Abstract.

This document describes an implementation of Standard LISP which runs on Tandem Computers Inc. NonStop II and NonStop TXP systems. It is intended to be a reference document to be used in conjunction with the "Standard Lisp Report", Marti et al, SIGPLAN Notices, Volume 14, Number 10, October 1979 (here after refered to as the Standard).

The document is organized into two sections. The first section contains information about features which are specific to this implementation. The second section is a glossary of LISP objects which provides a cross-reference between this document and the Standard.

Differences from the Previous Release

Assign's and Param's are now supported by LISP. They are to be found in the globals *ASSIGNS and *PARAMS, and are automatically passed to a new process as needed. User programs may modify these lists at will.

The optional heap size and swap file information are now passed to LISP using ASSIGN's and PARAM's.

A generalized file logging capability via the function IOECHO has been added.

Better control over break during interactive sessions is now possible using the function OLDBREAK.

A list of all file handles for open files is returned by OPENFILES.

An alternative way to access bytes in strings is available via GETBYTE and PUTBYTE.

Garbage collection may be quietly invoked by GCQ.

NonStop programming is now supported by FORK and BACKUPOPEN.

PSETQ and LAST have been added from Common LISP.

SETQ has been generalized as defined in Common LISP.

The LISP library functions are now documented here as they are normally bundled with the interpreter.

Introduction.

Why LISP? Why not! It is as an excellent tool for investigating algorithms, control structures, and data structures. It also allows rapid prototyping and interactive program development. Finally, compiled LISP programs have quite respectable performance. However, it is a language (like APL) that one is not neutral to. One either accepts its quirks and sees it as a powerful tool, or points to it's quirks as good reasons not to use it.

Given that one is tempted to at least look at it, the best book to read is "LISP" by Winston & Horn. Notes are provided within this document for using it with this LISP system. An alternative, which also is an introduction to A.I., is "Artificial Intelligence" by Winston. If this seems like too much work, then there is a non-traditional introduction to LISP in "The Little LISPer" by Friedman. For the dedicated, there are gory details about internals of LISP in "Anatomy of LISP" by Allen.

As there are some differences between LISP dialects, one should consult the glossary in this document to discern them as one reads these materials.

A major problem in implementing LISP is choosing a dialect. Unfortunately, there are no readily accepted standards in this area. This in turn tends to impede program and documentation interchange. There are currently several exotic LISP's such as INTERLISP or MACLISP which would be very nice to use. However, it is not clear that one wishes to undertake the implementation of same. There is also the original LISP 1.5 (defined in 1965) which is a more manageable implementation project, but lacks facilities such as arrays, floating point numbers, or file i/o.

Luckily, a nice compromise dialect is available, Standard LISP, which provides a dialect of LISP based upon LISP 1.5, but with some nice additions based upon experience over the last 10 to 15 years. This dialect, less a few features related to compiled functions and floating point numbers has been chosen as the implementation base.

Running LISP.

The interpreter is started on a NonStop II system by the command:

:[RUN] LISP [/ [IN <in file>] [,OUT <out file>] /] [<command string>]

The <in file> will be read into the basic interpreter. If the <in file> is not the same as the <out file>, then input records will be echoed on the <out file>. In either or both files are not specified, then the command interpreter defaults will be used. LISP will prompt terminals with the prompt "<digit>~" where <digit> is the current number of unmatched left parentheses.

The LISP interpreter is terminated when an end-of-file condition is detected on the initial <in file> at the top level of the interpreter.

If a command string is present in the RUN command, then it is interpreted by the LISP interpreter, or the user's top-level function (see SAVECODE).

The default swap file for the LISP node space is a temporary file created on the default volume. An alternative volume or file may be specified by the assign:

```
:ASSIGN LISPSWAP,{ <volume> | <file> }
```

The default heap size is 100 pages. If some other value in the range 50 <= x <= 10000 is desired, then it may be set by:

```
:PARAM LISPHEAP <heap size>
```

The current configured heap size will be printed by LISP on each start up.

"LISP" Chapter Notes.

The following page notes are for the first nine chapters of "LISP", by Winston and Horn, the first edition. They point out the basic differences between MACLISP and Standard LISP.

pg 10: Functions are defined using DE rather then DEFUN.

pg 14: SET will print a warning message the first time a variable is assigned a value.

The "fc" command may be used to correct lines.

Define NEWFRIEND using DE rather than DEFUN as follows:

```
(DE NEWFRIEND (NAME)
 (SET 'ENEMIES (DELETE NAME FRIENDS))
 (SET 'FRIENDS (CONS NAME FRIENDS)))
```


pg 16: SQRT and EXPT are not supported.

pg 18: This LISP system supports one type of number, a decimal value with up to 13 digits to the left of the decimal point and 6 digits to the right of the decimal point. As a result, one does not need the functions FIX and FLOAT.

pg 24: APPEND allows only two (2) arguments. To append three lists one would use: (APPEND L1 (APPEND L2 L3))

pg 33: Functions are defined using DE:

```
(DE <function name>
    (<parameter 1> <parameter 2> ... <parameter n>)
    (<process description>)
```

Rather than typing functions directly into the LISP interpreter, one should use the function (EDIT <file>) to enter a function into a file using EDIT. When editing is complete, exit using the E command to return to LISP. The file may then be read by typing (0 <file>). LISP considers all characters on a line after a "%" to be comments.

pg 38: BOUNDP is not supported.

pg 48: FUNCALL is not supported, so use: (APPLY f (LIST p1 p2)) instead of (FUNCALL f p1 p2).

pg 64: MAPCAR expects only two (2) arguments and in the opposite order. The correct expression is (MAPCAR '(1 2 3) 'ADD1).

pg 65: PLUS cannot be used with APPLY as shown in COUNTATOMS. Instead one could use EVAL as follows:

```
(DE COUNTATOMS (S)
    (COND ((NULL S) 0)
          ((ATOM S) 1)
          (T (EVAL (CONS 'PLUS (MAPCAR S 'COUNTATOMS))))))
```

DEPTH must be solved in a similar manner as MAX cannot be used with APPLY.

pg 72: PUT is used instead of PUTPROP and the parameters are in a different order:

(PUT <atom name> <property name> <property value>).

pg 75: Arrays are called vectors and may have only one-dimension. See sections 3.9 and 2.1 of the Standard LISP Report.

pg 84: MAPCAN expects only two (2) arguments and in the opposite order. The correct expression is (MAPCAN GROCERIES 'FRUITP).

pg 90: A "!" is used to indicate that the following character is to lose any special properties on input. The use of "|" is not supported.

pg 91: IMPLODE must be defined by the user as:

```
(DE IMPLODE (X) (INTERN (COMPRESS X)))
```

pg 92: PRIN2 is used instead of PRINC.

pg 97: FEXPR's are defined:

```
(DF <function name> (<single parameter>) <body>)
```

pg 99: MACRO's are defined:

```
(DM <function name> (<single parameter>) <body>)
```

pg 100: LEXPR's are not supported.

pg 114: DELETE is not a destructive operation.

The second edition of LISP by Winston and Horn uses Common LISP. The following set of notes for the first 12 chapters will help one get started:

pg 15: Use PLUS instead of "+".

pg 16: SETQ will print a warning message the first time a variable is assigned a value.

The "fc" command may be used to correct lines.

Define NEWFRIEND using DE rather than DEFUN and DELETE rather than REMOVE as follows:

```
(DE NEWFRIEND (NAME)
 (SETQ ENEMIES (DELETE NAME FRIENDS))
 (SETQ FRIENDS (CONS NAME FRIENDS)))
```

pg 18: Use TIMES for "*", QUOTIENT for "/". SQRT and EXPT are not supported.

