SH20-6477-0

Program Offering

LISP/VM User's Guide

Program Number: 5798-DQZ

LISP/VM is a general purpose, high-level language appropriate for use in artificial intelligence, expert systems, symbolic and natural language processing, and other advanced applications. It includes a unified, interactive, and user-friendly development environment with tools for the creation and maintenance of LISP/VM programs and data. This development environment includes both a LISP/VM interpreter and a semantically equivalent LISP/VM compiler.

This manual describes the function, language, and use of the LISP/VM program.

IBM

PROGRAM SERVICES

During a specified number of months immediately following initial availability of each licensed program, the customer may submit documentation to the designated IBM location below when he/she encounters a problem which his/her diagnosis indicates is caused by a defect in the licensed program. During this period only, IBM through the program sponsor(s), will, without additional charge, respond to an error in the current unaltered release of the licensed program by issuing known error correction information to the customer reporting the problem and/or issuing corrected or notice of availability of corrected code. However, IBM does not guarantee service results or represent or warrant that all errors will be corrected. Any onsite program services or assistance may be provided at a charge.

WARRANTY

THE LICENSED PROGRAM DESCRIBED IN THIS MANUAL IS DISTRIBUTED ON AN "AS IS" BASIS WITHOUT WARRANTY OF ANY KIND EITHER EXPRESSED OR IMPLIED.

Central Service Location: IBM Corporation
B/O 205
Two Riverway
Houston, TX 77056
Attn: Chris Bosman-Clark
IBM Tieline: 8/345-2523
Telephone: (713) 940-2523

Note: Non-US customers should contact their designated support group in their country.

Information concerning Program Services for this Program Offering can be found in Availability Notice G320-0349.

First Edition (July 1984)

CONTENTS

INTRODUCTION

1.0 Learning About LISP/VM

This manual is intended as a guide to the facilities and capabilities of LISP/VM. It contains reference material describing the functions available in that system, as well as material comprising a system programmer's guide. It also contains a certain amount of tutorial material that provides some motivation or explanation for why certain operations are performed in the way they are. The chapters describe the major components of the LISP/VM system, the compiler/interpreter, the editor, the debugger, and the file system. Appendices include an alphabetized summary list of functions and a glossary of terms.

1.1 Manual Description

1.1.1 Formal and Informal Descriptions

This document contains an overview of LISP/VM. An overview by its nature is high-level and incomplete. As such, it constitutes an informal description of LISP/VM and is intended to give the reader some background with which to read the complete, formal description which follows.

1.1.2 Formal Description

In describing LISP/VM features, the following conventions are used:

- { and } are used for meta-linguistic grouping.
- | is used to separate alternatives.
- [and] are used to indicate optional arguments.
- The ellipsis "..." is used to denote zero or more instances of the preceding object.
- Function names are written in upper case.
- To avoid multiple definitions, when an operator takes operands of different types, all definitions of the operator appear in one section of "Operations on Data Types", separated by "OR", and cross-references to the definitions appear in other relevant sections.

LISP/VM features (special forms, built-in functions, functions, function + compiler macros, macros, funargs, structures, jaunts, keywords, variables and system commands) will be described in the format shown by the following example:

GO —— *special form*

(GO **id**)

id must be associated with a label in an enclosing label-scope. The GO expression transfers control to the associated label. Control may not transfer out of an enclosing contour or label-scope or an error is signalled.

Operand descriptions will be abbreviated as follows:

Abbreviation	Data type
item	object of any type
id	identifier
pair	pair
list	any object interpreted as a list - any atom is an empty list - a pair is a list of at least one element
vec	vector of arbitrary objects
cvec	character vector
bvec	bit vector
bORcvec	bit vector or character vector
num	any number
sint	small integer
int	integer (fixed point number), including integers of arbitrary precision
rnum	real (floating point number)
hashtable	hashtable
readtable	readtable
app-ob	applicable object
pred	any **app-ob** treated as a predicate
exp	expression, any object in the context of evaluation
sd	state descriptor
bpi	compiled function or macro
label	identifier, interpreted as a label
bv-list	bound variable list
a-list	association list
boolean	any object treated as a truth value
stream	stream (I/O data structure)
rstrm	key-addressed stream
filespec	CMS file description - an identifier - a list of 1, 2, or 3 identifiers
filearg	CMS file description that may appear as 1, 2, or 3 separate arguments. - **filespec** - **id1 id2 [id3]**
sysdep-area	cvec or ivec (data vector expected by system-dependent interface)
arg	an argument described in more detail in subsequent text

Figure 1. Abbreviations for operands.

When an operator is designated as a function, built-in function, or CMS interface function, the operand descriptions always refer to the **values** of the operands in an invocation of the operator. The designation "function and compile-time macro" indicates a function that may be replaced by in-line code or by some other function in compiled programs.

When an operator is designated as a special form or macro, the operand descriptions will normally indicate which operands are evaluated and which ones are not.

Examples will show an expression to be evaluated followed on the next line by the result of evaluation. Some examples show the value prefixed by **Value** =. Other examples, show the value prefixed by **exp** = where exp is the expression being evaluated.

1.1.3 How to Read This Manual

The manual is composed of six main sections: Introduction, LISP/VM, The LISP/VM Editor, Debugging Aids, Input/Output, and System Capabilities. The Introduction describes this manual and points the reader to additional document and demonstration sources. LISP/VM includes an informal description of using the system and an overview of the language. It also includes a formal description of data types, evaluation, functions, macros, operator definition, predicates, control structures, and operations on data types. LISP/VM Editor includes a description of the user interface and files. An alphabetic listing of all editor commands is prefaced by lists of commands organized by function. Debugging Aids describes interactive evaluation, debugger facilities, call tracing, and performance monitoring and measurements. Input/Output describes the internal and external forms of data types, the Reader, Printer, readtables which define the syntax rules used by the Reader, and files. System Capabilities describes CALLBELOW, the operating system interface.

Appendices include MACLISP Compatibility Package, Generalized Iteration, Table of System Functions, Variables and Commands, Glossary, and Index.

If you are a new user of the system, it will probably be easiest for you to read the first chapter of the LISP/VM section which will show you how to get on and off the system within the system editor. You could also create and save expressions using your own favorite editor. The second chapter of the LISP/VM section gives an informal overview to give you a feeling for the system. The extensive glossary appendix is a useful place to start. It can be just read through to gain familiarity with terms or it can be used for finer detail by first examining one definition which will probably contain unknown terms and then using the glossary again to track down those terms.

As a more experienced user, you will probably want to concentrate on the formal descriptions in the manual and appendices, especially the lists of editor commands, table of system functions, and glossary.

1.2 External Requirements

LISP/VM runs under the CP/CMS system using a screen display console such as the IBM 3270 series.

1.3 Other Sources

1.3.1 Online Documentation

Once the LISP/VM system has been started, online documentation is available by typing:

```
help
```

The HELP documentation system will guide you through its use. It can be used directly by typing HELP followed by the function name in which you are interested. If you are not certain what function will be most useful, it is possible to ask for a list of topics:

```
help key
```

This will show the list of topics on which help is available. Typing HELP followed by one of the listed topics will cause a list of function names associated with that topic to be shown.

1.3.2 Other Documents

This manual is not intended as a basic primer for LISP. For that purpose, the reader should consult another publication such as Let's Talk LISP by Laurent Siklossy (Prentice-Hall, 1976), The Programmer's Introduction to LISP by W. D. Maurer (Elsevier, 1972), or LISP by P. H. Winston and B. K. P. Horn (Addison Wesley, 1981), all of which are textbooks presenting an introduction to LISP for the beginning LISP programmer. Other books, such as Artificial Intelligence by Patrick Winston (Addison-Wesley, 1977) and Computational Semantics by E. Charniak and Y. Wilks (Elsevier, 1976) contain chapters introducing LISP in the course of examining some of the application areas where LISP programs have been significant.

1.4 Demonstrations

A demonstration program is provided with the system, and may be run (on those installation sites which install it) by typing from CMS the command "LISPDEMO". The demonstration program will show how to initially input a program to compute Pig Latin, declare a variable, complete the program, run the program, examine the way the program runs, store the program on disk, compile the stored program, and run the compiled program.

LISP/VM

2.0 Getting in and out of LISP/VM

2.1 Starting Out

LISP/VM is a system which allows you to create, run and debug LISP/VM programs. It exists in the CMS environment under CP. This document assumes that you understand how to log into CP/CMS. You will probably be using a screen display console, such as the IBM 3270 series, for developing your programs since their rapid display capability makes working with an interactive system like LISP/VM more rewarding.

To bring the system up, first type

```
LISP
```

The system will respond with the initial screen display: the word "NIL" in the upper left hand corner, and a brightened line near the bottom of the screen which starts with the word "LISPEDIT". The area above the brightened line will show the results of your latest instructions. The lines starting with the brightened line will give you useful system information. The last line is the input area into which you will type.

We'll be going into greater detail on creating programs later in this chapter but, for now, let's look at a few fundamentals which will let us try out some small examples. Whatever you type will be input to the READ-EVAL-PRINT loop. A name, for example, will be read, evaluated, and its value printed. Typing INTEGER in the input area, when no value has been assigned to INTEGER, will cause

```
INTEGER = INTEGER
```

to appear in the area above the brightened line. Typing

```
(SETQ INTEGER 45)
```

in the input area will cause

```
(SETQ INTEGER 45) = 45
```

to appear. In this case, the variable INTEGER was assigned the value 45 and the entire input expression was evaluated and its value, the value of the object named by the last operand of the expression, was printed as the value of the input expression. Now, if you type INTEGER, you will see that

```
INTEGER = 45
```

2.2 Ending LISP/VM Sessions

2.2.1 Losing The Session Results

In the example above, you have created a variable, INTEGER, which has the value 45. If you were not interested in keeping any of this work, you would now type

```
EXITLISP
```

which takes you back to CMS. The next time you bring up LISP/VM, you would begin with an empty new screen. It would be most common to EXITLISP when one is just experimenting with LISP/VM but not creating any code which would be used again.

2.2.2 Continuing Sessions

If, rather than being willing to lose the variable INTEGER, you wanted to be able to save your work for later reuse, you could type

```
FILELISP file-name [= file-mode]
```

This takes a snapshot of the system at the moment you typed the command and places it in file-name. (The optional = says to use the standard file-type and file-mode allows an explicit file-mode to be specified.) The next time you want to continue your work, you can type

```
LISP file-name
```

and you will pick up from exactly where you stopped with INTEGER equal to 45. The next time you want to stop, you can just type

```
FILELISP
```

and the new snapshot will be saved in place of the last. This is the easiest way for the naive user to work. Eventually you will learn about indexed files, which require less disk storage, and can be given easily to other LISP programmers for incorporation into their programs.

2.2.3 Being Extra Conservative

Changes to a program can also be made tentatively without losing the original. It's possible to take a snapshot of the system before you make changes and keep it under another name. You can then continue with the more successful snapshot. To make a snapshot and then continue, type

```
SAVELISP [file-name [= file-mode]]
```

If the file-name is not specified, then the snapshot is saved under the current file-name.

2.2.4 Other Useful Information

If you have started up the system and want immediately to name the file which will be used to hold the snapshot, you can type

```
NAMELISP file-name [= file-mode]
```

This means you don't have to include the file-name when you FILELISP or SAVELISP.

Since FILELISP and SAVELISP can cause a new snapshot to be stored in place of a previous snapshot, it is useful to be able to query the system to remind yourself what file-name a snapshot will be filed under if no file-name is specified. This is done by typing

```
NAMELISP
```

3.0 LISP/VM Overview

3.1 List Processing

Every successful programming language has some principal area of strength where it is more convenient or more powerful than other languages. The principal strength of APL is in its powerful operators for handling arrays. The principal strength of COBOL is in handling files and records, sorting them, and printing reports. The principal strength of SNOBOL is in handling character strings and searching for patterns. For LISP, the principal strength is in handling irregular data structures, which are typically stored as lists whose elements are lists of lists of lists of.... The name LISP comes from its principal application as a *LIS*t *P*rocessing language.

The areas of application where LISP is strongest are those areas that typically have complex or irregularly structured data. Graphics, artificial intelligence, computational linguistics, theorem proving, formula manipulation, and development of advanced systems and languages are areas where LISP is widely used. For artificial intelligence in particular, a reading knowledge of LISP is essential to understand the literature and folklore of the field.

Besides its use in advanced applications, LISP is an important language to study for general background in computer science:

- It teaches programming techniques that can be adapted to other languages, such as PL/I, APL, or even assembler.
- It has stimulated a great deal of research that can be applied to other languages and operating systems.
- It contains features and concepts that have become part of the cultural milieu of computer science.

LISP is, perhaps, the most flexible programming language ever invented. As a result, it evolves and changes rapidly with each LISP system evolving a new variation. This document describes one variation, LISP/VM, which has been developed over a decade in a research environment. The principal aim of the design process was to produce a design which was clean, powerful, and flexible. Compatibility with previous designs was given careful consideration but was not allowed to control the design.

3.2 READ-EVAL-PRINT

A "program" is created by writing an expression which is READ by the system, EVALuated, and the result PRINTed. The form in which the expression is typed into the system or printed by the system is known as the external form. The form in which the expression is stored and evaluated by the system is known as the internal form. The external forms of expressions include external characters such as the parentheses surrounding elements which are internally lists and the angle brackets surrounding elements which are internally vectors. The external form of numbers may include the external characters +, -, and E.

3.3 Expressions

An expression is made up of atoms and pairs. Programs are built from exactly the same structures as data, and all the operators used for constructing and analyzing data can be used for constructing and analyzing programs. Atoms include NIL, identifiers, numbers, and vectors. They also include the object code resulting from the compilation or interpretation of a function or macro, expressions with the environment in which they are to be evaluated, specific environments with control, hashtables, readtables, and the end-of-file symbol. Pairs are composite objects which have two components, a CAR and a CDR.

3.4 List Notation

Each component of a pair can itself be a pair. The external form of a pair consists of a left parenthesis, the CAR component, a dot, the CDR component, and a right parenthesis.

Examples of pairs:

```
        (A . B)
        (A . (A . B))
        ((U . V) . (X . (Y . Z)))
```

Note how the recursive definition permits pairs to be nested inside other pairs to an arbitrary depth.

Although the pair notation has the advantage of simplicity, it can become tedious to write whenever there are more than two entities to be combined. List notation is usually a more convenient way of representing a list of arbitrary length. Like pairs, lists are defined recursively, allowing lists of lists of lists.

Lists are a special type of expressions determined by the following recursive definition:

- The list of length zero, called the empty list, is represented by the symbol NIL.
- If n>0, a list of length n is a pair of an expression and a list of length n-1.

According to this definition, if A, B, C, and D are expressions, then they may be combined to form the following list of length 4:

```
        (A . (B . (C . (D . NIL))))
```

To translate from pair notation into list notation, convert every occurrence of NIL to (). Then if any dot is followed by a left parenthesis, remove the dot, the left parenthesis, and its matching right parenthesis. The above list becomes simply

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

To translate a list back into pairs, first replace any occurrence of () with NIL; if the list has one or more elements, insert ". (" after the first element and add ")" at the end. Then repeat this procedure for any remaining sublists:

```
    ((A B) C)
    ((A B) . (C))
    ((A . (B)) . (C))
    ((A . (B . ())) . (C))
    ((A . (B . NIL)) . (C))
    ((A . (B . NIL)) . (C . ()))
    ((A . (B . NIL)) . (C . NIL))
```

The expressions in a list may be atoms, pairs, or other lists. Following are some lists expressed in both list notation and pair notation.

List Notation	Pair Notation
()	NIL
(A)	(A . NIL)
((A))	((A . NIL) . NIL)
(A B)	(A . (B . NIL))
(A B C)	(A . (B . (C . NIL)))
((A B) (C D))	((A . (B . NIL)) . ((C . (D . NIL)) . NIL))
(A B C . D)	(A . (B . (C . D)))

List notation is a essentially a compact way of expressing a long series of pairs. For most applications, list notation is used almost exclusively in preference to the pair notation. An understanding of pairs, however, helps to explain how some of the primitive functions work. Any expression more complex than an atom may be written as a pair, but not necessarily as a list. In fact, the basic pair (A . B) does not have a corresponding list form.

3.5 Expression Forms, QUOTE, and SETQ

There are several types of lists which can act as programs. In these lists, the first element is considered to be an operator and the following elements to be operands. In LISP/VM, the operator is always evaluated. This evaluation provides information on how to evaluate the operands. If, for example, the operator is evaluated and recognized as a function, the operands are evaluated in left to right order and the function which is the value of the operator is applied to the list of all the values of the operands. Operator evaluation is an important way in which LISP/VM differs from other LISPs. In most LISPs, the operator is not evaluated: in LISP/VM it always is. If the operator is evaluated and recognized as a special form, the evaluation of the remainder of the list follows the special rules defined for each special form.

One special form is the operator QUOTE. The effect of QUOTE is to return the operand unevaluated. If X is a variable with value 39, then (PRINT X) will cause X to be evaluated before it is passed to the PRINT function; therefore, (PRINT X) will print "39", and (PRINT (QUOTE X)) will print "X".

To reduce the amount of writing, (QUOTE X) may be abbreviated by a single quote:

```
       'X  is  (QUOTE X)
     ' 'X  is  (QUOTE (QUOTE X))
  '(A B C)  is  (QUOTE (A B C))
```

Only one quote mark is used. A closing quote must not be written. Note that the QUOTE may refer to an entire list from the starting "(" up to and including the ending ")".

