function prints the evaluation of its single argument to the current source of output. The value of PRINT is the value of its argument. The PRINT function always terminates with a carriage return.

(PRIN1 X)

The PRIN1 function prints the evaluation of its single argument without a terminating carriage return. If X is a quoted atomic expression, (e.g. "ABC"), then it is printed without the quotes. Thus:

```
LISP
(PRIN1 (QUOTE "ABC"))
```

yields

```
ABC"ABC"
```

Note the absence of a carriage return after the first ABC (which is followed by "ABC", the value of PRIN1).

Similarly:

```
LISP
(PRIN1 '(A B))
```

yields:

```
(A B)(A B)
```

(PRIN2 X)

which is similar to PRIN1 except that quoted atoms are printed with the quotes.

(TERPRI)

TERPRI (nmemonic for 'terminate print') prints a carriage return to the current output device, and returns NIL.

7.0 DEFINING NEW FUNCTIONS

Just as SET and SETQ establish values for atoms, DEFINE and DEFINEQ establish function definitions for atoms. For example, to define a function which doubles the value of a number you could type:

```
LISP (DEFINEQ DOUBLE (LAMBDA (N) (+ N N)))
```

INTER-LISP/65 responds with:

```
(LAMBDA (N) (+ N N))

LISP
```

Note that the syntax of DEFINEQ and DEFINE in this implementation differ slightly from the standard INTERLISP implementation. DEFINE is similar to DEFINEQ except it evalutes its first argument. Example:

```
LISP
(DEFINE 'DOUBLE (LAMBDA (N) (+ N N)))
```

You may now use your new procedure in other expressions or even directly from the supervisor. Try typing:

```
LISP
(DOUBLE 6)
```

and INTER-LISP/65 should respond with:

```
12

LISP
```

You could also write a quadrupling procedure using the DOUBLE expression as follows:

```
LISP
(DEFINEQ QUADRUPLE (LAMBDA (N) (DOUBLE (DOUBLE N))))
```

7.1 THE LAMBDA FUNCTION

The Lambda function allows construction of new user defined functions which are locally defined variables. The syntax is:

```
(LAMBDA (X1 X2 ... XN) BODY)
```

where, X1, X2 ... are the local variables of the procedure defined by the s-expression BODY. In other words, the first argument of the LAMBDA function is the list of local variables to be employed within the procedure body. Thus:

```
(DEFINEQ GREATER (LAMBDA (X Y)
                         (COND (> X Y) X)
                               (T Y)
                 )))
```

defines a function "greater" which returns the greater of the two "input" variables X and Y. When you "call" this new function by entering the s-expression:

```
(GREATER 3 1)
```

the values of the arguments, (in this case the numbers 3 and 1), are "bound" to the formal arguments X and Y which occurred in the definition of GREATER. The values of the arguments at the moment the function is called are referred to as the "actual arguments". The binding of actual arguments to the formal arguments is in effect only while the LAMBDA function is being evaluated. Once a LAMBDA function is exited, the formal arguments are reassigned the values they had prior to the evaluation of the LAMBDA function. This can have some surprising effects. For instance, try the following example on your ATARI:

```
LISP
(SETQ X 'FIRSTVALUE)

FIRSTVALUE

LISP
```

```
(DEFINEQ SETVAL (LAMBDA (X Y)
                        (SET X Y)
                ))

(LAMBDA (X) (SET X Y) )

LISP

(SETVAL 'X SECONDVALUE)

SECONDVALUE

LISP
X

FIRSTVALUE
```

7.2 THE NLAMBDA FUNCTION

The syntax of the NLAMBDA function is:

```
(NLAMBDA (A) BODY)
```

Note that only one formal argument is allowed. The NLAMBDA function is similar to the LAMBDA function in that it also defines user functions. The major difference is that while LAMBDA evaluates its arguments, NLAMBDA does not evaluate its single argument. An example should clarify this difference:

```
LISP
(DEFINEQ DEMONSTRATION (NLAMBDA (A) A))

(NLAMBDA (A) A)

LISP
(DEMONSTRATION OF THE NLAMBDA FUNCTION)

(OF THE NLAMBDA FUNCTION)
```

Note that the entire unevaluated list which followed the function name was returned as the value of DEMONSTRATION.

7.3 THE MACRO FUNCTION

The MACRO function is an extension of the NLAMBDA function. It too is of the general form:

```
(MACRO (A) BODY)
```

For a MACRO, the single argument A is bound to the entire s-expression associated with the call of the MACRO. Again the argument A is unevaluated. Finally, the evaluation of BODY is evaluated as the MACRO is exited, (hence the terminology MACRO). The main use for MACRO is to construct code and then execute it. An example should clarify this:

```
LISP
(DEFINEQ FIRST (MACRO (S) (CONS 'CAR (CDR S))))

(MARCO (S) (CONS (QUOTE CAR) (CDR S)))
```

If you execute FIRST by entering:

```
LISP
(FIRST '(A B))
```

then the value of S will be:

```
(FIRST (QUOTE (A B)))
```

Thus the result of the MACRO body:

```
(CONS (QUOTE CAR) (CDR S)))
```

will be:

```
(CAR (QUOTE (A B)))
```

which when evaluated gives:

```
A

LISP
```

Note that FIRST is equivalent to CAR.

7.4 HOW INTER-LISP/65 EVALUATES FUNCTIONS

Recall that when LISP encounters a list it expects the first element to refer to a function, thus:

```
(FUNCTION ARGUMENT1
          ARGUMENT2
          .
          .
          .
          ARGUMENTn)
```

If FUNCTION is an atom, as in

```
(+ 10 -5)
```

then the function definition is retrieved. The function retrieved must be a list beginning with one of the atoms

SUBR,NSUBR,LAMBDA,NLAMBDA or MACRO

which signal the beginning of the function definition. If no function is present, or the list obtained does not begin with one of the reserved function types, then an EVALUATION ERROR will result. (See Appendix B.)

If the function name is already a list then the function definition is not retrieved. In this case the list must begin with one of the function atoms given above. Typically, this will be a LAMBDA expression.

Note that on occasion it is convenient to perform a "computed" function call. For example, the atom FUNCTION may have a value which is the name of a function, say CAR. However:

```
(FUNCTION X)
```

will not return the CAR of the list X because EVAL will look for a function by the name of FUNCTION (instead of the value of FUNCTION which is CAR). To properly retrieve the function definition the APPLY* function is used. Thus:

```
LISP
(APPLY* FUNCTION '(A B))
```

will return:

```
A
```

if FUNCTION has the value of CAR; and it will return

```
(B)
```

if FUNCTION has the value of CDR.

7.5 RETRIEVING FUNCTION DEFINITIONS

When LISP encounters an atom, the context specifies whether the atom is used as a variable or as a function. Thus, when you type an atom at the supervisor level, LISP retrieves the value of the atom typed, (i.e. it treats the atom as a variable). To retrieve the function definition, use the GETD function instead:

```
LISP
(GETD 'DOUBLE)

(LAMBDA (N) (+ N N))

LISP
```

8.0 CONDITIONALS AND PREDICATES

8.1.0 RELATIONAL FUNCTIONS

INTER-LISP/65 contains several functions which determine relations among their arguments. The value of these functions is either true, (T), or false, (NIL), depending upon the statement made about the arguments.

8.1.1 THE NUMBER PROPERTY

(# N) determines whether the sole argument is a numeric atom, Example:

```
LISP
(# 1)

T

LISP
(# 'A)

NIL
```

8.1.2 THE GREATER PROPERTY

(> X Y) determines whether the first argument is numerically greater than the second. Non-numeric arguments evaluate to zero. Thus (> 1 'A) returns T.

8.1.3 THE EQ PROPERTY

(EQ A B) determines if the two arguments represent the same internal LISP pointer. Care must be applied here since two s-expressions may be displayed, (by PRINT), identically yet may be distinct lists in memory. (This care is not needed for atoms, since each string is stored only once; therefore, all references to them have the same internal pointer.)

