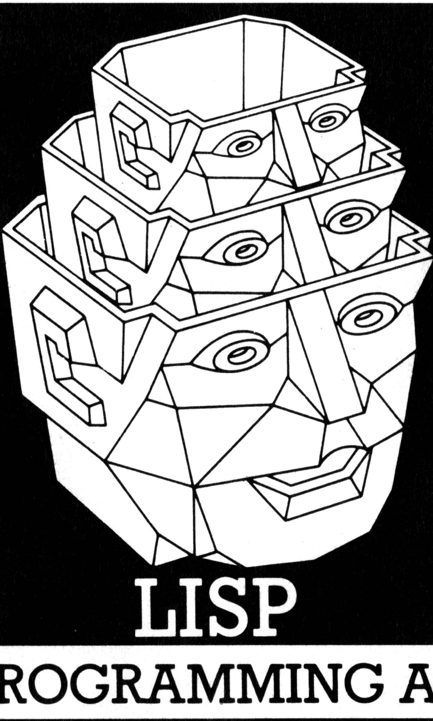
ENTERPRISE

PROGRAMS



LISP

PROGRAMMING AID

CONTENTS

IS-LISP for the Enterprise version 1

© 1985 Intelligent Software Ltd

Programming: M. Richer

Manual Writing R. Hurley, J. Hurley, R.D. Virgo

PREFACE

Although LISP (LIST PROCESSING) has been around for twenty or more years it is still highly popular in the fields of 'Artificial Intelligence' and 'Expert Systems', where its flexibility and expandability have made it an ideal choice.

This manual is not aimed to be a complete tutorial on the language but explains the implementation of IS-LISP on the Enterprise computer and looks closely at how it interacts with the user and the EXOS operating system. It will provide the complete beginner with a short introduction to programming in the language, but will assume that the reader is familiar with the operating of the machine and has at least a limited knowledge of programming in BASIC.

Because LISP is an interactive language, with each function being evaluated as and when it is entered, it is a simple matter to experiment with the various functions. The reader is advised to test the examples contained within the manual whenever possible.

Books on LISP

There are many good books on programming in LISP. The following are especially recommended:

"LISP on the BBC microcomputer" by A. Norman & G. Cattell, gives a good introduction to LISP programming. It describes BBC LISP which is very similar to IS-LISP.

"LISP" by P. Winston & B. Horn, describes a different dialect of LISP, but explains some advanced features such as macros.

Chapter 1
GETTING STARTED

To start programming in IS-LISP, plug your cartridge into the port on the left-hand side of the Enterprise microcomputer and switch on. If your computer is already turned on, press the Reset button twice. The normal cold start procedure will take place, with the machine performing check on the sections of memory, and a message as shown below will appear on the screen.

You are now ready to start programming and statements are entered via the keyboard in the same way as when using IS-BASIC. All communication between the user and the LISP interpreter is done through the 'editor' which means that the cursor keys and joystick can be used to move the cursor around the screen in order to enter, amend or delete text.

When a line of program has been correctly typed, pressing return key will cause 'EVAL' to come into operation and, providing it is syntactically correct, it will be evaluated and the result displayed on the screen.

For example, to add two numbers together we would simply type (PLUS 3 4) and the computer would reply with 7.

At this point it is important to remember that each line of LISP program is checked and evaluated as soon as it has been entered by pressing the return key. This is completely different to the procedure normally encountered when working in a BASIC environment, where a section or even the whole program is entered before an attempt is made to execute the code.

Bearing this in mind, you are now ready to start experimenting and programming in LISP and the next few chapters will provide you with a guide on the use of the some of the common LISP functions.

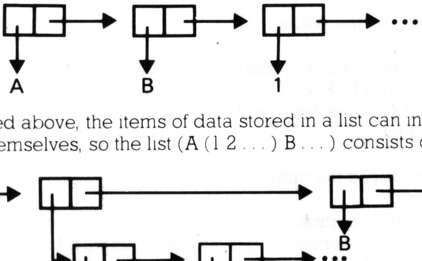
Chapter 2
AN INTRODUCTION TO LISP

In LISP, as with all high-level languages, the programmer requires the capacity to display textual messages on the screen and also to use such words as are appropriate as variable names. The name given to such words in LISP is either 'identifier' or 'atom', and these constitute the most elementary objects that can be used in a LISP program.

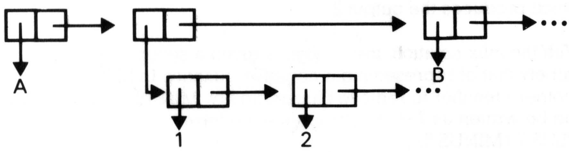
An identifier is an object that is constructed using characters (e.g. Z, DOG, T. NIL) and which can be used for either of the purposes described above, Atoms include all identifiers, together with numbers (e.g. 7, -24). Using these atoms we can construct the data structure that gives LISP its name. i.e. lists.

A list is a collection of items of data, each of which has a successor, except for the final item. It is useful to consider a system of pointers that will allow a user to traverse a list, with each pointer locating a memory cell containing a pair of pointers. The left-hand pointer indicates the atom (or list) that is the data, and the right-hand pointer where the remainder of the list starts. For the list (A B 1 . . .)

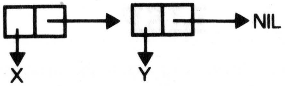
we get



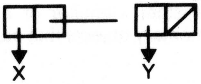
As indicated above, the items of data stored in a list can indeed be a list themselves, so the list (A (1 2 . . .) B . . .) consists of



It has already been mentioned that all items in a list have a successor, except for the last, for which LISP reserves a special identifier called 'NIL' to represent the empty list (). Thus the full diagram to represent the list (X Y) is



which is conventionally drawn



This all theoretical and forms the basis of how LISP operates, but for the moment let us consider how we can make some sense out of this new language.

Arithmetic

Arithmetic in LISP is performed by using identifiers rather than the conventional infix operators, such as '+' and 'x'. To add numbers (atoms) together we use the PLUS command.
To perform the addition 7+5, we would write
(PLUS 7 5) which returns the answer 12, and to calculate the value of 7+5+3+1, the correct LISP command is
(PLUS 7 5 3 1)

which returns the solution 16.

The reverse process to addition, namely subtraction, is achieved using the DIFFERENCE command, which subtracts the second argument from the first, i.e. 7-5 is given as
(DIFFERENCE 7 5)

which produces the output 2.

With the infix notation, the '–' sign is given a second meaning, namely that of representing a negative quantity. In LISP there is another identifier to remove this ambiguity, MINUS. So 7-5, which can be written as 7+(–5), could take the form
(PLUS 7 (MINUS 5))

and this will also produce the result 2.

Multiplication is performed using the TIMES, so to calculate 7x5 the command
(TIMES 7 5)

is used to return 35, and to multiply, say, 7x5x3x1, the command
(TIMES 7 5 3 1)

is issued and returns the answer 105.

LISP only allows the use of integers and this can create some difficulty when we wish to perform a division, e.g. 7 / 5 = 1.4 or, in a fractional form, 1 2/5. There are three different LISP commands related to division, DIVIDE, QUOTIENT and REMAINDER, each of which produces a different result.

DIVIDE returns a dotted pair (a special type of list), with the first element the quotient and the second the remainder,
e.g. (DIVIDE 7 5) --> (1.2)

QUOTIENT returns the whole number part of the solution and ignores any fractional part,
e.g. (QUOTIENT 7 5) --> 1

and REMAINDER returns the remainder after the division is performed, e.g. (REMAINDER 7 5) --> 2

Note that LISP allows the use of all integers in the range — 32768 to 32767 and that an error will occur if the result of an arithmetical calculation lies outside these limits.

Variables

In LISP the word ALPHA could represent either a variable or a pieces of data, and to inform the computer of the difference, a single quote can be used. Thus ALPHA would be the variable and the hence if accessed, its value would be displayed, and 'ALPHA would represent the word ALPHA as literal data. If a variable ALPHA has not been given a value, LISP will regard its value as UNDEFINED, which is a special identifier in the language reserved for this use.

To give a variable a value the identifier SETQ should be used. For example,
(SETQ ALPHA '(A B C D E))

will set the value of the variable ALPHA to be the list (A B C D E). [Note the quote ('), which means that, (A B C D E) should not be evaluated.] When the variable has been assigned it will display its value if its name is typed without parentheses, i.e. ALPHA.

For example,
(SETQ TEN 10)
(TIMES TEN TEN)

will set the value of TEN to 10 and then calculate the value of TEN squared, producing the answer 100. The value of the expression to be assigned can also be calculated within the SETQ command and so the instruction
(SETQ ANSWER (TIMES TEN TEN))

will assign the answer of the calculation, namely, 100 to the variable ANSWER.

List Handling

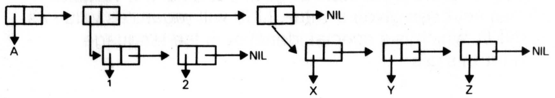
There are two basic functions for retrieving values from a list, namely CAR and CDR. IS-LISP allows either the command CAR or HEAD to extract the first element of a list and either CDR or TAIL to obtain the list which forms the remainder.

So for the list (X Y) we have
(CAR '(X Y)) --> X
(CDR '(X Y)) --> (Y)

Consider the list defined in the SETQ instruction
(SETQ EXAMPLE '(A (1 2) (X Y Z)))

The box diagram for this list is

EXAMPLE



Items can be extracted from the list EXAMPLE using the following:

(CAR EXAMPLE)	returns the identifier A
(CDR EXAMPLE)	returns the list with two members ((1 2) (X Y Z))
(CAR (CDR EXAMPLE)	returns the list (1 2)
(CDR (CDR EXAMPLE)	returns the list ((X Y Z))
(CAR (CAR (CDR EXAMPLE)))	returns the identifier 1

IS-LISP allows the user to shorten these statements by writing

(CAAR EXAMPLE)	instead of (CAR (CAR EXAMPLE))
(CDDR EXAMPLE)	instead of (CDR (CDR EXAMPLE))
(CADR EXAMPLE)	instead of (CAR (CDR EXAMPLE))
(CAAAR EXAMPLE)	instead of (CAR (CAR (CAR EXAMPLE)))
(CADAR EXAMPLE)	instead of (CAR (CDR (CAR EXAMPLE)))

for any combination of up to three of the CAR or CDR instructions.

The identifier T

The special identifier T is the value returned when a Boolean functon is evaluated to be true.