SETQ is the assignment operator. It takes two operands, the variable and the value to be assigned to the variable. The definition of SETQ provides for only evaluating the value to be assigned, not the variable. Therefore, if X has the value 10 and Y the value 20,

```
  (SETQ X Y)
```

evaluates Y to 20 and assigns that value to X, not the value of X.

Another form of list which acts as a program is the macro. This form is used to introduce new constructs into the language. A macro is passed an unevaluated expression and returns an expression which is then evaluated. Thus, the programmer can change the "syntax" of the language by defining a macro, which transforms the program from the new form to one understood by the original system.

It is apparent that to understand how an operator works, it is necessary to understand if it follows the rules for functions, those for macros, or has special rules of its own. For that reason, when an operator is defined in this document, the form of that operator is specified.

3.6 Constructing and Dissecting Expressions

There are three primitive functions for constructing and dissecting lists and pairs. (CONS A B) takes two operands A and B and constructs a pair whose first element is A and whose second element is B. If P is any pair, then (CAR P) is its first element, and (CDR P) is its second element.

The functions CAR and CDR are undefined for atoms, but they are defined for lists since lists are defined in terms of pairs. The null list NIL is a special case since it is both an atom and a list; by convention, (CAR NIL) and (CDR NIL) are both defined to be an error. The function CONS is defined for all expressions, including lists, atoms, and NIL.

The following examples show how CAR and CDR operate on various lists. Note that as an expression appearing in an operand position, we would expect to have to QUOTE 5 to avoid having it evaluated. For simplicity, however, numbers by convention are implicitly QUOTEd. Lists which are not meant to be evaluated, even if they contain numbers, must still be QUOTEd.

```
(CAR 5)                is undefined.
(CAR '(5))             is  5
(CDR 5)                is undefined.
(CDR '(5))             is  NIL
(CAR '(5 . 93))        is  5
(CDR '(5 . 93))        is  93
(CAR '(5 93))          is  5
(CDR '(5 93))          is  (93)
(CAR '(5 93 17 27))    is  5
(CDR '(5 93 17 27))    is  (93 17 27)
```

When applied to a pair, CAR extracts the first part, and CDR extracts the second part.

```
(CONS 5 27)                is  (5 . 27)
(CONS 5 '(27))             is  (5 27)
(CONS 5 NIL)               is  (5 . NIL) or (5)
(CONS NIL NIL)             is  (NIL . NIL) or (NIL) or (())
(CONS 5 '(27 15 78))       is  (5 27 15 78)
(CONS '(5 27 15) 78)       is  ((5 27 15) . 78)
(CONS '(5 27 15) '(78))    is  ((5 27 15) 78)
(CONS '(5 27 15) NIL)      is  ((5 27 15))
```

Since the functions CAR, CDR, and CONS are defined in terms of pairs, their operations on lists are not immediately obvious. In general, CAR produces the first element, or head, of a list, and CDR produces the rest of the list, the tail. When applied to an atom A and a list L, CONS produces a new list whose head is A and tail is L. In reading and writing LISP/VM notation, remember that parentheses are never optional: (((A))) is not the same as ((A)), which is not the same as (A), which is not the same as A.

What is amazing about LISP is its incredible simplicity. These three basic functions are all that is necessary for a universal system that can build data structures of arbitrary complexity. A variety of other list handling functions for convenience, but all of them ultimately reduce to the basic operations of CAR, CDR, and CONS. There is an abbreviation for repeated application of the functions CAR and CDR. If E is any expression, then

```
(CAAR E)    is  (CAR (CAR E))
(CDDR E)    is  (CDR (CDR E))
(CADR E)    is  (CAR (CDR E))
(CDAR E)    is  (CDR (CAR E))
(CADAR E)   is  (CAR (CDR (CAR E)))
```

and so on for CDADR, CAADR, etc.

Although CONS is useful for constructing pairs, it is convenient to have a function that combines two lists to form another list. The function APPEND takes two or more lists as its operands and concatenates them into a single list:

```
(APPEND '(1 2 3 4) '(5 6 7))        is (1 2 3 4 5 6 7)
(APPEND () '(1))                    is (1)
(APPEND '((1 2) 3) '((4)))          is ((1 2) 3 (4))
(APPEND '(1 2 3) '(4) NIL '(5 6))   is (1 2 3 4 5 6)
(APPEND '(1 2) '((3)) () ())        is (1 2 (3))
```

LIST is another useful function, which takes an arbitrary number of operands and combines them all into a list:

```
(LIST 1 2 3)              is (1 2 3)
(LIST)                    is NIL
(LIST '(1 2 3) '(4 5))    is ((1 2 3) (4 5))
```

Note the difference between LIST and APPEND; for this last example,

```
      (APPEND '(1 2 3) '(4 5))  would be  (1 2 3 4 5).
```

If L is a list, then (CAR L) is the first element, (CADR L) is the second, (CADDR L) is the third, (CADDDR L) is the fourth. For long lists, the function ELT is often easier to use than counting D's. If N is any positive integer, (ELT N L) is the Nth element of L, that is, ELT uses zero-based indexing.

If L has fewer than N elements, then the result is NIL. For any non-empty list L, (LAST L) is a list containing only the last element of L.

The function LENGTH gives the length of a list. If the list happens to contain sublists, LENGTH only gives the number of elements in the topmost list. LENGTH is also defined for other expressions: length of an atom is 0, and the length of a pair that is not also a list is 1.

```
(LENGTH '(1 2 3 4))          is  4
(LENGTH '((1 2) (3 4)))      is  2
(LENGTH '((1 2) . (3 4)))    is  1
(LENGTH '((1 2 3 4))         is  1
(LENGTH NIL)                 is  0
```

3.7 EVAL

The use of the READ-EVAL-PRINT loop for expressions entered from the console has been described above. Evaluation is actually performed by the EVAL function, which can be called explicitly. If X is a variable, then (EVAL 'X) is the value of the variable. If the value of F is a function, then (EVAL '(F A B C)) normally evaluates each of the operands A, B, and C and passes the results to the function F, which it then evaluates as well.

When an expression is entered from the terminal, the system automatically passes it to EVAL. Therefore, for the variable of our previous example with value 39, if the input is X, the output will be (EVAL 'X) or 39. If the input is (QUOTE X), the output will be X:

Input from terminal	Output to terminal
`X`	`39`
`(QUOTE X)`	`X`
`(QUOTE (QUOTE X))`	`'X`
`(EVAL (QUOTE X))`	`39`
`(EVAL (EVAL 'X))`	`X`
`(EVAL (EVAL ' 'X))`	`'X`
`'(A B C)`	`(A B C)`
`'(EVAL 'X)`	`(EVAL 'X)`

EVAL is like the execute operator in APL, which takes a quoted character string and executes it. QUOTE is used to delay or prevent evaluation.

3.8 Three Kinds of Equality

Because LISP/VM deals with complex data objects, there is a hierarchy of equality operations.

- Two objects may be identical: they overlay the same storage and a change to one is a change to the other. This is the strongest kind of equality, and is tested by the predicate EQ.

- Two data objects may have the same shape and the same values, this is tested by the UEQUAL predicate. If two data objects, which are UEQUAL do not share sub-structure and the same updating operations are done to their components, then they will still be UEQUAL.

  The predicate UEQUAL can be thought of as testing structural equality, although it actually tests whether if the same update operations are done to both data objects they will still be UEQUAL.

- The weakest kind of equality is expressed by the predicate EQUAL. If two objects are EQUAL, then using the same series of access functions, like CAR, CDR, and ELT, will yield components which are EQUAL. Two objects may be equal even if they do not have the same shape.

  ```
  (SETQ B (CONS A 5))
  (SETQ C (CONS B B))
  (SETQ D (CONS (CONS A 5) (CONS A 5)))
  ```

  C and D are EQUAL but not UEQUAL.

In general the stronger the equality predicate, the more efficient. EQ always takes a fixed amount of time, UEQUAL takes time linear in the size of the object, and in the case of complex patterns of sharing, EQUAL can take time cubic in the size of the object (it is almost always linear).

There are two kinds of functions which search data objects looking for equality with other data objects: those functions suffixed with a Q which test for EQ, and those functions which do not use a naming convention for which EQUAL is the default.

3.9 Predicates and Conditionals

A predicate is a function whose result is either true or false. False is represented by NIL and true is represented by any other value. The term "non-NIL" is used in this manual to convey the multiplicity of values which represents true.

AND takes an arbitrary number of operands, which it evaluates in order from left to right. As soon as it finds an operand with value NIL, it stops evaluating any of the others and returns NIL. If none of its operands evaluate to NIL, it returns the value of its last operand. With no operands, (AND) has the value non-NIL.

OR takes an arbitrary number of operands, which it also evaluates in order from left to right. As soon as it finds an operand with value non-NIL, it stops evaluating any of the others and returns that value. If none of its operands evaluate to non-NIL, it returns NIL. With no operands, (OR) has the value NIL.

NOT changes truth to falsity and falsity to truth. (NOT NIL) is non-NIL, and (NOT 'ELSE) is NIL. When applied to anything other than NIL, NOT also produces the result NIL.

The primary use for predicates is in conditional expressions. The basic conditional has the following form:

```
    (COND (p1 e1) (p2 e2) ... (pn en))
```

The predicate p1 is evaluated first. If the value of p1 is not NIL, then e1 is evaluated, and its value is returned as the value of (COND ...); none of the other predicates or expressions are evaluated. If the value of p1 is NIL, then e1 is not evaluated, and control passes to the next pair (p2 e2). Evaluation continues in this way through the entire list: the p's are evaluated until the first non-NIL one is reached, and then the corresponding e is evaluated and returned as the value of (COND ...). If all p's are NIL, then none of the e's are evaluated, and the result of COND is NIL.

COND corresponds to a string of IF-THEN-ELSE statements:

```
    (COND (p1 e1)  ('ELSE e2))
```

corresponds to:

```
    IF p1 THEN e1 ELSE e2;
```

and

```
    (COND (p1 e1) (p2 e2) (p3 e3) ... ('ELSE en))
```

corresponds to:

```
    IF p1 THEN e1; ELSE IF p2 THEN e2;
       ELSE IF p3 THEN e3; ... ELSE en;
```

One of the basic predicates is ATOM, which tests to see if its operand is an atom. (ATOM op) is non-NIL if op is an atom; otherwise, it is NIL.

EQ is the predicate that tests for equality of two objects. If A and B are atoms, then (EQ A B) is true if A and B are the same and false if they are not. If one operand is an atom and the other is composite, then (EQ A B) is false. If both operands are composite, then (EQ A B) is false unless both operands occupy the same location in storage; i.e. (EQ A A) would be true, but (EQ (CONS A B) (CONS A B)) would be false since the two newly created objects occupy different locations. For the same reason after (SETQ C 1) and (SETQ D 1) then (EQ C D) is true, but (EQ '(C) '(D)) is not.

After the evaluation of (SETQ B A), the value of (EQ B A) is true. (EQ 2 2) is also true. Since the question of when the operands occupy the same storage location is not obvious, EQ should primarily be used to test equality of atoms.

The predicate EQUAL tests any two expressions to see if they are equal. For the types of data discussed so far, it can be defined in terms of EQ:

```
    (EQUAL X Y)
```

is equivalent to

```
    (COND
       ((ATOM X) (EQ X Y))
       ((ATOM Y) NIL)
       ((EQUAL (CAR X) (CAR Y)) (EQUAL (CDR X) (CDR Y)))
       ('ELSE  NIL))
```

This is an example of a recursive definition, where EQUAL is defined in terms of itself. There are examples of data on which the above program will never finish. The EQUAL defined by the system always does. Note that the last expression of the COND could be omitted and the semantics would be the same. It is added for readability.

NULL is a predicate that tests for the null list. (NULL X) is equivalent to (EQ X NIL). In practice, the predicates NOT and NULL always give identical results.

3.10 Arithmetic

Numbers are stored in three forms: reals, and large and small integers. The size of large integers is limited by the size of memory. If all operands of an arithmetic operation are integer, the result is integer. If any or all of them are floating, the result is floating point. Numeric literals are represented as in FORTRAN: 29, +29, -29 and +29E0 are integers; and 29.0, +29.E0, and -0.29E+2 are real numbers in external forms.

FUNCTION	VALUE
(PLUS op1 op2 ... opn)	op1 + op2 + ... + opn
(MINUS op)	op with sign changed
(DIFFERENCE op1 op2)	op1 - op2
(TIMES op1 op2 ... opn)	op1 × op2 × ... × opn
(QUOTIENT op1 op2)	op1 ÷ op2 If both operands are integer, the result is a truncated integer
(REMAINDER op1 op2)	remainder if op1 and op2 are integer, and residue if either or both are real.
(DIVIDE op1 op2)	value of (CONS (QUOTIENT op1 op2) (REMAINDER op1 op2))
(EXPT op1 op2)	op1 raised to the power of op2. If op2 is real, op1 must be positive.
(ADD1 op)	op + 1
(SUB1 op)	op - 1
(ABSVAL op)	absolute value of op
(MAX op1 op2 ... opn)	largest operand
(MIN op1 op2 ... opn)	smallest operand
(RNUM2INT op)	op converted to integer
(FIX op)	identical to RNUM2INT
(INT2RNUM op)	op converted to real
(FLOAT op)	identical to INT2RNUM

The following predicates perform tests on numbers and return non-NIL or NIL if the test is true or false. NUMP and EQUAL may be applied to any operands, but all of the others are defined only for numeric operands. EQ is used when both operands are small integers.

PREDICATE	TRUE WHEN
(NUMP op)	op is a number
(GREATERP op1 op2)	op1 > op2
(LESSP op1 op2)	op1 < op2
(EQUAL op1 op2)	EQUAL predicate defined before
(ZEROP op)	op = 0
(MINUSP op)	op < 0
(PLUSP op)	op >= 0
(INTP op)	op is integer
(RNUMP op)	op is real

3.11 Function Definition

A function is an expression for which certain variables are specified as bound variables. A list with LAMBDA as its operator is a function. When the variable SQUARE has as its value,

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

this corresponds to the PL/I format:

```
SQUARE: PROCEDURE(X); RETURN(X*X); END;
```

SQUARE can be given that value by evaluating:

```
(SETQ SQUARE '(LAMBDA (X) (TIMES X X )))
```

Here X is the bound variable, and the expression (TIMES X X) is the body of the function. After the function has been defined, it can be evaluated by having its name appear as the first element after a left parenthesis in some expression that is to be evaluated. For example, if (SQUARE 7) is typed from the terminal, the function SQUARE is invoked, the argument 7 is bound to the bound variable X, and the body of the function, (TIMES X X), is evaluated to produce the result 49. More generally, a function is an expression that is composed of three parts: an identifier which evaluates to LAMBDA, a bound variable list, and a body.

When a function is evaluated and when the bound variable list is a simple list, each bound variable is bound to a value from one of the arguments in the function invocation. Then the expression in the body of the function is evaluated, and the value of that expression is returned as the value of the function. One of the primary differences between programming styles in LISP/VM and PL/I is that a single expression in PL/I is usually rather simple, but a single expression in LISP/VM can span an entire program. More generally, when a function is invoked, all of its arguments are evaluated and a list of the values is created. That list is passed to the function. The function matches its bound variable list against that list. If the bound variable list is a single variable, then that variable matches the entire list. Otherwise, the CAR of the bound variable list is matched against the CAR of the list passed in and the CDRs of the lists are matched. A bound variable list is often of the form:

```
(X1  X2  ...  Xn . Y)
```

X1 through Xn will match the first n arguments and Y will match all remaining (possibly zero) arguments.

When a function is invoked, each operand is evaluated and only the values are associated with the bound variables. This method of function invocation is known as call by value, and there is no way for the function to change the values of the variables used as arguments in the calling program.

LISP/VM allows the programmer more flexibility by the use of macros. A macro (MLAMBDA) has the same form as a function (LAMBDA), however it is passed the entire expression unevaluated. It can then manipulate the expression and return a new expression. That expression is then evaluated. Macros are used to extend the syntax of the language. The looping constructs are all implemented by macros which insert GO expressions around the body of the loop. A macro which implements a simplified version of SETQ is shown below.

```
        (MLAMBDA (OPERATOR TARGET VALUE)
                 (LIST 'SET (LIST 'QUOTE TARGET) VALUE))
```

3.12 The Program Feature

In principle, all programs can be written as single expressions, which may become quite complex. The program feature, in conjunction with assignment expressions, is particularly convenient. It has the following form:

```
        (PROG (local variables)
              (expression1)
              (expression2)
                  &dots.
              (expression))
```

PROG is a macro; it is followed by a list of identifiers that name variables that are local to the list of expressions that follow. Evaluation of the PROG list causes the expressions to be evaluated in order unless one of them contains the special forms RETURN or GO.

If the last expression on the list does not evaluate a RETURN or GO, then after that expression is evaluated, the PROG is terminated with the value NIL. If any expression happens to be (RETURN expression), then when the expression is evaluated, the PROG will be terminated, and the value of the expression is returned as the value of the PROG.

The special form GO is used in conjunction with a construction known as a *label*: a label may precede an expression in any PROG. After evaluation of (GO label) in a PROG, the next expression to be evaluated is the one following that label.