For example —

```
LISP
(EQ '(A B) '(A B))

NIL
```

while the following demonstrates that A and B are EQual.

```
LISP
(SETQ A '(A))

(A)

LISP
(SETQ B A)

(A)

LISP
(EQ A B)

T
```

Also,

```
LISP
(EQ 'A 'A)

T
```

Also,

```
LISP
(EQ 1 1)

T
```

8.1.4 THE ATOM PROPERTY

(ATOM X) returns T if the argument X is not a list. For example:

```
LISP
(ATOM 'X)

T
```

8.1.5 THE MEMBER PROPERTY

(MEMBER X Y) tests the list Y for the first occurrence of the element X. The entire sub-list, for which X is the CAR is returned as the value of MEMBER.

```
LISP
(SETQ FRUITS '(BANANNA ORANGE APPLE LEMON))

  (BANANNA ORANGE APPLE LEMON)

LISP
(MEMBER 'APPLE FRUITS)

  (APPLE LEMON)

LISP
(MEMBER 'COMPUTER FRUITS)

NIL
```

8.1.6 THE COND FUNCTION

It is, perhaps, an understatement to say that COND is the most useful of all of the LISP functions. COND provides all of the power of the if-then-else construct found in other structured languages such as Pascal. The syntax of COND is:

```
(COND (TEST1 RESULT1)
      (TEST2 RESULT2)
            .
            .
            .
      (TESTn RESULTn))
```

COND evaluates each of the test-result pairs sequentially and returns the result which corresponds to the first non-NIL test. If the result is a sequence of expressions then each of those are also evaluated sequentially. COND is equivalent to the BASIC statement:

```
IF TEST1 THEN RESULT1 ELSE IF TEST2 THEN RESULT2 ...
ELSE IF TESTn THEN RESULTn.
```

If a RESULTn expression is absent, the value of the non-NIL TESTn will be returned as the value of the COND. Hence:

```
(COND ((PRINT X) X))
```

is equivalent to:

```
(COND ((PRINT X)))
```

since PRINT returns the value of its argument.

As a detailed example, consider the following COND which returns the decimal value of a given hex digit assigned to the atom HEX (remarks are to the right of the semicolon and are not part of the COND expression):

```
(COND ((# HEX) HEX) ;A NUMBER JUST RETURN THE NUMBER
      ((EQ HEX 'A) 10) ;ELSE IF A RETURN 10
      ((EQ HEX 'B) 11) ;ELSE IF B RETURN 11
      ((EQ HEX 'C) 12) ;ELSE IF C RETURN 12
      ((EQ HEX 'D) 13) ;ELSE IF D RETURN 13
      ((EQ HEX 'E) 14) ;ELSE IF E RETURN 14
      ((EQ HEX 'F) 15) ;ELSE IF F RETURN 15
      (T (PRINT '(NOT A HEX DIGIT)))) ;ELSE SHOW AN ERROR
```

8.2.0 THE SEQUENTIAL EXECUTION FUNCTIONS

8.2.1 THE PROGN FUNCTION

The PROGN function allows evaluation of a sequence of functions. The syntax is:

```
(PROGN X1 X2 X3 ... Xn)
```

Each expression XI is evaluated in order. The value of Xn is returned as the value of PROGN.

Example:

```
(PROGN (PRINT 'FIRST) (PRINT 'SECOND) (PRINT 'THIRD))

FIRST
SECOND
THIRD
THIRD

LISP
```

Note that all three prints are evaluated but that the value of PROGN is 'THIRD (the value of the last argument).

8.2.2 THE PROG FUNCTION

The PROG function allows construction of iterative procedures. The syntax is:

```
(PROG (X1 X2 ... Xn)
        STATEMENT1
        STATEMENT2
        STATEMENT3
        .
        .
        .
        STATEMENTn )
```

Each statement is evaluated sequentially. The statements can be functions or atoms. The atoms serve as labels for looping via the GO function.

When a PROG is evaluated, each of the local variables, (such as XI in the above example), are initialized to NIL. They are restored to their old values when the PROG is exited.

If, at the end of STATEMENTn, the PROG has not encountered a RETURN, then the PROG is terminated and the value of STATEMENTn is the value of the PROG. If a RETURN function is encountered, its argument is returned as the value of the PROG and the PROG is terminated.

The GO function can be used to transfer control to the statement immediately following an atomic label. If the GO attempts to transfer control to a label outside the currently executing PROG, an error will result.

As an example, consider the following function which iterates a given function "N" times:

```
(DEFINEQ REPEAT (LAMBDA (N FN)
                        (PROG (M)
                              (SETQ M N)
                              LOOP
                              (COND ((EQ M 0) (RETURN))
                                    ((APPLY* FN)))
                              (SETQ M (SUB M 1))
                              (GO LOOP)
                        )))
```

8.2.3 THE AND FUNCTION

The syntax of AND is:

(AND X1 X2 ... Xn)

AND evaluates its arguments Xi sequentially until a NIL argument is encountered. In that case the value of AND is NIL. If none of the Xi are NIL then the value of Xn is returned as the value of the AND. In INTER-LISP/65 (AND) returns NIL.

8.2.4 THE OR FUNCTION

The syntax of OR is:

(OR X1 X2 ... Xn)

OR evalutes its arguments Xi, (where i is an integer), sequentially until a non-NIL argument is encountered. In this case the value of OR is the Xi thus encountered. If all of the Xi are NIL then NIL is returned as the value of the OR. In INTER-LISP/65, (OR) returns NIL.

9.0 SYSTEM AND MISCELLANEOUS FUNCTIONS

(NEW)

Reinitializes INTER-LISP/65 and deletes all user defined expressions.

(OBLIST)

Returns the list of currently entered atoms.

(PEEK A)

Returns the contents of the byte at the memory location specified by A. If A is not numeric, the content of location 0 is returned, (a usually meaningless result).

(POKE A N)

Stores the eight bit (in the range 0-255) quantity specified by N in the memory location specified by A. Only the low byte of N is used; thus (POKE A −1) is equivalent to (POKE A 255) while (POKE A 256) is equivalent to (POKE A 0).

(MEM)

Returns the number of unused bytes of memory. MEM forces a garbage collection and is the slowest of all INTER-LISP/65 functions.

(DIR DRIVENO)

Displays the contents of the disk directory on drive specified by DRIVENO, and returns NIL. (DIR) is equivalent to (DIR 1).

(SOUND VOICE PITCH DISTORTION VOLUME)

The SOUND function causes the specified note to be played. Sounds are generated continuously until another SOUND function is evaluated with the same value for VOICE. Legal values for the arguments are as follows:

VOICE: specifies which voice (0-3).

PITCH: specifies the pitch (0-255); high values correspond to low notes.

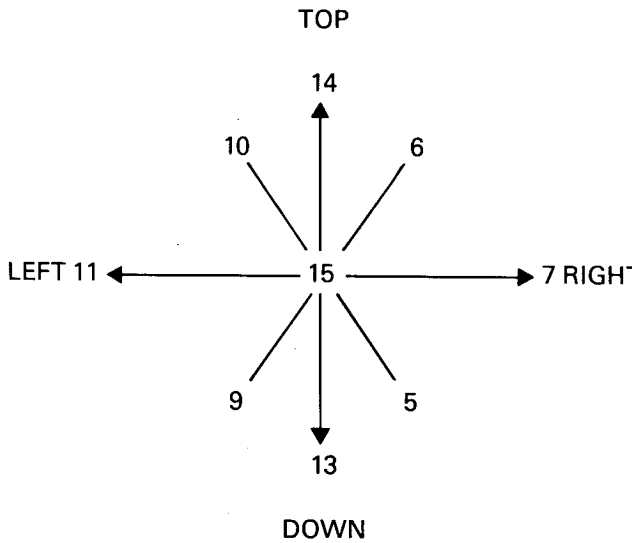
DISTORTION: specifies the distortion level; must be an even number between 0-14.

VOLUME: specifies the volume; must be a number between 1 and 15, with 15 corresponding to the loudest volume.

A table of notes and their associated PITCH values is provided in Appendix G.

(STICK PORT)

The STICK function returns the number corresponding to the deflection of the specified PORT (0-3). The correspondence of STICK values and deflections is as follows:



(STRIG N)

Returns T if the button on joystick N is pressed, otherwise NIL is returned.