The characer "!" indicates that the following character gets special treatment.

pg 19: A unary "-" is the function MINUS, and subtraction is done using the function DIFFERENCE.

pg 21: This LISP system supports one type of number, a decimal value with up to 13 digits to the left of the decimal point and 6 digits to the right of the decimal point. As a result, one does not need the functions FIX and FLOAT.

pg 22: TRUNCATE is not implemented, but can be defined by:

```
(DE TRUNCATE (x) (QUOTIENT x 1))
```

The remainder function is REMAINDER.

pg 30: APPEND allows only two (2) arguments. To append three lists one would use: (APPEND L1 (APPEND L2 L3))

pg 39: Functions are defined using DE:

```
(DE <function name>
    (<parameter 1> <parameter 2> ... <parameter n>)
    (<process description>)
```

Rather than typing functions directly into the LISP interpreter, one should use the function (EDIT <file>) to enter a function into a file using EDIT. When editing is complete, exit using the E command to return to LISP. The file may then be read by typing (0 <file>). LISP considers all characters on a line after a "%" to be comments.

pg 45: LISTP is not implemented, but can be defined as:

```
(DE LISTP (x) (or (null x) (pairp x)).
```

pg 46: Use EQN instead of "=" to compare two numbers.

pg 48: Use LESSP for "<" and GREATERP for ">".

pg 49: EVENP is not implemented, but can be defined as:

```
(DE EVENP (x) (ZEROP (REMAINDER x 2)))
```

pg 54: (sub)Standard LISP uses dynamic scoping.

pg 56: FUNCALL is not supported, so use: (APPLY f (LIST p1 p2)) instead of (FUNCALL f p1 p2).

pg 57: LET and LET* are defined in the LISP library.

pg 79: MAPCAR expects only two (2) arguments and in the opposite order. The correct expression is (MAPCAR '(1 2 3) 'ODDP), where one must define ODDP as:

```
(DE ODDP (x) (ONEP (REMAINDER x 2)))
```

pg 81: PLUS cannot be used with APPLY as shown in COUNTATOMS. Instead one could use EVAL as follows:

```
(DE COUNTATOMS (L)
    (COND ((NULL L) 0)
          ((ATOM L) 1)
          (T (EVAL (CONS 'PLUS (MAPCAR L 'COUNTATOMS))))))
```

DEPTH must be solved in a similar manner as MAX cannot be used with APPLY.

pg 83: MAPCAN takes two arguments which are in the opposite order.

pg 84: DO and DO* are defined in the LISP library.

pg 96: SETF is not defined in (sub)Standard LISP. Use (PUT <symbol> <property name> <value>) to put an item onto the property list.

pg 100: DEFSTRUCT is not supported, nor are keyword arguments.

pg 104: The #' shorthand for FUNCTION is not supported.

pg 114: Use "!" to preceed a special character that you want to treat as a letter in an identifier.

pg 115: Use (PRIN2 " ") to print a string. This is a good time to examine the Standard and Notes to the Standard concerning I/O as every LISP system is different.

pg 121: (sub)Standard LISP does not have optional parameters or a backquote facility. However it does have macros so you should not ignore this chapter.

pg 131: DELETE is not a destructive operation.

pg 142: This discussion of EQL and EQUAL only applies to Common LISP. See the Standard for a similar discussion of EQ, EQN, and EQUAL.

pg 143: (sub)Standard LISP does not have keywords.

pg 151: Arrays in (sub)Standard LISP may have only one dimension and are accessed in a different manner. You best study the Standard before trying to read this chapter.

pg 173: Use a different name for EXPAND as it is already defined.

A - Primitive data types.

Primitive data types are as described in the Standard with a few exceptions. First, the interpreter supports only one type of number, namely decimal, with a maximum of 13 decimal places to the left and 6 places to the right of the decimal point. This provides some level of decimal numbers, allows the interpreter to run without the floating point option, and removes a whole lot of headaches for the implementer.

Second, since no compilation of functions is being done, the concept of LOCAL binding and the type function-pointer do not exist.

Identifiers longer than 24 characters and strings longer than 80 characters will be supported. However, they must fit in one input and output record.

The notation "stringid" is used to denote a value which may be either a string or an id. If an id is provided, then it's print name will be used as a character string.

The notation "stringNIL" is used to denote a value which may be either a string or NIL.

The notation "numberNIL" is used to denote a value which may be either a number or NIL.

A "subr" is an EXPR which is internal to the LISP interpreter.

A "fsubr" is a FEXPR which is internal to the LISP interpreter.

A "msubr" is a MACRO which is internal to the LISP interpreter.

B - Structures.

A more liberal (similar to LISP 1.6) version of the cond-form is allowed:

```
( S0 [ Sn ] )
```

where Sn is zero or more S-expression's. Similarly, the body of a lambda expression may contain more than one S-expression. The value of the final S-expression is the value that will be returned.

C - Error and Warning Messages.

The interpreter has a somewhat simplistic view of error recovery. When an error is detected, a message will be printed on the initial output file and an upexit made to the top level of the interpreter. Any files that have been opened will be closed with any partial outputs written to the file. This mechanism may be overridden by using CATCH or CATCHALL

All error messages are preceeded by "*****". At the user's option, the current stack of functions may be dumped. This is enabled by (DUMPSTACK T) and disabled by (DUMPSTACK NIL) which is the default. The function returns the current setting of the stack dump.

If these actions are not desired, then the user should read the following section which describes the facilities for dynamic uplevel exits.

D - Dynamic Non-local Exits.

A mechanism for allowing arbitrary function exits is provided by the "catch" and "throw" mechanisms from Common LISP. In general terms, a "catch" defines a place for a "throw" to return a value. The function

(THROW <tag> <result>)

is used to throw the value of <result> to the value of <tag>, which is expected to be an id.

The simplist form of a "catch" is:

(CATCH <tag> [<any>])

where <tag> is first evaluated to produce the id which names the catch. The optional <any>'s are then evaluated as an implicit PROGN and the result of the last <any> is returned unless a throw occurs to the <tag> which names the catch. In that case, the value of the function is the result which was thrown.

A more general "catch" is:

(CATCHALL <function> [<any>])

which catches any throw. The value of <function> is expected to be a function of two arguments. The optional <any>'s are then evaluated as previously described. If no throw occurs during the evaluation of the optional <any>'s, then it acts just like CATCH, returning the value of the last <any>. Any throw which is not caught by an inner CATCH or CATCHALL will cause the function to be evaluated with the thrown <tag> and <result> as it's arguments and it's result will be the value of the CATCHALL. Note that it is possible to relay the throw during the execution of the catch function.

An almost identical function is:

(UNWINDALL <function> [<any>])

which is just like CATCHALL except that the <function> will always be called. If there was no throw, then the value of the thrown <tag> will be NIL.

A program can assure that cleanup code is always run by using the form:

(UNWINDPROTECT <protected-any> [<cleanup-any>])

which will cause <protected-any> to be evaluated and then evaluate the optional <cleanup-any>'s and discard their values. The value returned will be that obtained in evaluating the <protected-any>. The <cleanup-any>'s will always be evaluated even if the evaluation of <protected-any> is aborted by a throw of any kind.

Errors generated by the LISP interpreter are reported by "throwing" them. Any error will cause a "throw" with a tag of *LISPERROR* and the result equal to the error message string. A break interrupt from a terminal will cause a "throw" with a tag of *BREAK* and a result of "* BREAK *". It is important to note that either of these "throws" may be caught, examined, and then relayed in a transparent manner.