The following function illustrates the use of the PROG feature for writing a function that reverses a list; i.e. the output list contains the same elements as the input list, but in reverse order.

```
(LAMBDA (INLIST)
   (PROG (OUTLIST)
LOOP  (COND (NULL INLIST) (RETURN OUTLIST))
      (SETQ OUTLIST (CONS (CAR INLIST) OUTLIST))
      (SETQ INLIST (CDR INLIST))
      (GO LOOP) ) ) )
```

In the above example, the expression labeled LOOP is equivalent to the PL/I statement:

```
        LOOP: IF INLIST=NULL THEN RETURN(OUTLIST);
```

If a PROG has no bound variables and no RETURNs, it can be replaced by the simpler and more fundamental construct SEQ, which is a PROG without bound variables. The difference between PROG and SEQ is like the difference between a BEGIN block in PL/I, which defines local variables, and a DO group in PL/I, which merely groups statements. SEQ is usually more efficient because it doesn't have to allocate space for the variables. An EXIT expression terminates a SEQ and causes the SEQ to return a value in the same way that RETURN terminates PROGs and LAMBDAs and causes them to have certain values.

3.13 Applying a Function

When the result of a function is another function, the new function can be evaluated immediately if it is the first element in a list. Consider a program, DERIV, which takes the name of a function as input and generates the derivative of the function as its result. The derivative can then be evaluated immediately, as in the next example:

```
        ((DERIV F) 3)
```

First DERIV is invoked with F as its operand. Then DERIV returns the derivative of F as its result, and that new function is invoked with 3 as its operand.

In most languages, there is no way to apply a function to a list or vector without taking the list apart. In PL/I, for example, if F is a function with 3 parameters and A is a vector of 3 elements, the elements of A must be separated when F is applied to A:

```
F(A(1),A(2),A(3))
```

In LISP/VM, however, the function APPLY will apply a function to the elements of a list:

```
(APPLY 'F A)
```

Note that F must be quoted to keep it from being evaluated before it is applied to the elements of A. The reason for the quote is that APPLY might take an expression whose result after evaluation is a function:

```
(APPLY (DERIV F) '(3))
```

The result of this expression is the same as

```
((DERIV F) 3)
```

When a function is applied to a list, either the function could operate on the list as a whole, or it could be applied successively to the individual elements of the list. In LISP/VM, the default case is to apply the function to the whole list. To apply a function successively to sub-parts of the list, the mapping operators MAPCAR and MAPLIST may be used. If F is a function of one argument and L is a list, then

```
(MAPCAR 'F L)
```

constructs a list whose first element is F applied to the first element of L, whose second element is F applied to the second element of L, and so forth. For example,

```
(MAPCAR 'ADD1 '(2 4 6 8))  is  (3 5 7 9)
```

MAPCAR applies F first to (CAR L), then to (CADR L), then to (CADDR L), etc. MAPLIST, however, first applies F to L as a whole, then to (CDR L), then to (CDDR L), etc. For example,

```
(MAPLIST 'LENGTH '(2 4 6 8))  is  (4 3 2 1)
```

Both MAPCAR and MAPLIST analyze only to the top level of a list; if L is a list whose elements are also lists, neither MAPCAR nor MAPLIST analyze sub-parts of those lists. See the function SIMPLIFY in a later section for an application of MAPCAR.

Unlike most languages, LISP/VM supports functions that have no names. For example, the following expression is legal.

```
((LAMBDA (X) (PLUS X 5)) 7)
```

The unnamed function is a function that adds 5 to its operand. That function is then applied to 7 to generate the final result 12. For this simple case, it would be easier to write

```
(PLUS 5 7)
```

The practical uses for unnamed functions occur when a function is being passed as an argument to another function or when a function generates another function as its result. Then there is no reason to define a function and give it a unique name just for a single use. The following expression, for example, maps an unnamed function onto a list to add 5 to each element:

```
(MAPCAR (LAMBDA (X) (PLUS X 5)) '(2 4 6 8 10))
```

The result is (7 9 11 13 15).

The great flexibility in creating functions, analyzing functions, and using them as operands and results is one of the major strengths of LISP/VM. In artificial intelligence, these features are used in many creative ways. In a problem solving system, for example, a program might translate the statement of a problem into a function and then evaluate the function to generate the answer.

3.14 Data Types

LISP/VM supports a number of data types which were not in the original Lisp 1.5 description. Sometimes one data type can be an interpretation on another. This distinction allows different operators for the same kind of functions, depending on the operand type. For example, CAR and CDR will be used to access parts of a pair. ELT may be used if the object is viewed as a list. (ELT FOO 0) will return the CAR of FOO. (ELT FOO 1) the CADR, (ELT FOO 2) the CADDR, (ELT FOO 3) the CADDDR, etc.. ELT returns the ith element of a list. Note that accessing the ith element of a list takes time proportional to i. When speed of access is important, vectors may be the more appropriate data type.

3.15 Vectors

Vectors are data objects which hold collections of data. To create the most general type of vector, use the function MAKE-VEC. (SETQ FOO (MAKE-VEC 5)) will return a vector capable of holding five objects. One can access them by using (ELT FOO 0) through (ELT FOO 4). Initially, the values will be NIL. To assign values, we would input for example,

```
(SETELT FOO 2 5)
(SETELT FOO 3 (MAKE-VEC 2))
(SETELT FOO 4 '(A . B))
```

FOO would then print as

```
<NIL NIL 5 <NIL NIL> (A . B)>.
```

Conversions may be done between lists and vectors using the functions LIST2VEC and VEC2LIST. The value of (LIST2VEC (LIST 'A 'B 'C) is <A B C>.

There are several other types of vectors. These are all more specialized and can only hold one kind of data. For example, bit vectors and character vectors store boolean values and characters much more efficiently than the general vector. They are created by the functions MAKE-BVEC and MAKE-CVEC.

A string of characters may be conveniently input as a character vector by surrounding it with double quote marks. The ELT function extracts individual characters in a character vector as identifiers.

```
(ELT "Mary had a little lamb."  0)
```

returns the identifier M.

```
(ELT "Mary had a little lamb."  3)
```

returns the identifier y. Note that the identifier y is lower case and the identifier M is upper case.

Vectors can be CONCatenated by using CONC (CONC also works on lists and is synonymous with APPEND). INDEX can be used to find one substring in another, and SUBSTRING can be used to create a character vector which is a substring of another. Two character vectors can be compared for lexicographical ordering using GT, LT, or GE and equality, of course, can be checked using EQUAL.

Bit vectors can be anded, ored, or xored with BITAND, BITOR, and BITXOR.

3.16 Structured Data Definitions

Structured data definitions allow complex data objects to be manipulated in uniform and mnemonic ways. Suppose we have a data structure that contains information about an an employee in a firm. We may think of this information as consisting of three main parts: personal information, department information and salary information. Each of these parts may be further subdivided into employee number and name, department name and manager, salary, deductions and tax rate. One way to store this information in an object might be as follows:

```
(SETQ X '((1234 Smith) (Shipping Cosgrove) (120.34 11.22 12)))
```

We can extract the current deductions with the expression

```
(CAR (CDR (CAR (CDR (CDR X)))))
```

and other information with comparable expressions. An alternative is to define a single operator that allows structured access to any data object of this particular shape. The expression

```
(SETQ EMP-DATA (STRDEF EMP-DATA
       ( EMP      ( NUM , NAME ) ,
         DEPT     ( NAME , MGR ) ,
         ACCT     ( SAL , DED , TAX ) ) ) )
```

defines EMP-DATA as such an operator. With this operator, we extract information by evaluating expressions of the form:

```
(EMP-DATA DED X)        is 11.22
(EMP-DATA EMP NAME X)   is Smith
(EMP-DATA DEPT NAME X)  is Shipping
(EMP-DATA ACCT X)       is (120.34 11.22 12)
```

To change information in this data object, enter

```
(SET-S (EMP-DATA TAX X) 14)
```

and to create a new instance of a data object of this kind, evaluate

```
(EMP-DATA)  is  ((NIL NIL) (NIL NIL) (NIL NIL NIL))
```

There are many more options available in the creation and use of these operators. These are described later.

3.17 Associative Memory Operations.

In additional to the property list and association lists common in all LISP systems, LISP/VM supports a variety of hashtables and file organizations that provide associative access to stored data.

The simplest of these objects is the association list. This is simply a list of pairs in which the CAR of each pair is considered to be a key and the CDR a value. The functions ASSOC and ASSQ return a key-value pair given an association list and a key. Keys may be any objects, although identifiers are the most common example. The function ASSOC uses EQUAL when comparing keys, the function ASSQ uses EQ. While EQ allows a faster test, it also limits the data objects that can be used as keys, i.e. identifiers and small integers.

When the access functions return only the value, instead of the name-value pair, we call an association list a property list. The functions GET and PUT treat association lists in this style. The internal form of each stored identifier includes a property list. While many LISP systems use this property list to store system information, LISP/VM reserves it exclusively for application programs.

LISP/VM also has hashtables that allow more general, and much more efficient storage representations for collections of key-value associations. Whereas access to an item in an association list is proportional to the length of the list, access time in hashtables is typically a constant.

EQ hashing allows values to be associated with arbitrary data objects, while UEQUAL hashing allows values to be associated with objects of a particular shape. One common use of UEQUAL hashing is to hash on the contents of character vectors. As with most other fundamental operations, UEQUAL hashing can deal with circular data objects.

```
(SETQ FOO (MAKE-HASH-TABLE 'UEQUAL))
```

creates a hash table with tests on UEQUAL.

```
(PUT FOO (CONS "Mary" "had") 5)
```

stores the value 5 with the key ("Mary" . "had"), and

```
(GET FOO (CONS "Mary" "had"))
```

returns the value 5. If we had used EQ hashing, the value would be NIL since the value 5 in FOO is then associated with a particular instance of a pair.

3.18 Input and Output

Input and output is performed through a stream. A stream is an object that contains information about the source or destination of the data, as well as buffers and other information. The functions MAKE-INSTREAM and MAKE-OUTSTREAM are used to create streams as needed.

Output, by default, goes to a stream called CUROUTSTREAM, input to CURINSTREAM. One of the output functions is PRINT. PRINT takes either one or two arguments. The first argument should evaluate to an expression, which will then be output to the stream specified by the second argument, if present, or to CUROUTSTREAM if no second argument exists. Both CUROUTSTREAM and CURINSTREAM are initialized by the system to the terminal.

To create an output stream to a CMS file, and produce some output, we say

```
  (SETQ FOO (MAKE-OUTSTREAM '(EXAMPLE DATA A))
  (PRINT (CONS 'A 'B) FOO)
```

This sequence will place the characters "(A . B)" on the first line of the CMS file: "EXAMPLE DATA A". (PRINT 5 FOO) will place the number 5 on the second line. (SHUT FOO) will inform CMS that LISP/VM is done with the file, and let other programs, including LISP/VM itself, input data from the file.

To input data we again create an input stream

```
  (SETQ FOO (MAKE-INSTREAM '(EXAMPLE DATA A)))
```

(READ FOO) will return the pair containing the identifiers A and B. (READ FOO) a second time returns the number 5.

3.19 Entering, Editing, and Observing Programs

The normal way of using LISP/VM will be through the system editor, Lispedit. To edit the value of an expression simply input

```
  E expression
```

The expression will be evaluated and the value displayed to you. If the expression is an identifier, you may change its value using edit commands, and the value of the identifier will be different for the duration of the session, or until it is changed again.

You can create a function with a given name by editing the identifier with that name. You can then enter the definition of that function, and later evaluate it. To enter the definition, you would input the character I and press the enter key. As you type the definition pressing the enter key will cause the current value of the function to be displayed. To exit insert mode, the mode started by typing I, press the enter key, when no characters have been input. The function may now be invoked by using the function name and arguments in an expression.

The function definition may be changed by placing the cursor under the expression to be changed and pressing the PF key labeled Sel (for select). The definitions of the PF keys may be found highlighted at the bottom of the screen. Selecting a piece of the function will cause future commands to refer to that piece. That piece is referred to as the focus, because it is presumed to be the part of the function the user is focusing attention on. The focus is highlighted. DEL (the delete command) will delete the focus. Inserting may be done by entering insert mode as above, and changing may be done by using the REP command. REP with an argument will replace the focus.

To observe the evaluation of the program, input the TRAP command, followed by the name of the function. Now whenever the function is invoked, it will be displayed and the focus will be the part of the function which is about to be evaluated. The command RUN will evaluate the entire focus. The command STEP will move the focus to the first component of the current focus and evaluate it. UNTRAP is used to prevent future invocations from being displayed.

A function may be compiled by the command COMPILE function_name. After the COMPILE command has been evaluated, editing the function will actually be equivalent to editing the source, except that changes will not be reflected into the running system. The DEFINE command may be used to switch to the interpreted version of the current source. The COMPILE command will recompile the version and thus also reflect the changes back into the running system.

On-line documentation is available for editor commands. The command HELP will give a short description of the system. HELP and a command name will give a description of that command. HELP HELP will describe the HELP function. A description is simply an expression and may be edited if it is too large to fit on a screen. To do so, the system will prompt you to push PF1. HELP WHAT gives the names of all commands, grouped into coherent collections. HELP followed by the name of the collection will give a one line description of those commands. The user should try to be familiar with all the commands described in the BASIC collection.

While we recommend using the editor supplied with the system, if the user is only a casual user, one of the CMS text editors may be used. The function EXF stands for "evaluate file". (EXF '(FOO BAR A)) will evaluate all expressions in the file FOO BAR on the user's A disk. That file may contain SETQ's of function names to expressions, uses of the COMPILE function (described later), or uses of the DEFUN function (for those familiar with some LISPs other than LISP/VM.)

The CMS command may be used to evaluate commands to the CMS environment while in LISP/VM.

3.20 Saving, and Loading Programs

A set of Lispedit commands support a CMS file organization we call Lispedit Indexed Files, or indexed files for short. Indexed files allow a number of named property lists to be stored in a CMS file. We normally use the notation 'xfn' to indicate the name of an indexed file.

To save the function function type

```
SAVE xfn function_name
```

where xfn is the name of an indexed file. An indexed file is composed of a number of CMS files, and is used to store a collection of functions and data, which are logically related. The command

```
LOAD xfn *
```

will load all members of an indexed file.

The source filed by the SAVE command is stored in the file in the property list associated with id, as the property associated with the key VALUE. Other items in this property list indicate how this value should be treated. To cause id to be compiled upon loading, you set its property DISPOSITION to the value COMPILE in the property list associated with id in the file. The DI command performs this operation conveniently.

LISP/VM also contains the functions DEFUN, DEFINE and COMPILE that support a more primitive style of function definition. Note that since a function is defined by setting the value of an identifier to a functional object, a particular name can be only a function, or only a variable at a given moment. In addition, a compiled definition will cause the source definition to be discarded.

3.21 Running Programs

The simplest way to cause an expression to be evaluated is to type it in and press the enter key. For example to find the CAR of an identifier A, type

```
(CAR A)
```

and press the enter key. If the expression is an identifier and the identifier is also the name of a Lispedit command, it will be necessary to prefix the identifier with the V command. This will cause the identifier to be evaluated instead of being treated as a command. V can prefix any expression.

Most errors which can occur while the expression is being evaluated will cause an automatic return (actually a THROW) to Lispedit, with an indication that an error occurred, and some debugging information. More information may be obtained by prefixing the expression with the command VB. VB will cause the expression to be evaluated, but if an error occurs the program will be left in the Break-Loop. In the Break-Loop variable values and the program stack may be examined.

To return from the Break-Loop to Lispedit use the UNWIND operator. This will occasionally be necessary even when the VB command is not used. The FIN operator is used to continue evaluation past the error, possibly with new values. The & operator is used to examine the values of variables, and the stack used in evaluation.

If a program is looping, it is possible to force it into the Break-Loop. If a Lispedit screen is displayed with a locked keyboard (or input-inhibited), press RESET and CLEAR. The result will be a blank screen with VM READ in the lower right-hand corner. Press ENTER to switch to RUNNING mode, then PA1 to switch to CP READ mode. Finally, type EXT and wait for the Break-Loop to respond. If the screen is already in RUNNING mode, you can omit the RESET, CLEAR, and ENTER steps.

> While the screen is in VM READ or RUNNING mode, any input entered at the terminal is saved in the CMS console stack. LISP/VM will evaluate this input the next time it expects input from the console. Null and blank lines are normally ignore by the Reader, but other input should not be entered inadvertantly.

After examining variables in the Break-Loop, entering (FIN) will cause the program to continue. UNWIND may be used to return to Lispedit.