For example,

```
(NULL NIL)      --> T          since NIL is ()
(ATOM 3)        --> T          since 3 is an atom
(EQ 5 5)        --> T          since 5 = 5
```

Decision Making

In BASIC decision making is usually performed with IF . . .THEN . . .ELSE; which is employed in the following way.

```
IF <predicate.1> THEN <expression.1>
ELSE IF <predicate.2> THEN <expression.2>
ELSE IF <predicate.3> THEN <expression.3>
...
ELSE <otherwise-expression>
```

In LISP this simply becomes

```
(COND  (<predicate.1> <expression.1>)
       (<predicate.2> <expression.2>)
       ...
       (T <otherwise-expression>))
```

where T is the special identifier used to represent a true statement. Thus the COND function works by evaulating each predicate in turn until it finds one that is true. When this occurs it evaulates the corresponding expression and returns that its value.

Example

```
(COND  ((ATOM '(X Y)) (TIMES 4 9))
       ((ATOM 4) (TIMES 5 8))
       (T (TIMES 6 7)))
```

Here, since '(X Y) is a list and not an atom, the first predicate returns NIL and is ignored. The second predicate, however, is true, since 4 is an atom, and so the result of (TIMES 5 8), i.e. 40, is returned as the value of the statement.

Looping

Repetition of a group of expressions in LISP is obtained by using the LOOP command. So the following statement

```
(LOOP(PRINT(PLUS 7 5))
   (PRINT(DIFFERENCE 7 5)))
```

produces a loop which displays 12 then 2 then 12 . . . until the STOP key is pressed. A more useful use of the LOOP facility is to perform a sequence of events until some terminating condition is satisfied, and to do this the function WHILE or UNTIL is used.

When a loop contains either a WHILE or an UNTIL statement, it will continue to be executed as long as a WHILE predicate yields T, or an UNTIL predicate is not NIL.

Set the value of the variable C to be 7 by using the command (SETQ C 7) and set A to be 0 using
(SETQ A 0)

Then the use the loop

```
(LOOP (WHILE (GREATERP C 0) A)
        (SETQ A (PLUS A C))
        (SETQ C (SUB1 C)))
```

This will return the value of 7+6+5+4+3+2+1=28, since the loop is repeated all the time that C>0.

Alternatively, set the value of C to be 0 by using the command (SETQ C 0) and reset A to be 0 using
(SETQ A 0)

Then use the loop

```
(LOOP (UNTIL (EQ C 8) A)
        (SETQ C (PLUS A C))
        (SETQ C (ADD1 C)))
```

which produces the answer to 1+2+3+4+5+6+7=28, since the loop is repeated until C=8.

Chapter 3
INPUT/OUTPUT

The most important commands in any language are the input/output functions, as without these we would not be able to enter any values or obtain the results of our calculations.

Input Syntax

The most common way of getting data/program into LISP is by using the READ function. This will allow dotted pairs, lists, variables and numbers in the range -32768 to 32767 to be entered via the current input channel. When the system is first switched on the input channel is number '0' connected to the keyboard device, however this can be changed by using the OPEN and RDS commands, as explained in Chapter 6.

Note that if a "%" character is encountered while reading then everything up to the next end of line is treated as a comment and ignored.

The syntax of the input of identifiers is such that the computer will accept the characters A . . . . Z, a . . . z, 0 . . . 9 and the following special characters.

-	MINUS	_	UNDERSCORE	=	EQUALS
,	COMMA	;	SEMICOLON	:	COLON
[	LEFT SQUARE BRACKET	*	ASTERISK	&	AMPERSAND
$	DOLLAR	~	TWIDDLES	#	HASH
@	AT SIGN	/	SLASH	?	QUESTION MARK
+	PLUS SIGN	|	VERTICAL BAR	{	CURLY BRACKET
↑	UP ARROW	<	LESS THAN	>	GREATER THAN
\	BACKSLASH				

For example, A?BC, ::><:<*-40:*, + are all legal identifiers.

All other characters must be preceded by the escape character '!' if they are to be included in an indentifier. Normally each left bracket "(" must be closed by a matching right bracket ")", however, the "superbracket" "]" can be used to close *all* open left brackets. For example:

```
(DEFUN SQUARE (X) (TIMES X X]
```

The usual way to quote identifier is by preceeding them with the ' character: e.g. 'ABC. However an alternative is to surround the identifier with double quotes: e.g. "ABC". This is an easy way to get unusual characters like spaces into identifiers.

e.g. (PRINTC "A B")

Output Format

Output is generally produced by using the four print statements PRIN, PRINC, PRINT, PRINTC.

There are two different formats: the PRIN type and the PRINC type. With PRIN the output will be suitable to be read back in again using READ, and identifiers will be escaped with '!' where appropriate. However, the PRINC type instructions do not escape identifiers but produce them as given.

An alternative way of printing things out is with the SPRINT function, in which items are printed on one line where convenient or otherwise in the form:

```
(<function-name>
   <arg 1>
   <arg 2>
   <arg 3>
   . . .
   <arg n>)
```

When the definition has been produced SPRINT will print two blank lines and return the value NIL.

Chapter 4
DEFINING FUNCTIONS

As previously mentioned, LISP is a very flexible language which can be expanded to suit the needs of the individual user.

When we first enter the LISP environment we have a series of functions that are already defined and which can be called by simply typing the appropriate function name
e.g. (PLUS 4 7)

The function 'PLUS' has been defined as part of the LISP interpreter and has the effect of causing the 'evaluator' to find the sum of the two numbers 4 and 7. On occasions it is highly likely that we will want to use some function that has not been pre-defined and in such a case it is necessary to define it ourselves by using the 'DEFUN' command.

The syntax of the command is:
(DEFUN NAME (PARAMETERS) (BODY))

where NAME represents the name by which the function can be called;
PARAMETERS are the values passed to the function; and
BODY consists of one or more previously defined LISP functions.

EXAMPLE 1

Consider a situation in which the value of 2x+3y has to be evaluated several times during the operation of a program. It would be possible to use the following section of code each time that the operation was required:

```
(PLUS (TIMES 2 X) (TIMES 3 Y))
```

This obviously takes a long time to enter and it would be much better if we had a single function to perform the operation. This could be achieved using the statement.

```
(DEFUN OPERATE (P Q) (PLUS (TIMES 2 P) (TIMES 3 Q)))
```
The LISP interpreter would then reply with
```
(LAMBDA (P Q) (PLUS (TIMES 2 P) (TIMES 3 Q)))
```

indicating that the function had successfully been defined.

If we now wished to evaluate 2x+3y, we would simply enter

```
(OPERATE x y)
```

The values of 'x' and 'y' would be passed to 'p' and 'q' in the function which would then be calculated with the result being returned by the evaluator.

e.g. (OPERATE 4 5) --> 23

The use of the function in this way leads to a very structured form of programming known as the 'TOP-DOWN' approach, in which one function is defined that can then be used as part of a definition for a second function that in turn can be used as part of a definition for a third and so on, and it is in this way that most LISP programs are written.

EXAMPLE 2

Consider the problem of defining a second function 'OPERATE2', which will calculate 3(2X+3Y). This second function could be defined very easily by making use the first function 'OPERATE',

i.e. (DEFUN OPERATE2 (P Q) (TIMES 3 (OPERATE P Q)))

The value of 3(2X+3Y) could then be calculated by using the single function call

```
(OPERATE2 X Y)
```

Recursion

Recursion is a facility available to the LISP programmer, which gives enormous flexibility in the construction of programs. By using recursion it is possible to define a function that repeatedly calls itself until a certain condition is attained. It is very similar in action to a standard LOOP, but generally results in the production of shorter and more efficient code.

EXAMPLE

Consider the problem of defining a recurring function which takes a LIST (A B C D E) and finds the number of atoms within the list. This could be achieved as follows:

```
(DEFUN LEN (X)
   (COND
      ((ATOM X) 0)
      (T (ADD1 (LEN (CDR X]
```

Optional Arguments

We can define a function that may be called with (say) one *or* two arguments:

```
(DEFUN ADDON (A (B.2))
   (PLUS A B))
```

Here the first argument must be supplied while the second is optional — if it is not supplied then it takes the value of 2.

```
e.g. (ADDON 4 8) --> 12
     (ADDON 5) --> 7
```

A function can have any number of ordinary and optional arguments, with the restriction that all the optional ones must follow the ordinary ones.

Note that the default value (the number 2 in the above example) is evaluated if the parameter is not supplied, so the example could just as well have been written:

```
(DEFUN ADDON (A (B . (PLUS 1 1)))
   (PLUS A B))
```

Local Variables

If a function is defined with say 1 ordinary argument and 2 optional arguments:

e.g. (DEFUN A-FUN (A (B . 1) (C . NIL)) . . .)

and then always called with just one argument then the variables B and C are effectively local variables: they can be used as "temporary storage" within A-FUN and disappear when the function returns.

Editing Functions

When writing a LISP program you will often find a mistake and need to change the definition of a function. For example suppose we type in:

```
(DEFUN SQUARE (X) (TIMES X Y))
```

accidently typing a Y instead of an X.

To edit the function SQUARE we type:

```
(FEDIT SQUARE)
```

the screen will be cleared and we will see:

```
(LAMBDA (X) (TIMES X Y))
```

this is the machine's representation of the function: the word DEFUN and the name of the function have been replaced by the word LAMBDA.

The edit the function we move the cursor to the Y as normal and type an X. When we have finished our corrections we press the ESCAPE key. SQUARE is now redefined to its correct form.

If an error is detected while reading in the new definition (for example excess "." or ")", or end of file which means too few right brackets) the error message is displayed at the top of the screen for a few seconds. This then disappears and the cursor returns to allow the error to be corrected.

Chapter 5
SAVING and LOADING

The SAVE and LOAD commands allow you to transfer the state of the system to cassette so that it can be used at a later time. Before attempting to save the system you should ensure that a blank cassette is ready for data transfer and that the leads from the output sockets of your Enterprise microcomputer are connected to the 'MIC' and 'REM' sockets of your tape recorder.

You need a filename for the current state of the system and if this is, say, TUESDAY then issuing the command.

```
(SAVE "TUESDAY")
```

will cause this state to be saved onto cassette, ready to be loaded at a later time. The reloading of the state is achieved simply by issuing the command

```
(LOAD 'TUESDAY")
```

and then all the user defined expressions will be reloaded into the computer's memory. Remember to connect the 'EAR' and 'REM' sockets on the tape recorder to the input sockets of the Enterprise.