Needless to say, some examples are in order:

```
(catch 'a (catch 'b 1))
1

(catch 'a (catch 'b (throw 'a 1)))
1

(catch 'a (catch 'b (unwindprotect 1 (print 'protected))))
PROTECTED
1

(catch 'a (catch 'b (unwindprotect (throw 'a 1) (print 'protected))))
PROTECTED
1

(catchall '(lambda (tag value) (print (list tag value)) value) 1)
1

(catchall
 '(lambda (tag value) (print (list tag value)) value)
 (throw 'a 1))
(A 1)
1

(catchall
 '(lambda (tag value) (print (list tag value)) value)
 (car))
(*LISPERROR* "*****  CAR  Wrong number of parameters")
"*****  CAR  Wrong number of parameters"

(unwindall '(lambda (tag value) (print (list tag value)) value) 1)
(NIL 1)
1
```



```
(unwindall
 '(lambda (tag value) (print (list tag value)) value)
 (throw 'a 1))
(A 1)
1
```

The Standard LISP functions ERROR and ERRORSET can be implemented as follows:

```
(de ERROR (x) (throw '*LISPERROR* x))

(dm ERRORSET (x)
 (subst
  (cadr x)
  '<code>
  '(progn
    (put 'emsg* 'value nil)
    (catchall
     '(lambda (f v) (put 'emsg* 'value v) nil)
     (list <code>)))))
```

E - Functions on Dotted-Pairs.

Taking a queue from other LISP systems, CAR, CDR and their composites return NIL rather than causing an error when applied to an atom.

Additional functions for generating lists have been introduced so as to be compatible with the Standard LISP compiler. Their definitions are as follows:

```
(de  NCONS (x) (cons x nil))

(de  LIST2 (v w) (list v w))

(de  LIST3 (v w x) (list v w x))

(de  LIST4 (v w x y) (list v w x y))

(de  LIST5 (v w x y z) (list v w x y z))

(de  XCONS (x y) (cons y x))
```

F - GENSYM Print Names.

The print names for id's produced by GENSYM will be of the form Gxxxx where xxxx is a 4 digit number.

G - Property List Operations.

GET has been extended to return the whole property list by using:

(GET <id> T).

Function definitions are on the property list, but the functions GETD, PUTD, and REMD are supplied so as to be compatible with the standard. Flags are not implemented.

H - Variables and Bindings.

All variables are fluid. If the variable is not found on the ALIST, then its value is on the variable's property list with the indicator "VALUE". An assignment to a variable which is neither on the ALIST nor has the indicator "VALUE" will cause the value to be placed on the property list with the indicator "VALUE" and a warning message will be issued.

The function (FLUID id-list) is defined as follows:

(DE FLUID (idlist) T)

It is included so that it may be used to note fluid variables within functions which may be either interpreted or compiled.

The form SETQ has been extended to allow multiple identifier/ expression pairs. The expressions are evaluated and assigned in a left-to-right order. The value returned will be the value of the right most expression. A degenerate form (SETQ) is allowed whose value is NIL.

A form PSETQ has been added from Common LISP. It's form is identical to SETQ, but all expressions are evaluated before any assignments are made and the value of the form is always NIL. For example, the values of three identifiers could be rotated by:

(PSETQ a b b c c a)

I - WHILE.

A "while" construct is provided of the form:

(WHILE <p> [<any>])

<p> is evaluated and if it is equal to NIL then NIL is returned. Otherwise, the optional <any>'s are evaluated as an implied PROGN and then the test on <p> is repeated.

J - The Interpreter.

The function FUNCTION will result in the current A-list being bound with the function and a FUNARG will be formed. That A-list will then be used when the function is evaluated. The mechanics of this are exactly as defined for LISP 1.5 in the "LISP 1.5 Programmer's Manual", McCarthy et al.

K - Input / Output.

The Standard LISP "stream" input and output is decidedly lacking in more general file access capabilities. To help correct this deficiency, additional file open options are supported. The OPEN macro is generalized to allow a list of open options:

(OPEN <filename> <qualifier> [<qualifier>])

where <filename> is as defined in the Standard, or a crtpid, and the <qualifier>'s are selected from the following list:

INPUT	read access (implies read-only shared access).
OUTPUT	write access (implies exclusive access).
EXCLUSIVE	exclusive access (overrides implied access).
SHARED	shared access (overrides implied access).
OLD	file must exist.
NEW	create file if it does not exist, or purgedata if it does. If SEXPR is not specified, then an EDIT file will be created.
SEXPR	the file contains arbitrary length S-expressions stored in multiple records using "hidden keys".
UPDATE	records which already exist may be overwritten.
ECHO	records which are read are to be echoed on the current output file. This is the same as (IOECHO file T).
PADTEXT	short records will be blank padded.

The result of this operation is a filehandle which is used with RDS, WRS, EXPANDFNAME, and CLOSE.

To ease the linkage with compiled code, OPEN is defined as a macro of the form:

```
(dm OPEN (x) (list 'OPENLIS (cons 'list (cdr x))))
```

and a function OPENLIS is defined which accepts as it's argument a list of open options.

The current input and output files are selected and positioned by the functions RDS (for input) and WRS (for output) respectively. In all cases, the value of the function is the previous file handle. The first case simply designates the file and no positioning is done:

```
(RDS <filehandle>)

(WRS <filehandle>)
```

Additional positioning options are defined in the following sections which describe disc file positioning and $RECEIVE handling.

As with OPEN, WRS and RDS are defined as macros of the form:

```
(dm RDS (x) (list 'RDSLIS (cons 'list (cdr x))))

(dm WRS (x) (list 'WRSLIS (cons 'list (cdr x))))
```

and the functions RDSLIS and WRSLIS are defined which accept a list of arguments as their argument.

- File Input Operations.

READ has been extended to allow special characters to appear in id's which allows easier input of GUARDIAN file names. The syntax for the scanner is now the following grammar.

```
<scan token>     ::= space <scan token>
                 ::= EOL <scan token>
                 ::= <comment> <scan token>
                 ::= EOF
                 ::= <string>
                 ::= <number>
                 ::= <magic char>
                 ::= <id>

<comment>        ::= % <comment body>

<comment body>   ::= EOL
                 ::= <any character> <comment body>

<string>         ::= " <characters> "

<characters>     ::= >=0 <anycharacter>'s with " represented by ""

<number>         ::= <number1>
                 ::= <sign> <number1>

<number1>        ::= . <integer>
                 ::= <integer> . <integer>
                 ::= <integer> .
```



```
<sign>          ::= +
                ::= -

<integer>       ::= <digit>
                ::= <digit> <integer>

<magic char>    ::= any character in the string "(.)[,]'%"

<id>            ::= <start id>
                ::= <start id> <restids>

<start id>      ::= ! <any character>
                ::= any alphabetic character
                ::= any special character which is not a <magic char>

<restids>       ::= <restid>
                ::= <restid> <restids>

<rest id>       ::= <start id>
                ::= .
                ::= <any character> not a space or <magic char>

<any character> ::= any character except EOL or EOF
```

An ESC character may be entered on input from the terminal which will result in an upexit as though break were pressed while the interpreter was running.

The "FC" command is supported if and only if it is entered as the first two characters of the input line.

The next character in the input stream may be examined without advancing the scanner by:

```
(PEEKCH)
```

It returns the same values as (READCH).

The function:

```
(READRECORD)
```

is provided to read an entire record as a string. It will return either a string representing the record or EOF.