3.22 Examples

The following two functions both compute factorials. The first is defined recursively, and the second uses the program feature and a GO TO. There is little difference in performance when interpreted, because the overhead caused by the function call is balanced by the extra expressions evaluated in the looping version. However, there is a dramatic difference when the functions are compiled.

```
FACTR is
    (LAMBDA (N)
     (COND ((ZEROP N) 1)
           ('ELSE (TIMES (FACTR (SUB1 N)) N))))

FACTL is
  (LAMBDA (N)
   (PROG (X)
         (SETQ X 1)
    LOOP (COND ((ZEROP N) (RETURN X)))
         (SETQ X (TIMES X N))
         (SETQ N (SUB1 N))
         (GO LOOP)  ))
```

To input the first of these functions, evaluate it, and then compile and evaluate it, you may type the following lines pressing the enter key wherever the string [ENTER] occurs:

```
e factr [ENTER]
I [ENTER]
(LAMBDA (N) [ENTER]
(COND ((ZEROP N) 1) [ENTER]
('ELSE (TIMES (FACTR (SUB1 N)) N))))))) [ENTER]
[ENTER]
(factr 5)
compile [ENTER]
(factr 5)
```

The next two functions show some typical LISP/VM programming techniques in a merge sort. The function MERGE merges two sorted lists of numbers while preserving the order. The function SORT calls MERGE.

```
Let MERGE be
(LAMBDA (LIST1 LIST2)
  (COND ((NULL LIST1) LIST2)
        ((NULL LIST2) LIST1)
        ((LESSP (CAR LIST1) (CAR LIST2))
         (CONS (CAR LIST1) (MERGE (CDR LIST1) LIST2)) )
        ('ELSE (CONS (CAR LIST2) (MERGE (CDR LIST2) LIST1))) ))
```

SORT first checks the length of the list; if the list is shorter than 2, there is nothing to sort. Then it scans the list up to the midpoint and splits the list into two halves. Finally, it merges the results of sorting each half.

```
Let SORT be
(LAMBDA (L)
  (PROG (I MID TAIL)
        (SETQ I (LENGTH L))
        (COND ((LESSP I 2) (RETURN L)))
        (SETQ MID L)
  SCAN  (COND ((LESSP I 4) (GO SPLIT)))
        (SETQ MID (CDR MID))
        (SETQ I (DIFFERENCE I 2))
        (GO SCAN)
  SPLIT (SETQ TAIL (CDR MID))
        (RPLACD MID NIL)
        (RETURN (MERGE (SORT L) (SORT TAIL))) ))
```

The merge sort illustrated here is much faster than a bubble sort. For a list of 100 numbers, this routine runs 6.5 times faster than a bubble sort. For a list of 400 numbers, it is 23 times faster.

The following function may be used to check the evaluation time for a LISP/VM expression.

```
(TIMER 'expression)
```

will return the evaluation time of the expression in milliseconds. This example illustrates the use of the PROG feature with assignment expressions. It also illustrates the use of EVAL to change the normal order of evaluation.

```
TIMER is
(LAMBDA (EXPRESSION)
      (PROG (TIME)
            (SETQ TIME ($V-TIME))
            (EVAL EXPRESSION)
            (RETURN ($V-DELTA TIME)) ) )
```

As an example of the use of TIMER, try computing factorial 50 with the interpreted form and then with the compiled form. Type the following three lines:

```
      (TIMER (FACTR 50))
      (COMPILE (LIST 'FACTR FACTR))
      (TIMER (FACTR 50))
```

The first call evaluates the interpreted form of FACTR, then COMPILE generates the machine language form, and the third line evaluates it. Note the **exact** placement of parentheses and quote.

When the built-in function COMPILE is used to compile a function, the source definition is lost. The COMPILE command, however, stores that value where it can be retrieved by the editor. The following function saves the interpreted source form on the property list under the property SOURCE before calling COMPILE. Remember to call this function with the quoted name of the function to be compiled.

```
COMPSAVE is
(LAMBDA (FUNC)
      (PROG (EXPR)
            (SETQ EXPR (EVAL FUNC))
            (PUT FUNC 'SOURCE EXPR)
            (COMPILE (LIST FUNC EXPR) ) ) )
```

The following set of functions may be used to take the derivatives of arithmetic functions. They illustrate the ease of using LISP/VM for accessing the source form of a function, modifying it, and generating new functions.

```
Let DERIV be
(LAMBDA (EX)
      (PROG (X)
            (SETQ X (CAADR EX))
            (SETQ EX (CADDR EX))
            (RETURN (LIST 'LAMBDA (LIST X) (DIFIATE EX X))) ))
```

The function DERIV takes a monadic function as input, extracts the parameter symbol and the expression, and then passes them to the function DIFIATE to be differentiated. Finally, it combines the parameter symbol with LAMBDA and the derivative of the expression to form a function expression that represents the derivative. For the function SQUARE defined as (LAMBDA (X) (TIMES X X)),

```
       (DERIV SQUARE)
```

would produce the result

```
       (LAMBDA (X) (PLUS X X))
```

The calling program may either evaluate the derivative directly or use it to define a new function. Since SQUARE is a monadic arithmetic function, the following expression would evaluate the derivative of SQUARE for the value 3:

```
       ((DERIV SQUARE) 3)
```

The next expression defines a new function DSQ, which is the derivative of the function SQUARE:

```
  (SETQ DSQ (DERIV SQUARE))
```

Once DSQ has been defined, (DSQ 3) gives the result 6. Then to get the second derivative of SQUARE, (DERIV DSQ) gives a function expression for the constant function that produces the result 2 for any input:

```
       (LAMBDA (X) 2)
```

The following function differentiates an expression with respect to a given variable symbol. It is called by the function DERIV, but it could be used by itself. This function is not complete; it only recognizes the special functions PLUS and TIMES. As an exercise, the reader should extend the function to recognize and differentiate DIFFERENCE, MINUS, QUOTIENT, EXP, SIN, COS, LOG, etc.

```
DIFIATE is
(LAMBDA (EX X)
    (SIMPLIFY
      (COND ((ATOM EX)
              (COND ((EQ EX X) 1) ('ELSE 0)) )
            ((EQ 'PLUS (CAR EX))
             (LIST
                'PLUS
                (DIFIATE (CADR EX) X)
                (DIFIATE (CADDR EX) X) ) )
            ((EQ 'TIMES (CAR EX))
             (LIST
                'PLUS
                (LIST 'TIMES (DIFIATE (CADR EX) X) (CADDR EX))
                (LIST 'TIMES (CADR EX) (DIFIATE (CADDR EX) X)) ))))))
```

The function SIMPLIFY may be used to simplify arithmetic expressions. It is another illustration of the ease with which expressions may be taken apart and recombined. To appreciate the power of LISP/VM for such operations, the reader should try writing equivalent functions for some other language. APL would be easier to use than PL/I, but even for APL the analysis would be much more complex than for LISP/VM.

```
SIMPLIFY is
(LAMBDA (EX)
    (PROG (FCN ARGS SIMPX)
      (COND ((ATOM EX) (RETURN EX)))
      (SETQ FCN (CAR EX))
      (SETQ ARGS (MAPCAR SIMPLIFY (CDR EX)))
      (SETQ SIMPX
        (COND
          ((EQ 0 (CAR ARGS))
             (COND
               ((EQ FCN 'PLUS) (CADR ARGS))
               ((EQ FCN 'TIMES) 0)
               ((EQ FCN 'MINUS) 0)
               ((EQ FCN 'DIFFERENCE) (LIST 'MINUS (CADR ARGS)))
               ((EQ FCN 'QUOTIENT) 0)
               ((EQ FCN 'EXP) 1)
               ((EQ FCN 'EXPT) 0)
               ((EQ FCN 'SIN) 0)
               ((EQ FCN 'COS) 1)
               ('ELSE (CONS FCN ARGS)) ))
          ((EQ 0 (CADR ARGS))
             (COND
               ((EQ FCN 'PLUS) (CAR ARGS))
               ((EQ FCN 'TIMES) 0)
               ((EQ FCN 'DIFFERENCE) (CAR ARGS))
               ((EQ FCN 'EXPT) 1)
               ('ELSE (CONS FCN ARGS)) ))
          ((AND (EQ 1 (CAR ARGS)) (EQ FCN 'TIMES)) (CADR ARGS))
          ((AND (EQ 1 (CADR ARGS)) (EQ FCN 'TIMES)) (CAR ARGS))
          ((AND (EQ 1 (CAR ARGS)) (EQ FCN 'EXPT)) 1)
          ((AND (EQ 1 (CADR ARGS)) (EQ FCN 'EXPT)) (CAR ARGS))
          ((AND (EQ 1 (CAR ARGS)) (EQ FCN 'LOG)) 0)
          ((NUMP (CAR ARGS))
             (COND
               ((EQ FCN 'MINUS) (MINUS (CAR ARGS)))
               ((AND (EQ FCN 'PLUS) (NUMP (CADR ARGS)))
                    (PLUS (CAR ARGS) (CADR ARGS)))
               ((AND (EQ FCN 'TIMES) (NUMP (CADR ARGS)))
                    (TIMES (CAR ARGS) (CADR ARGS)))
               ((AND (EQ FCN 'DIFFERENCE) (NUMP (CADR ARGS)))
                    (DIFFERENCE (CAR ARGS) (CADR ARGS)))
               ('ELSE (CONS FCN ARGS)) ))
          ('ELSE (CONS FCN ARGS)) ))
      (RETURN SIMPX) ))
```

4.0 Data Types

LISP/VM contains many different types of data objects. Unlike languages like Pascal, LISP/VM data objects are typed rather than variables. A type is a set of data objects.

Types are arranged in a hierarchical fashion. The set of all numbers, for example, is a subtree of the hierarchy which includes the sets of integers and reals. The set of integers contains the set of small integers (snum) and large integers (bnum).

Objects may be of the following types:

```
identifier
   gensym
num
   rnum (floating point number)
   int  (fixed point number)
      sint (small integer)
      bint (big integer)
bpi
   fbpi (compiled function)
   mbpi (compiled macro)
funarg
state descriptor
pair
   list (alternate view of pair)
   stream (alternate view of pair)
      character stream
      key-addressed stream
vector
   integer vector
   character vector
   real vector
   bit vector
hashtable
readtable
%.EOF
NIL
```

Figure 2. Hierarchy of Types

For operations on arbitrary objects, seepage 141 .

4.1 Identifier

An **identifier** type has three components, a name, a value, and possibly an a-list, called a property list. A **name** is any character string. It is also known as the **print name** or **pname**. A **value** is the result of evaluating an expression. For an identifier, a value is the value of the binding which is associated at any point in evaluation with the binding through identifier resolution rules. The **property list** is composed of **properties** which are themselves composed of **property name**-value pairs.

The internal form of a print name is any character string. If the print name would be confused with any other sequence of characters, the external form can contain the **escape character**, "|", which denotes that the following character should be interpreted as itself and is part of the internal form. The external form is both input and output.

```
|(       The identifier whose print name is "("
+|1      The identifier whose print name is "+1"
```

An identifier which occurs as an element of a SEQ or PROG expression is a **label**. It is not evaluated, but rather serves as the point to which control can transfer through a GO.

For operations on identifiers, see page 97.

Identifiers may be grouped into two categories: stored and non-stored.

4.1.1 Stored Identifiers

Stored identifiers have a name, a value, and a property list and can be further divided into two categories: interned and uninterned. An **interned identifier** has the relation between the internal and external forms maintained in a master index called the **obarray**, or **object array**. Two interned identifiers with identical print names will always be EQ, even for different invocations of the Reader. An **uninterned identifier** has no relationship maintained.

4.1.1.1 Identifiers with Special Meaning

Some identifiers have pre-defined meanings. It is possible to redefine these identifiers but that would not be a normal practice. Certain identifiers are conventionally distinguished, such as characters and digits.

A **special form** is a macro-like operator, or the expression containing this identifier as the expression operator, which is recognized directly by LISP/VM. The special forms are:

```
CLOSEDFN        FUNCTION        PROGN
COND            GO              QUOTE
EVALQ           GVALUE          RETURN
EXIT            LAMBDA          SEQ
F*CODE          MLAMBDA         SETQ
```

A **built-in function** is a function-like operator, or the expression containing this identifier as the expression operator, which is recognized directly by LISP/VM. The built-in function are:

```
APPLY           CONS            IVECP           RPLACD
ATOM            CVECP           LISTP           RVECP
BINTP           EQ              MBPIP           SET
BFP             EVAL            MDEF            SFP
BVECP           FBPIP           NULL            SINTP
CALL            FUNARGP         NUMP            STATE
FUNCALL         GENSYMP         PAIRP           STATEP
CAR             HASHTABLEP      READTABLEP      VECP
CDR             IDENTP          RNUMP
CLOSURE         INTP            RPLACA
```

4.1.2 Non-stored Identifiers, Gensyms

Non-stored identifiers are generated symbols, **gensyms**, which are unique. This uniqueness is enforced by the system rather than by programmer convention. Gensyms do not have property-lists, and their internal print names are numbers. Gensyms can have either lexical or non-lambda bindings but they cannot be used as free or fluid variables in compiled programs.

The external input form of a gensym is:

```
%Gn
```

where **n** is a number of the user's choice. This external form is converted into an internal form where **n** is translated into **m** in the range 2^{20}-1. This internal form has never before existed in the system. During a single invocation of READ, all uses of a particular external form are associated with this unique internal form. During a subsequent invocation of READ, the use of the same external form will be associated with an entirely new **m**.

All invocations of PRINT for a gensym will print the same output external form of the internal form.

```
%Gm
```

There is a one-to-one mapping, on output, between the output external form and the internal form.

```
X
Value = %G1325
Y
Value = %G1325
(EQ X Y)
Value = non-NIL
(SETQ A '%G1)
Value = %G1401
(SETQ B '%G1)
Value = %G1402
(EQ A B)
Value = NIL
(SETQ C '(%G1 %G2 %G1 %G2))
Value = (%G1403 %G1405 %G1403 %G1405)
```

There is only a finite number of possible internal forms of gensyms, $.75*2^{20}$, and an error is signalled if an attempt is made to generate more than that number.

4.2 Numbers

There are three types of **numbers**: small integers, large integers, and real numbers.

Small integers have type SNUM and have a range -2^{26} to $2^{26}-1$ (-67,108,864 to 67,108,863). **Large integers** have type BNUM and a range which includes all values not in SNUM. The arithmetic operators work on all numbers and allow mixed-mode arithmetic. The small integer type achieves greater efficiency in computation and storage than the large integer type. All integers have exact representation with the amount of space available in the heap being the only limitation on integer size.

The input/output external format for integers is:

```
[±] {d...}
```

where **d** specifies a decimal digit and no spaces may appear within the integer.

Real numbers are stored using System/370 double precision floating point format, yielding 53 to 56 bits of precision for the mantissa and a range of approximately 10^{74}.

The input/output external format for real numbers is:

```
[-] [d...] . [d...] [E [+|-] {d...}]
```

where **d** specifies a decimal digit, no spaces may appear within the real number, and at least one digit must be present.

There are two parameters which control the way in which real numbers are translated into their output external forms, FUZZ and NDIGITS. FUZZ refers to a value used to define the intended precision of real number operations. Two real numbers, X and Y, are equal if

```
||X| - |Y|| ≤ FUZZ * maximum (|X|, |Y|)
```

When a real number X is printed, the external form used is that which would have the shortest character string.

```
X-FUZZ*|X|  to  X+FUZZ*|X|
```

This external form may include an exponent, in which case there will be exactly one decimal digit before the decimal point, or in cases where the number of digits (exclusive of decimal point and a possible minus sign) needed to represent the numeric value is less than NDIGITS, no exponent will be printed and the decimal point will be placed wherever required.

The user may specify values for FUZZ and NDIGITS by using the function SETFUZZ.

For operations on numbers, see page 101.

4.3 Binary Program Images

The result of compilation of an expression or assembly of a LISP Assembly Language program (LAP) is a binary program image, or **bpi**. Function binary program images are known as **fbpis** and macro binary program images are known as **mbpis**.

The external form of a bpi is output only. If an attempt is made to input a bpi, READ will signal an error.

```
  %.FBPI.bpiname   or   %.MBPI.bpiname
```

4.4 Funargs

A funarg type consists of an expression with a specific environment in which that expression is to be evaluated. It is represented as a pair-like object. The type of the expression in the funarg characterizes the funarg: identifier funarg, gensym funarg, number funarg, function funarg, macro funarg, etc.

The external form of a funarg is output only.

```
  %.FUNARG.(expression . sd)
```

4.5 State Descriptor

A state descriptor is a data object which contains a state, that is, an environment, control chain, and control point.

State descriptors are created by the STATE built-in function which saves the current state. This current state is then retained even after the currently evaluating object terminates. When combined with an expression into a funarg, it can be used to provide the environment in which an expression is evaluated. It can also be used to transfer control back to the saved state with evaluation continuing at the expression following the saved control point.

The external form is output only.

```
    %.SD.xxxxxxxx
```

4.6 Pairs

A pair is a stored data object having two component objects which are referred to as the CAR component and the CDR component (for historical and compatibility reasons). Each component may be an arbitrary object.

Sometimes it is useful to illustrate data objects. The convention for pairs is a box diagram. Given a pair with a CAR consisting of an arbitrary object, **a**, and a CDR consisting of an arbitrary object, **b**, it can be drawn as:

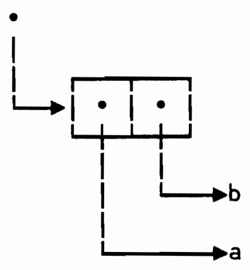


Figure 3. Box representation of pairs.

Two basic functions are provided for accessing part of a pair. CAR or CDR applied to a pair returns as its value the corresponding component of the pair.

The external form of a pair consists of a left parenthesis followed by the external form of the first element of the pair, a blank, a period, a blank, the external form of the second element of the pair, and finally a right parenthesis. The external form of the previous example is

```
(A . B)
```

For operations on pairs, see page 109.