(BREAK MESSAGE)

BREAK halts the current evaluation thread and passes control to the error handler (Chapter 13.0). The list MESSAGE (which is not evaluated) is printed at the current output device. The main use of this function is for debugging LISP expressions. By inserting a BREAK you can halt execution and thus examine the current values of atoms.

(XIO CMD IOCB AUX1 AUX2 FILEDEV)

This function invokes a general call to the CIO function resident in the ATARI operating system. IOCB specifies the I/O control block associated with the call. CMD is the command number as follows:

TABLE 9-1
XIO COMMAND NUMBERS

MD	OPERATON	EXAMPLE
3	OPEN	SAME AS OPEN
17	DRAW	SAME AS DRAW
18	FILL	SEE CHAPTER 10
32	RENAME	(XIO 32 1 0 0 'D:OLD,NEW)
33	DELETE	(XIO 33 1 0 0 'D:JUNKFILE)
35	LOCK FILE	(XIO 35 1 0 0 'D:SAFE)
36	UNLOCK FILE	(XIO 36 1 0 0 'D:SAFE)
37	POINT	SAME AS POINT
38	NOTE	SAME AS NOTE
254	FORMAT DISK	(XIO 254 1 0 0 'D2:)

The AUX1 and AUX2 values depend upon the specific function or device which is being accessed, you should refer to the appropriate manual for the specific values required by each command.

10.0 TEXT AND GRAPHICS

10.1 CHANGING SCREEN MODES

(GR MODE)

This expression places the ATARI computer into the graphics mode specified by MODE. The following table summarizes the characteristics of the various graphic modes:

TABLE 10-1

MODE	TYPE	COLUMNS	ROWS (SPLIT)	ROWS (FULL)	COLORS	RAM REQUIRED (BYTES)
0	TEXT	40	N/A	24	2	993
1	TEXT	20	20	24	5	913
2	TEXT	20	10	12	5	261
3	GRAPHICS	40	20	24	4	273
4	GRAPHICS	80	40	48	2	537
5	GRAPHICS	80	40	48	4	1017
6	GRAPHICS	160	80	96	2	2025
7	GRAPHICS	160	80	96	4	3945
8	GRAPHICS	320	160	192	1/2	7900

The split screen modes are obtained by adding 16 to the base mode number. Thus, to enter split screen mode 3 you would set mode to 19 (3+16). To inhibit a screen erase going into a given mode, add 32 to the base mode number. Thus (GR 51) places the ATARI into split screen mode 3 and does not erase the screen.

LISP uses IOCB number 6 for all graphics functions; therefore, you shouldn't open any other device using IOCB number 6.

The GR function returns the system status number as the value of the function. Typically this will be a one (1) indicating a successful open. A table of system error numbers is provided in Appendix E.

10.2 GRAPHIC CONTROL

(COL N)

where N is an integer from 0 to 255 representing a color to be used by the PLOT and DRAW functions. The interpretation of the color number N is dependent upon the particular graphics mode of the ATARI computer. In text mode 0-2, N may be a number from 0 to 255 and determines the character to be displayed and its color. In the graphics modes (3-8) N can be a number within the range of permissable colors for the particular mode (see Table 10-2).

(COL) is equivalent to (COL 0).

(SETCOL REGISTER HUE LUMINANCE)

This function is used to assign the hue and luminance to a given color register specified by REGISTER. The interpretation of, REGISTER is dependent upon the current graphics mode. In the graphics modes REGISTER is associated with the value in the COL function. The interpretation of REGISTER for the text modes is shown in Table 10-2.

Table 10-3 displays the values of HUE and the associated colors. The LUMINANCE is a number between 0 and 7 with 7 corresponding to the brightest luminance.

TABLE 10-2
GRAPHICS MODE CHARACTERISTICS

MODE 0:

COLOR REGISTER	DEFAULT	COL N VALUE	COMMENTS
0	—	0-255	
1	LIGHT BLUE	0-255	CHARACTER LUMINANCE
2	DARK BLUE	0-255	BACKGROUND COLOR
3	—	0-255	
4	BLACK	0-255	BORDER COLOR

TABLE 10-2
GRAPHICS MODE CHARACTERISTICS (Continued)

MODES 1,2:

COLOR REGISTER	DEFAULT	COL N VALUE	COMMENTS
0	ORANGE	0-255	CHARACTER
1	LIGHT GREEN	0-255	CHARACTER
2	DARK BLUE	0-255	CHARACTER
3	RED	0-255	CHARACTER
4	BLACK	0-255	BORDER AND BACKGROUND

MODES 3,5,7:

COLOR REGISTER	DEFAULT	COL N VALUE	COMMENTS
0	ORANGE	1	GRAPHICS POINT
1	LIGHT GREEN	2	GRAPHICS POINT
2	DARK BLUE	3	GRAPHICS POINT
3	—	—	
4	BLACK	4	BACKGROUND

MODES 4,6:

COLOR REGISTER	DEFAULT	COL N VALUE	COMMENTS
0	ORANGE	1	GRAPHICS POINT
1	—	—	
2	—	—	
3	—	—	
4	BLACK	—	BACKGROUND

TABLE 10-2
GRAPHICS MODE CHARACTERISTICS (Continued)

MODE 8:

COLOR REGISTER	DEFAULT	COL N VALUE	COMMENTS
0	—	—	
1	LIGHT GREEN	1	GRAPHICS POINT LUMINANCE
2	DARK BLUE	0	GRAPHICS POINT
3	—	—	
4	BLACK	—	BORDER

TABLE 10-3
SETCOL HUE NUMBERS

GRAY	0	BLUE	8
LIGHT ORANGE	1	LIGHT-BLUE	9
ORANGE	2	TURQUOISE	10
RED-ORANGE	3	GREEN-BLUE	11
PINK	4	GREEN	12
	5	YELLOW-GREEN	13
PURPLE-BLUE	6	ORANGE-GREEN	14
BLUE	7	LIGHT ORANGE	15

(PLOT X Y)

The PLOT function displays the current color (specified by the last evaluated COL function) at the specified graphic location. X and Y are the coordinates of the point to be plotted. The graphic coordinate system is identical to the coordinate system described in the ATARI Basic* programming manual.

(DRAW X Y)

The DRAW function draws line from the current cursor location to the specified point using the current color (specified by the last evaluated COL function) at the specified graphic location. X and Y are the coordinates of the destination point. The graphic coordinate system is identical to the coordinate system described in the ATARI* BASIC programming manual.

(XIO 18 6 0 0 'S:)

This function invokes the special graphics "fill" command resident in the ATARI screen hander (S:). Prior to calling this function you should POKE into location 765 the color register number to be used for the fill operation. To cause the fill to work properly move to the lower right corner of the figure then draw to the upper right, then to the upper left. Finally move to the lower left corner and perform the XIO call.

The following PROG illustrates the technique:

```
(PROG NIL
    (GR 5)
    (COL 3)
    (PLOT 70 45)
    (DRAW 50 10)
    (DRAW 30 10)
    (PLOT 10 45)
    (POKE 765 3)
    (XIO 18 6 0 0 'S:)
    LOOP
    (GO LOOP))
```

10.3 TEXT CONTROL

(PAGE)

Erases the text screen and places cursor in the upper left hand corner of the display. Returns NIL.

(TAB N)

Tabs the cursor to column N on the CRT display. Zero corresponds to the left most column.

10.4 OTHER USEFUL SCREEN COMMANDS

(POKE 752 1)

Supresses screen cursor.

(POKE 752 0)

Enables screen cursor.

(POKE 128 0)

Disables LISP prompt.

(POKE 128 ASCII)

Changes prompt to the character specified by ASCII

(POKE 82 LEFT)

Sets left margin to LEFT.

(POKE 83 RIGHT)

Sets right margin to RIGHT.

(POKE 201 WIDTH)

Sets width of the ATARI tab key.

11.0 LOAD AND SAVE

INTER-LISP/65 contains two internal expressions to LOAD and SAVE s-expressions from an external device. The LOAD expression is similar to the BASIC LOAD command. For example, to LOAD s-expressions from the disk file HANOI just type:

```
(LOAD 'D:HANOI)
```

The file name is quoted because LOAD evalutes its arguments.