The input record size may be changed or checked by the function:

```
(READLENGTH <length>)
```

If the parameter is a number, then it sets a new input record length and returns the old input record length as it's value. A NIL parameter allows the current length to be queried.

The character position for the next character in the input record is returned by:

(RPOSN)

The default input prompt for terminal and process I/O may be changed by setting the fluid variable *PROMPT to a stringid. Those characters as printed by PRIN2 will then be used to prompt input. The default prompt will occur when *PROMPT is equal to NIL.

- Misc. I/O Operations.

The current default volume and subvol may be obtained by:

(DEFAULTVSV nil)

It may be set to a new value by:

(DEFAULTVSV <string>)

where the <string> is of the form: "[\<system>.]$<volume>.<subvol>".

A fully expanded file name may be obtained by:

(EXPANDFNAME <stringid>)

where the <stringid> is a file name or process' crtpid. The fully expanded file name of an open file may be obtained:

(EXPANDFNAME <filehandle>)

A list of the file handles for all open files is returned by the function:

(OPENFILES)

A file may be purged by:

(PURGE <stringid>)

where the stringid is a file name. The function returns T if the file was purged or NIL if the file did not exist.

A GUARDIAN CONTROL operation may be performed on a file by:

(IOCONTROL <filehandle> <number> <number>)

The value returned by this function will be the GUARDIAN error number.

Logging or echoing of one file on another can be controlled by:

(IOECHO <filehandle1> <option>)

If <option> is equal to T, then all records read and written to <filehandle1> will be written to the current output file. If <option> is equal to <filehandle2>, then all records read and written to <filehandle1> will also be written to <filehandle2>. Finally, if <option> is NIL, then no file logging will be done. The value of the function is the previous logging option.

A GUARDIAN SETMODE operation may be performed on a file by:

```
(IOSETMODE <filehandle> <number> <number> <number>)
```

The value returned by this function will be a list of three numbers: the GUARDIAN error number, oldvalue[0], and oldvalue[1]. See the GUARDIAN manual for more information on the parameter values for these two functions.

If the standard input file is a terminal, then break will be held during execution, but returned to the previous owner when the terminal is being read. The previous break owner may be queried and set by:

```
(OLDBREAK <value>)
```

where <value> is expected to be either NIL to check the current value, or a list of two integers which will the two parameters supplied to SETMODE when break is returned during input. One can disable break during terminal input by:

```
(OLDBREAK '(0 0))
```

or one can have LISP keep break by:

```
(OLDBREAK
 '((plus
    (times 256 (getbyte $myprocess$ 6))
    (getbyte $myprocess$ 7))
   0))
```

The value of the function is always a list of two integers which is the previous value of OLDBREAK. The user of this function is expected to be familar with GUARDIAN's break handling.

- Disc File Positioning.

For disc files, a GUARDIAN POSITION operation to be done along with the file selection:

```
(RDS <file handle> <record specifier>)

(WRS <file handle> <record specifier>)
```

where <file handle> is as previously defined and <record specifier> is a numeric value.

Another option for disc files causes the GUARDIAN KEYPOSITION operation to be performed along with the file selection. It is done by:

(RDS <file handle> <key> <key specifier> <mode>)

(WRS <file handle> <key> <key specifier> <mode>)

The <file handle> is as previously defined and the <key> is either a stringid or an integer (which will be considered to be a 4-byte key). For files which have not been designated SEXPR at OPEN time, the <key specifier> designates which key is to be used and it expected to be either a numeric value or a one or two character string or id. Finally, the mode is expected to be: APPROXIMATE, GENERIC, or EXACT. For APPROXIMATE and GENERIC searches, the length of the <key> is passed to KEYPOSITION as the <compare length> and <key length>. For an EXACT position, <key> is be padded with nulls to the length of the key.

It should be realized that when a file is open for both INPUT and OUTPUT that the RDS and WRS positioning modes interact as only one GUARDIAN open is done by the LISP interpreter.

The standard LISP I/O functions are then used to read and write records. Under most circumstances, it is the user's responsibility to put the keys in the right places and worry about record lengths. The exception is files designated SEXPR when they are opened. This type of file provides an analog of property lists which resides on disc. It allows the storage and retrieval of arbitrary S-expressions by key without having to concern oneself with record sizes.

Files of this type are key-sequenced files with one key which starts in the first byte of the record. Following the key will be a record which makes up all or part of the S-expression. S-expressions may be read from this type of file by the following mechanism. First, indicate and position the file by doing a:

(RDS <file handle> <key> 0 'GENERIC)

This results in the key being padded by nulls until it is the key length-2 of the file. A generic KEYPOSITION is then be done. As each record is read from this type of file, the key is be stepped over before the S-expression is scanned. As many records as needed to complete the S-expression are read. Note that if the key does not exist, then an end-of-file indication is returned.

S-expressions are written to this type of file by first selecting the file and defining the <key> by:

(WRS <file handle> <key> 0 'EXACT)

This results in the <key> being null padded to its defined length and the placed into the output buffer for the file. As each record making up the s-expression is written, the last two bytes of the key are

incremented. Thus, the S-expression may occupy up to 2**16 records in the file.

A record in an S-expression file may be deleted by following the WRS with a TERPRI.

- $RECEIVE Handling.

$RECEIVE may be accessed using the LISP I/O system. If it is opened for INPUT, then it may only be used to pick up messages. However, if it is opened for OUTPUT as well, then one may write to the file which will result in a REPLY to the sending process.

When $RECEIVE is currently selected by RDS, the information concerning the last read message is available via the function:

```
(RMESSAGETAG)
```

It will return a messagetag which is a three element list of:

message flag:	NIL = user message T = system message
sender's id:	a string which is the GUARDIAN crtpid of the sender process
reply flag:	NIL = message needs no reply <> NIL = message needs a reply

If it is called when $RECEIVE is not the current file, then its value is NIL.

A process may check to see if any messages are queued on $RECEIVE by:

```
(WAITFORRECEIVE <timeout>)
```

where <timeout> is either a -1 indicating wait indefinitely, or <timeout>/100 is the number of seconds to wait until something shows up. The function returns T if there are messages waiting, in which case it may not have waited for the whole timeout, or NIL if it timed out without any messages being queued.

Normally messages are replied to in arrival order. Thus, when:

```
(WRS <$receive handle>)
```

is used, the program will be replying to the last message. It is possible to reply to messages out of order. This is done by:

```
(WRS <$receive handle> <messagetag>)
```

where <messagetag> was returned by (RMESSAGETAG) when the message was read. Finally, a program may reply to a message with an error by:

(WRS <$receive handle> <messagetag> <error number>)

The following function is an example of how one can drive a process with a command string using $RECEIVE:

```
(de PROMPT (crtpid command recvhandle)
 (prog (r ords owrs)
  (setq ords (rds recvhandle))
  LOOP
  (setq r (readrecord))
  (or (checkprocess crtpid) (return nil))
  (or (equal (cadr (rmessagetag)) crtpid) (return nil))
  (or (car (rmessagetag)) (prin2 r))
  (cond ((null (caddr (rmessagetag))) (terpri) (go loop)))
  (prin2 command)
  (terpri)
  (setq owrs (wrs recvhandle (rmessagetag)))
  (prin2 command)
  (terpri)
  (rds ords)
  (wrs owrs)
  (return t)))
```

The final facility related to $RECEIVE is that one may attach a "demon" function which will be automatically invoked whenever the LISP system reads a message from $RECEIVE. The function is set by:

(ONRECEIVE <function>)

where:

<function>: NIL: remove any current demon
form: demon function

The function will be APPLYed with the following four arguments:

message:	a LISP string holding the message
message flag:	NIL = user message T = system message
sender's id:	a string which is the GUARDIAN crtpid of the sender process
reply flag:	NIL = message needs no reply <> NIL = message needs a reply

The function returns T if the message is to be queued for later reading, or NIL if the message is to be flushed. The function may also use THROW to cause an upexit on a break message or other appropriate message. If "reply flag" is T and a THROW is done, the demon function must first reply to the message via WRS.