4.6.1 Lists

Lists are composite objects created from pairs by taking an alternate view of the pair data type. Thus each pair is a list whose CAR component is interpreted as the first element of that list, and whose CDR component is interpreted as the remainder of that list.

The distinguished object NIL is used to denote an empty list. Thus, if the CDR of a pair is NIL, there are no remaining elements in that list.

Having NIL as its CDR component is only one way in which a pair may be the end of a list. If the CDR of a pair is any data object other than a pair, that pair terminates a list. This definition of list is more general than the traditional LISP definition and accepts all traditional lists which end in NIL as well as lists which end in non-NIL.

For the purposes of functions which operate on lists, the CDR component of the pair terminating the list is not considered to be part of the list.

Using the box notation, a list of three elements, **A**, **B** and C, would have the following structure:

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

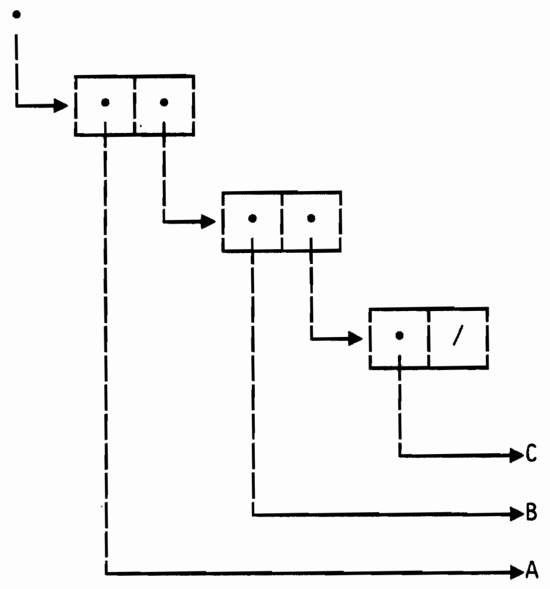


Figure 4. Box representation of list, corresponding to dot notation.

Note the convention of representing a pointer to NIL by a / in the appropriate box.

The external form of a list is a modification of the representation of its component pairs as described above. This modification is intended to improve readability by eliminating some of the parentheses and divulging the sharing of data; however, the inclusion of some (or all) of the deleted parentheses is always acceptable in input data. During printing, when a pair is pointed to from the CDR of another pair, the separating period and blank of the original pair and the enclosing right and left parentheses of the CDR are not printed. In addition, when the terminating pair of a list has NIL as its CDR component, that NIL and the space, period and space which would separate it from the CAR value are not printed. (Note, that if NIL is represented by its alternative form, (), the first rule has the same results automatically.

Thus, the list

```
(A . (B . (C . NIL)))
```

would appear as

```
(A B C)
```

when printed. While

```
(A . (B . (C . D)))
```

will print as

```
(A B C . D)
```

List notation will be presented horizontally in future box diagrams. Thus, the previous list will be drawn as:

```
(A B C)
```

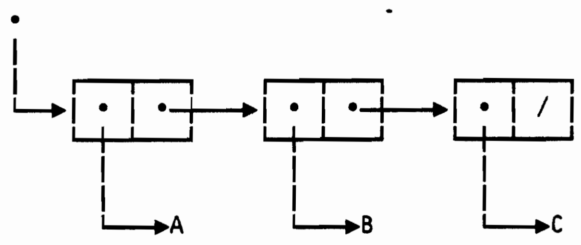


Figure 5. Box representation of list, corresponding to list notation.

For operations on lists, see page 113.

4.6.1.1 Circular Lists

Since a pair is a perfectly reasonable element of a list, it is possible to create lists which include themselves, or parts of themselves, as elements. LISP/VM uses a general scheme for input/output which indicates the sharing of data. This sharing scheme, as well as other aspects of the input/output system, makes use of a break character which is defined in the standard system as percent (%). An input expression written:

```
%L1=(A . %L1)
```

generates a pair whose CAR component is a pointer to the identifier A and whose CDR component is a pointer to the pair itself. The structure is:

```
%L1=(A . %L1)
```

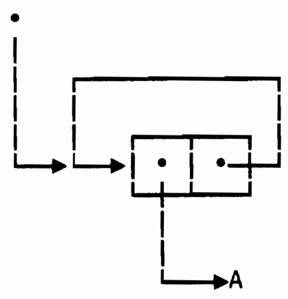


Figure 6. Box representation of a circular list.

The list interpretation of this pair would be a circular list -- effectively an infinite list of A's.

This sharing notation need not generate a circular list. For example, the expression:

```
(%L1=(A) %L1)
```

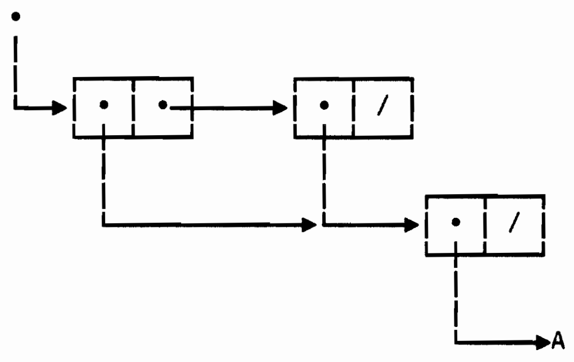
generates a list containing two elements. The first element is the list containing a single element -- the identifier A -- and the second element is another identical pointer. This is to be distinguished from the expression:

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

which also generates a list of two elements, each of which is a list containing the single identifier A. In this case, however, the two elements are different pointers, although they point to equal (but separately stored) lists.

These two structures are:

```
(%L1=(A) %L1)
```



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

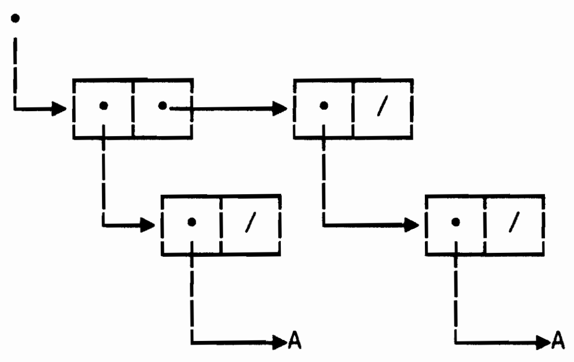


Figure 7. Box representation of equivalent shared and non-shared lists.

For purposes of accessing the elements of the list, both expressions are equivalent (but note that the list having the shared data requires less storage). These two lists are not equivalent with respect to updating. That is, the product of updating one may not be the same as the product achieved by the same updating operation applied to the other.

There are two primary output operators, PRINT and PRETTYPRINT. PRINT produces a continuous sequence of characters, with *all* shared structure exposed, whether circular or not. PRETTYPRINT produces a formatted representation, with blanks and new-lines inserted were it is deemed appropriate. However, PRETTYPRINT does *not* expose non-circular sharing. Thus the two (different) structures presented here would appear the same when PRETTYPRINTed. PRETTYPRINT has no rules for displaying circular structures, and defaults to the unformatted, PRINT, representation if any are present in its operand.

Note that the system provided READ-EVAL-PRINT supervisor uses PRETTYPRINT for echoing its input and displaying the results of evaluation. This, in turn, means that non-circular sharing will not be visible during interactions with the supervisor. LISPEDIT, also, does not explicitly show sharing, circular or not, but it does inform the user of its presence.

In general, if it is true of two structures that corresponding accesses yield equivalent values then it can be said that the structures are equivalent trees (see EQUAL function). If it is true that the products of some updating operation applied to two structures would leave them EQUAL, then the structures can be said to be equivalent rooted directed graphs (see UEQUAL function).

4.6.2 *Streams*

Streams are an alternate view of pairs. There are two distinct types of streams, each with its own set of operators.

The streams which are managed by READ, PRINT and their related operators can be viewed as producers or receivers of characters. These are the streams which allow LISP/VM to communicate with the console, and to read and write files which can be edited by the user.

The streams which are managed by RREAD, RWRITE and their related operators can be viewed as producers or receivers of arbitrary data objects. The DASD files which correspond to these streams are, in general, not readable by the user.

All operations on streams are updating operations, that is, operations which change values, e.g., the CAR of a stream uses a RPLACA operation. This insures that all processes which have access to a particular stream remain in synchronization.

For operations on streams, see page 277 and page 283.

4.6.2.1 Character Streams

All character streams are pairs, with the CAR of the stream being the current character, or an "end-of-line" flag.

The simplest form of a character stream is a list of characters:

```
(A B C)
```

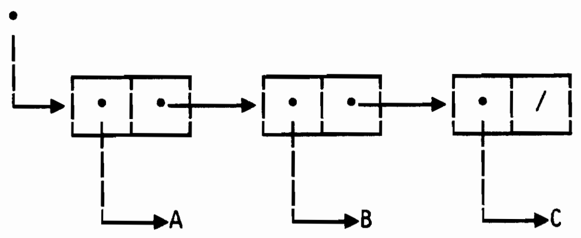
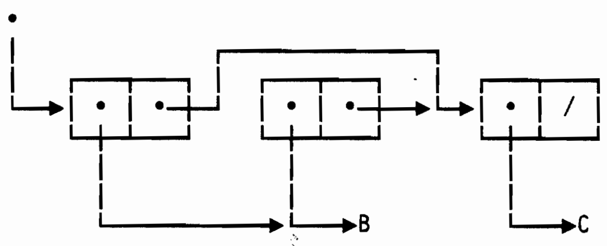


Figure 8. A list, interpreted as a character stream.

The primitive operations on such a stream are NEXT and PRINT-CHAR.

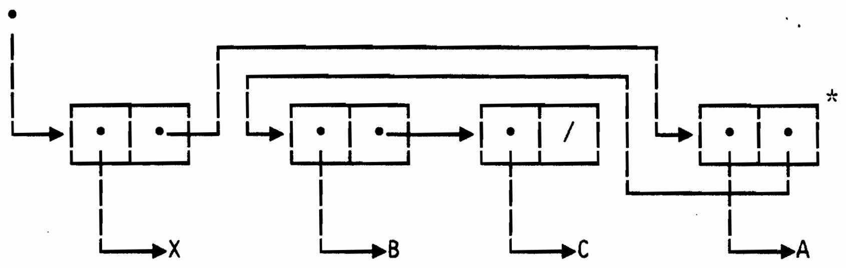
After an application of NEXT

(B C)



After an application of PRINT-CHAR, with X as first operand

(X A B C)



(Note, only the pair marked by * is newly created.)

Figure 9. Effect of NEXT and PRINT-CHAR on stream, Figure 8.

It should be noted that NEXT and PRINT-CHAR are mirror operations. If a list is produced by a series of PRINT-CHARs, the repeated application of NEXT will produce the characters in the reversed order.

The most common form of character stream is called a *fast stream*. This is a list, usually of a single element, the final CDR of which is a vector containing the information required for performing I/O to or from a DASD device or a terminal.

Fast streams contain a number of components which may be accessed or changed by various operators. These include an association list, with entries specifying the direction of the stream (INPUT or OUTPUT), the device type, the file name, etc., as needed. There is a buffer, containing the current line from the device, with an index designating the current character in the buffer. A *stream specific function* defines the action to be taken when the buffer is to be disposed of, either refreshed from an input device, or written to an output device.

Two special configurations of a fast stream indicate end-of-line and end-of-file conditions. These may be tested with the predicates EOLP and EOFP.

4.6.2.2 Key-Addressed Streams

The second type of stream supports key addressed, random access, files or *libraries*. Each member in such a file represents a single data object. Associated with each item is a key and a class designation. The key is either a string or a GENSYM identifier, while the class is a small integer in the range 0 (zero) to 255.

The usual file type for such files is LISPLIB.

Unlike character steams, libraries may contain an external form of a bpi, and are often used as an analogue to VM TXTLIB files. It must be noted that an existing bpi can *not* be written to a library, but can only be placed in one by the assembler.

The basic operators which deal with libraries are distinguished by a prefixed R on their names, for *R*andom access. In addition there exist a set of operators, including LOADVOL, SUBLOAD etc, which manage these files and allow operator definitions to be loaded from them.

4.7 Vectors

Vectors are one-dimensional objects containing components, or **elements**, with efficient indexed access and a compact storage representation. The specific vector types include the set of integer vectors, real vectors, character vectors, and bit vectors. If the vector is not identified as a vector of specific element type, then the elements of the vector can be any arbitrary data objects.

Except for bit vectors, vectors may have any length (including zero) for which sufficient space exists in the heap. All vectors have a **length**-- the number of elements in the existing vector object. They also have a **capacity**, the maximum number of elements in the string. Character vectors and bit vectors are known as **strings**.

The input/output external form of a vector of arbitrary elements is:

```
<E0 E1 ... Ei>
```

where Ei is the ith element of the vector.

A box diagram can be used to describe a vector.

```
<A B C>
```

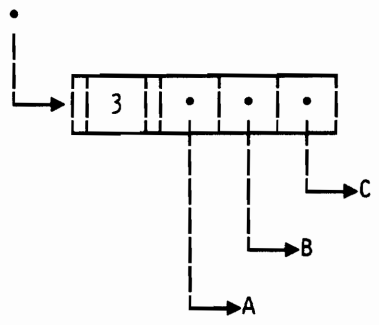


Figure 10. Box representation of a vector.

For operations on vectors, see page 123 and page 129.

4.7.1 Integer Vector

An **integer vector** is a vector containing integer elements.

The input/output external form of an integer vector is:

```
%I<INT...>
```

where **INT** specifies an integer element.

The length and capacity of an integer vector are always the same. The form

```
%Ik<INT...>
```

can be used for a vector of k elements. As an input external form, it can be used to create a vector of k elements initialized to NIL.

4.7.2 Real Vector

A **real vector** is a vector containing real number elements.

The input/output external form of a real vector is:

```
%F<R...>
```

where **R** specifies a real number element.

The length and capacity of an real vector are always the same. The form

```
%Fk<R...>
```

can be used for a vector of k elements. As an input external form, it can be used to create a vector of k elements initialized to NIL.

4.7.3 Character Vectors

A **character vector** is a vector containing character elements. It is also known as a **character string**. The delimiters and escape character can be used as characters within a character string by preceding them with the escape character. Delimiters and escape characters are not counted in the length.

All character strings have both a length and a capacity. Due to storage representation, the capacities are quantized to multiples of four plus one.

The main input/output external form for character vectors is:

```
"C..."
```

where **C** specifies a character element. When this form is used, the capacity becomes the quantized capacity which is greater than or equal to the length.

For example,

```
v "ABCD"
"ABCD" = "ABCD"
(SIZE "ABCD")
(SIZE "ABCD") = 4
(CAPACITY "ABCD")
(CAPACITY "ABCD") = 5
(CONCAT "ABC" "DEF")
(CONCAT "ABC" "DEF") = "ABCDEF"
```

The user can explicitly specify a capacity through operations and through a second input external form. The same form is used on output, in some cases, to display the capacity.

```
%k"C..."
```

where **C** specifies a character element and **k** is the capacity designator. When the capacity designator is specified on input, it is rounded up to the next quantized capacity and becomes the capacity of the string.

- If the length equals the capacity, or is up to three less than the capacity, it is displayed on output in the simple, "..." form.
- If the length is more than three less than the capacity, it is displayed on output in the %k"..." form.
- If the length is greater than the capacity, an error is signalled.

For example,

```
v %6''ABCDE''
%9''ABCDE'' = %9''ABCDE''
(SIZE %6''ABCDE'')
(SIZE %9''ABCDE'') = 5
(CAPACITY %6''ABCDE'')
(CAPACITY %9''ABCDE'') = 9
v %6''ABCDEF''
''ABCDEF'' = ''ABCDEF''
v %6''ABCDEFG''
''ABCDEFG'' = ''ABCDEFG''
v %6''ABCDEFGH''
''ABCDEFGH'' = ''ABCDEFGH''
v %6''ABCDEFGHI''
''ABCDEFGHI'' = ''ABCDEFGHI''
v %6''ABCDEFGHIJ''
error is signalled
(UNWIND 1)
```

4.7.4 Bit Vectors

A **bit vector** is a vector containing bit elements. It is also known as a **bit string**.

All bit strings have a length and capacity. Length is the number of elements, or bits, in the bit string. Capacity is the maximum number of elements, or bits, a bit string can have. Due to storage representation, the capacities are quantized to multiples of thirty-two plus eight.

The main input/output external form for bit strings is:

```
%B''H...''
```

where **H** is a hexadecimal digit representing four bit elements. When this form is used, the length is inferred to be the number of bits from the leftmost bit of the bit string to the rightmost non-zero bit. This length is known as the **inferred length**. The capacity is inferred to be the quantized capacity greater than or equal to the number of hexadecimal digits in the bit string times four. This capacity is known as the **inferred capacity**. When this form is used as the input external form, the value returned may be displayed in one of the forms described below.

For example,

```
v %B''01''
%B''01'' = %B''01''
(SIZE %B''01'')
(SIZE %B''01'') = 8
(CAPACITY %B''01'')
(CAPACITY %B''01'') = 8
v %B''80''
%B''8'' = %B''8''
(SIZE %B''80'')
(SIZE %B''8'') = 1
(CAPACITY %B''80'')
(CAPACITY %B''8'') = 8
```

The user can explicitly specify a capacity and a length through operations and other input external forms. The same forms are used on output, in some cases, to display the capacity and length.

```
%B:c''H...''
%Bk''H...''
%Bk:c''H...''
```

where **H** stands for a hexadecimal digit which represents four bit elements, **k** is the capacity designator, and **c** is the length. When the length is specified on input, it becomes the length of the bit string. When the length is not specified, the inferred length becomes the length of the bit string. When the capacity designator is specified on input, it is rounded up to the next quantized capacity and becomes the capacity of the string. When the capacity designator is not specified, the inferred capacity becomes the capacity of the string.