NOTE

When a new file is loaded from cassette, any information currently in the computer will be overwritten. If this information is important, however, then it should first be saved using the SAVE function, described above.

Chapter 6 EXOS

EXOS is the extendible operating system for the Enterprise microcomputer. It provides an interface between IS-LISP and the hardware of the machine. The main features of EXOS are a channel-based input/output (I/O) system and sophisticated memory mangement facilities. The I/O system allows device-independent communications with a range of built-in devices, as well as any additional device drivers which may be attached.

The built-in devices are:

1. Video driver providing text and graphics handling.
2. Keyboard handler providing joystick, autorepeat and programmable function keys.
3. Screen editor with word processing capacbilities.
4. Comprehensive stereo sound generator.
5. Cassette tape.
6. Centronics compatible parallel interface.
7. RS232 type serial interface.
8. Networking interface.

Many features of the system are controlled by a number of single byte values called "EXOS variables". A full list of these variables can be found in appendix 1.

The beginner to programming in LISP will not be too concerned about communications with EXOS. However, there will come a time when such communication will be required and the remainder of this chapter will consider how this can be achieved.

Screen Editor

When the machine is first switched on with the IS-LISP cartridge in place, the user is in communication with the 'screen editor', which in turn communicates with the LISP interpreter. This allows the user to move the cursor around the screen, and scroll both up and down.

Channels

As was mentioned in the beginning of this chapter, all I/O takes place through channels. LISP has been designed so that most simple things can be done without any knowledge of this fact. However, full use of the system can only be made with an understanding of how these channels and their associated EXOS variables can be manipulated.

When the machine is first switched on, the following channels are opened automatically:

CHANNEL 0 EDITOR
CHANNEL 1 VIDEO (GRAPHICS ONLY) - opened by the GRAPHICS command
CHANNEL 2 VIDEO (TEXT ONLY)
CHANNEL 3 SOUND
CHANNEL 5 KEYBOARD

And communications to the various devices will always take place along these channels until for one reason or another they are closed.

Channels 0,2,3 and 5 should be left alone. Unless there is a good reason, they should not be closed.

New Channels

On occasions we may need to open a new channel to a device driver so that communication can take place to some peripheral other than those set up as default channels. There are two functions to do this: OPEN and CREATE. For most purposes these two are equivalent, however when opening a file on tape or disc, CREATE will make a new file and OPEN will assume that the file already exists.

EXAMPLE

Consider the problem of defining a function that will SPRINT some function to a parallel printer. This can be overcome by using the following section of code

```
(DEFUN SPRINTER (EXP)
  (OPEN 10 "PRINTER:")
  (WRS 10)
  (SPRINT EXP)
  (WRS 0)
  (CLOSE 10))
```

This works by opening channel 10 to the parellel printer port and then using the WRS 10 instruction to select 10 as the current output stream. The function defined by the parameter 'EXP' is then SPRINTed onto the printer, after which the output stream is switched back to its default value and the channel is closed.

NOTE

When using channels it is easy to forget that the input or output will automatically be sent to the default channels, as shown below, and in order to redirect the commands SNDS, RDS, WRS and GRS must be used.

DEVICE	**DEFAULT**	**COMMAND**
SOUND	3	SNDS
TEXT (INPUT)	0	RDS
TEXT (OUTPUT)	0	WRS
GRAPHICS	1	GRS

EXAMPLE

To Input from channel 12 we would require

```
(OPEN 12 "DEVICE:")
(RDS 12)
```

Filenames

When a channel is open we use a command such as
(OPEN 12 "<device>:<filename>.<extension>")
where <device>, <filename> and <extension> are all optional.

The devices that can be used are listed below, and if no value is given, the system will default to tape.

KEYBOARD:	The Keyboard
VIDEO:	Video screen device
EDITOR:	Word processor device
PRINTER:	Centronics port
SOUND:	Sound device
SERIAL:	RS232 port
NET:	Network device
TAPE:	Cassette device

TAPE FILE HANDLING

On occasions we may wish to save information or functions onto tape without having to save the complete environment. This can be achieved by defining a function such as:

```
(DEFUN FILE EXP)
 (CREATE 11 "TAPE:RH")
 (WRS 11)
 (PRINT EXP)
 (WRS 0)
 (CLOSE 11))
```

When 'FILE' is called, the content of the parameter 'EXP' is saved onto a FILE called 'RH'. This can then be used at some later date by using a second function to extract the expression from the tape file.

EXAMPLE

```
 (DEFUN EXTRACT ((TEMP))
 (OPEN 11 "TAPE:RH")
 (RDS 11)
 (SETQ TEMP (READ))
 (RDS 0)
 (CLOSE 11)
 TEMP)
```

By typing (EXTRACT), the expression within the file 'RH' will be returned.

EXOS Commands

There are three LISP functions EXOS-READ, EXOS-WRITE and EXOS-TOGGLE that can be used by the experienced programmer to interrogate or change certain system variables. Appendix 1 lists these variables, together with their locations, and gives a brief description of what they are used for.

As an example of the use and importance of 'EXOS-VARIABLES', consider the problem of SPRINTing a function to a printer connected to the RS232 serial port and operating at the speed of 1200 BAUD. An investigation of the LISP commands shows that there is no function for changing the BAUD rate, but from Appendix 1 we see that there is an associated EXOS variable which can be used as follows:

```
(DEFUN SERPRINT (EXP)
  (EXOS-WRITE 16 8) % Set BAUD rate to 1200
  (OPEN 11 "SERIAL:")
  (WRS 11)
  (SPRINT EXP)
  (WRS 0)
  (CLOSE 11))
```

Chapter 7 INTERRUPT HANDLING

The Enterprise version of IS-LISP provides a special identifier, which enables a LISP programmer to write functions that utilize the interrupt system.

When we first switch on the microcomputer, the variable HANDLER is UNDEFINED, and if any software interrupt is received, then a SOFTWARE INTERRUPT (ERROR 4) will be produced. However, it is possible for the user to define some function, such as FRED, with one argument, and make this the interrupt HANDLER by typing:

```
(SETQ HANDLER 'FRED)
```

When a software interrupt is received, FRED will be called with the interrupt type (see below) as its only argument.

EXAMPLE

If we define FRED as:

```
(DEFUN FRED (TYPE)
  (COND
      ((EQ TYPE 24) (PRINTC "UNDEFINED FUNCTION")))
      ((EQ TYPE 48) (PRINTC "NETWORK INTERRUPT"))
      ((EQ TYPE 64) (PRINTC "TIME-OUT"))
      (T NIL)))
```

we see that a simple function such as this can provide the user several useful facilities:

1. A time facility.
2. An information facility when the network needs service.
3. The ability to interrupt the program by pressing (say) shift-F1 and then continue on as if nothing had happened.

The following table represents the values sent to HANDLER when a software interrupt occurs:

CODE	INTERRUPT	CODE	INTERRUPT
16	FUNCTION KEY 1	26	FUNCTION KEY 11
17	FUNCTION KEY 2	27	FUNCTION KEY 12
18	FUNCTION KEY 3	28	FUNCTION KEY 13
19	FUNCTION KEY 4	29	FUNCTION KEY 14
20	FUNCTION KEY 5	30	FUNCTION KEY 15
21	FUNCTION KEY 6	31	FUNCTION KEY 16
22	FUNCTION KEY 7	32	STOP KEY
23	FUNCTION KEY 8	33	ANY KEY
24	FUNCTION KEY 9	48	NETWORK
25	FUNCTION KEY 10	64	TIMER

(Function keys 9-16 are shifted function keys)

Function Key Interrupts

If a function key is programmed with the null string ("" in LISP), it will generate a software interrupt when pressed. When LISP powers up, certain keys are preprogrammed to useful commands, such as FEDIT, and the rest are set to the null string.

Stop Key Interrupt

If the EXOS variable STOP IRQ (number 8) is made non-zero, then the stop key will just return a code like any other key, effectively disabling the stop key. Otherwise, when the stop key is pressed, software interrupt 32 is received.

Any Key Interrupt

If the EXOS variable KEY IRQ (number 9) is set to zero (default value = 255), then when any key is pressed it will cause a software interrupt 33, as well as returning the code.

Network Interrupt

When a message is received on the network a software interrupt is usually generated (unless the EXOS variable NET IRQ (number 19) is set to zero). This is useful as it enables the user to read the message and then act upon it.

N.B. The EXOS variable ADDR NET (number 18) will then contain the network channel from which the data is to be read.

Timer Interrupt

The EXOS variable TIMER (number 5) is usually set to zero. If, however, it is made non-zero by using the EXOS-WRITE command, then it will start to count down by one each second. When it reaches zero, software interrupt 64 is generated. This can be very useful in the production of on-screen timers, as shown below.

EXAMPLE

The following section of code will produce a debugging aid. When shift-F1 is pressed LISP will run the function DEBUG which allows the user to type in any LISP expression which is then evaluated and the result printed out. For example the values of variables can be found by just typing in their names. When the user has finished he types 'BYE' and the program carries on whatever it was doing before.

```
(SETQ HANDLER 'FRED)

(DEFUN FRED (TYPE)
  (AND (EQ TYPE 24) (DEBUG)))

(DEFUN DEBUG ((INPUT))
  (PRINTC "Lisp debugger - type BYE to leave")
  (LOOP
            (PRINTC "DEBUG>")
            (LOOP (WHILE (ATOM (SETQ INPUT (ERRORSET
            (READ))))))
            (SETQ INPUT (CAR INPUT))
            (UNTIL (EQ INPUT 'BYE)
                   (PRINTC "***Leaving Debug"))
            (ERRORSET (PRINT (EVAL INPUT)))))
```

Interrupt handling is a very powerful tool for the more experienced programmer and it is possible to create complex functions which can be brought into operation when a particular software interrupt occurs.

N.B. Unfortunately LISP does not respond to interrupts while it is waiting for the user to type something in. (i.e. while the cursor is flashing).

Chapter 8
THE FUNCTIONS OF IS-LISP

This chapter contains an alphabetical list of the LISP functions, explains their structure and operation, and gives some examples of their use. Each function is identified as one of the following:

> Subr - a ordinary function evaluating its arguments;
> Fsubr - a function whose arguments are not necessarily evaluated;
> Id - an identifier with special significance in LISP;
> Var - a system-provided variable.

Arguments for LISP functions are shown in brackets '<>', or square brackets '[]', the latter being optional.

The result of the function is also given and this is a list (i.e. (a b c . . . z)), a whole number in the range
-32768 to 32767, or the word 'any', which allows any LISP type as the result.