The disk will 'whirl' for a few seconds while the s-expressions are read from the disk. After LISP has loaded the file it will return the system status of the function. Typically this will be a one (1) indicating a successful open. A table of system error numbers is provided in Appendix E. In this case LISP responds with:

```
1

LISP
```

You can determine which s-expressions have been loaded by typing

```
D:HANOI
```

Whenever expressions are saved in the file, FILE, a directory list is saved as the first s-expression on the file. When you re-load the first s-expression in the file it is assigned to the atom which represents the file's name.

Saving S-expressions is performed by entering

```
(SAVE PROPS FILE)
```

where PROPS is a list of the expressions to be saved on file FILE. For example, if you have made changes to the HANOI program, (after you have loaded it), entering:

```
(SAVE D:HANOI 'D:HANOI)
```

will save all of the expressions in the list D:HANOI onto the file D:HANOI. Similarly:

(SAVE '(A B C) 'D:STUFF)

will save the expressions A, B and C in file STUFF. The values saved will be both the values bound to the atoms (in this case A,B and C) and any function definitions with the names A,B and C. If you reload STUFF by entering (LOAD 'D:STUFF) then D:STUFF will be set to the list (A B C). Note that the directories, D:HANOI, D:STUFF, etc., are only assigned at the moment of a LOAD, not at the moment of a SAVE.

Note that s-expression can also be stored on a cassette tape using, for example:

(SAVE '(A B C) 'C:)

SAVE can also be used to save s-expression onto the screen! This is useful for obtaining a quick list of the s-expression in a file's directory list.

For example:

(SAVE D:HANOI 'E:)

would display all of the expressions specified by D:HANOI on the ATARI video display.

In a similar manner,

(SAVE D:HANOI 'P:)

yields a listing of all the properties in the list D:HANOI on the printer.

12.0 DEVICE INPUT AND OUTPUT

This chapter provides a description of the device I/O functions which are available to the INTER-LISP/65 programmer. All of these functions behave similarly to their corresponding BASIC counterparts. You are advised to read the appropriate device manual and the ATARI DOS manual prior to using these functions.

12.1 SEQUENTIAL FILE/DEVICE ACCESS

(OPEN IOCB ACCESS FILESPEC)

This function creates an I/O control block for the sequential file or device specified by FILESPEC. Some examples of valid FILESPECs are:

C:	THE CASSETTE
D:FILE	THE FILE BY NAME FILE ON DRIVE NUMBER 1
P:	THE PRINTER
E:	THE KEYBOARD/SCREEN EDITOR
S:	THE SCREEN

ACCESS specifies the type of I/O to be associated with the file or device. Values for ACCESS are as follows:

ACCESS	OPERATION
4	OPEN FOR INPUT
8	OPEN FOR OUTPUT
12	OPEN FOR INPUT AND OUTPUT
6	DISK DIRECTORY READ
9	OPEN FOR APPENDED OUTPUT

(CLOSE IOCB)

Causes the file or device associated with IOCB to be closed. CLOSE returns the system status number as its value. (See Appendix E.)

(IN# IOCB)

Causes all subsequent LISP input to be received from the file or device specified by IOCB. The association of IOCBs with files is performed by the OPEN function described above. IN# returns NIL.

The following LISP evaluations:

```
(IN# 4) (SETQ X (READ)) (SETQ Y (READ))
```

are equivalent to the BASIC commands:

```
INPUT #4;X
INPUT #4;Y
```

```
(PR# IOCB)
```

Causes all subsequent LISP output to be sent to the file or device specified by IOCB. The association of IOCBs with files is performed by the OPEN function described above. PR# returns NIL.

The following LISP evaluations:

```
(PR# 4) (PRINT 'X= ) (PRINT X)
```

are equivalent to the BASIC command:

```
PRINT #4;"X=";X
```

12.2 DISK RANDOM ACCESS FUNCTIONS

(POINT IOCB SECTOR BYTE)

This function can be used to specify the sector and byte at which the next LISP I/O function will read or write. Typically this function is used in conjunction with the note command described next. POINT returns the system status number. (See Appendix E).

(NOTE IOCB)

This function returns the current sector and byte position of the file specified by IOCB. The sector and byte are returned as a dotted pair. For

example, if the current sector and byte were 32 and 120 respectively, then an evaluation of NOTE would return:

```
(32 . 120)
```

As an illustration of the technique required for using POINT and NOTE, consider the following example.

The following PROG creates a data file (D:PEOPLE) and a memory of the locations of the SALLY and HENRY data using the NOTE function to write a directory (or index) file (D:PEOPLE.DIR) :

```
(PROG NIL
  (OPEN 4 12 'D:PEOPLE)
  (PR# 4)
  (SETQ HENRY (NOTE 4))
  (PRINT '(HENRY (AGE 32)))
  (SETQ SALLY (NOTE 4))
  (PRINT '(SALLY '(AGE 28)))
  (CLOSE 4)
  (OPEN 5 12 'D:PEOPLE.DIR)
  (PR# 5)
  (PRINT HENRY)
  (PRINT SALLY)
  (CLOSE 5)
  (PR# 0))
```

The following PROG reads the data file created by the previous PROG using the POINT function:

```
(PROG NIL
  (OPEN 4 4 'D:PEOPLE.DIR)
  (IN #4)
  (SETQ HENRY (READ))
  (SETQ SALLY (READ))
  (CLOSE 4)
  (OPEN 5 12 'D:PEOPLE)
  (IN# 5)
  (POINT 5 (CAR SALLY) (CDR SALLY))
  (PRINT (READ))
  (POINT 5 (CAR HENRY) (CDR HENRY))
  (PRINT (READ))
  (CLOSE 5)
  (IN# 0))
```

13.0 ERROR HANDLING

When LISP encounters an error, an error message is displayed and control is passed to the error handler. Error messages are of the form:

```
4 ERR          <---------- the type of error
A              <---------- the current value
(CDR X)        <---------- the expression which encountered the
                           error
ERR>           <---------- the error handler prompt
```

In this example, A was passed as the current binding of the variable X in the (CDR X) expression. Since A (the literal) is not a list, an error has occurred. All of the LISP error messages can be found in Appendix B. Note that the error handler uses a different prompt character. Its function is to remind you that the LISP supervisor still has "unfinished" business residing on the evaluation stack.

Once within the error handler, you may proceed in several ways by using one of the error processing expressions described below:

(RESET)

You can use this command to "abort" the current evaluation thread. Control is immediately passed back to the LISP supervisor (not to be confused with the RESET key).

(RETURN X)

You can use this expression to continue evaluation of the current evaluation thread. The single argument is evaluated and used as the value of the expression which produced the error. Control is then passed back to the supervisor which continues evaluation from the point which produced the error.

(BAKTRACE)

This command displays the sequence of evaluations which lead to the error. Each evaluation displayed is preceded by two "plus" signs, as follows:

```
LISP
(BAKTRACE)

    + + (BAKTRACE)
    + + (CDR X)
    + + (PROGN (SETQ Y (CDR X)))
    NIL
```

Additionally, while within the error handler you may perform any operation which would normally be available from the supervisor. This includes defining new functions, loading files, setting variables, etc.

14.0 SAMPLE PROGRAMS

Your INTER-LISP/65 system disk is supplied with sample programs to demonstrate some of the capabilities of LISP.

14.1 THE TOWERS OF HANOI

This program graphically displays the solution of the famous Towers of Hanoi puzzle and provides an excellent example of the power of recursive programming. The object of the puzzle is to move all of the disks from their starting post to the third post under the restrictions that only one disk may be moved at a time, and that a disk may only be placed upon a larger disk.

Running The Program

Power up LISP and enter

```
(LOAD 'D:HANOI)
```

After a few moments LISP should respond with

```
1
```

which indicates a successful load.

Now type,

```
(HANOI)
```

Your ATARI should now show its intelligence by solving the Towers of Hanoi puzzle with methodical speed. After the "program" is finished you can return to text mode by entering (GR 0).

14.2 THE LISP EDITOR

Your INTER-LISP/65 system disk contains a powerful LISP expression editor to aid in the creation and manipulation of LISP programs. The editor contains a general "pretty printer" algorithm which is invaluable during the debugging of LISP functions.