- If the length of the bit string is greater than the inferred length, then the length is printed on output.
- If the length of the bit string is equal to the inferred length, then the length is not printed on output.
- If the length of the bit string is less than the inferred length, then an error is signalled.
- If the length equals the capacity, or is up to thirty-one less than the capacity (or seven for capacity of eight), then the capacity is not printed on output.
- If the length is more than thirty-one less than the capacity (or seven for capacity of eight), then the capacity is printed on output.
- If the length is greater than the capacity, then an error is signalled.

For example,

```
v %B:13''001''
%B:13''0010'' = %B:13''0010''
(SIZE %B:13''001'')
(SIZE %B:13''0010'') = 13
(CAPACITY %B:13''001'')
(CAPACITY %B:13''0010'') = 40

v %B:12''001''
%B''001'' = %B''001''

v %B:11''001''
error is signalled
(UNWIND 1)

v %B32''001''
%B''001'' = %B''001''

v %B32''800''
%B40''8'' = %B40''8''

v %B11''001''
error is signalled
(UNWIND 1)

v %B10:4''8''
%B10:4''8'' = %B10:4''8''
```

4.8 Hashtables

Hashtables are composite objects containing a collection of key-value pairs. They are designed to allow relatively uniform access times to any of their components, as contrasted with a-lists, where the range of access times is proportional to the number of components.

Each hashtable has a specified key class. There are four classes: two for arbitrary keys and two for special key sets.

The two arbitrary key classes differ in the comparison operator, EQ or UEQUAL, used to search the buckets for a match.

The two special key set classes (and corresponding hashing functions) are provided for character strings and for identifiers. EQUAL is used for comparison of character strings and EQ is used for comparison of identifiers.

In addition to the key class, every hashtable is either "strong" or "weak". In a **strong hashtable**, once a key/value pair has been added it will persist until it is explicitly removed. In a **weak hashtable**, in contrast, any such pair whose key is not accessible from some place other than the hashtable itself will be deleted at garbage collection time. This allows a table to be built with keys drawn from various data objects, and have the corresponding entries vanish when any of the objects is discarded.

A **bucket** is the collection of the keys which have been hashed to the same hashcode. The number of key-value pairs in each bucket is under user control, allowing a maximum average bucket length (and hence search time) to be maintained. Whenever a key/value pair is added or deleted from a hashtable the count/size ratio is computed and compared with growth and shrinkage ratios. As long the actual ratio is between the specified ratios, the table is left alone. Whenever it deviates the bucket array is rebuilt, either larger or smaller, to return the ratio to the specified range.

One set of hashtable access functions treat them as a-lists. In other words, they follow the precedent set by ASSOC and return a key-value pair. Another set of functions follow the example of GET and treat hashtables as property lists, returning only the value associated with a key. The advantage of the a-list style is that it is possible to distinguish between a missing key-value pair and a value of NIL. The advantage of the property list style is that the value can be used immediately.

The external form of a hashtable is output only.

```
%.HASHTABLE.xxxx
```

For operations on hashtables, see page 137.

4.9 Readtables

A readtable is an object that defines the syntax rules to be used by the Reader. Readtables are built and modified by the operators defined in the User-Defined Syntax chapter of the INPUT/OUTPUT section.

The external form of a readtable is output only.

```
%.READTABLE.xxxx
```

For operations on readtables, see page 259.

4.10 Miscellaneous Types

There are two miscellaneous types, NIL and %.EOF. NIL denotes the empty list or the boolean value 'false'. %.EOF. is the value returned by READ on an empty stream.

5.0 Evaluation

5.1 Object Evaluation

All objects can be evaluated.

- Numbers, state descriptors, vectors, hashtables, readtables, and NIL evaluate to themselves.
- The evaluation of an identifier involves finding the binding of the identifier using the identifier resolution rules and returning the value of that binding.
- Lists, pairs, and streams evaluate using rules which depend on the value of the operator.
  - **special form invocation**: Occurs when one of the following identifiers appears as the value of the operator:

CLOSEDFN	FUNCTION	PROGN
COND	GO	QUOTE
EVALQ	GVALUE	RETURN
EXIT	LAMBDA	SEQ
F*CODE	MLAMBDA	SETQ

    Each special form has its own evaluation rules which are described with the description of the operator. If the identifier corresponding to a special form has been redefined, evaluating the identifier will result in the application of the new definition. Evaluating the quoted identifier will result in the application of the original built-in definition.
  - **function invocation**: Occurs when the value of the operator is a function expression or an fbpi. Evaluation of the function invocation causes the arguments to be evaluated and bound to the bound variables. The environment and control chain are created for the function invocation. The body of the function expression or fbpi is evaluated. The value of the body becomes the value of the function invocation.
  - **built-in function invocation**: Occurs when one of the following identifiers appears as the value of the operator:

APPLY	CONS	IVECP	RPLACD
ATOM	CVECP	LISTP	RVECP
BINTP	EQ	MBPIP	SET
BFP	EVAL	MDEF	SFP
BVECP	FBPIP	NULL	SINTP
CALL	FUNARGP	NUMP	STATE
FUNCALL	GENSYMP	PAIRP	STATEP
CAR	HASHTABLEP	READTABLEP	VECP
CDR	IDENTP	RNUMP	
CLOSURE	INTP	RPLACA	

    The rules for evaluation of a built-in function invocation follow the rules for the evaluation of a function invocation.

    If an identifier corresponding to a built-in function has been redefined, evaluating the identifier will result in the application of the new definition. Evaluating the quoted identifier will result in the application of the original built-in definition.
  - **identity rule**: If the value of the operator is not a special form or a built-in function and the value is EQ to the operator, an error is signalled.
  - **macro invocation**: Occurs when the value of the operator is a macro expression or an mbpi. Evaluation of the macro invocation causes the single argument to be bound to the single bound variable. The environment and control chain are created for the macro invocation. The body of the macro expression or mbpi is evaluated. The value returned by the body is evaluated again.

- **function expression evaluation**: Occurs when the value of the operator is the identifier LAMBDA. The value of the function expression evaluation is a funarg, which contains the function expression and the current environment.
- **macro expression evaluation**: Occurs when the value of the operator is the identifier MLAMBDA. The value of the macro expression evaluation is a funarg, which contains the macro expression and the current environment.
- **funarg invocation**: Occurs when the value of the operator is a funarg. Application of the expression in the funarg proceeds normally in the environment contained in the funarg.
- **erroneous atom rule**: If the value of the operator is a non-identifier atom that fits none of the above rules, then an error is signalled.
- **re-evaluation of the operator**: Occurs when the value of the operator is none of the above. The operands are evaluated and their values saved, then the value of the operator is evaluated. The above function and funarg rules are used if the new value is one of the above. Macros and special forms may not be the result of the re-evaluation of the operator. The operands will only be evaluated once. If the new value is also none of the above, it is itself re-evaluated. This process may result in an indefinite repetition.

- The value of a bpi evaluation is a funarg, which contains the bpi and the current environment.
- The evaluation of a funarg involves evaluating the expression in the environment contained in the funarg.

5.2 Environment and Control Chain

5.2.1 Invocation

Bindings are the associations of identifiers with values. **Non-lambda bindings** are those bindings which are present prior to any invocations by the user. The set of all non-lambda bindings is known as the **non-lambda environment**. LISP/VM provides for multiple non-lambda environments to allow such features as compile-time choice of function or macro expressions, based on expected efficiency. Since this would not affect the normal user, all discussions of environments will assume a single non-lambda environment. A **lambda binding** is an association of a bound variable to an argument. The set of all lambda bindings for bound variables specified within a function or macro expression are known as a **local environment**.

Every invoked object has three parts: an environment, a control chain and a current control. These parts are called the **state** of the invoked object. Every control point has associated with it a current environment and a current control chain. When an applicable object is invoked, a record is kept of the control point to which evaluation should return when this invocation terminates. These records are known as the **control chain**. The point of current evaluation is known as the **current control**. The **environment** is the set of all bindings which exist during invocation. A **resolution boundary** can exist within an environment to limit searches during identifier resolution.

When an applicable object is invoked,

- A pointer to the local environment of the invoked object is added to a copy of the environment of the invoking object or, if the invocation was at the top level, to a pointer to the non-lambda environment. This becomes the environment of the invoked object. (Environments consist of pointers to local and non-lambda environments so that the actual bindings are shared and changes to a binding by one object can be visible to another object.) A resolution boundary is created.

- The control point which immediately follows the invocation in the invoking object or in the top level is added to a copy of the control chain of the invoking object or top level. This becomes the control chain of the invoked object.

- The control point changes from the invocation to the first expression in the body of the invoked object.

If an object A invokes an object B, then B is said to be **immediately dynamically nested** within A. If B then invokes C, C is immediately dynamically nested in B and **dynamically nested** in A. A would be the **outermost** nested object and C would be the **innermost** nested object.

> This semantic model has been chosen for simplicity of description. The actual implementation mirrors the model in a highly efficient manner by using shallow binding cells to effect efficient access to the environment and by minimizing both copying and creation of funargs.

5.2.2 Invocation in Non-standard Environments

Application of STATE causes a state descriptor to be created containing the current state. This is the only way for an invoked object to prevent its environment from disappearing upon invocation termination. If APPLY, EVAL, or MDEF are then invoked with an applicable object and the state descriptor as arguments, the result is to invoke the object in the environment contained in the state descriptor..

- A copy of the environment of the state descriptor is made and a pointer to the local environment of the invoked object is added to the copy and becomes the environment of the invoked object. A resolution boundary is not created.
- A copy of the control chain of the object which evaluated the APPLY, EVAL, or MDEF is made, the control point which immediately follows the invocation in the invoking object is added, and this becomes the control chain of the invoked object.
- The control point changes from the invocation to the first expression in the body of the invoked object.

5.2.3 Invocation Termination

When an invoked object terminates through evaluation of a contained RETURN or EXIT expression or through evaluation of the last expression in the body:

- control returns to the control point specified in the control chain,
- the environment and control chain of the terminated object are deleted, and
- the environment and control chain associated with the specified control point are resumed.

5.2.4 Changing the Environment and Control Chain

There are several ways to change states. Application of STATE causes a state descriptor to be created containing the current state. This is the only way for an invoked object to prevent its environment from disappearing upon invocation termination.

Application of JAUNT to a state descriptor causes the saved state to become the current state.

It is also possible to replace the current state with the state of an outer dynamic object. This allows non-local transfers of control and error recovery. Evaluation of a catch operator sets a catch point in the control chain. There can be multiple catch points. Evaluation of a THROW causes the control chain to be outwardly searched for a catch point and causes the current control point to be the expression following the catch point, and the environment and control chain which would be current during the evaluation of the catch point to become the current environment and the current control chain.

5.3 Identifier Resolution

Identifier resolution is the process of finding a binding for a particular instance of an identifier. When an identifier is encountered,

- If a binding occurs in the current local environment, then the identifier resolves to that binding.
- If a binding does not occur in the current local environment, then the search continues outward in all previous local environments until a resolution boundary is found. If a binding is found before a resolution boundary occurs, then the identifier resolves to that binding.
- If a binding is not found before a resolution boundary occurs, then the search continues outward only for a fluid binding.
- If a binding is not found in the outermost previous local environment, then it is guaranteed to be found in the non-lambda environment.

5.4 Evaluation Operators

EVAL —— built-in function

(EVAL **exp** [**sd**])

If only one argument is specified, The EVAL operator causes the value of its operand, **exp**, to be evaluated as if it had occurred in the place of the EVAL expression. If the **sd** argument is specified, EVAL causes the value of **exp** to be evaluated in the environment captured by **sd**. Both evaluations have access to the lexical bindings in **sd.**

The following example illustrates the effect of EVAL when invoked with one argument:

```
(SETQ X 'A)
Value = A
( (LAMBDA (A B) (EVAL B)) 1 X)
Value = 1
```

The sequence of evaluations in the last expression is:

1. X evaluates to A
2. The identifiers A and B are bound to the objects 1 and A respectively.
3. B evaluates to A
4. EVAL applied to A evaluates to 1
5. The function expression returns 1

Note that even though the identifier A was not lexically present in the function expression, its lexical value was found.

```
The following example shows how a state descriptor retains a
```

```
(SETQ X ( (LAMBDA (Y) (STATE)) 10))
Value = %.SD.xxxxxxxx
(SETQ Y 100)
Value = 100
(EVAL 'Y)
Value = 100
(EVAL 'Y X)
Value = 10
(EVAL '(PLUS Y Y))
Value = 200
(EVAL '(PLUS Y Y) X)
Value = 20
```

EVALFUN —— built-in function

(EVALFUN **exp** [**sd**])

The EVALFUN operator causes the value of its operand, **exp**, to be evaluated as if it had occurred in a function of no arguments called from the place of the EVALFUN expression.

```
The following example shows how EVALFUN differs from EVAL.
(SETQ X 'A)
Value = A
(SETQ A 10)
Value = 10
( (LAMBDA (A B) (EVALFUN B)) 1 X)
Value = 10
```

The sequence of evaluations in the last expression is:

1. X evaluates to A
2. The identifiers A and B are bound to the objects 1 and A respectively
3. B evaluates to A
4. EVALFUN applied to A does not see the lexical binding of A to 1. Instead, the outer, non-lambda binding of A, namely 10, is returned
5. The function expression return 10

Here, unlike the previous case, the lexical binding of the variable A in the function expression is not found by the evaluation, and the previous binding is seen. If **sd** is specified, the same sort of evaluation takes place in the environment of **sd**

EVAL-ID —— function

(EVAL-ID **id**)

The value of this operator is exactly the same as that of EVALFUN when applied to the same argument. It differs in that its operand must have an identifier as value, (there is no restriction on the value of the identifier, just on the value of the operand in the EVAL-ID expression), and in its efficiency. Where EVALFUN must be able to evaluate any expression, EVAL-ID, being restricted to identifiers, can use a much faster mechanism.

EVAL-LEX-ID —— function

(EVAL-LEX-ID **id**)

This operator is the (special case) identifier evaluator corresponding to EVAL. Unlike EVAL-ID it can access those lexical bindings which are visible from the context of application.

```
(SETQ X (SETQ Y 10))
Value = 10
( (LAMBDA (X (FLUID Y)) (EVAL-ID 'X)) 100 100)
Value = 10
( (LAMBDA (X (FLUID Y)) (EVAL-LEX-ID 'X)) 100 100)
Value = 100
( (LAMBDA (X (FLUID Y)) (EVAL-ID 'Y)) 100 100)
Value = 100
( (LAMBDA (X (FLUID Y)) (EVAL-LEX-ID 'Y)) 100 100)
Value = 100
```

EVAL-GLOBAL-ID —— function

(EVAL-GLOBAL-ID **id**)

This operator returns the value of its argument as bound in the non-lambda environment. No values bound on the stack, either FLUID or LEXical, are seen.

CEVAL-ID —— *function*

(CEVAL-ID **id**)

CEVAL-ID searches for a FLUID binding of **id** in the local environments of the invoking objects, using the control chain to identify these local environments. If no binding is found in a previous local environment, the non-lambda environment is searched; if no binding is found there, the value is **id**.

> In the actual implementation, the stack frame is maintained as a spaghetti stack containing both environment and control chain. It also contains pointers to the previous environment and previous control. These can be different, though that is rare in practice. CEVAL-ID breaks the normal evaluation search rule, and follows the control hierarchy at the splits.
>
> The primary purpose of this abnormal evaluation is to search the control chain for values which control exception handling. The error break mechanism is controlled by the value of a free (hence non-lambda or FLUIDly bound) variable, PROGRAM-EVENTS.
>
> It is felt that the error break specifications found in the control chain are more meaningful than those found in the environment chain in the cases when they differ. Thus, a use of EVAL will not temporarily switch exception handling to the status which existed when the **sd** operand was created, but will retain the current status.

CEVAL-LEX-ID —— *function*

(CEVAL-LEX-ID **id**)

This operator differs from CEVAL-ID only in having access to those LEXical bindings visible from the application context.

GVALUE —— *special form*

(GVALUE **id**)

This special form returns the contents of a unique global value cell associated with its argument. **id** is not evaluated, and must be an identifier. Otherwise an error is signaled.

EVAL-GVALUE —— *function*

(EVAL-GVALUE **id**)

This operator returns the contents of the global value cell associated with the identifier which is the value of its argument, **id**. If the value of the argument is not an identifier an error is signaled.

This is the functional equivalent of the special form GVALUE.

SET-GVALUE —— *function*

(SET-GVALUE **id item**)

This operator stores the value of **item** into the global value cell associated with the identifier **id**. If the first argument is not an identifier, an error is signalled. No check is made on the value of **item**.

5.5 Creating Funargs

Funargs are created by:

- evaluating a function expression
- evaluating a macro expression
- evaluating a bpi
- applying CLOSURE to an expression and a state descriptor
- evaluating a FUNARG expression
- applying FUNCTION to an expression

STATE —— built-in function

(STATE [**a-list**])

This operator creates a state descriptor which saves the current environment, control chain and control point.