(ABS <number>) --> number **Subr**

This returns the absolute value of the argument.

```
e.g. (ABS  23) --> 23
     (ABS -19) --> 19
```

(ADD1 <number>) --> number **Subr**

This returns the value of the argument plus one.

```
e.g. (ADD1 23) --> 24
```

(AND <exprl> <expr2> . . .) --> NIL or any **Fsubr**

This evaluates its arguments in turn. Should one of them evaulate to NIL, the function will return that value; otherwise, it will return the value of the last argument.

```
        e.g. (AND (NUMBERP 10 (ATOM 'A B))) --> NIL
                           (AND (LISTP '(A B)) 'XYZ) --> XYZ
```

(APPEND <x> <y>) --> list **Subr**

If <x> and <y> are lists, this function returns the list obtained by putting the elements in <y> after those in <x>.

```
            (DEFUN APPEND (X Y)
              (COND
                ((ATOM X) Y)
                  (T (CONS (CAR X) (APPEND (CDR X) Y ]
          e.g. (APPEND '(A B) 'C D)) --> (A B C D)
```

(APPLY <fn> <args>) --> any **Subr**

This returns the value of the function <fn> with the list of parameters <args>

```
            e.g. (APPLY NUMBERP '(10)) --> T
```

(ASSOC <key> <a list>) --> NIL or (<key>. value) **Subr**

This searches the association list <a list> of dotted pairs for the given key. If the search is successful, it returns (<key>. value) pair is; otherwise, it returns NIL.

```
(DEFUN ASSOC (U ALIST)
          (COND
              ((ATOM ALIST) NIL)
              ((ATOM (CAR ALIST) ERROR "BAD
                ASSOCIATED LIST"))
              ((EQUAL U (CAAR ALIST)) (CAR ALIST))
              (T (ASSOC U (CDR ALIST]

          e.g. (ASSOC 'A '((B.2) (A. -3))) --> (A. -3)
```

(AT <row> <column>) --> NIL **Subr**

This moves the cursor on the current output channel to (<row>, <column>).

e.g. (AT 10 20) --> NIL

(ATOM <x>) --> T or NIL **Subr**

This returns T if <x> is not a dotted pair.

```
e.g. (ATOM '(A.B)) --> NIL
     (ATOM 'A) --> T
     (ATOM 10) --> T
     (ATOM CAR) --> T
```

AUTOLOAD **Id**

When EVAL is trying to evaluate an expression of the form

(<identifier> <args>)

and finds that <identifier> is UNDEFINED it usually results in an error. However, the use may write an autoloader, which is a LISP function that will be evaluated whenever the user attempts to call an undefined function.

The autoloader can be set by using the command

(SETQ AUTOLOAD 'FRED)

The function FRED will then be called whenever an undefined function is obtained with the name of the undefined function as its single argument. This can be most useful in top-down programming, as the upper level functions can be tested without defining those at the lower level.

(BAND 2 <x> <y>) --> number **Subr**

This returns the bitwise AND of the two numbers <x>, <y> when they are expressed in binary form.

```
e.g. Since 7 = 111 and 3 = 11
     (BAND2 7 3) --> 3
```

(BEAM <exp>) --> NIL **Subr**

If <exp> is not NIL, then the beam at the current graphics channel is switched on; otherwise, it is swtiched off.

```
        e.g. (BEAM NIL) --> NIL to switch beam off
             (BEAM T) --> NIL to switch beam on
```

BLANK **Var**

This represents the space character, " ".

```
        e.g. (PRINC BLANK) displays a blank character
```

(BNOT <number>) --> number **Subr**

This returns the bitwise NOT of the number, i.e. the value of x becomes (−x) −1.

```
        e.g. (BNOT 7) --> -8
             (BNOT -3) --> 2
```

(BORDER <number>) --> NIL **Subr**

If 0 < (number) < 255, this will change the border to the colour represented by <number>.

```
        e.g. (BORDER 42)
```

(BOR2 <x> <y> --> number **Subr**

This returns the bitwise OR of the two numbers <x>, <y>.

```
        e.g. Since 7 = 111 and 3 = 11
             (BOR2 7 3) --> 7
```

(BXOR2 <x> <y>) --> number **Subr**

This returns the bitwise exclusive — OR of the two numbers <x>, <y>.

```
        e.g. Since 7 = 111 and 3 = 11
             (BXOR2 7 3) --> 4
```

(CAPTURE <old> <new>) --> NIL **Subr**

This redirects all read operations from the <old> channel to the <new> channel.

```
e.g. (CAPTURE 34 102) --> NIL
```

(CAR <x>) --> **any** **Subr**

This returns the first field of the dotted pair <x>. Error 25 occurs if <x> is not a dotted pair. A synonym for CAR is HEAD. The CAR function can be extended by using abbreviations such as CAAR or CAAAR to represent CAR CAR and CAR CAR CAR respectively.

```
e.g. (CAR '(A.B)) --> A
     (CAR '((A.B).C)) --> (A.B)
     (CAAR '((A.B).C)) --> A
```

(CDR <x>) --> **any** **Subr**

This returns the second field of the dotted pair <x>. Error 25 occurs if <x> is not a dotted pair. A synonym for CDR is TAIL. The CDR function can be extended by using abbreviations such as CDDR or CDDDR to represent CDR CDR and CDR CDR CDR respectively.

```
e.g. (CDR'(A.B)) --> B
     (CDR '(A.(B.C))) --> (B.C.)
     (CDDR '(A.(B.C))) --> C
```

(CHARACTER <number>) --> **id** **Subr**

This returns the identifier defined by the ASCII code <number>. It is the LISP equivalent of CHR$ in BASIC.

```
e.g. (CHARACTER 65) --> A
```

(CHARP <x>) --> **T or NIL** **Subr**

This returns T if <x> is an identifier and NIL otherwise.

```
e.g. (CHARP 'A) --> T
     (CHARP 10) --> NIL
```

(CHARS <exp>) --> number **Subr**

This returns the number of characters an atom would generate if it were printed. The value returned depends on the type of the argument.

list --> 0
identifier --> the number of characters in the print name
(equivalent to the BASIC LEN)
number --> 6
codepointer --> 10

```
e.g. (CHARS 'A) --> 1
     (CHARS 10) --> 6
     (CHARS CAR) --> 10
```

(CLEAR) --> NIL **Subr**

This clears the video page of the current graphics channel.

(CLOSE <number>) --> <number> **Subr**

If <number> is a valid channel number of an open file, then the file is closed.

e.g. (CLOSE 1) will close the file on channel 1

(CODEP <x>) --> T or NIL **Subr**

This returns T if <x> is a codepointer; otherwise, it returns NIL.

```
e.g. (CODEP (A.B.)) --> NIL
     (CODEP CAR) --> T
```

(COMMENT <any1> <any2> ...) --> NIL **Fsubr**

This is used to place textual comments in an expression.

e.g. (COMMENT THIS WILL BE IGNORED) --> NIL

(COND (predicate action) (predicate action) ...) --> ANY **Subr**

This is a method for control structure and is similar to the BASIC IF THEN ELSE command.

```
        i.e. IF <pred1> THEN <exp1>
             ELSE IF <pred2> THEN <exp2>
             . . .
             ELSE <otherwise exp>
```

is written in LISP as

```
          (COND (<pred1> <exp1>)
                (<pred2> <exp2>)
            . . . (T<otherwise exp>))

          (COND ((EQ A 3) (PRINT "A=3"))
                ((EQ A 4) (PRINT "A=4"))
                (T NIL))
```

(CONS <car part> <cdr part>) --> list **Subr**

This creates the new list from the two given expressions.

```
        e.g. (CONS 'A 'B) --> (A.B)
             (CONS 'A (CONS 'B NIL)) --> (A B)
```

(CONSTANTP <x>) --> T or NIL **Subr**

This returns T if <x> is a number or codepointer; otherwise, it returns NIL.

```
(DEFUN CONSTANTP (OR (NUMBERP X) (CODEP X]
```

```
        e.g. (CONSTANTP 10) --> T
             (CONSTANTP 'A) --> NIL
```

(COPY <x>) --> <x> **Subr**

This returns a copy of <x>.

```
(DEFUN COPY (X)
  (COND
    ((ATOM X) X)
```

```
        (T (CONS (COPY (CAR X)) ((COPY (CDR X]

        e.g. (COPY '(ABCDE)) --> (ABCDE)
```

(CREATE <number> <filename>) --> <number> **Subr**

This function opens <filename> on channel <number>. It is equivalent to the IS-BASIC OPEN command with "Access Output" specified — see the BASIC manual.

```
        e.g. (CREATE 15 "NAME")
```

CRLF **Var**

The value of CRLF is a carriage return/line feed.

(CURSOR <exp>) --> NIL **Subr**

If <exp> is not NIL, then the cursor on the current output channel is switched on; otherwise, it is turned off.

```
        e.g. (CURSOR NIL) --> NIL to turn cursor off
             (CURSOR T) --> NIL to turn cursor on
```

(DEFLIST <dlist> <ind>) --> list **Subr**

The argument <dlist> is a list in which each element is a two-element list; (id prop). Each identifier in the dlist has its prop placed on its property list under the indicator <ind>. A list of the <ind>s is returned.

```
(DEFIN DEFLIST (U IND)
  (COND
    ((ATOM U) NIL)
    (T (PUT (CAAR U) IND (CADAR U))
      (CONS (CAAR U) (DEFLIST (CDR U) IND)))))

        e.g. (DEFLIST '((V DD) (H RG)) 'NAME) --> (V H)
             which is the same as:
             (PUT 'V 'NAME 'DD) --> DD
             (PUT 'H 'NAME 'RG) --> RG
         and can be retrieved by
```

```
(GET 'V 'NAME) --> DD
```

(DEFMAC <name> <parameter> <body> . . .) --> <name> **Fsubr**

This is the usual way of defining a macro.

```
e.g. (DEFMAC IF X
        (LIST
          "COND"
          (LIST (CADR X) (CADDR X))
          (LIST T (CAR (CDDDR X)))))
```

This defines a macro IF which is called with three arguments:

```
e.g. (IF A B C)
```

'A' is evaluated: if it is true then 'B' is returned, otherwise 'C' is returned.

```
e.g. (IF (ZEROP X) 4 5) --> 4 if X=0
     (IF (ZEROP X) 4 5) --> 5 if X<>0
```

(DEFUN <name> <parameters> <body> . . .) --> <name> **Fsubr**

This is the way of defining a function.

```
e.g. (DEFUN SQUARE (X) (TIMES2 X X)) --> SQUARE
```

(DEFVIDEO <+-mode> <g-mode> <g-col>) --> NIL **Subr**

This defines the parameters to be used only by the TEXT and GRAPHICS functions:
<+mode> should be either 40 or 80 and is the number of columns in a screen.