USING THE EDITOR:

With INTER-LISP/65 running, enter:

(LOAD 'D:EDIT)

INTER-LISP/65 will respond with:

1

after the editor is loaded. To enter the editor, type:

(EDIT EXPRESSION)

where EXPRESSION is the list you wish to edit. Note that to edit a function you can enter (EDIT (GETD 'FUNCTION)). You can also enter the editor by just typing:

(EDIT)

in which case, you can use the E and EF commands desrcibed below to specify the s-expression to be edited.

Once in the editor you may use the following commands:

A - Advances to next element of list

D - Go down to the first element of current element (must be a list)

B - Backup to previous element

R s1 - Replaces current element by s1.

X s1 s2 - Exchanges all occurences of s1 by s2.

DEL - Deletes current element

I S1 - Inserts S1 after current element

LI - Places one level of parentheses about current element

RE - Removes one level of parentheses from current element

G - Groups items into a list. Enter "A" command until last element to be grouped is displayed. List is then closed by entering another "G".

PRE s1 - Inserts s1 in front of current item

PP - Pretty prints the entire list being edited

P - Pretty prints current expression and all items following it

T - Returns to top element of list

EX - Exits editor

C - Displays disk catalog

E s1 - Edits the list specified by s1.

EF s1 - Edit the function specified s1.

S - Save expressions in list s1 on file s2 disk. The editor prompts for the property list to save and file name

L - load expressions from disk file s1. The editor prompts for the file name

PU S1 - Deletes file S1 from the disk.

LOCK S1 - Locks file S1

UNL S1 - Unlocks file S1.

H - Displays edit commands

The following text gives a detailed example of use of the editor program. User input is indicated by a terminating [CR] symbol. Explanatory notes are surrounded by brackets [like this]. All other text (including the prompts) is output from LISP or the editor.

[User loads editor]

```
LISP
(LOAD 'EDITOR) [CR]

1
```

[user enters editor]

```
LISP
(EDIT) [CR]

NIL
```

[since no argument was supplied the editor responds with NIL. User now decides to write a factorial function. Note that "Edit> " is the editor's prompt]

```
Edit> EF ! [CR]

Creating--> !
```

[! is currently an udefined function so the editor indicates that the function ! is being created.]

```
Input> (LAMBDA (C)) [CR]

LAMBDA
```

[the user has begun the defintion of ! by indicating it is a lambda expression with an argument of C. Note that the prompt is "Input> " while in the create mode].

```
Edit> A [CR]

(C)
```

[move across to the (C)]

```
Edit> I [CR]
```

[goes to insert mode. the editor will now insert the next s-expression read after the current item].

```
?(COND ((EQ N 0) 1)) [CR]

(COND ((EQ N 0) 1))
```

[inserts a COND expression. Note that the editor prompts with a question mark, "?", since the expression to insert was not supplied.]

```
Edit> PP [CR]
```

[and pretty prints to see how it looks]

```
(LAMBDA (C)
   (COND ((EQ N 0)
          1)))

Edit> T [CR]
```

[goes to top of the expression]

```
LAMBDA

Edit> A [CR]
```

[moves across]

```
(C)

Edit> R (N) [CR]
```

[replaces incorrect argument]

```
(N)

Edit> A [CR]
```

```
(COND ((EQ N 0) 1))

Edit> D [CR]

COND

Edit> A [CR]

((EQ N 0) 1)

Edit> I [CR]
```

[to insert rest of factorial]

```
?(T (* N (! (SUB N)))) [CR]

(T (* N (! (SUB N))))

Edit> PP [CR]

(LAMBDA (N)
    (COND ((EQ N 0)
           1)
          (T (* N (! (SUB N)))))
```

[everthing looks fine now]

```
Edit> S

Properties> '(!)

File> D:FACTOR [CR]
```

[decides to save the single factorial property on file named FACTOR. Note that the editor displays "Properties> " to prompt for the properties to be saved and displays "File> " to prompt for the file name.]

```
Saved
```

[the editor responds with the fact that the file has been saved]

Edit> EX [CR]

EXIT

[the user exits the edit program and returns to LISP>

LISP
(! 4) [CR]

24

[exits the editor and tests the factorial !]

.
.
.

[some time later user re-enters editor and decides to write more functions]

LISP

(EDIT) [CR]

NIL

Edit> L

File> D:FACTOR [CR]

Loaded--> (!)

[editor indicates successful load and indicates which properties have been loaded by displaying the file's directory list]

Edit> EF ABS [CR]

Creating--> ABS

Input> (LAMBDA (N)) [CR]

```
    LAMBDA

    Edit> A [CR]

    (N)
```

[moves across]

```
    Edit> I [CR]

    ?(COND ((> N 0) N) (T (SUB 0 N)))[CR]

    (COND ((> N 0) N) (T (SUB 0 N)))

    Edit> PP [CR]

    (LAMBDA (N)
       (COND ((> N 0)
              N)
             (T (SUB 0 N))))
```

[user decides to replace the formal argument name N by X]

```
    Edit> T [CR]

    LAMBDA
```

[goes to top since the Xchange command replaces only occurrences starting at current position in the s-expression up to end of list]

```
    Edit> X N X [CR]
```

[replaces all ocurrences of N by X]

```
    LAMBDA

    Edit> PP [CR]
```

```
(LAMBDA (X)
   (COND ((> X 0)
          X)
         (T (SUB 0 X))))
```

[DONE!]

Edit> E D:FACTOR [CR]

[decides to add ABS to directory list of FACTOR. Note that this does not affect the file only the in-core list]

!

Edit> PP [CR]

(!)

Edit> I ABS [CR]

ABS

Edit> PP [CR]

(! ABS)

Edit> S

Properties> D:FACTOR

File> D:MATHEXP [CR]

[all of the s-expressions in the list D:FACTOR, i.e. ! and ABS, will be saved on the file MATHEXP. Again note that only the file MATHEXP is affected even though the modified directory list of the file FACTOR is used as the properties of the S command]

Saved

Edit> EX [CR]

EXIT

```
LISP
(ABS -1) [CR]

1

LISP
```

[END OF SAMPLE EDIT SESSION]<

14.3 DOCTOR

This program is a translation of the DOCTOR program as described in Winston and Horn on page 219. This file also contains the powerful string matching function as described on page 230 of Winston and Horn. This version of the program is rather primitive but you should have fun exercising your LISP programming skills to create more elaborate (and intelligent) conversational programs.

Enter the command:

```
(LOAD 'D:DOCTOR)
```

After the file has loaded entering

```
(DOCTOR)
```

will start the program. Note that all user input to the program must be in the form of lists.

14:4 THE RPN CALCULATOR

This function can be found in the file CALCULATOR. To load the program type:

```
(LOAD 'D:CALCULATOR)
```

The function is evaluated by typing:

```
(CALC)
```

The program now enters entry mode by displaying the prompt:

ENTER>

If you enter a number it is pushed onto the stack (which is displayed after each entry). If you enter an operator then the operation is performed on either the top two numbers for binary operators or just the top number for unary operators. The operators recognized by CALC are:

BINARY OPERATIONS

+	add top two stack numbers
–	subtract top two stack numbers
*	multiply top two stack numbers
/	divide top two stack numbers
p	power of the top two stack numbers

UNARY OPERATIONS

e	exponential
l	natural log
s	square root
n	negative
d	duplicate top of stack
q	quit program

14.5 MACLISP FUNCTION SIMULATOR

The file MACLISP contains INTER-LISP/65 functions which will simulate the MACLISP functions:

FUNCALL, DEFPROP, GET, REMPROP, MAPCAR, DEFUN, REVERSE, APPLY, SUBST, NCONC and DELETE.

These functions will be available after loading the file MACLISP. Note that these functions do not behave altogether the same as their corresponding MACLISP counterparts since they are just simulators.

The property list is simulated by attaching an ordinary LISP list to an atom's value cell. All properties which are not function types will be appended onto this ordinary LISP list. If one of the property access functions detects a MACLISP function type of EXPR, FEXPR, or MACRO the proper INTER-LISP/65 function list is constructed and inserted in the atom's function definition cell.