If the optional argument, **a-list**, is present, it is used as the non-lambda environment. **a-list** differs form normal a-lists in the interpretation of its final (non-pair) CDR. If **a-list** terminates in a state descriptor, the non-lambda environment component of that state descriptor becomes an extension of the non-lambda environment which holds **a-list**. While this extension (and any extension it may have in turn) is searched for bindings, only the portion of the non-lambda environment represented by **a-list** will have new bindings added to it.

CLOSURE —— built-in function

(CLOSURE **exp sd**)

This operator creates a funarg composed of **exp** and **sd**.

FUNARG —— special form

(FUNARG **exp sd**)

This operator creates a funarg composed of its two arguments, unevaluated. It is included for compatibility with older programs.

FUNCTION —— special form

(FUNCTION **exp**)

FUNCTION is exactly equivalent to

```
  (CLOSURE 'exp (STATE)).
```

It is included for compatibility with older programs.

5.6 Using Funargs

If the expression in the funarg is an invocable object, then the funarg can appear as the value of an operator and evaluation of this funarg invocation results in the application of the invocable object in the environment contained in the funarg. If the expression is not an invocable object, then evaluation of this funarg results in the evaluation of the expression in the environment contained in the funarg.

5.7 Effect of Compilation

A high degree of semantic equivalence exists between the interpreted and compiled versions of an expression. In particular, the rules for variable binding and evaluation are identical in interpreted and compiled programs. There are some incompatibilities, however.

During interpretation, a list whose operator is a macro will be macro-expanded each time it is encountered. During compilation, such a form will be macro-expanded once, and the result will be compiled. In most instances this will not affect the semantics. However, a macro which retains information between invocations and expands differently at different times will not have its effects maintained. In particular, a macro which uses an identifier bound FLUIDly by the program in which it is used as a free variable, will have different values available during interpretation and compilation.

If a macro is redefined, the new definition has no effect on previously compiled programs.

The first phase of the compilation process is to expand all macros found in the program, and to replace the forms by their corresponding expansions. Certain other transformations are performed during this phase. The only one which could have an effect on the meaning of the program is an optimization of function invocation. When the compiler encounters a list of the form, (**id item** ...), it evaluates **id** in the environment of compilation. If the value found is not a special form, built-in function or macro, it is assumed to be a function. If, further, **id** is not bound as a variable in the object being compiled and is not used as a free variable, the run time evaluation of **id** is deferred until after the evaluation of the arguments. The run time evaluation of **id** is short circuited, via **id**'s shallow binding cell. This transformation is innocuous in all but one situation. If, during the evaluation of one of the arguments, an assignment is made to **id** by some other function, then during interpretation (where the operator is evaluated before the arguments) the effect is not seen, whereas during the running of the compiled code, it is. The user may explicitly force interpreter order of evaluation by writing (FUNCALL **id item** ...).

The special form CLOSEDFN is equivalent to QUOTE in interpreted programs. In compiled programs, CLOSEDFN may be used to specify that a nominally QUOTEd function or macro expression is to be compiled into a quoted bpi.

Programs which, in their interpretive forms, contain cyclic structure other than wholly within QUOTEd objects, cannot be compiled at all. The compiler will not detect this condition, and will loop.

Non-cyclic shared substructure in a program may compile correctly, unless it is used by the program for self-modification, in which case the compilation will terminate with no apparent errors but the resulting bpi will not behave in the same way as the original program.

5.7.1 Examples

The evaluation rules allow a great amount of flexibility for the advanced user, but can be used safely and easily by the casual user. We include a few examples to illustrate the rules, and to give the advanced user a feel for how to use the rules, and the naive user some guidelines and some appreciation of how the rules have enabled certain facilities to be built.

A function has arguments, builds up temporary values, and returns a final value. Those temporary values are often stored in local variables. Suppose a function has an argument "X" and the programmer chooses to call the local variable "Y". The function would probably be written:

```
(LAMBDA (X)
    (PROG (Y)
     ...
))
```

PROG is a macro and when it is the operator of an expression, that expression expands to an expression containing a function expression. If we do the expansion in place (as the compiler does) the result looks like:

```
(LAMBDA (X)
   ((LAMBDA (Y)
    ...)
     NIL)
)
```

The identifier Y is initially bound to the value NIL, when the innermost function expression is evaluated. The programmer expects that when the variable X is used in the body of the inner lambda its value is the value of the argument to the entire function. With the rules of evaluation, it is. The inner function expression evaluates to a funarg, and hence no resolution boundary is raised between the use of X and its intended binding.

Suppose the programmer wanted to re-define the "=" operator to mean assignment as it does in a language like Pascal. Evaluating "(SETQ = 'SETQ) will cause that re-definition. After that evaluation, evaluating "(= A 5)" will cause the binding for A to have a value of 5.

The programmer may have written a function P, in the course of which = should mean assignment, and yet want to be able to return to the original meaning of = when the function is finished. This may be accomplished by evaluating the following expression:

```
((LAMBDA ((FLUID =))
   (P ...))
 'SETQ)
```

P will be evaluated on the otherside of a resolution boundary, but since = is bound to SETQ using FLUID, that binding will be found by the identifier resolution rules.

The next example is furnished for the readers who wish to verify their understanding of the rules. Let us assume that FOO has as its value:

```
(LAMBDA () (PRINT X))
```

and BAR has as its value:

```
(LAMBDA ((FLUID X))
   (SETQ X 6)
   ((LAMBDA (X)
      (SETQ X 5)
      (FOO)
      (SETQ BAZ '(LAMBDA () (PRINT X)))
      (PRINT (EQUAL BAZ FOO))
      (BAZ)
      ('(LAMBDA () (PRINT X)))
      ((LAMBDA () (PRINT X)))
      (SETQ GRITCH (LAMBDA () (PRINT X)))
      (GRITCH))))
```

(BAR ()) will print 6, TRUE, 6, 6, 5, and 5. FOO is another function whose invocation raises a resolution boundary. Hence, the binding of X to 5 is not seen while the expression "(FOO)" is being evaluated. The further away binding of X to 6 is seen. The rules were constructed so that equal functions would result in equal values. The value of FOO is equal to the value of BAZ, and hence "(BAZ)" also evaluates to 6. The quoted function expression, hence an unevaluated function expression, in operator position also follows the same rule. The application of an unevaluated function expression does not see bindings which are not fluid.

However, an evaluated function expression (e.g. the two in the example which are not preceeded by the quote mark) returns a funarg, which does not raise a resolution boundary. These follow a different rule. That rule allows the programmer, when writing a function, to use local functions which can see and modify local bindings.

6.0 Functions

A function is a built-in function, a function expression, or an fbpi.

A **built-in function** is one which has an implicit definition for the interpreter and compiler. It can be invoked by evaluating a built-in function invocation.

> Arguments which are passed to a built-in function are not maintained in a stack frame but are kept directly in registers. This means that if an error occurs during evaluation of a built-in function, the use of TRACE will not necessarily find a stack frame.

A **function expression** is a list which contains LAMBDA as its CAR at the time when the interpreter recognizes the operator. A function expression can be created using the LAMBDA operator described below. It can be invoked by evaluating a function invocation. In this case, the value of the operator will be the function expression.

An **fbpi** is a compiled function expression. It can be invoked by evaluating a function invocation. In this case, the value of the operator will be the fbpi.

6.1 Creating Function Expressions

LAMBDA —— special form

(LAMBDA **bv-list** [**exp**] ...)

This form is used to define new functions. The two main parts of a function expression are the bound variable list and the group of expressions known as the *function body*. The above notation uses Backus-Naur Form to describe the bound variable list. The **bv-list** part of a function expression must have one of the forms shown below in Backus-Naur Form.

```
bv  =  ()
       var
       (= var bv)
       (ibv . bv)

ibv =  bv
       (ibv . ibv)
       (= var ibv)
       <ibv ...>

var =  id
       (LEX id)
       (FLUID id)
```

The evaluation of a function expression differs depending on whether the evaluation results from the recognition of the function expression in the operator position of a list or in the operand position.

The evaluation of a function expression in operator position of a list consists of the following steps:

1. The bound variables are created, the arguments are associated with the bound variables by rules which follow, and the arguments are evaluated from left to right, based on the associated bound variables, and become the initial values for the bound variables.

2. The following rules define argument binding.

   Given a function expression (LAMBDA bv . body) being applied to values A1, A2, ..., form the list,

```
(A1 A2 ...)
```

Now, treating this list as a single argument, examine it and the bound variable list, bv, in parallel.

- If bv is neither a pair, a vector or an identifier, discard the argument and terminate.
- If bv is an identifier, bind the argument to that identifier.
- If bv is a list of two elements, of the form (LEX **id**), bind the argument to **id**.
- If bv is a list of two elements, of the form (FLUID **id**), bind the argument to **id**, and note that **id** is declared FLUID in the environment created by the function expression.
- If bv is a list of three elements, of the form (= **id** bv'), bind the argument to the identifier **id**, and repeat the process with bv' and the argument.
- If **id** in the above step is a list of the form (LEX **id1**) or (FLUID **id1**), bind the argument to **id1** and note the FLUID declaration if appropriate.
- If bv is any other pair and the argument is not a pair, signal an error.
- If both bv and the argument are pairs, repeat the process on their respective CARs and CDRs.
- If both bv and the argument are vectors, and if the argument is at least as long as bv, repeat the process for each element of bv and the corresponding element of the argument.

(Note that NIL is not an identifier.)

Note that because of the "constant" rule, excess arguments are simply discarded, not considered an error, while missing arguments are treated as an error.

3. The environment and control chain are created using the rules onpage 44 .
4. The expressions in the function-body are evaluated from left to right.
5. In normal termination, all expressions in the function-body are evaluated and the value of the function expression is the value of the last expression in the function-body.
6. In abnormal termination, all expressions in the function-body are evaluated until a transfer of control expression is evaluated. The value of the function expression is the value defined by the transfer of control construct.

The evaluation of a function expression in operand position of a list consists of creating a funarg consisting of the function expression as the funarg expression and the current state as the state of the funarg state descriptor.

CASE 1:

The environment created allows access to the variables in the environment of the expression. Thus, as in the funarg, free variables will resolve to immediate bindings.

A function expression which is found by evaluating an operator, (an identifier, say), does not have access to the local environment. In order that a variable be accessible from an invoked function it must be declared to be FLUID. This corresponds to SPECIAL in many other LISP systems. The difference is that the declaration of SPECIAL is made outside of any specific function expression, while the declaration of FLUID applies to one specific instance of a variable, in one specific function expression.

In order to declare a variable FLUID, the position usually taken by the variable, say X, in the bound variable list is filled by the list (FLUID X). Suppose that:

```
FOO = (LAMBDA (X) (BAZ))
BAR = (LAMBDA ((FLUID X)) (BAZ))
BAZ = (LAMBDA () X)
```

then the following sequence would ensue.

```
(SETQ X 10)
Value = 10
(FOO 1)
Value = 10
(BAR 1)
Value = 1
```

In the first case, (FOO 1), the free variable, X, in BAZ does not have access to the binding of X in FOO, while in the second case, (BAR 1), it does have access to the FLUID binding in BAR. This accessibility of the FLUID binding is not just limited to the immediately invoked function, but is also effective at deeper levels of calling.

CASE 2:

When the resulting funarg is applied, free variables in the function expression will be resolved, lexically, in the environment of the state-descriptor.

Suppose the value of the identifier FOO is

```
(LAMBDA (X) (LAMBDA (Y) (PLUS X Y)))
```

and the expression (FOO 7) is evaluated.

The value of this expression will be

```
%.FUNARG.((LAMBDA (Y) (PLUS X Y)) . %.SD.xxxxxxxx)
```

where the state descriptor part of the funarg captures the binding of X (from the original function expression) to 7. The resulting funarg will now add 7 to any argument that it is applied to.

```
((LAMBDA U U)                               1 2 3) = (1 2 3)
((LAMBDA (U) U)                             1 2 3) = 1
((LAMBDA (U . V) (LIST U V))                1 2 3) = (1 (2 3))
((LAMBDA (U V . W) (LIST U V W))            1 2 3) = (1 2 (3))
((LAMBDA (U V W) (LIST U V W))              1 2 3) = (1 2 3)
((LAMBDA (U V W . X) (LIST U V W X))        1 2 3) = (1 2 3 ())
((LAMBDA (U V W X) (LIST U V W X))          1 2 3) = Error break
((LAMBDA ((U . V)) (LIST U V))             '(1 2)) = (1 (2))
((LAMBDA ((U V)) (LIST U V))               '(1 2)) = (1 2)
((LAMBDA ((= U (V W)) (LIST U V W))        '(1 2)) = ((1 2) 1 2)
((LAMBDA (= U (V . W)) (LIST U V W))        1 2 3) = ((1 2 3) 1 (2 3))
((LAMBDA (<U V>) (LIST U V))             '<1 2 3>) = (1 2)
((LAMBDA (<U V W>) (LIST U V W))           '<1 2>) = Error break
```

Figure 11. Examples of LAMBDA bv-lists.

6.2 Naming Functions

A function expression is an object, and therefore can be the value of an identifier. The association of a function expression with an identifier is made by the SETQ function.

Because there are so many operators, naming conventions are necessary to allow the programmer to develop expectations of the probable meaning of an operator name. Conventions are not rigidly followed. In addition, it is important to avoid accidental re-definitions that might change the behavior of the system.

A simple rule rule of thumb is as follows. If an identifier is described in this document, then the identifier is a reserved name of LISP/VM. In a LISP/VM environment with no user programs, if the value of an identifier is not that identifier, then the identifier is a reserved name of LISP/VM. Changing the value of reserved identifiers may change the behavior of LISP/VM.

Predicates have names ending in P.

Argument-modifying operators have names beginning with N (for *N*on-copying). Example: NREVERSE as contrasted with REVERSE.

System operators have a comma in their name. These functions must be avoided or used with caution since most of them make extensive assumptions about the environment in which they are called.

Optimized versions of operators have names beginning with Q for QUICK. In all cases it implies the existence of a macro in the compile environment which produces in-line code. In many cases a quick operator avoids type checking. Another group of Q operators are the QS... operators. These (QSPLUS, QSADD1, etc.) assume that their arguments and values are small integers, (numbers between -(2*26) and 2*26-1). In general, they do arithmetic modulo 2*26, forcing correct small integer type codes on their results. They are also aware of each other, and will skip the forcing of type codes on some intermediate results when nested. QE operators are quick operators which relate to EBCDIC representations of characters, QV operators relate to vectors, and QC operators relate to character vectors. Note that there are a few functions whose names start with Q which are neither in line nor unchecked. These include QUOTE and QUOTIENT.

Functions whose names end in Q are usually version of other functions which use EQ rather than EQUAL. Such pairs include MEMBER/MEMQ, ASSOC/ASSQ, UNION/UNIONQ.

Functions whose names begin with MAKE usually create data objects. Examples are MAKE-VEC and MAKE-INSTREAM.

When an abbreviation of a type name appears in a function name, the function either takes that type as an operand type or returns it as the result type.

The prefix H in a function name relates to hashtable functions.

6.3 Using Functions

To cause a function to be invoked, we write expressions of the form:

```
(id [exp] ... )
```

where **id** must evaluate to a function.

The evaluation of a function invocation causes the argument expressions to be evaluated and associated with the bound variables of the function. The current environment and control chain are created and the evaluation continues at the current control point, the first expression in the body of the function.

Two other forms which are used to invoke functions are:

```
(function-expression [exp] ... )
or
('function [exp] ... )
```

In the first case, the function-expression is optimized to prevent a funarg being created. In the second case, the function must be quoted since the evaluation of a function produces a funarg as its value.

6.4 Special Ways of Creating and Using Functions

Application, on the other hand, treats one object as an operator and one or more other objects as operands, computing the value which results from the action of the operator upon the operands.

APPLY —— built-in function

(APPLY **app-ob list [sd]**)

The operator APPLY applies **app-ob** to the elèments of **list** in the environment captured by **sd**. **app-ob** must not be a macro-expression (an macro expression or a mbpi), nor may it evaluate to a macro-expression.

If **app-ob** requires more operands than are contained in **list** an error is indicated.

```
(APPLY PLUS '(1 2 3 4 5))
Value = 15
(SETQ X ( (LAMBDA (Y) (STATE)) 10))
Value = %.SD.xxxxxxxx
(SETQ Y 100)
Value = 100
(APPLY '(LAMBDA (Z) (PLUS Y Z)) '(1))
Value = 101
(APPLY '(LAMBDA (Z) (PLUS Y Z)) '(1) X)
Value = 11
```

Note that the **app-ob** (the function expression) **must** be QUOTEd. If it had not been it would have evaluated to a funarg, which when applied would have supplied its own environment of evaluation, ignoring that supplied by APPLY.

CLOSEDFN —— special form

(CLOSEDFN **item**)

This operator is exactly equivalent to QUOTE during interpretation. It acts as a signal to the compiler to compile its operand. This allows a QUOTEd bpi to be constructed. In compiled code, the value of

```
(CLOSEDFN (LAMBDA (X) (PLUS X 7)))
```

is a bpi, which adds 7 to its argument. This differs from

```
(QUOTE (LAMBDA (X) (PLUS X 7)))
```

which is the specified list structure, and

```
(LAMBDA (X) (PLUS X 7))
```

which results in a compiled function, a bpi, but one which is enclosed in a funarg.