L - String Primitives.

A number may be converted to a one-character string which is its ASCII equivilant by:

(INTTOCHAR <number>)

and a one-character stringid may be converted to a number by:

(CHARTOINT <stringid>)

The length of a stringid is returned by:

(LENGTHSTR <stringid>)

A new string can be formed by:

(MAKESTR <length> <stringid>)

which will make a string of <length> bytes which will be initialized to <stringid>. If <stringid> is the null string, then the new string will be blank filled. Otherwise it will be copied as many times as needed to fill the new string.

Two strings may be joined to form a new string by the function:

(APPENDSTR2 <stringid> <stringid>)

and more than two may be joined by the macro:

(APPENDSTR <stringid> <stringid> [<stringid>])

The final set of operations work on a subset of string. It is designated by the stringid, a numeric offset into the string where 0 is the first character of the string, and a character count. A string can be copied by:

(COPYSTR <stringid> <offset> <count>)

Two strings can be compared by the boolean predicates:

(EQUALSTR <stringid> <offset> <stringid> <offset> <count>)

(GREATERPSTR <stringid> <offset> <stringid> <offset> <count>)

(LESSPSTR <stringid> <offset> <stringid> <offset> <count>)

Finally, characters in a string may be destructively replaced by:

(PUTSTR <string> <offset> <stringid> <offset <count>)

Note that the destination must be a string as one is not allowed to overwrite an id's print name.

A byte within a string may be accessed as a number by:

(GETBYTE <stringid> <offset>)

and a byte in a string may be destructively replaced by:

(PUTBYTE <string> <offset> <number>)

M - Function and Variable Tracing.

A trace facility is provided which will trace the input parameters and values returned for selected functions or the values assigned to identifiers by SET or SETQ. It expects one or more id's as its arguments which are the names of functions or identifiers to be traced:

(TRACE [id])

All functions will be traced by:

(TRACE T)

and tracing will be turned off by:

(TRACE)

The value returned by TRACE is list supplied on the previous call to TRACE. The trace records will always be written to LISP's standard output file, rather than to the current output file.

N - Process Creation Functions.

A process is created by the function:

(CREATEPROCESS <program> <pri> <cpu> <name> <in> <out> <command>)

where the operands are:

<program>: stringid name of the program file

<pri>: NIL: use LISP's priority
number: relative priority to LISP's priority

<cpu>: NIL: use LISP's processor
number: processor number

<name>: NIL: process is to be unnamed
stringid: process name

<in>: NIL: use LISP's standard in file
stringid: in file name


```
<out>:      NIL:  use LISP's standard out file
            stringid:  out file name

<command>:  NIL:  no command string
            stringid:  command string
```

The value returned is an 8-character string which is the GUARDIAN CRTPID of the process. This value may be used as a file name for any functions requiring one, as well as with the functions CHECKPROCESS and STOPPROCESS.

A program may check to see if a process still exists by the boolean:

```
(CHECKPROCESS <crtpid>)
```

which returns T if it does, or NIL if it does not.

A process may be stopped by the following function which always returns NIL.

```
(STOPPROCESS <crtpid>)
```

The crtpid for the LISP process is available as the value of the global variable $MYPROCESS$.

A list of assigns is maintained in *ASSIGN and is composed of elements of the form:

```
( { <progam unit name>       | NIL }
    <logical file name>
  { <tandem file name>       | NIL }
  { <primary extent size>    | NIL }
  { <secondary extent size>  | NIL }
  { <file code>              | NIL }
  { <exclusion spec>         | NIL }
  { <access spec>            | NIL }
  { <record size>            | NIL }
  { <block size>             | NIL } )
```

where the first three items are either strings or NIL and the rest are either integers or NIL.

A similar mechanism exists to support params. They are kept on the list *PARAMS which is composed of elements of the form:

```
( <parameter name> <parameter value> )
```

where <parameter name> and <parameter value> are strings.

*ASSIGNS and *PARAMS initially contain any assigns and params passed to LISP when it started. Their contents will be used to generate messages that will be sent to processes created by CREATEPROCESS. LISP programs may modify these lists as desired, but they must contain valid elements when CREATEPROCESS is called. Additional information

on assigns and params is to be found in the GUARDIAN Operating System Programming Manual.

The LISP system supports a form of process creation analogous to a UNIX "fork" which may be used to build multi-process LISP systems, or NonStop LISP process-pairs. A LISP process forks by calling:

(FORK <form> <processor> <swap file>)

where the operands are:

<form>: S-expression to evaluate in the forked process

<processor>: NIL: use LISP's processor
number: processor number

<swap file>: NIL: put heap on LISP's program file volume
volume: put heap on that volume
file: put heap in that file

The function will return the crtpid of the fork process. The fork process will have a copy of the LISP space of the original process, but it will not have a copy of the stack, nor will it have opened any files. File opens and checkpoints must be handled by the form that was passed to the fork process.

A process may have it's fork perform a backup open on a file by evaluating:

(BACKUPOPEN <file handle>)

in its process and then checkpointing the resultant form to the fork who will then evaluate it. Needless to say this is a very concise explanation of these functions. Users are strongly encouraged to examine the library functions CREATEBACKUP, CHECKPOINT, and MONITORPRIMARY to see how they may be used.

O - OBLIST.

The OBLIST in this LISP system is organized as an array of hash buckets which hold lists of interned id's. It is visible via the global id *OBLIST*. In addition, one may change the oblist by setting *OBLIST* to some different array. This could allow one to have multiple name spaces in the LISP system.

The hash function used to intern an id is:

(SXHASH <any>)

which will return a number which is the hash value of the LISP object. One use of this is to combine it with the vector primitives to provide hashed arrays.

P - Misc. Additions.

A special form, EDIT, has been added which allows access to the editor. It may be used as:

```
(EDIT [ <stringid> ])
```

Another special form provides an obey file capability. It is used:

```
(O <filename>)
```

It has a "quiet" form which does not echo the input on the current output device:

```
(OQ <filename>)
```

An edit of a file followed by an obey of it is provided by the form:

```
(EO <filename>)
```

The function:

```
(GC)
```

will run the garbage collector and cause the garbage collector message to be printed. Garbage collection may also be invoked and no message printed by the function:

```
(GCQ)
```

The function:

```
(LISPVERSION)
```

will return the version string for LISP interpreter. The version string is currently "15dec83Djfb".

The last dotted-pair of any expression is returned by the function:

```
(LAST any)
```

For example: (last '(a b c)) = (c) (last '(a . b)) = (a . b)

The function:

```
(REVERSIP <any-list>)
```

will reverse in place any list. It's LISP definition is as follows:

```
(de  REVERSIP  (u)
 (prog (x y)
  (while u (setq x (cdr u)) (setq y (rplacd u y)) (setq u x))
  (return y)))
```


The function:

(TIMEOFDAY)

will return a seven-item list which is the values returned by the GUARDIAN procedure TIME.

One may save the current state of the interpreter in a program file via the function:

(SAVECODE <filename> <function>)

When <filename> is run, the function <function> will be evaluated and then the program will stop without printing the results of the function. This allows one to build handy utility programs with LISP that may be given to others who know nothing about LISP.