<g-mode> is one of the following:

1	high-resolution graphics
5	low-resolution graphics
15	attribute mode

<g-col> is one of the following: 0 for the two-colour mode
1 for the four-colour mode
2 for the 16-colour mode
3 for the 250-colour mode

e.g. (DEFVIDEO 40 1 0) --> NIL

(DEL <ch>) --> <ch> **Subr**

This closes EXOS channel <ch>. For most devices it is equivalent to CLOSE.

e.g. (DEL 1) --> 1

(DELETE <x> <y>) --> list **Subr**

This returns the list <y> with the first occurrence of <x> deleted.

```
(DEFUN DELETE (A L)
  (COND
    ((ATOM L) L)
    ((EQUAL A (CAR L)) (CDR L))
    (T (CONS (CAR L) (DELETE A (CDR L))))))
```

e.g. (DELETE 'A '(B A C A) --> (B C A)

(DIFFERENCE <x> <y>) --> number **Subr**

This subtracts the number y from the number x and returns the result.

(e.g. (DIFFERENCE 7 3) --> 4

(DIGIT <x>) --> T or NIL **Subr**

This returns T if <x> is an identifier with a digit (0-9) as the first character in its print-name; otherwise, its value is NIL.

e.g. (DIGIT 'A) --> NIL
(DIGIT '!0A) --> T

(DISPLAY <chan> <from> <ln> <at>) --> NIL **Subr**

This displays <ln> lines, starting at line <from> of the page on channel <chan>. They are displayed from line <at> onwards on the screen. All arguments should be integers.

> e.g. (DISPLAY 4 10 7 1) --> NIL
> will display lines 10 to 16 of the page on channel 4, starting at the top line of the screen.

(DIVIDE <x> <y>) --> (quo.rem) **Subr**

The number <x> is divided by <y> and the (quotient.remainder) pair is returned.

> e.g. (DIVIDE 7 3) --> (2.1)

DOLLAR **Var**

This is the dollar ($) character.

(EDIT <exp>) --> any **Subr**

This creates a text page and displays <exp> using SPRINT. You can then edit the <exp> by using all of the word processor facilities, pressing 'ESCAPE' to finish. The modified structure is then read back and returned. The usual way to edit functions is using the FEDIT function (qv).

(ELLIPSE <x> <y>) --> NIL **Subr**

This draws an ellipse centred on the current beam position on the current graphics channel in the current ink colour. <x> and <y> specify the x-radius and y-radius respectively.

> e.g. (ELLIPSE 200 100) --> NIL

(ENVELOPE <en> <er> [<cp> <cl> <cl> <pd>]) --> NIL **Fsubr**

A full description of envelope can be found in the BASIC manual. <en> is the envelope number (0-250)

<er> is the number of phases before release (255 for no release)
The parameters in square brackets define a phase and can be repeated up to 12 times.
<cp> - the change in pitch in semitones
<cl> - change in left volume (-63. . .63)
<cr> - change in right volume -63. . .63)
<pd> - phase duration in 1/50th second

e.g. (ENVELOPE 1 1 10 30 20 10) --> NIL

(EOF) --> T or NIL **Subr**

This tests to see if the end of a file has been reached on the current input channel. Its value is T if it has, and NIL otherwise.

(EQ <expl> <exp2>) --> T or NIL **Subr**

This returns T if one of the following is true about its arguments.
1. They are the same identifier.
2. They are equal numbers.
3. They are identical lists in LISP memory.

```
e.g. (EQ 1 'A) --> NIL
     (EQ 10 10) --> T
     (EQ '(A B) '(A B)) --> NIL
```

(EQUAL <expl> <exp2>) --> T or NIL **Subr**

This tests to see if two general expressions are equal.

```
(DEFUN EQUAL (X Y)
  (COND
    ((EQ X Y) T)
    ((OR (ATOM X) (ATOM Y)) NIL)
    ((EQUAL (CAR X) (CAR Y)) (EQUAL (CDR X) (CDR Y)))
    (T NIL)))
```

```
e.g. (EQUAL 1 'A) --> NIL
     (EQUAL '(A B) '(A B)) --> T
```

(ERROR <number>) --> no value **Subr**

This causes LISP error number <number> to be triggered.

(ERRORSET <any>) --> **number or any** **Fsubr**

This evaluates the argument. If no error occurs in the evaluation of <any>, it returns this value as a dotted pair with NIL; otherwise, it returns the error number.

```
        e.g. (ERRORSET (ATOM 10)) --> (T)
             (ERRORSET (A.B)) --> 17
```

(EVAL <any>) --> <any> **Subr**

This causes a second evaluation of its argument to be performed.

```
        e.g. (EVAL '(CAR '(A B))) --> A
```

(EVLIS <list>) --> **list** **Subr**

This evaluates each member of the list and returns a list of the results.

```
(DEFUN EVLIS (ARGS)
  (COND
    ((ATOM ARGS) NIL)
    (T (CONS (EVAL (CAR ARGS))
             (EVLIS (CDR ARGS))))))
```

```
        e.g. (EVLIS '((CHARP 'Z) (EQ 'A 'B))) --> (T NIL)
```

(EXOS-READ <var>) --> <number> **Subr**

This returns the current value of the EXOS variable number <var>.

```
        e.g. (EXOS-READ 10) --} 234
```

(EXOS-TOGGLE <var>) --> <number> **Subr**

This acts like a switch and toggles the appropriate EXOS variable. The value returned by the function is the ones complement of its current status.

e.g. (EXOS-TOGGLE 8) --> 0

(EXOS-WRITE <var> <val>) --> <val> **Subr**

This sets the EXOS variable number <var> to the value <val>.

e.g. (EXOS-WRITE 10 234) --> 234

(EXPAND <list> <function>) --> list Subr

<function> should be a function requiring two arguments. If there are n elements in the list L(1), L(2), . . ., L(n), then the value of the resulting list is:
(<function> L(1) (<function> L(2) (. . . (<function> L(n-1) L(n)) . . .)))

```
(DEFUN EXPAND (L FN)
  (COND
    ((ATOM (CDR L)) (CAR L))
    (T (CONS FN
      (CONS (CAR L)
      (CONS (EXPAND (CDR L) FN) NIL))))))
```

e.g. (EXPAND '(A B C D) 'PLUS2) --> (PLUS2 A (PLUS2 B (PLUS2 C D)))

(EXPANDMACRO <macro function> <test>) --> any **Subr**

This is a useful debugging tool to be used when working with macros.

e.g. (EXPANDMACRO IF '(IF(EQ X 3) 4 5))
(see the entry for DEFMAC for the definition of IF)

(EXPLODE <id>) --> list **Subr**

This returns a list of the characters in the print name of <id>.

e.g. (EXPLODE 'ABCDE) --> (A B C D E)
(EXPLODE "") --> NIL

(FEDIT <fun>) --> NIL **Fsubr**

This is similar to EDIT, but allows you to alter the function <fun>. Press 'ESCAPE' when the editing is finished to re-define the function.

(FKEY <kn> <str>) --> NIL **Subr**

This sets function key <kn> to be the string <str>, of up to 23 characters.
<kn> is in the range 1-16: 9-16 are shifted 1-8.

e.g. (FKEY 1 "I'm now programmed!") --> NIL

(FLAG <idlist> <ind>) --> NIL **Subr**

This marks each identifier in the list with the flag <ind>.

e.g. (FLAG '(A B C D) 'FINE) --> NIL

(FLAGP <id> <ind>) --> T or NIL **Subr**

This is used to detect if the identifier <id> has been flagged with the indicator <ind>.

e.g. (FLAGP 'A 'FINE) --> T or NIL

(FLATTEN <x>) --> list **Subr**

This removes all of the subtree structure of <x>.

```
(DEFUN FLATTEN (X)
  (COND
    ((NUL X) NIL)
    ((ATOM X) (CONS X NIL))
    (T (NCONC (FLATTEN (CAR X))) (FLATTEN (CDR X))))))
```

e.g. (FLATTEN '(A((B)) NIL (C.D) E)) --> (A B C D E)

(FLUSH <ch>) --> NIL **Subr**

This routine flushes the network channel <ch>

e.g. (FLUSH 17) --> NIL

(FSUBRP <X>) --> T or NIL **Fsubr**

This returns T if <x> is a codepointer to an Fsubr and NIL otherwise.

e.g. (FSUBRP COND) --> T
(FSUBRP 'A) --> NIL

FUNARG **Id**

FUNARG has the value UNDEFINED. It has a special significance within the interpreter, denoting FUNARG closures.

(FUNCTION <fn>) --> (FUNARG <fn> environment **Fsubr**

This acts just like QUOTE <qu> but should only be applied to functions. When the resulting form (called a FUNARG closure) is applied to some arguments all free variables will have the same values as they did at the time the closure was created.

(GENSYM) --> id **Subr**

This returns a unique identifier of the form G0000, G0001, etc.

e.g. (GENSYM) --> G0042

(GET <id> <ind>) --> any **Subr**

This returns the property on the list of identifier <id> under the indicator <ind>, or NIL if there is non stored.

e.g. (GET 'V 'NAME) --> DD

(GETCHAR) **Subr**

A single character is read from the current input stream and returned. Note that this is not the same as INKEY$ in BASIC. The LISP equivalent of INKEY$ is:

```
(DEFUN INKEY ()
  (RDS 5)
(PROG1 (AND (ZEROP (READSTATUS)) (GETCHAR))
       (RDS 0)))
```

e.g. (GETCHAR) --> J

(GRAPHICS) --> NIL **Subr**

If a graphics channel exists, then it is displayed on the screen; otherwise, a standard graphics page is opened and displayed in the top 20 lines of the screen. The graphics mode and colours are defined by the DEFVIDEO command.

(GREATERP <x> <y>) --> T or NIL **Subr**

T is returned if number <x> is greater than number <y>; otherwise, NIL is returned.

e.g. (GREATERP 7 3) --> T

(GRS <ch>) --> <number> **Subr**

This makes <ch> the current graphics channel, returning the previous one.

e.g. (GRS 12) --> 42

HANDLER **Id**

When LISP is executing it may receive 'software interrupts' from the operating system. The variable HANDLER can be set so that the system performs some function when an interrupt occurs. See chapter 7.