14.6 THE CLISP FUNCTION

The file CLISP contains a function which will convert an s-expression containing algebraic arithmetic expression to Cambridge prefix notation used by INTER-LISP/65. For example the expression:

```
(A * B)
```

will be converted to:

```
(* A B)
```

while

```
(X = (A - B))
```

will be converted to:

```
(SETQ X (SUB A B))
```

Other exprssion are converted similarly.

The syntax of the function is

```
(CLISP (QUOTE FUNCTIONNAME))
```

where FUNCTIONNAME is the name of the function to be converted. CLISP will retrieve the function definition and redefine the function to be the converted function.

As an example if ! is defined as follows:

```
(DEFINEQ ! (LAMBDA (N)
                   (COND ((EQ N 0) 1)
                         (T (N * (! (N - 1))))
          ))
```

then

```
(CLISP '!)
```

will convert ! as if it had been defined as follows:

```
(DEFINEQ ! (LAMBDA (N)
                   (COND ((EQ N 0) 1)
                         (T (* N (! (SUB N 1))))
          ))
```

14.7 LISP LIGHTS

This program demonstrates the use of high resolution graphics in LISP.

To load the program type:

```
(LOAD 'D:LIGHTS)
```

To run the program enter:

```
(LIGHTS)
```

The program runs forever, to terminate execution press control-B twice in quick succession.

APPENDIX A
INTER-LISP/65 COMMAND SUMMARY

(AND X1 X2 ...XN)

The logical 'AND' of the operands, X1, X2 ...; if any of the XI are NIL, then NIL is returned. Otherwise, XN is returned.

(APPEND LIST1 LIST2 LIST3 ...)

Concatenates the lists specified by its arguments and returns the concatenated list as its value.

(APPLY* FUNCTION ARGUMENT1 ARGUMENT2 ...)

Applies the function specified by the value of FUNCTION to the arguments, ARGUMENT1, ARGUMENT2 ... Thus if the value of FUNCTION is '+' then the following two expressions are equivalent:

```
(+ 2 7)
```

and

```
(APPLY* FUNCTION 2 7)
```

(ASSOC A L)

Searches the second argument L for the sublist whose CAR matches A; if a match is found, then the sublist is returned. Otherwise, NIL is returned.

(ATOM X)

Returns T if X is not a list. Examples:

(ATOM 'A) returns T

(ATOM '(A B)) returns NIL.

(BAKTRACE)

Displays the sequence of evaluations leading to the current evaluation. Returns NIL.

(BREAK MESSAGE)

Prints the unevaluated MESSAGE and stops execution of the current expression. Control is passed to the error handler.

(CAR X)

Returns the first element of list X. If X is a non-NIL atom, an error will result. (CAR NIL) returns NIL.

(CDR X)

Returns list obtained by omitting the first element of the list X. If X is not a list, an error will result. (CDR NIL) returns NIL.

(CLOSE IOCB)

Closes the disk file specified by FILE, and returns the system status number as its value.

(COL X)

Sets current graphic color to the color register specified by X. If X is not numeric, the color is set to color register zero. (COL) is equivalent to (COL 0).

(COND (P1 V1) (P2 V2) ... (PN VN))

Evaluates the 'predicates' of each argument and returns the corresponding value for the first non-NIL predicate, (i.e. it returns V1 if P1 is non-NIL), else if P2 is non-NIL it returns V2, etc. Each VN is an "implicit" PROGN. This expression is equivalent to the BASIC statement:

IF P1 THEN V1, ELSE IF P2 THEN V2 ...

ELSE IF PN THEN VN

(CONS X Y)

Returns the dotted pair whose CAR is X and whose CDR is Y.

(DEFINEQ FUNCTION FUNCTIONDEF)

Establishes FUNCTIONDEF as the function associated with the atom FUNCTION and returns FUNCTIONDEF.

(DEFINE FUNCTION FUNCTIONDEF)

Similar to DEFINEQ except FUNCTION is evaluated. Returns FUNCTIONDEF as its value.

(DRAW X Y)

Draws from current cursor location to the point (x , y) specified by X and Y; returns NIL.

(EQ X Y)

Returns T if X and Y are the same atom, (either numeric or symbolic), or X and Y point to the same list. (Two lists may PRINT equally but reside at distinct memory locations.) (EQ X) is equivalent to (NOT X).

(EVAL X)

Returns the value of X.

(EXP N) Returns e (2.718289...) to the N th power.

(GETD NAME)

Returns the function list which is attached to the atom specified by name. NAME is evaluated. If NAME is not an atom then NIL is returned as the value of GETD.

(GO LABEL)

Used within a PROG to transfer control to the expression following LABEL. LABEL is not evaluated and it must be symbolic. GO is only valid inside a PROG and it cannot be used to branch to a label which is outside of the currently executing PROG.

(GR MODE)

Sets screen to graphic mode specified by MODE. Returns the number equal to the system status of the mode change. (A 1 indicates success.)

(INT N)

Returns the nearest integer of N by rounding N. Thus −.5 returns −1 while −.4 returns 0 and .5 returns 1.

(IN# IOCB)

Causes all subsequent input to be received from the file or device associated with IOCB. Returns NIL.

(LAMBDA (X1 ... XN) BODY)

The 'lambda' expression or user defined procedure LAMBDA is evaluated by passing the evaluated argument list (X1 ...XN) to the procedure body BODY.

(LAST X)

Returns the last CDR of the expression X. If X is an atom or NIL then NIL is returned.

(LENGTH X)

Returns the number of elements in the list X. If X is an atom or NIL then zero is returned.

(LIST A B C ...)

Returns the list whose elements are A, B, C ...

(LOAD FILE)

Accepts s-expressions from file FILE until a NIL expression is read. Returns the system status number as its value.

(LOG N)

Returns the natural logarithm of N.

(MACRO (A) BODY)

The MACRO expression: A MACRO is evaluated by passing the entire calling expression as the value of the single argument A. The value of the MACRO is the evaluation of the results of the evaluation of BODY. In somewhat loose terminology MACRO behaves like "EVAL squared". The main use of MACRO is to construct code and then execute the code.

(MEM)

Returns the number of bytes of free memory. (MEM) causes a garbage collection, consequently it should not be used too liberally in functions.

(MEMBER X Y)

Searches the list Y for an element which matches X and returns the sublist of Y for which X is the CAR if a match is found.

(NEW)

Erases all user defined lists from memory. NEW reinitializes LISP and thus has no value.

NIL

The empty list, logical 'false' or end of list. NIL is an atom.

(NLAMBDA (A) BODY)

The 'NLAMBDA' procedure. NLAMBDA is similar to LAMBDA except that the single argument A is not evaluated before passing it to BODY.

(NOT X)

See (EQ X Y).

(OBLIST)

Returns the list of currently entered atoms.

(OPEN IOCB ACCESS FILESPEC)

Creates an I/O control block (specified by IOCB) for the file or device specified by FILESPEC sequential file FILE, (i.e. sets up a file or device for subsequent I/O operations). It returns the system status number as its value.

(OR X1 X2 ... XN)

Returns the value of the first non-NIL XI. If all XI are NIL, then NIL is returned.

(PACK L)

Returns the atom whose characters are the concatenation of the atoms in the list L. Example:

```
(PACK '(A T O M))
```

returns

```
ATOM.
```

(PAGE)

Clears the CRT screen. Returns NIL.

(PEEK A)

Returns the contents of memory location A.

(PLOT X Y)

Plots at the graphics point at coordinates (X, Y) in the current color. Returns the system status number as its value.

(POKE X Y)

Sets the memory location X to the number Y. Returns NIL.

(PRINT X)

Prints and returns the value of X.

(PRIN1 X)

Prints the expression X with no carriage return. If X is a quoted string, e.g. "ABC", then it is printed without the quotes. Returns the value of X.

(PRIN2 X)

Prints the expression X with no carriage return. Returns the value of X.

(PROG (X1 X2 ... XN) S1 S2 ...SN)

Evaluates each of the "statements" SI sequentially. Each of the arguments XI is set to NIL prior to execution of the PROG body and are restored to their old values upon exiting the PROG. PROG returns the value of the SI evaluated before an exiting statement is encountered.

(PROGN X1 X2 ... XN)

Evaluates each "statement" XI sequentially and returns the value of XN.