If the value of <function> is equal to NIL, then the normal top-level processing loop will be used. This allows one to save the LISP system between working sessions.

A sorted copy of a list may be created by the function:

(SORT <list> <function>)

where <list> is any list (including NIL) and <function> is a boolean with two parameters which returns T if the first item should be before the second. The actual sort is done using GUARDIAN's procedure HEAPSORT.

Q - The LISP Library.

The LISP library consists of a number of functions written in LISP which enhance the basic LISP system. As they are usually included, they are defined in this document.

Additional functions extend the function definition and editting facilities. When a function is defined, the file containing it's definition will be saved on the function name's property list under the indicator *SOURCEFILE. The function definition may then be editted and the file reloaded by the functions:

(EO <function name>)

(EOQ <function name>)

where the later form "quietly" reloading the file. If no function with that name is found, then EO and EOQ will assume that the name is the name of a file. One can avoid the function check by supplying the file name as a string. One can easily redefine EO and EOQ to use alternatives to EDIT.

A list of all functions defined in a file can be obtained by:

(DEFINEDIN file)

The library also provides protection against the redefinition of a function. The default is to only allow the function to be redefined by a definition coming from the same file as the original definition. Any attempt at an illegal redefinition will result in a warning message and the new definition will be ignored.

One can enable function redefinition by setting the global variable *REDEFINE to either a list of all function names that one wishes to redefine, or T in which case any function may be redefined.

A minimal debug package is provided by the functions SBPT, CBPT, and BREAK. A breakpoint is set at the start and the return of a function by:

(SBPT <function> [<condition>])

where the <condition> is an optional boolean expression which provides a conditional breakpoint capability. A breakpoint can be removed from a function by:

(CBPT <function>)

When a breakpoint is hit, the function name is printed and then the user is prompted for input. If #R is entered, then execution will continue. If #QUIT is entered, then an upexit to the top level of the interpreter will occur. Any other input will be evaluated and the value will be printed. The value of the function is in #RESULT. It may be changed via SETQ to return a different value. #RESULT will have the value "#CALL" when the function is called. Finally, breakpoints may be explicitly placed within a function definition by:

(BREAK <any>)

which will result in the message <any> being printed and then normal breakpoint processing will commence.

Large LISP systems often want to have several programs within them, yet still avoid conflicts on identifier names. For this reason a "multiple oblist" package is provided. The first set of functions provide analogs to the property list functions which allow access into another oblist.

(GETXO <oblist> <identifier> <indicator>)

(GETDXO <oblist> <identifier>)

(PUTXO <oblist> <identifier> <indicator> <value>)

(PUTDXO <oblist> <identifier> <indicator> <value>)

(REMPROPXO <oblist> <identifier> <indicator>)

(REMDXO <oblist> <identifier>)

The following function will add an ID (or list of ID's) and it's (their) property list(s) to the current oblist. In doing so, the id WILL BE REMOB'ed from the current oblist. The result will be that the id in one space will be EQ the id in the other space and thus shared between the spaces.

(INTERNXO <oblist> <identifier>)

The current value of *OBLIST* may be saved on *OBLIST* under the indicator PDL (push down list) and *OBLIST* set to either a copy of it, or a previously saved oblist by the following function:

(PUSHXO [<existing oblist>])

The *OBLIST* is "popped" and optionally saved in an ID by the following function:

(POPXO [<oblist name>])

An example of the use of this package can be seen in the LISP compiler documentation where it is used to "hide" the compiler from the user's program.

Functions may be pretty-printed by:

(PPFUN [<function> ...] ["<list file>"])

which will print each definition in a form that can be read by READ. If a string defining the list file is supplied, then the definitions will be appended to that file.

Any expression may be pretty-printed on the current output file by the function:

(PPRINT <any>)

whose value is always NIL.

Formatted output can be produced by the macro PRINTUSING. Each argument will be examined and processed from left-to-right as follows. A "/" will cause a new line to be started. TAB will skip to the column equal to the value of the following argument. All others arguments will be evaluated and then printed using PRIN2. The function will normally terminate the print line unless the last argument is a "\".

Text files containing LISP definitions can be TGAL'ed in a pleasing manner by the form:

(TGAL [<file name> ...] <output file>)

A copy of any S-expression may be made by:

(COPYSEXPR <any s-expression>)

The decimal fraction of two integers is the value of:

(QUOTIENTF <integer 1> <integer 2>)

A string comparison function suitable for use with SORT is:

(SORTSTRINGP <stringid 1> <stringid 2>)

The library contains the following functions from Common LISP:

(LIST* [<arguments>])

which constructs a list like LIST except that the last cons of the constructed list is "dotted". That is: (list* a b c) => (a b . c).

Local variables can be created and optionally assigned values (the default is NIL) by:

(LET ({ <identifier> | (<identifier> <value>) }) form)

(LET* ({ <identifier> | (<identifier> <value>) }) form)

LET will assign all values in parallel, whereas LET* will do left-to-right sequential assignment.

Short hand forms of COND are provided:

(IF <test> <then> <else>) => (cond (<test> <then>) (T <else>))

(WHEN <test> <form>) => (cond (<test> <form>) (T NIL))

(UNLESS <test> <form>) => (cond ((not <test>) <form>) (T NIL))

Good iteration forms are provided by:

(DO ({var[init[step]]}...) (end-test result ...) statement ...)

(DO* ({var[init[step]]}...) (end-test result ...) statement ...)

where DO will do assignments in parallel and DO* will do assignments sequentially. Either "LISP" or "Common LISP: The Language" should be consulted for the definitions.

Two functions are provided to support NonStop process-pairs. A process creates a backup by:

(CREATEBACKUP <on takeover> <cpu> <swap>)

where <on takeover> is to be evaluated when the process takes over, <cpu> is the processor to run the backup in, and <swap> is the swap file as defined in FORK. The result of this function is a two element list consisting of the crtpid and the file handle for the backup

process. The backup will have opened all files currently open by the primary.

Checkpointing is done by sending an s-expression to the backup where it will be evaluated. The function:

(CHECKPOINT <createbackup result> <any>)

which will return T if the checkpoint was successful, or NIL if it was not.

User SUBR's and FSUBR's.

All subrs, fsubrs, and msubrs (compiled macros) are denoted by tags on the procedure name. The tags are of the form:

^F	indicates a FSUBR
^M	indicates a MSUBR
^Snn	indicates a SUBR where "nn" is the number of arguments that the function expects.

A set of 15 "registers" are defined which are used to pass arguements into a function. All functions return their value in REG1. The register save protocol is that the caller must save all needed values before calling other routines UNLESS that routine is defined as not destroying the register values.

The functions are added to the interpreter by binding them into a copy of it. The interpreter supports multi-space object files so extensive additions can be made.

N.B. Any user who uses this facility will be expected to find their own problems! It is not recommended that one supply such code unless one has a good understanding of the interpreter.