(HEAD <x>) --> any **Subr**

This function is identical to car.

e.g. (HEAD '(A.B)) --> A

(IMPLODE <idlist>) --> id **Subr**

This returns the identifier obtained by concatenating the atoms in <idlist>.

e.g. (IMPLODE '(A B CD EF G)) --> ABCDEFG

(IN <ioport>) --> number **Subr**

This returns the value obtained from Z80 input/output port <ioport>.

e.g. (IN 129) --> 3

(INK <col>) --> NIL **Subr**

This changes the foreground colour of the current graphics channel to the 'logical' colour <col> selected from the page's PALETTE.

e.g. (INK 10) --> NIL

(INTERN <id>) --> <id> **Subr**

This searches the oblist for <id>: if it is successful, then <id> is added to the olbist and returned.

e.g. (INTERN 'WRITE) --> WRITE

(JOY <num>) --> number **Subr**

This returns the status of the joystick number <num> as defined in the BASIC manual.

e.g. (JOY 1) --> 17

LAMBDA **Id**

This has special significance within the interpreter denoting a lambda expression.

(LAST <x>) --> any **Subr**

The returns the last member of the list <x>.

```
(DEFUN LAST (X)
  (COND
    ((ATOM X) X)
    ((NULL (CDR X)) (CAR X))
    (T (LAST (CDR X)))))
```

e.g. (LAST 'A B C)) --> C

(LENGTH <x>) --> number **Subr**

This returns the top level length of the list <x>.

```
(DEFUN LENGTH (X)
  (COND
    ((ATOM X) 0)
    (T (ADD1 (LENGTH (CDR X))))))
```

```
e.g. (LENGTH 'A) --> 0
     (LENGTH '(A B (C.D) E)) --> 4
```

(LESSP <x> <y>) --> T or NIL **Subr**

This returns T if number <x> is strictly less than number <y>; otherwise, it returns NIL.

e.g. (LESSP 7 3) --> NIL

(LINELENGTH <any>) --> number **Subr**

This affects the length of a line as used by SPRINT.

e.g. (LINELENGTH 60) --> 64 (the previous value)

(LIST <arg1> <arg2> . . .) --> list **Subr**

This returns the list (<arg1> <arg2> . . .).

e.g. (LIST 'A 'B '-9 'C) --> (A B -9 C)

(LISTP <x>) --> T or NIL **Subr**

This returns T if <x> is a dotted pair; otherwise, it returns NIL.

e.g. (LISTP '(A B)) --> T

(LITER <x>) --> T or NIL **Subr**

This returns T if <x> is an identifier whose print name has a letter as its first character; otherwise, it returns NIL.

e,g, (LITER 'A) --> T
(LITER '!0A) -->NIL

(LOAD <filename>) **Subr**

This will load a memory image saved using the function SAVE. it will also load an ASCII file consisting of a series of LISP expressions.

e.g. (LOAD "NAME")

(LOOP <action1> <action2> . . .) --> any **Subr**

This will keep executing <action1> <action2> . . . until a clause in a WHILE or UNTIL becomes true/false.

e.g. (LOOP (PRIN '*) (SETQ CT(SUB1 CT)) (UNTIL (ZEROP CT))) will produce a row of * if prceeded by (SETQ CT <number>)

LPAR **Var**

The value of LPAR is the character "(".

MACRO **Id**

This has special significance within the interpreter, denoting a Macro expression.

```
(DEFUN MAP (X FN)
  (LOOP
    (UNTIL (ATOM X) NIL)
    (FN X)
    (SETQ X (CDR X))))
```

e.g. (MAP 'ABC D) '(LAMBDA (x) (PRIN x))) --> NIL
will produce the output (A B C D) (B C D) (C D)
(D) NIL

(MAPC <x> <fn>) --> NIL **Subr**

This applies the function <fn> to each member of the list <x> in turn. Then it returns NIL.

```
(DEFUN MAPC (X FN)
  (LOOP
    (UNTIL (ATOM X) NIL)
    (FN (CAR X))
    (SETQ X (CDR X))))
```

e.g. (MAPC '(A B C D) '(LAMBDA (x) PRIN (x)))
--> NIL
will produce the output ABCD NIL

(MAPCAN <x> <fn>) --> list **Subr**

It is required that every application of <fn> produces a list. MAPCAN returns the concatenation of the output lists for each of the elements of <x> taken in turn.

```
(DEFUN MAPCAN (X FN)
  (COND
    ((ATOM X) NIL)
    (T (NCONC (FN (CAR X)) (MAPCAN (CDR X) FN)))))
```

e.g. (MAPCAN '(A B C D) '(LAMBDA (x) (LIST xx)))
-->
(A A B B C C D D)

(MAPCAR <x> <fn>) --> list **Subr**

This returns the list of the results of applying <fn> to every member of <x> in turn.

```
(DEFUN MAPCAR (X FN)
  (COND
    ((ATOM X) NIL)
    (T (CONS (FN (CAR X)) (MAPCAR (CDR X) FN)))))
```

e.g. (MAPCAR 'A B C D) '(LAMBDA (x) (CONC x x)))
-->
((A.A) (B.B) (C.C) (D.D))

(MAPCON <x> <fn>) --> list **Subr**

It is required that each application <fn> will produce a list. MAPCON applies <fn> to <x>, (CDR <x>), CDDR <x>, . . . until the list is exhausted and returns the list constructed by concatenating the results.

```
(DEFUN MAPCON (X FN)
  (COND
    ((ATOM X) NIL)
    (T (NCONC (FN X) (MAPCON (CDR X) FN)))))
```

e.g. (MAPCON '(A B C D) '(LAMBDA (x) (LIST (CONS 1 x))) --> ((1 A B C D) (1 B C D) (1 C D) (1 D))

(MAPLIST <x> <fn>) --> list **Subr**

This returns the list of the results of applying <fn> to <x>, (CDR <x>, (CDDR <x>), . . . until the list is exhausted.

```
(DEFUN MAPLIST (X FN)
  (COND
    ((ATOM X) NIL)
    (T (CONS (FN X) (MAPLIST (CDR X) FN)))))
```

e.g. (MAPLIST '(A B C D) '(LAMBDA (x) (LENGTH x))) --> (4 3 2 1)

(MAX2 <x> <y>) --> number **Subr**

This returns the larger of the two numbers <x> and <y>.

e.g. (MAX2 7 3) --7

(MEMBER <x> <y>) --> NIL or list **Subr**

This returns NIL if <x> is not a member of the list <y>; otherwise, it returns the remainder of <**y**>, starting at <x>.

```
(DEFUN MEMBER (X Y)
  (COND
    ((ATOM Y) NIL)
    ((EQUAL X (CAR Y)) Y)
    (T (MEMBER X (CDR Y)))))
```

```
            e.g. (MEMBER 'A '(B C A D E)) --> (A D E)
```

(MEMQ <x> <y>) --> NIL or list **Subr**

This is the same as MEMBER but uses EQ instead of EQUAL for the comparison.

```
(DEFUN MEMQ (X Y)
  (COND
    ((ATOM Y) NIL)
    ((EQ X (CAR Y)) Y)
    (T (MEMQ X (CDR Y)))))
```

```
            e.g. (MEMQ 'A B) '(A B (A B) C)) --> NIL
```

(MESSOFF <number>) --> number **Subr**

Messoff and Messon are used to control the printing of certain system messages. They control bits in a control byte, as follows:

1	Garbage collection bytes collected	OFF
2	Garbage collection number	OFF
4	Error number (and message if any)	ON
8	Error backtrace	ON
64	Controls conversion of (QUOTE <id>) to "<id>"	ON

For example, (MESSON 2) causes the garbage collection number to be printed, while (MESSOFF 8) causes the error backtrace to be suppressed.

```
            e.g. (MESSOFF 10) --> 255
```

(MESSON <number>) --> number **Subr**

See MESSOFF.

(MIN2 <x> <y>) --> number **Subr**

This returns the smaller of the two numbers <x> and <y>.

```
e.g. (MIN2 7 3) --> 3
```

(MINUS <number>) --> number **Subr**

This returns <number> negated.

```
e.g. (MINUS 7) --> -7
     (MINUS -3) -->  3
```

(MINUSP <x>) --> T or NIL **Subr**

This returns T if x a number and x<0; otherwise, it returns NIL.

```
e.g. (MINUSP -3) --> T
     (MINUSP 4) --> NIL
```

(MKQUOTE <x>) --> list **Subr**

This returns the list (QUOTE <x>).

```
e.g. (MKQUOTE '(A B C) --> (QUOTE (A B C))
```

(NCONC <x> <y>) --> list **Subr**

This concatenates <x> to <y> without copying <x>. It is the same as APPEND but is not as reliable. If in doubt, use APPEND!

```
(DEFUN NCONC (A B (W))
  (COND
    ((ATOM A) B)
    (T (SETQ W A)
      (LOOP
       (UNTIL (ATOM (CDR W)))
       (SETQ W (CDR W)))
```

```
        (RPLACD W B)
        A)))
```

e.g. (NCONC '(A B C) '(D E)) --> (A B C D E)

NIL **Id**

This is used to represent false.

(NOT <x>) --> T or NIL **Subr**

If <x> is NIL, then NOT returns T; otherwise, it returns NIL.

e.g. (NOT NIL) --> T

(NULL <x>) --> T or NIL **Subr**

If <x> is NIL, then NULL returns T; otherwise, it returns NIL. It is exactly equivalent to the function NOT (see above).

e.g. (NULL 1) --> NIL

(NUMBERP <x>) --> T or NIL

This returns T if <x> is a number; otherwise, it returns NIL.

e.g. (NUMBERP 7) --> T

(OBLIST) --> list **Subr**

This returns a list containing all identifiers known to the system. (FLATTEN (OBLIST)) produces a simpler list.