(PR# IOCB)

Causes all subsequent output to be directed to the device or file associated with IOCB. Returns NIL.

(QUOTE X)

The literal X. An equivalent form on input is 'X.

(READ)

Accepts an S-expression from the current source of input and returns the S-expression as its value.

(READA)

Accepts one atomic expression from the current source of input. Returns the atom as its value.

(READC)

Accepts one character from the current source of input and returns the single character entered as the value of the READC. READC flushes the LISP input buffer so that the "A" in the following will be ignored:

```
LISP
(READC)A <CR>
```

(RPLACA X Y)

Replaces the CAR of X with Y and returns the modified list as its value.

(RPLACD X Y)

Replaces the CDR of X with Y and returns the modified list as its value.

(RESET)

Cancels the current thread of expressions being evaluated. Returns control to the LISP supervisor. RESET has no value.

(RETURN X)

Causes an exit from the current PROG with the value of X. RETURN can also be used to return from the error handler. In this case (RETURN X) evaluates X and returns it as the value of the expression which produced the error.

(SAVE PROPS FILE)

Saves the expressions specified by the list PROPS on the file or device FILE. Returns the sytem status number as its value.

(SETCOL REGISTER HUE LUMINANCE)

Sets the hue and luminance of the specified color register. Returns NIL.

(SET X Y)

Sets the value of X, (which must be a legal variable), to the value of Y. Returns the value of Y.

(SETQ X Y)

Sets the "variable" X to the value of Y and returns the value of Y.

(SOUND VOICE PITCH DISTORTION VOLUME)

Sets the pitch, distortion and volume to be played by the specified voice. Returns NIL.

(STICK N)

Returns the number corresponding to the current deflection of the joystick specified by N.

(STRIG N)

Returns T if the button on joystick N is depressed; NIL otherwise.

(SUB X Y)

Returns the number X − Y.

T

Logical "TRUE".

(TAB N)

Tabs horizontally N spaces on CRT screen. Returns NIL. (TAB) is equivalent to (TAB 0).

(TERPRI)

Prints a carriage return and returns NIL.

(UNPACK X)

Returns the list whose elements are the characters of the atom X. Example:

(UNPACK 'ATOM)

returns

(A T O M).

(XIO COMMAND IOCB AUX1 AUX2 FILESPEC)

Performs a general call to the ATARI operating system function CIO (Central I/O utility). The function to be performed is specified by COMMAND. Returns the system status number as its value.

(# N)

Returns T if N is a numeric atom.

(+ X Y)

Returns the number X+Y.

(> X Y)

Returns T if X is greater than Y.

(* X Y)

Returns the product of X and Y.

(/ X Y)

Returns the number X divided by Y.

(@ X I)

Returns the Ith CDR of the list X. This expression is equivalent to the expression:

(CDR (CDR (CDR ... (CDR X) ...)))

!<---- I TIMES ------>!

Note that (@ X 0) returns the value of X. This expression is useful for array processing. If X is the list:

(1 2 3 4)

then (CAR (@ X I)) will return the number I+1. If X is the list

((1 2 3) (4 5 6) (7 8 9))

then (CAR (@ (CAR (@ X I)) J)) returns the (I,J) element of the 3 x 3 matrix represented by X.

APPENDIX B
LISP ERRORS BY ERROR NUMBER

ERROR 1: "EVALUATION ERROR"

LISP has encountered an undefined function. A somewhat trivial example which would produce this error is:

```
LISP
(T)
```

since T is not a function.

ERROR 2: "SYSTEM ERROR"

A bad system error has occurred. If you have, unsaved S-expressions it is better if you save them on a scratch disk and re-initialize LISP.

ERROR 3: "MEM FULL"

LISP has used all available memory.

ERROR 4: "LIST ERROR"

LISP expects a list and has not encountered one.

ERROR 5: "UNUSED"

This error message is not used.

ERROR 6: "ATOMIC ERROR"

LISP expects an atomic expression and has not encountered one. This error is also displayed for invalid assignments as in

```
(SETQ T 5)
```

ERROR 7: "UNUSED"

This error message is not used.

ERROR 8: "BAD SUBR ERROR"

LISP has encountered a SUBR or NSUBR whose CDR is not of the correct form.

ERROR 9: "FORMAL ARGUMENT ERROR"

LISP has encountered an illegal argument to LAMBDA, NLAMBDA or MACRO. Examples are:

```
(LAMBDA (T) (SETQ T NIL))
          !---------- T is reserved.

(NLAMBDA ((N)) (SETQ N 1))
            !-------- (N) is not atomic
```

This error is considered fatal and causes the current evaluation to be aborted; control is passed back to the LISP supervisor, NOT the error handler.

ERROR 10: "UNUSED"

This error message is not used.

ERROR 11: "UNUSED"

This error message is not used.

ERROR 12: "NO LABEL ERROR"

A GO expression references a label which is not defined within the currently executing PROG.

ERROR 13: "REPLACE ERROR"

A RPLACA or RPLACD has attempted to modify a non list structure.

ERROR 14: "ARITHMETIC OVERFLOW"

Evaluation of an arithmetic expression has resulted in a number larger than the largest number which LISP can store. The most frequent cause of this error is an attempt to divide by zero.

ERROR 15: "STACK FULL ERROR"

LISP has used all available memory during evaluation of a function. If this error occurs during an extensively recursive function then it may be impossible to repeat other extensively recursive functions even though (MEM) may indicate a large pool of free memory. You should still be able to execute any of the predefined LISP functions. To fully recover from this error you need to save any unsaved expressions on a disk, re-initialize LISP, and then reload your files.

Certain LISP functions return the system status error numbers, a description of these errors can be found in Appendix E.

APPENDIX C
SOME UTILITY LISP FUNCTIONS

The following utilities are provided on your LISP system disk in file UTILS.

(VTAB R)

Tabs to row R

(LAMBDA (R) (POKE 656 R))

(GOXY R C)

Positions cursor at row R and column C

(LAMBDA (R C) (PROGN (VTAB R) (TAB C)))

(REFLECT)

Prints charaters upside down

(LAMBDA NIL (POKE 755 4))

(NORMAL)

Prints character right side up

(LAMBDA NIL (POKE 755 2))

(HLIN X1 X2 Y)

Draws a horizontal line from (x1 ,y) to (x2 ,y)

(LAMBDA (X1 X2 Y) (PROGN (PLOT X1 Y) (DRAW X2 Y)))

(VLIN Y1 Y2 X)

Draws a vertical line from (x, y1) to (x, y2)

(LAMBDA (Y1 Y2 X) (PROGN (PLOT X Y1) (DRAW X Y2)))

(MOD X Y)

Returns the number x modulo y

(LAMBDA (X Y) (SUB X (* (INT (SUB (/ X Y) .5)) Y)))

(RAN RANGE)

Returns random number between 0 and RANGE

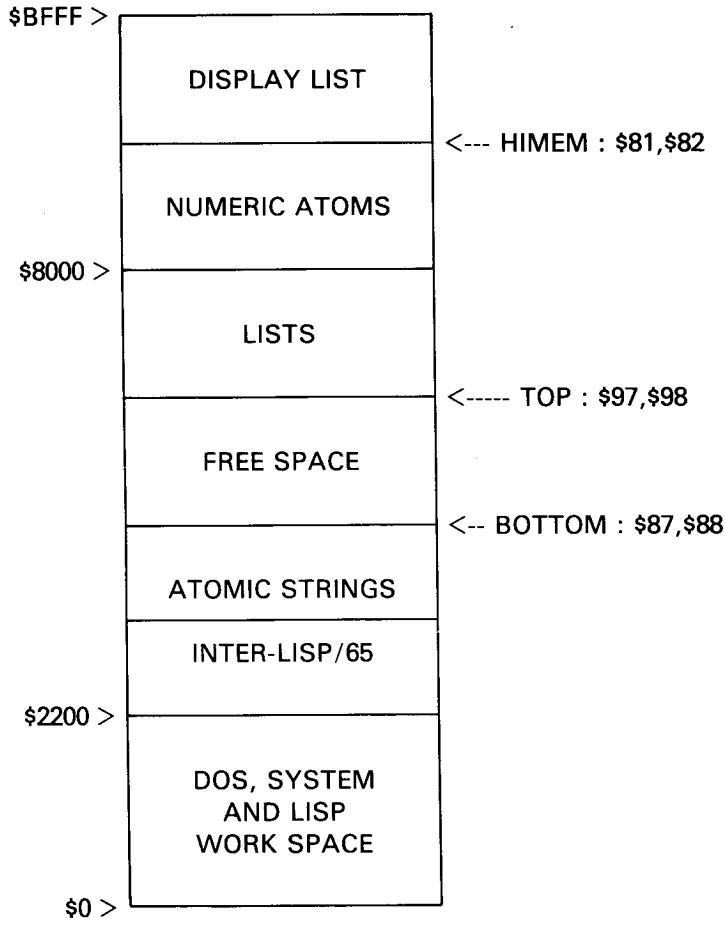
(LAMBDA (RANGE) (* (/ (+ (PEEK 53770) (* 256 (PEEK 53770))) 65536) RANGE))

(TIME)

Prints the time since power on in the format HH:MM:SS and returns the number of seconds since power on.