LISP OBJECT	TYPE		STANDARD LISP	(sub) STANDARD
(ABS number)	subr	=> number	3.11	
(ADD1 number)	subr	=> number	3.11	
(AND any)	fsubr	=> extra-boolean	3.10	
(APPEND list list)	subr	=> list	3.13	
(APPENDSTR stringid ...)	macro	=> string		L
(APPENDSTR2 stringid stringid)	subr	=> string		L
(APPLY function list)	subr	=> any	3.14	
APPROXIMATE	id			K
*ASSIGNS	id			N
(ASSOC any alist)	subr	=> {pair, NIL}	3.13	
(ATOM any)	subr	=> boolean	3.1	
(BACKUPOPEN filehandle)	subr	=> list		N
(BREAK any)	subr	=> any		Q
BREAK	id			D
(CAAAAR any)	subr	=> any	3.2	E
(CAAADR any)	subr	=> any	3.2	E
(CAAAR any)	subr	=> any	3.2	E
(CAADAR any)	subr	=> any	3.2	E
(CAADDR any)	subr	=> any	3.2	E
(CAADR any)	subr	=> any	3.2	E
(CAAR any)	subr	=> any	3.2	E
(CADAAR any)	subr	=> any	3.2	E
(CADADR any)	subr	=> any	3.2	E
(CADAR any)	subr	=> any	3.2	E
(CADDAR any)	subr	=> any	3.2	E
(CADDR any)	subr	=> any	3.2	E
(CADDDR any)	subr	=> any	3.2	E
(CADR any)	subr	=> any	3.2	E
(CAR any)	subr	=> any	3.2	E
(CATCH id any)	fsubr	=> any		D
(CATCHALL func any)	fsubr	=> any		D
(CBPT function)	fsubr	=> any		Q
(CDAAAR any)	subr	=> any	3.2	E
(CDAADR any)	subr	=> any	3.2	E
(CDADAR any)	subr	=> any	3.2	E
(CDADDR any)	subr	=> any	3.2	E
(CDDAAR any)	subr	=> any	3.2	E
(CDDADR any)	subr	=> any	3.2	E
(CDDDAR any)	subr	=> any	3.2	E
(CDDDDR any)	subr	=> any	3.2	E
(CDAAR any)	subr	=> any	3.2	E
(CDADR any)	subr	=> any	3.2	E
(CDAR any)	subr	=> any	3.2	E
(CDDAR any)	subr	=> any	3.2	E
(CDDDR any)	subr	=> any	3.2	E
(CDDR any)	subr	=> any	3.2	E

(CDR any)	subr	=>	any	3.2	E
(CHARTOINT stringid)	subr	=>	num		L
(CHECKPOINT any <any>)	subr	=>	boolean		Q
(CHECKPROCESS crtpid)	subr	=>	boolean		N
(CLOSE filehandle)	subr	=>	filehandle	3.15	
(CODEP ...	-			3.1	
*COMP	-			4.	A
(COMPRESS id-list)	subr	=>	atom	3.3	
(COND cond-form)	fsubr	=>	extra-boolean	3.10	B
(CONS any any)	subr	=>	pair	3.2	
(CONSTANTP any)	subr	=>	boolean	3.1	
(COPYSEXPR any)	subr	=>	any		Q
(COPYSTR string offset count)	subr	=>	string		L
(CREATEBACKUP any cpu swap)	subr	=>	list		Q
(CREATEPROCESS programfile relpriority processor processname infile outfile commandstring)	subr	=>	crtpid		N
(DE id id-list any)	fsubr	=>	id	3.5	
(DEFAULTVSV stringNIL)	subr	=>	string		K
(DEFINEDIN stringid)	subr	=>	list		Q
(DELETE any list)	subr	=>	list	3.13	
(DEFLIST dlist id)	subr	=>	list	3.13	
(DF id id-list any)	fsubr	=>	id	3.5	
(DIFFERENCE number number)	subr	=>	number	3.11	
(DIVIDE number number)	subr	=>	pair	3.11	
(DIGIT any)	subr	=>	boolean	3.13	
(DM id id-list any)	fsubr	=>	id	3.5	
(DO ({var[init[step]]}...) (end-test [result]) [statement])	msubr	=>	any		Q
(DO* ({var[init[step]]}...) (end-test [result]) [statement])	msubr	=>	any		Q
(DUMPSTACK any)	subr	=>	boolean		C
ECHO	id				K
(EDIT stringid)	fsbur	=>	NIL		P
(EO stringid)	macro	=>	NIL		P Q
(EOQ stringid)	macro	=>	NIL		Q
EOF	uid			4.	
EOF	id	=	EOF	4.	
EOL	uid			4.	
EOL	id	=	EOL	4.	
(EJECT)	subr	=>	NIL	3.15	
(ERROR ...	-			3.8	C D
(ERRORSET ...	-			3.8	C D

(EQ any any)	subr	=>	boolean	3.1	
(EQN any any)	subr	=>	boolean	3.1	
(EQUAL any any)	subr	=>	boolean	3.1	
(EQUALSTR string offset string offset count)	subr	=>	boolean		L
(EXPT ...	-			3.11	A
(EVAL any)	subr	=>	any	3.14	
(EVLIS any-list)	subr	=>	any-list	3.14	
EXACT	id				K
EXCLUSIVE	id				K
(EXPAND list function)	subr	=>	list	3.14	
(EXPANDFNAME stringid)	subr	=>	string		K
(EXPANDFNAME filehandle)	subr	=>	string		K
(EXPLODE atom)	subr	=>	id-list	3.3	
EXPR	id			2.2	
FEXPR	id			2.2	
(FIX ...	-			3.11	A
(FIXP ...	-			3.1	A
(FLAG ...	-			3.4	G
(FLAGP ...	-			3.4	G
(FLOAT ...	-			3.11	A
(FLOATP ...	-			3.1	A
(FLUID idlist)	subr	=>	T	3.6	H
(FLUIDP ...	-			3.6	H
(FORK any cpu swapfile)	subr	=>	crtpid		N
FSUBR	id				A
FUNARG	id				J
(FUNCTION function)	fsubr	=>	funarg	3.14	J
*GC	id	:=	NIL	4.	
(GC)	subr	=>	NIL		P
(GCQ)	subr	=>	NIL		P
GENERIC	id				K
(GENSYM)	subr	=>	uid	3.3	F
(GET any any)	subr	=>	any	3.4	G
(GETXO oblist any any)	subr	=>	any		Q
(GETBYTE string offset)	subr	=>	number		L
(GETD id)	subr	=>	any	3.5	G
(GETDXO oblist id)	subr	=>	any		Q
(GETV vector offset)	subr	=>	any	3.9	
(GLOBAL ...	-			3.6	H
(GLOBALP ...	-			3.6	H
(GO id)	fsubr			3.7	
(GREATERP number number)	subr	=>	boolean	3.11	
(GREATERPSTR string offset string offset count)	subr	=>	boolean		L

(IDP any)	subr	=>	boolean	3.1	
(IF test then else)	msubr	=>	any		Q
INPUT	id			3.15	K
(INTERN stringid)	subr	=>	id	3.3	
(INTERNXO list/id id)	subr	=>	list/id		Q
(INTTOCHAR number)	subr	=>	string		L
(IOCONTROL filehandle number number)	subr	=>	number		K
(IOECHO filehandle any)	subr	=>	any		K
(IOSETMODE filehandle number number number)	subr	=>	number-list		K
LAMBDA	id			2.3	B
(LAST any)	subr	=>	any		P
(LENGTH any)	subr	=>	number	3.13	
(LENGTHSTR stringid)	subr	=>	number		L
(LESSP number number)	subr	=>	boolean	3.11	
(LESSPSTR string offset string offset count)	subr	=>	boolean		L
(LET ({ id | (id expr) }) [any])					Q
(LET* ({ id | (id expr) }) [any])	macro	=>	any		Q
(LINELENGTH numberNIL)	subr	=>	number	3.15	
LISPERROR	id				D
(LISPVERSION)	subr	=>	string		P
(LIST [any])	fsubr	=>	any-list	3.2	
(LIST* [any])	macro	=>	any-list		Q
(LIST2 any any)	subr	=>	any-list		E
(LIST3 any any any)	subr	=>	any-list		E
(LIST4 any any any any)	subr	=>	any-list		E
(LIST5 any any any any any)	subr	=>	any-list		E
(LITER any)	subr	=>	boolean	3.13	
(LPOSN)	subr	=>	number	3.15	
MACRO	id			2.2	
(MAKESTR number stringid)	subr	=>	string		L
(MAP list function)	subr	=>	any	3.12	
(MAPC list function)	subr	=>	any	3.12	
(MAPCAN list function)	subr	=>	any	3.12	
(MAPCAR list function)	subr	=>	any	3.12	
(MAPCON list function)	subr	=>	any	3.12	
(MAPLIST list function)	subr	=>	any	3.12	
(MAX number-list)	macro	=>	number	3.11	
(MAX2 number number)	subr	=>	number	3.11	
(MEMBER any list)	subr	=>	extra-boolean	3.13	
(MEMQ any list)	subr	=>	extra-boolean	3.13	
(MIN number-list)	macro	=>	number	3.11	