(ONEP <x>) --> T or NIL **Subr**

This returns T if <x> is the number one; otherwise, it returns NIL.

e.g. (ONEP 1) --> T

(OPEN <num> <filename>) --> <number> **Subr**

This function opens <filename> on channel <num>. It is similar to the Basic OPEN command.

e.g. (OPEN 15 "NAME")

(OR <exprl> <expr2> <expr3> . . .) --> any **Fsubr**

This returns the first of the arguments that is non-NIL, or NIL if it exhausts these arguments.

e.g. (OR (NUMBERP 'A) (CONS A B) (ZEROP 'T)) --> (A.B)

(ORDERP <id 1> <id 2>) --> T or NIL **Subr**

This returns T if the ASCII code of the printname of <id 1> is greater than that of <id 2>; otherwise it returns NIL.

```
e.g. (ORDERP 'A 'B) --> NIL
     (ORDERP 'B 'A) --> T
     (ORDERP 'AB 'AA) --> T
```

(ORDINAL <id>) --> number **Subr**

This returns the ASCII code of the first character of the printname <id>.

e.g. (ORDINAL 'APPLE) --> 65

(OUT <value> <ioport>) --> <value> **Subr**

This sends <value> to the Z80 input/output specified by <ioport>

e.g. (OUT 10 254) --> 254

(PAINT) --> NIL **Subr**

This causes a fill on the current graphics page of any shape starting at the current beam position, and stopping at any boundary that is not the same colour as the current beam position.

(PAIR <x> <y>) --> a list **Subr**

<x> and <y> must be lists with the same number of elements. PAIR returns a list in which each element is a dotted pair, the CAR taken from <x> and the CDR taken from <y>.

e.g. (PAIR '(A B) '(1 2)) --> ((A. 1) (B. 2))

(PALETTE <c0> <c1> <c2> <c3> <c4> <c5> <c6> <c7>) --> NIL **Subr**

This specifies the palette for the current graphics channel, with <c0> to <c7> being integers in the range 0-255, each specifying a colour.
NOTE: In 2 colour mode c2 to c7 are redundant, in 4 colour mode c4 to c7 are redundant, in 16 colour mode the other eight colours are derived from c0 to c7 in the usual way.

e.g. (PALETTE 10 20 40 5 73 122 5 0) --> NIL

(PAPER <col>) --> NIL **Subr**

This changes the paper colour of the current graphics channel to <col>, the colour taking effect when the channel is next CLEARed.

e.g. (PAPER 29) --> NIL

(PEEK <address>) --> number **Subr**

This returns the contents of <address>.

e.g. (PEEK 10) --> 6

PERIOD **Var**

The value of PERIOD is the character '.'

(PLIST <id>) **List**

This returns the property list of the identifier <id>.

e.g. (PLIST 'A) --> ((PROP1.VALUE1) (PROP2.-19))

(PLOT <x> <y>) --> NIL **Subr**

This moves the graphics beam to (<x>, <y>). If the beam is on, it will draw a line.

e.g. (PLOT 200 100) --> NIL.

(PLOTR <x> <y>) --> NIL **Subr**

This moves the graphics beam by <x> in the x-direction and <y> in the y-direction from the current beam position. If the beam is on, it will draw a line.

e.g. (PLOTR 100 200) --> NIL

(PLOTMODE <num>) --> NIL **Subr**

This sets the plotting mode on the current graphics channel as follows:

0. PUT plotting (default)
1. OR plotting
2. AND plotting
3. XOR plotting

e.g. (PLOTMODE 2) --> NIL

(PLOTSTYLE <num>) --> NIL **Subr**

This specifies the linestyle on the current graphics channel. <num> is an integer in the range 1-14, with 1 being a solid line and the others various dotted lines.

e.g. (PLOTSTYLE 5) -->NIL

(PLUS <arglist>) --> number **Fsubr**

The sum of the integers in <arglist> is returned.

e.g. (PLUS 10 3 16) --> 29

(PLUS2 <x> <y>) --> number **Subr**

A more efficient way of adding just two numbers <x> and <y>.

e.g. (PLUS2 7 3) --> 10

(POKE <address> <value>) --> <value> **Subr**

This places the <value> in a memory location determined by <address>.

e.g. (POKE 10 23) --> 23

(PRIN [<arglist>]) --> any **Subr**

The arguments in <arglist> are evaluated and displayed without intervening blanks. Special characters are escaped.

e.g. (PRIN 'A BLANK 'B) --> B will display A!   B

(PRINC [<arglist>] --> any **Subr**

This is the same as PRIN but with the special characters not escaped.

e.g. (PRINC 'A BLANK 'B) --> B will display A   B

(PRINT [<arglist>]) --> NIL **Subr**

This is the same as PRIN but including the CR/LF and returning NIL.

e.g. (PRINT 'A BLANK 'B) --> NIL will display A!   B

(PRINTC [<arglist>]) --> NIL **Subr**

This is the same as PRINT but with the special characters not escaped.

e.g. (PRINTC 'A BLANK 'B) --> NIL will display A   B

(PROG1 <exp1> <exp2>) --> <exp1> **Subr**

This returns the first argument.

e.g. (PROG1 'A 'B) --> A

(PROG2 <exp1> <exp2>) --> <exp2> **Subr**

This returns the second argument.

e.g. (PROG2 'A 'B) --> B

(PROGN <exp1> <exp2> . . . <expn>) --> any **Fsubr**

This evaluates <exp1> <exp2> . . . <expn> in turn and then returns <expn>.

e.g. (PROGN 'A 3 4 'B) --> B

(PUT <id> <ind> <prop>) --> <prop> **Subr**

This places the property <prop> on the property list of <id> under the indicator <ind>.

e.g. (PUT 'V 'NAME 'DD)

(QUOTE <any>) --> unevaluated <any> **Fsubr**

This stops evaluation and is written '<any> throughout LISP.

e.g. (QUOTE A) --> A
or
'A --> A

(QUOTEP <x>) --> T or NIL **Subr**

This returns T if <x> is a quoted expression; otherwise, it returns NIL.

e.g. (QUOTEP '(QUOTE Z)) --> T

(QUOTIENT <x> <y>) --> number **Subr**

This divides number <x> by number <y>, ignoring the remainder, and returning the answer.

e.g. (QUOTIENT 7 3) --> 2

(RANDOM <num>) --> number **Subr**

This returns a random number in the range 0 . . . <num> - 1 unless <num> = 0 when the range is 0-32767. The maximum value for <num> is 2000.

e.g. (RANDOM 1967) --> 1510

(RANDOMISE <seed>) --> <seed> **Subr**

This can be used to control the production of random numbers. If the <seed> is zero, then the sequence is unpredictable, but if it is non-zero, then a specific repeatable sequence is obtained.

e.g. (RANDOMISE 43) --> 764

(RDS <ch>) --> <cr> **Subr**

This selects <ch< as the current input stream, returning the previous input stream.

e.g. (RDS 1) --> 0

(READ) --> any **Fsubr**

The function returns the result of reading the next s-expression from the current input channel.

(READLINE) --> any **Fsubr**

This reads off the current input channel up to the next newline character, and forms a single identifier, which is returned.

(READ-STATUS) --> <num> **Subr**

This returns the status of the current read channel. its values are:

```
-1 - end of file reached
 0 - character ready to be READ
 1 - otherwise
```

(RECLAIM) --> number **Subr**

This forces a garbage collection. It returns the number of LISP cells free. Multiply this figure by 5 to get an idea of the number of bytes free.

(REDIRECT <old> <new>) --> NIL **Subr**

This redirects all output operation for the <old> channel to the <new> channel.

```
e.g. (REDIRECT 42 104) --> NIL
```

(REMAINDER <x> <y>) --> number **Subr**

If x>0, then x MOD (ABS y) is returned. If x>0, then y-(ABS x) MOD (ABS y) is returned.

```
e.g. (REMAINDER 7 3) --> 1
     (REMAINDER –7 3) --> 2
```

(REMFLAG <id list> <ind>) --> NIL **Subr**

This 'unflags' the list of identifiers <idlist>.

```
e.g. (REMFLAG '(A B) 'FINE) --> NIL
```

(REMOB <id>) --> <id> **Subr**

This searches the oblist for <id> and removes it if it is present.

```
e.g. (REMOB 'A) --> A
```

(REMPROP <id> <ind>**) --> any** **Subr**

This removes the property <ind> from the property list of the identifier <id>. It returns NIL if the property cannot be found.

e.g. (REMPROP 'V 'NAME) --> DD

(REPEAT <count> <exp>) --> NIL **Fsubr**

This evaluates <exp>, <count> times.

e.g. (REPEAT 5 (PRINC 'AB)) --> NIL displays ABABABABAB

(REVERSE <x>**) --> list** **Subr**

This returns a copy of the list <x> with the elements in reverse order.

e.g. (REVERSE '(A B (C D) E)) --> (E (C D) B A)

(REVERSEIP <x>**) --> list** **Subr**

This does the same as REVERSE and is much quicker, though less reliable.

.e.g. (REVERSEIP '(A B C D)) --> (C D B A)

RPAR **Var**

The value of RPAR is the character ')'.

(RPLACA <mode> <exp>**) --> any** **Subr**

This replaces the CAR field of <mode> with <exp>.

e.g. (RPLACA '(A B) 1) --> (1 B)

(RPLACD <mode> <exp>**) --> any** **Subr**

This replaces the CDR field of <mode> with <exp>.

e.g. (RPLACD '(A B) 1) --> (A.1)

(SASSOC <key> <alist>) --> (<key> <value>) or any **Subr**

This searches the <alist> for a given <key> and returns the key value pair if it is present. If the key is not found, then <fn> is evaluated with no arguments.

e.g. (SASSOC 'A '((B.27) (A.-3)) FN) --> (A.-3)
(SASSOC 'A '((B.27) '(LAMBDA NIL 5)) --> 5

(SAVE <filename>) --> <filename> **Subr**

This will save the current state of the system (to be retrieved later by LOAD).

e.g. (SAVE "NAME")

(SET <id> <exp>) --> <exp> **Subr**

This changes the value of <id> to <exp>.
e.g. (SET 'X 42) --> 42

(SETATTRIBUTES <num>) --> NIL **Subr**

This sets the graphics attribute flag byte (on the current graphics channel) to <num>. Basically this byte of flags affects the way things are plotted in the attribute mode. For a full description please refer to the EXOS specifications. The function returns NIL.

e.g. (SETATTRIBUTES 3) --> NIL

(SETCOLOUR <num> <col>) --> NIL **Subr**

This makes the 'logical' colour <num> on the palette into colour <col>.

e.g. (SETCOLOUR 3 24) --> NIL

(SETQ <id> <exp>) --> <exp> **Fsubr**

This is the same as SET but the first argument is automatically quoted.

e.g. (SETQ X 42) --> 42

(SET-TIME <id>) --> NIL **Subr**

This allows the system clock to be set (see TIME). The argument <id> should be an identifier eight characters long with the time in the format:

hh:mm:ss

e.g. (SET-TIME "01:30:42") --> NIL

(SETVIDEO <ind> <col> <x> <y>) --> NIL **Subr**

This defines a video page and should be called just before the page is OPENED or CREATED>

<ind> should be:

0 - 128 low-resolution characters (42 characters per line)
1 - high resolution pixel graphics (872 pixels)
2 - 128 high-resolution characters (84 characters per line)
5 - low-resolution pixel graphics (436 pixels)
15 - attribute mode.