APPENDIX D
INTER-LISP/65 MEMORY MAP

All adresses are in hexadecimal.



APPENDIX E
ATARI SYSTEM ERROR MESSAGES

ERROR #	MEANING
1	SUCCESSFUL COMPLETION
128	BREAK ABORT: User hit break key during I/O
129	IOCB is already open.
130	Nonexistent device.
131	IOCB open for write only.
132	Invalid command for device.
133	File or device not open.
134	Illegal device number
135	IOCB open for read only.
136	END OF FILE detected.
137	Truncated record.
138	Device time out.
139	Device NAK: grabage at serial port or bad disk drive.
140	Serial bus framing error.
141	Cursor out of range
142	Serial bus data frame overrun.
143	Serial bus checksum error.

144 Device done error.

145 Read after write compare error.

146 Function not implemented.

147 Insufficient ram for screen mode.

160 Drive number error.

161 Too many files open

162 Disk full

163 Unrecoverable system I/O error.

164 File number mismatch: (trashed disk)

165 File name error.

166 POINT commad data length error

167 File locked.

168 Special command invalid.

169 Disk directory full (64 files).

170 File not found.

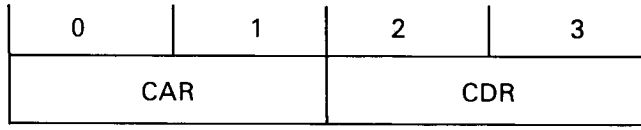
171 POINT command invalid.

APPENDIX F
OVERVIEW OF LISP STORAGE

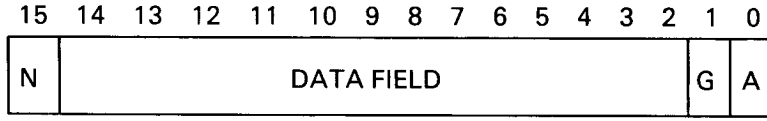
INTER-LISP/65 stores data in three distinct memory areas, according to the data type. Atomic strings or "print names" are stored starting at low memory in increasing order; numbers are stored starting at address $8000 in increasing order; and lists are stored in the heap (which starts at $7FFF) in decreasing order. A memory full error occurs when the print names collide with the heap. This storage arrangement is shown in Appendix D. The following discussion provides the particulars of the storage of the various types.

Print names are stored as zero byte terminated strings with the sign bit of each character equal to zero.

Each list element occupies four contiguous bytes in the list area of memory. Two bytes represent the CAR and two bytes represent the CDR, thus:



The format of the two byte CAR or CDR cell is:



where N = NUMERIC BIT

G = GARBAGE COLLECTION BIT

A = ATOM BIT

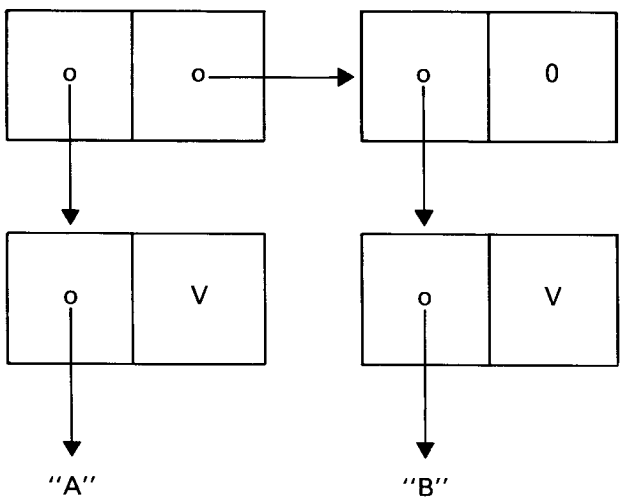
If the numeric bit is set then the data field points to the address where the numeric value is stored. Numeric atoms require eight bytes of storage for the actual numeric value of the atom.

If the atom bit is set, then the data field points to a cell whose CAR points to the print name of the atom and whose CDR points to the value of the atom.

A list which PRINTs as

(A B)

is represented in memory as



In this diagram V represents a pointer to the "property" list of the atoms A and B. Each property list requires a minimum of eight bytes of storage. Since this diagram also represents the manner in which new atoms are entered into the atomic symbol table, it should be clear that defining a new atom requires eight bytes of memory in addition to the length of the atom's print name. Thus defining the atom ATARI requires eight plus eight plus six or twenty-two bytes of storage. Assigning a value or function definition to an atom ,of course, requires additional memory.

APPENDIX G
NOTES AND PITCH VALUES FOR SOUND FUNCTION

C	29
B	31
A#	33
A	35
G#	37
G	40
F#	42
F	45
E	47
D#	50
D	53
C#	57
C	60
B	64
A#	68
A	72
G#	76
G	81
F#	85
F	91
E	96
D#	102
D	108
C#	114
MIDDLE C	121
B	128
A#	136
A	144
G#	153
G	162
F#	173
F	182
E	193
D#	204
D	217
C#	230
C	243

*ATARI is a registered trademark of ATARI COMPUTER INC.

*IBM 7090 is a registered trademark of IBM INC.

Interlisp/65 V2.2

Documentation Addendum, July, 1982

Version 2.2 corrects an error in the (GR) function which did not allow variable arguments as in: (GR X).

Also, user memory has been increased by 1152 bytes.

Documentation corrections for V2.2:

Page 53: Change catalog command description to read "C S1 - displays catalog on drive specified by S1".

Change UNL to UNLOCK.

Page 56: Change all occurrences of (SUB N) to (SUB N 1).

Page 60: Section 14.4; The function CALC is on file CALC. Change all occurrences of CALCULATOR to CALC.

Page 80: Change the "$22ØØ" to read "$1D8Ø".

Additional demonstration programs:

* The random access functions on page 48 are included in the file RANDOM. The function WRITE_PEOPLE creates the file. READ_PEOPLE will read the file.

* File BOX contains a GTIA demonstration; after loading type (BOXER) to execute the demo. Terminate with Control B.

* File GTIA contains two more GTIA demonstrations. Typing (GTIA1 9) or (GTIA1 11) will demonstrate GTIA modes 9 and 11. Terminate with Control B.

SOFTWARE OPPORTUNITY

Datasoft is offering a unique opportunity to software authors. Send us your program or program concept for evaluation. If it is accepted for publication we will enter into a marketing agreement to sell your product through our Domestic and International distribution channels.

And the opportunity does not end there. We offer you something few other publishers can. We call it "Product Roll-Over". We have the capability to take a program and transfer it to other popular microcomputers (Atari, Apple, TRS-80 and NEC). We can even plan distribution on machines still in development that we feel will be a large part of tomorrow's market.

Datasoft works with several large microcomputer manufacturers on new and exciting projects. We are involved with many "famous-name" companies entering our industry for the first time.

So get the most exposure for your programming efforts. Write us for a free programmer's package and get a start on a rewarding future. It's waiting for you today.

Send your name, address and phone number to:

Datasoft Inc.®
Programmer's Package
9421 Winnetka Avenue
Chatsworth, CA 91311

Or call us at (213) 701-5161 and ask for our Software Manager.