(MIN2 number number)	subr	=>	number	3.11	
(MINUS number)	subr	=>	number	3.11	
(MINUSP any)	subr	=>	boolean	3.1	
(MKVECT number)	subr	=>	vector	3.9	
$MYPROCESS$	id	=	LISP's crtpid		N
(NCONC list list)	subr	=>	list	3.13	
(NCONS any any)	subr	=>	pair		E
NEW	id				K
NIL	id	=	NIL	4	
(NOT any)	subr	=>	boolean	3.10	
(NULL any)	subr	=>	boolean	3.1	
(NUMBERP any)	subr	=>	boolean	3.1	
(O stringid)	fsubr	=>	NIL		P
OBLIST	id	:=	oblist array	2.1	O
OLD	id				K
(OLDBREAK any)	subr	=>	list		K
(ONEP any)	subr	=>	boolean	3.1	
(ONRECEIVE function)	subr	=>	boolean		K
OUTPUT	id			3.15	K
(OR any-list)	fsubr	=>	extra-boolean	3.10	
(OPEN stringid id [id])	macro	=>	filehandle	3.15	K
(OPENLIS id-list)	subr	=>	filehandle		K
(OPENFILES)	subr	=>	filehandle list		K
(OQ stringid)	fsubr	=>	stringid		P
PADTEXT	id				K
(PAGELENGTH numberNIL)	subr	=>	number	3.15	
(PAIR list list)	subr	=>	alist	3.13	
(PAIRP any)	subr	=>	boolean	3.1	
*PARAMS	id				N
(PEEKCH)	subr	=>	id		K
(PLUS number-list)	macro	=>	number	3.11	
(PLUS2 number number)	subr	=>	number	3.11	
(POPXO [oblist-name])	fsubr	=>	id		Q
(POSN)	subr	=>	number	3.15	
(PPFUN [<func>] ["<list>"])	fsubr	=>	NIL		Q
(PPRINT any)	subr	=>	NIL		Q
(PRIN1 any)	subr	=>	any	3.15	
(PRIN2 any)	subr	=>	any	3.15	
(PRINC onechar-id)	subr	=>	onechar-id	3.15	
(PRINT any)	subr	=>	any	3.15	
(PRINTUSING [any])	macro	=>	any		Q
(PROG id-list any)	fsubr	=>	any	3.7	
(PROG2 any any)	subr	=>	any	3.7	
(PROGN [any])	subr	=>	any	3.7	
*PROMPT	id	:=	NIL		K
(PSETQ [id any])	fsubr	=>	NIL		H
(PURGE stringid)	subr	=>	boolean		K
(PUSHXO [existing-oblist])	fsubr	=>	any		Q
(PUT id id any)	subr	=>	any	3.4	
(PUTXO oblist id id any)	subr	=>	any		Q

(PUTBYTE string offset number)	subr	=>	string		L
(PUTD id id any)	subr	=>	any	3.5	G
(PUTDXO oblist id id any)	subr	=>	any		Q
(PUTSTR deststring offset sourcestringid offset count)	subr	=>	string		L
(PUTV vector offset any)	subr	=>	any	3.9	
(QUIT)	subr	=>		3.16	
(QUOTE any)	fsubr	=>	any	3.14	
(QUOTIENT number number)	subr	=>	number	3.11	
(QUOTIENTF integer interger)	subr	=>	number		Q
*RAISE	id	:=	T	4	
(RDS filehandle [any])	macro	=>	filehandle	3.15	K
(RDSLIS any-list)	subr	=>	filehandle		K
(READ)	subr	=>	any	3.15	K
(READCH)	subr	=>	id	3.15	
(READLENGTH numberNIL)	subr	=>	number		K
(READRECORD)	subr	=>	string or EOF		K
(REMAINDER number number)	subr	=>	number	3.11	
(REMD id)	subr	=>	any	3.5	G
(REMDXO oblist id)	subr	=>	any		Q
(REMFLAG ...	-			3.4	G
(REMOB id)	subr	=>	id	3.3	
(REMPROP any any)	subr	=>	any	3.4	
(REMPROPXO oblist any any)	subr	=>	any		Q
(RETURN any)	fsubr	=>		3.7	
(REVERSE list)	subr	=>	any	3.13	
(REVERSIP list)	subr	=>	any		P
(RMESSAGETAG)	subr	=>	list		K
(RPLACA pair any)	subr	=>	pair	3.2	
(RPLACD pair any)	subr	=>	pair	3.2	
(RPOSN)	subr	=>	number		K
(SASSOC any list function)	subr	=>	any	3.13	
(SAVECODE file function)	subr	=>	stringid		P
(SBPT function [condition])	fsubr	=>	any		Q
(SET id any)	subr	=>	any	3.6	H
(SETQ [id any])	fsubr	=>	any	3.6	H
SEXPR	id				K
SHARED	id				K
(SORT list function)	subr	=>	list		P
(SORTSTRINGP stringid stringid)	subr	=>	boolean		Q
(STOPPROCESS crtpid)	subr	=>	NIL		N
(STRINGP any)	subr	=>	boolean	3.1	
(SUB1 number)	subr	=>	number	3.11	
(SUBLIS alist any)	subr	=>	any	3.13	
SUBR	id				A

(SUBST X forY inZ)	subr	=>	any	3.13	
(SXHASH any)	subr	=>	number		O
T	id	=	T	4	
(TERPRI)	subr	=>	NIL	3.15	
(TGAL [<file>] <output>)	fsubr	=>	NIL		Q
(THROW id any)	fsubr	=>			D
(TIMEOFDAY)	subr	=>	number-list		P
(TIMES number-list)	macro	=>	number	3.11	
(TIMES2 number number)	subr	=>	number	3.11	
(TRACE id ...)	fsubr	=>	id-list		M
(UNFLUID ...	-			3.6	H
(UNLESS test form)	msubr	=>	any		Q
(UNWINDALL function [any])	fsubr	=>	any		D
(UNWINDPROTECT [any])	fsubr	=>	any		D
(UPBV vector)	subr	=>	number	3.9	
UPDATE	id				K
VALUE	id				H
(VECTORP any)	subr	=>	boolean	3.1	
(WAITFORRECEIVE number)	subr	=>	boolean		K
(WHEN test form)	macro	=>	any		Q
(WHILE any-list)	fsubr	=>	NIL		I
(WRS filehandle [any])	macro	=>	filehandle	3.15	K
(WRSLIS any-list)	subr	=>	filehandle		K
(XCONS any any)	subr	=>	pair		E
(ZEROP any)	subr	=>	boolean	3.1	