The resolutions given in parentheses are for full-screen displays in the two colour mode. The vertical resolution for a full-screen is 27 characters or 243 pixels.

<col> should be:

0 - 2 colours
1 - 4 colours
2 - 16 colours
3 - 256 colours

<x> and <y> determine the size of the page.

1<x<42
0<y<255

e.g. (SETVIDEO 1 2 40 20) --> NIL

(SNDS <ch>) --> <num> **Subr**

This makes <ch> the current channel, returning the previous one.

e.g. (SNDS 56) --> 34

(SOUND <env> <p> <vl> <vr> <sty> <ch> <d> <f>) --> NIL **Subr**

This actually produces a sound (provided that the current sound channel is open to the SOUND device). The meaning of these parameters is:

<env>	- The envelope to use for the sound (see envelope). An envelope of 255 will produce a 'beep' of constant amplitude and pitch for the duration of the sound.
<p>	- The starting pitch of the sound in semitones.
<vl>	- Overall left amplitude (0 . . . 255)
<vr>	- Overal right amplitude (0 . . . 255)
<sty>	- The sound style byte. The effect of this byte is best determined by experiment! Zero gives a pure tone for tone channels and white noise for the noise channel.
<ch>	- The source for the sound - 0, 1 or 2 for the appropriate tone channel and 3 for the noise channel.
<d>	- The duration of the sound in 'ticks'. (1/50ths of a second)
<f>	- Flags byte. 0 => sound is queued up 128 => sound overides any sound queued for this channel.

The routine returns NIL.

e.g. (SOUND 255 20 3 10 30 40 0 0) --> NIL

(SPRINT <exp>) --> NIL **Subr**

This is the program formatter. It displays <exp> in a neater format.

(SUB1 <number>) --> number **Subr**

This returns <number> less one.

e.g. (SUB1 23) --> 22

(SUBLIS <alist> <exp>) --> any **Subr**

The value returned is the result of substituting the CDR of <alist> for every occurance of the CAR part in <exp>.

e.g. (SUBLIS '((A. 10) (B.C)) '(H A (B) A)) --> (H 10 (C) 10)

(SUBRP <x>) --> T OR NIL **Subr**

This returns T if <x> is a codepointer to a subr, otherwise NIL.

e.g. (SUBRP CAR) --> T

(SUBST <x> <y> <exp>) --> list **Subr**

This substitutes 'x' for 'y' in <exp>.

e.g. (SUBST 'A 'B (C B (A) (B A))) --> (C A (A) (A))

T **Id**

This represents TRUE in the system.

(TAIL <x>) --> any **Subr**

This function is identical to CDR.

e.g. (TAIL '(A . B)) --> B

(TERPRI) --> NIL **Subr**

This prints a carriage return on the screen.

(TEXT) --> NIL **Subr**

This opens up a new text page, with the number of columns being defined by the last DEFVIDEO call.

(TIME) --> identifier **Subr**

This returns the current time as an identifier, eight characters long, in the format:

hh:mm:ss

The clock starts at 00:00:00 when the machine is turned on. it may be set with the function SET-TIME (qv).

e.g. (TIME) --> 00:10:38

(TIMES <arglist>) --> number **Fsubr**

The evaluates the elements of <arglist> and then multiplies them together, returning the result.

e.g. (TIMES 2 5 -3) --> -30

(TIMES2 <x> <y>) --> number **Subr**

This is a more efficient way of multiplying just two numbers <x> and <y>.

e.g. (TIMES2 3 7) --> 21

UNDEFINED **Id**

When an identifier is first used it is given the value 'UNDEFINED'.

(UNTIL <exp>) **Fsbur**

this is used in conjunction with a LOOP.

e.g. (UNTIL (EQ A 3))

VERSION **Id**

this is a string describing the version of LISP in operation.

(WHILE <exp>) **Fsubr**

this is ued in conjunction with a LOOP.

e.g. (WHILE (EQ A 3))

(WRS <handle>) --> <handle> **Subr**

This selects channel <handle> as the current output stream.

e.g. (WRS 2) --> 3

(ZEROP <x>) --> T or NIL **Subr**

This returns T if <x> is the number zero; otherwise, it returns NIL.

e.g. (ZEROP 0) --> T

Appendix 1
EXOS VARIABLES

The following list gives the EXOS variables that can be manipulated using the EXOS-READ, EXOS-WRITE and EXOS-TOGGLE commands.

Any variable can be set to any value in the range 0-255. However, many of the variable act as switches to turn something on or off, and in these cases zero corresponds to 'on' and 255 to 'off'.

0 IRQ_ENABLE_STATE	bit 0 - set to enable sound IRQ bit 2 - set to enable 1Hz IRQ bit 4 - set to enable video IRQ bit 6 - set to enable external IRQ

Bits 1,3,5 and 7 must be zero. This variable should not be altered in normal use.

1 FLAG_SOFT_IRQ	This is set to non-zero by a device to cause a software interrupt
2 CODE_SOFT_IRQ	This should be inspected by a software interrupt routine to determine the reason for the interrupt
3 DEF_TYPE	Type of default device 0->TAPE 1->DISK
4 DE_CHAN	Default channel number
5 TIMER	This is a 1Hz down counter, which causes a software interrupt when it reaches zero, whereupon it will stop
6 LOCK_KEY	Current keyboard lock status
7 CLICK_KEY	0--> This enables key click <>0--> This disables key click

8 STOP_IRQ	0--> Stop key causes software IRQ <>0--> Stop key returns its code
9 KEY_IRQ	0--> Any key pressed causes software IRQ, as well as returning its code
10 RATE_KEY	Auto repeat rate in units of 1/50th second
11 DELAY_KEY	Delay until auto-repeat starts 0--> No auto-repeat
12 TAPE_SND	0--> This enables tape sound
13 WAIT_SND	0--> Sound driver waits when queue full <>0--> Returns SQFUL error
14 MUTE_SND	0--> This activates internal speaker <>0--> This disables internal speaker
15 BUF_SND	Sound envelope storage size in phases
16 BAUD_SER	This defines serial BAUD rate 6=>300 baud 12=>4800 baud 8=>1200 baud 14=>9600 baud 10=>2400 baud
17 FORM_SER	This defines serial word format bit 0 - no. of data bits 0=8 bits, 1=7 bits bit 1 - parity enable. Clear for no parity bit 2 - parity select. 0=even, 1=odd. bit 3 - no of stop bits 0=2, 1=1
18 ADDR_NET	Network address of machine
19 NET_IRQ	0--> This causes data received on the network to produce a software interrupt

20 CHAN_NET	Channel number of network block received
21 MACH_NET	Source machine number of network block
22 MODE_VID	Video mode
23 COLR_VID	Colour mode
24 X_SIZ_VID	X page size
25 Y_SIZ_VID	Y page size
26 ST_FLAG	0=> This displays the status line
27 BORD_VID	Border colour of screen
28 BIAS_VID	Colour bias for pallette colours 8 . . . 16
29 VID_EDIT	Channel number of video page for editor
30 KEY_EDIT	Channel number of keyboard for editor
31 BUF_EDIT	Size of editors buffer (in 256 byte pages)
32 FLG_EDIT	Flag to control reading from editor
33 SP_TAPE	Forces slow tape saving when non-zero
34 PROTECT	Makes a protected file when non-zero
35 LV_TAPE	This controls tape level output
36 REM.1	Sets state of cassette remote control 1 0-->OFF <> 0-->ON
37 REM.2	Sets state of cassette remote control 2

The following EXOS variable numbers should not be modified by the user: 0,1,2.

Appendix 2
ERROR MESSAGES

The following list represents the errors that can be produced by the IS-LISP interpreter.

1. Out of memory error
2. Execution interrupted (stop key has been pressed during calculation)
3. Interrupt during print
4. Software interrupt (handler has not been defined)
5. Arithmetic overflow
6. Division by zero
7. Number expected as an argument
8. Identifier expected
9. Byte expected (a number in the range 0-255)
10. Byte or negative number expected
11. Channel expected (number in the range 0-255)
12. Indicator expected
13. Excess "." or ")" while reading
14. Illegal dot notation
15. Number too large to be read
16. String too long (all strings have a maximum length of 255 chrs.)
17. Undefined function
18. Apply given Fsubr as function
19. Apply given number as function
20. Bad LAMBDA expression
21. Bad FUNARG expression
22. Too many arguments for LAMBDA expression
23. Too few arguments for LAMBDA expression
24. Optional arguments must follow simple arguments
25. Attempt to take CAR or CDR of an atom
26. Badly formed COND expression
27. Badly formed association list
28. Badly formed property list
29. Wrong number of arguments for primitive function
30. Unable to change the values of system variables
31. System identifier in LAMBDA/MACRO parameter list
32. MACRO form with a null parameter
33. MACRO parameter must be an atom

34. Badly formed MACRO expression
35. Lists not the same length for function PAIR
36. Bad argument to random function (must be an integer 0-2000)
37. Number expected for repeat factor
38. Bad argument to quote
39. List expected for RPLACA/RPLACD
40. Not given good list of identifiers for IMPLODE
41. Too many characters given to IMPLODE
42. Identifier expected by SET or SETQ
43. Not enough memory to load file: try closing unwanted channels and trying again
44. Cannot save with extensions loaded
45. Bad argument to SET-TIME

Appendix 3
FUNCTION KEYS

The Enterprise microcomputer has eight function keys, labelled f1 to f8. When used on their own or in conjunction with the SHIFT-KEY, these produce 16 functions, which can be defined by the user using the FKEY function.

For example, (FKEY 5 "HELLO. KEY5 PRESSED")

programs KEY5 to produce a message when function key 5 is pressed.

When the system is switched on or reset, f1 to f8 are given default values as shown below:

KEY	**FUNCTION**
1	(FLATTEN (OBLIST))
2	(DEFUN
3	(FEDIT
4	(EXOS-TOGGLE 36) toggles cassette remote 1
5	(TEXT)
6	(GRAPHICS)
7	(EXOS-TOGGLE 7) toggles keyclick
8	(RECLAIM)
9-16	the null string ""

N.B. By default these will cause a software interrupt when pressed