FUNCALL —— built-in function

(FUNCALL **app-ob [item** ...])

The FUNCALL operator applies **app-ob** to its remaining operands, **item** etc.

Interpretively,

(FUNCALL **app-ob item1 item2** ...)

is exactly equivalent to

(**app-ob item1 item2** ...)

In compiled programs this is not true. The compiler transforms expressions with identifiers as operators into CALL expressions, with the operator evaluated after the operands. In most cases these are equivalent, but if an operand assigns a new value to the operator they differ, the original form not "seeing" this side effect, while the compiled form does.

If the initial evaluation of the operator is important the FUNCALL form should be used.

Note that the transformation to CALL expressions is only done for operators which are identifiers that are not bound in the program containing the expression.

```
(FUNCALL LIST '1 '2 '3)
Value = (1 2 3)
```

CALL —— *built-in function*

(CALL [**item** ...] **app-ob**)

The CALL operator applies **app-ob** to its remaining operands, **item** etc.

The CALL operator is used as an intermediate form by the compiler, to reflect the usual function invocation mechanism used by compiled code.

```
(CALL '1 '3 '4 LIST)
Value = (1 2 3)
```

VAPPLY —— *function*

Apply a function and verify the arguments and value.

(VAPPLY pred0 **app-ob** [pred1 **item1**] ...)

The effect of this function is very similar to that of the expression

(**app-ob** **item1** ...)

but before **app-ob** is invoked, each of the predicates pred1, ... is invoked for the corresponding arguments &item1, ... If any predicate returns a value of NIL, an error is signalled. The predicate pred0 is invoked on the value returned by **app-ob** and if the value returned by the predicate is NIL, an error is also signalled.

By using VAPPLY, you can perform selective type checks on the arguments and values of functions without cluttering the body of the function. One typical use of VAPPLY is in the interpreted expression in F*CODE forms emitted by Q-macros. The interpreted expression verifies that the arguments and values are in the right domain and range, while the compiled machine instructions simply perform the operations on the assumption that the constraints are met.

7.0 Macros

A **macro** is a special form, a macro expression, or an mbpi. A **special form** is a macro-like operator, or the expression containing this operator as the expression operator, which is recognized directly by LISP/VM. It can be invoked by evaluating a special form invocation. A **macro expression** is a list which contains MLAMBDA as its CAR at the time when the interpreter recognizes the operator. A macro expression can be created using the MLAMBDA operator described below. It can be invoked by evaluating a macro invocation. In this case, the value of the operator will be the macro expression. An **mbpi** is a compiled macro expression. It can be invoked by evaluating a macro invocation. In this case, the value of the operator will be the mbpi.

A **macro-applicable object** is an invocable object (a macro or a macro-funarg) which is used by the programmer to extend the syntax.

7.1 Compilation of Macros

Compilation involves a partial interpretation of the expression being compiled. In particular, all macro operators must be macro-applied, that is expanded, at the time of compilation. The resulting macro-free form is then translated into machine code.

The **environment of compilation** consists of the operator recognition environment and the macro application environment.

The **operator recognition environment** is an environment used by the compiler which adds and replaces bindings from the non-lambda environment. It is used to selectively redefine operators during compilation, for reasons of efficiency and to achieve macros local to a file. Macros are recognized here and applied in the macro application environment. **Macro application** is the process of passing the argument to a macro and of returning the value of the macro. Macro application takes place during evaluation when the result of macro application is normally evaluated again. It also takes place during the first stage of compilation. **Macro expansion** is the effect of macro application. The **macro application environment** is an environment in which the compiler performs all macro applications. It is used to add temporary definitions or redefine existing definitions of macros and functions invoked during the macro applications which are triggered by the actual compilations. This serves to isolate the compiler from the compiled expression. This environment is normally empty.

7.2 Creating Macro Expressions

MLAMBDA —— special form

(MLAMBDA **bv-list** [exp ...])

Like the function, the macro expression behaves differently when used in operator or operand position.

A macro expression, used as an operator, specifies a macro application. This differs from ordinary application (such as of a function expression) in three ways.

1. No evaluation of operands occurs before the macro application
2. The entire expression, including the (unevaluated) operator, becomes the operand of the macro operator.
3. The value returned by the macro operator is itself evaluated, as if it had occurred in the place of the original expression.

The expression (which becomes the operand) is treated as if it were the conceptual operand list described above, under functions. Thus to bind the entire expression to a variable, one would write

```
( (MLAMBDA X
    (PRETTYPRINT (LIST '*** X '***))
    (CONS 'CAR (CDR X)))
  '(A . B))
(***
  ( (MLAMBDA X
      (PRETTYPRINT (LIST '*** X '***))
      (CONS 'CAR (CDR X)))
    '(A . B))
  ***)
Value = A
```

It is as if an outer pair of parentheses had been supplied. One can use a structured bound variable list, as well.

```
( (MLAMBDA (() . X)
    (PRETTYPRINT (LIST '*** X '***))
    (CONS 'CAR X))
  '(A . B))
(*** ('(A . B)) ***)
Value = A
```

In these example we see the side effects of the macro application, and the value of the resulting expression, but not the value of the macro itself. In both cases that is

```
(CAR '(A . B))
```

but that expression is reevaluated before any value is returned.

As in the function expression, a macro which is written explicitly as an operator has access to the lexical bindings of its environment.

> This is true of an macro expression which is the result of a macro application. Thus, one macro application may create a new expression which contains a macro operator, resulting in a second macro application.

An macro expression which is the immediate value of an operand does not have such lexical access.

If an macro expression is found as a result of repeated evaluations of an operator, an error is signalled. The rational for this action, is that elements of the expression, putative operands, will have been evaluated, making it impossible to recover the original form, the correct operand of the macro expression.

This rule applies to mbpis as well.

If an operand which the compiler assumed would have an applicable value during evaluation has, in fact, a macro applicable operand, the same error is signalled.

LAM —— *macro*

(LAM **bv-list** [**exp** ...])

Unlike most other LISP systems, functions which receive their arguments unevaluated, FEXPRs, are not fundamental to LISP/VM. This capability is realized through the LAM macro. A LAM macro acts like a function expression with the bound variable list augmented with declarative information. In the case of an augmented bound variable list, the LAM macro expands to a macro expression which processes the variables according to the declarations in the bound variable list and then applies a function expression incorporating the body from the original LAM.

The augmented bound variable list of the LAM expression may be defined in Backus-Naur Form as:

```
bv*  =  bv
        (QUOTE bv)
        (bv . bv*)
        ((QUOTE bv) . bv*)

bv  =  ()
       var
       (= var bv)
       (ibv . bv)

ibv =  bv
       (ibv . ibv)
       (= var ibv)
       <ibv ...>

var =  id
       (LEX id)
       (FLUID id)
```

where an instance of (QUOTE bv) specifies an unevaluated argument which is to be bound to the bound variable.

(LAM (QUOTE X) [**exp** ...]*)*	indefinite number of arguments, all QUOTEd
(LAM (X (QUOTE Y)) [**exp** ...]*)*	two arguments, the first evaluated, the second QUOTEd
(LAM ((QUOTE (X . Y))) . [**exp** ...]*)*	one unevaluated argument which is to be decomposed, with its CAR bound to X and its CDR to Y
(LAM (X . (QUOTE Y)) . [**exp** ...]*)*	one evaluated argument, indefinite number of trailing QUOTEd arguments

Note that the last case (as the trailing FLUID binding discussed under LAMBDA) would print as

```
(LAM (X QUOTE Y) . [exp ...])
```

Given A and B with values of 1 2 respectively:

```
((LAM ((QUOTE X) Y) (LIST X Y)) A B)
Value = (A 2)
```

The expansion of this expression would result in the expression:

```
((LAMBDA (X Y) (LIST X Y)) (QUOTE A) B)
```

To avoid having the resultant mbpi contain the entire body of the original LAM as a QUOTEd expression, DEFINE and COMPILE perform an optimization on the macro expression generated by the LAM. The LAM is expanded (by use of the system operator MDEF) and the QUOTEd function expression is extracted from it. A system generated name is generated and inserted in place of the function expression and the system generated name is then given the function expression as its value.

When LAM does not expand to a macro expression, the transformation described is not possible, and the expansion (which ultimately results in a function expression) is assigned to the name (in the case of DEFINE) or compiled.

Note that LAM may be redefined by the user, but it requires careful consideration of the interaction between LAM and DEFINE.

7.3 Naming Macros

The naming conventions for macros follows those for functions.

7.4 Using Macros

To cause a macro to be invoked, a macro invocation is written. This is a list of the form

```
(id [arg] . . . )
```

where **id** evaluates to a macro expression, an mbpi, or a macro funarg.

Evaluation of the macro invocation causes the entire invoking expression to be bound bound to the single bound variable of the macro. The environment and control chain are created for the macro invocation. The body of the macro expression or mbpi is evaluated. The value returned by the body is evaluated again.

MDEF —— *built-in function*

(MDEF **arg item [sd]**)

This operator allows only the macro invocation stage of evaluation to be performed in isolation. The argument **arg** must be a macro expression, an mbpi, or a macro funarg. Otherwise an error is signalled. The second argument is treated as a macro invocation and is bound to the bound variable of **arg** The result of the macro invocation is returned as the value and is not evaluated as would be the case in ordinary macro invocation.

If the **sd** argument is specified, the macro invocation is done in the environment of **sd**.

8.0 Operator Definition and Transformation

An identifier may be established as an operator by simple assignment. For various reasons this is rarely done in practice. Instead one of a group of definition and transformation operators is usually used, COMPILE, ASSEMBLE, DEFINE, or DEFINATE.

These operators interact with a data object, the option list, which is bound to the variable OPTIONLIST. This is an a-list which specifies various types of processing. The option list may be augmented by the user in various ways, as will be noted below.

8.1 Definition and Transformation Operations

COMPILE —— function

(COMPILE **list1** [**list2**])

Where **list1** must be of the form

(**id exp**)

or of the form

((**id1 exp1**) . . .)

For each **id** and **exp**, COMPILE performs a transformation of **exp** and sets **id** to have the transformed expression as its value. The COMPILE operator performs a five-stage transformation of expression. In the first stage, any outermost LAM is expanded and the **id exp** list is transformed into two **id exp** lists, with the second having a name of the form **LAM,id**. In the second stage, the expression operator is evaluated. If the result is a macro invocation, it is invoked. This process continues until the value of the operator is no longer a macro invocation. If the result is then a function or macro expression, no further transformation occurs. Otherwise, the expression is enclosed as the body of a function expression. The third stage macro-expands all macros in the expression. The fourth stage transforms the expression to LAP code. The fifth stage transforms LAP code into object code and produces a bpi and/or an entry in a LISPLIB.

The transformation is affected by the value of the option list.

ASSEMBLE —— function

(ASSEMBLE **list1** [**list2**])

Where **list1** has the form

(**id exp**)

or the form

((**id exp**) . . .)

For each **id** and **exp**, ASSEMBLE performs a transformation of expression and sets name to have the transformed expression as its value. The ASSEMBLE operator performs a single stage transformation of expression. Expression must be LAP code. ASSEMBLE transforms LAP code into object code, and produces a bpi and/or an entry in a LISPLIB.

The transformation is affected by the value of the option list.

DEFINE —— function

(DEFINE **list1** [**list2**])

Where **list1** has the form

(**id exp**)

or the form

((**id exp**) ...)

For each **id** and **exp**, DEFINE performs a transformation of expression and sets name to have the transformed expression as its value. The DEFINE operator performs a two-stage transformation of expression. In the first stage, any outermost LAM is expanded and the **id exp** list is transformed into two **id exp** lists, with the second having a name of the form **LAM,id**. In the second stage, the expression operator is evaluated. If the result is a macro invocation, it is invoked. This process continues until the value of the operator is no longer a macro invocation. If the result is then a function or macro expression, no further transformation occurs. Otherwise, the expression is enclosed as the body of a function expression.

The transformation is affected by the value of the option list, primarily the FILE and NOLINK key words.

DEFINATE —— function -- FOR THE EXPERT LISP/VM USER

(DEFINATE **list1 id1 id2 list2**)

Where **list1** has the form

(**id exp**)

or the form

((**id exp**) ...)

The values of the second and third operands are key words, indicating the form of the first operand and the desired processing, respectively. These are the specification and the action.

The fourth operand is a list of key-value pairs which is to be temporarily added to the head of the option list.

There are five specifications and six actions, and they are interrelated. For example, COMPILATION specifies that the input is in the same form that would have been produced by the action COMPILE. The actions are ordered, each requiring the result of the previous one. DEFINATE sequentially performs them in order, starting with the action which will accept the first operand as specified, and continuing until the requested action has been reached.

The final disposition of the value is controlled by the option list.

The specification describes the **exp** parts of the first operand.

EXPRESS	*EXPRESSION*	An EXPRESSION is any LISP expression. As an action, EXPRESS is the identity operator, no processing is done.
DEFINE	*DEFINITION*	The action, DEFINE, examines an EXPRESSION for the explicit operator LAM. If it finds one it performs the macro expansion of the LAM into a macro expression and transforms that into two **id exp** lists, with the second having a name of the form **LAM,id**. This prevents the body of the LAM expression from being included at each instance of its use. It also causes

		the body of the LAM expression to be not lexically present, and thus to behave in the same way as function expressions with regard to variable evaluation.
REALIZE	*REALIZATION*	The action, REALIZE, evaluates the operator of the DEFINITION and if the result is a macro invocation, it is invoked. This process continues until the value of the operator is no longer a macro invocation. If the result is then a function or macro expression, no further action occurs. If the result is any other expression, it is enclosed as the body of a function expression.
REDUCE	*REDUCTION*	The action, REDUCE, replaces all subexpressions in the REALIZATION by their macro expansions. This is done top down. As in REALIZE, the value of the operators in the compilation environment determines whether a macro expansion is done.
COMPILE	*COMPILATION*	The action, COMPILE, transforms the REDUCTION to LAP code. LAP is a high level assembler, accepting S/370 machine instructions (in parenthesized form) and various pseudo-operations. It is not interpretable.
ASSEMBLE		The action, ASSEMBLE, transforms LAP code into object code, and produces either a bpi, or an entry in a LISPLIB (or both). There is no specification, ASSEMBLY, as there is no READable representation for the result of this action.

8.1.1 EXF

Many LISP systems have a variety of operators which deal with text files of expressions. They may have separate LOAD and COMPILE command, for example. In LISP/VM the exists a single such function, EXF, which simply reads successive expressions from a file and evaluates them.

In order to achieve the variation in control provided by multiple file reading operators in other systems, LISP/VM contains a collection of functions, described in the chapter on Operator Definition, which adjust their behavior according to the value of the variable OPTIONLIST, a property list.

Thus, to cause the compilation of a factorial function, a file might contain the following sequence of lines.

```
(COMPILE '(
(FACTORIAL
  (LAMBDA (N)
    (COND
      ( (EQ N 0) 1 )
      ( 'T (TIMES N (FACTORIAL (SUB1 N))) )))) ))
```

To define a function the following form would be used.

```
  (DEFINE '(
  (NREVERSE
    (LAMBDA (X)
      (PROG (U V)
  TOP   (COND ((NOT (PAIRP X)) (RETURN U)))
        (SETQ V (CDR X))
        (SETQ U (RPLACD X U))
        (SETQ X V)
        (GO TOP)))) ))
```

The actual action taken depends on two entries in OPTIONLIST. If there is a FILE property, the value of which is a library, then a bpi (if the operator was COMPILE), or the given function expression (in the case of DEFINE), will be placed in the corresponding library, with keys of "FACTORIAL" and "NREVERSE". If there is a NOLINK property with a value of non-NIL, then no bpi will be created in the currently running LISP/VM system. The bpi which is placed in the library is fast-loadable in that it does not have to be parsed. One can thus produce any combination of fast-loadable and currently loaded versions of the function.

The operators FILEQ and FILEACTQ may be used to place expressions in the library, with flags indicating whether they are to be assigned or evaluated at load time.

In addition, there exist operators such as ADDTEMPDEFS and TEMPDEFINE, which make predefined macro and function definitions (possibly compiled) available to the compiler and to compile-time macros. These definitions persist only for the duration of the processing of the current file.

EXF —— macro

(EXF **arg1** [**arg2** [FILE **filespec**] [**id item**] ...])

The EXF operator receives its argument unevaluated. It rebinds OPTIONLIST, CURREADTABLE, OUTREADTABLE, and CURRENT-SYNTAX to have fluid binding, with their previous values. This allows the values of these variables to be changed within the EXF without affecting evaluations following its completion. Side-effects on these values will affect the invoking environment, unless the values are copied.

EXF analyzes its arguments, uses them to augment OPTIONLIST, and then invokes the system's top-level READ-EVAL-PRINT loop, with appropriate input and output streams.

The first argument, **arg1**, is required. It may be NIL, an identifier or a list of the form (**id1** [**id2** [**id3**]]), where the **ids** are CMS filename, filetype and filemode respectively. If **arg1** is NIL, the READ-EVAL-PRINT loop is given the current value of CURINSTREAM as its input stream. If **arg1** is the identifier *, EXF reads from the console. If **arg1** is a list, it is used as a file spec. to create an input stream to read from disk. If it is an identifier, it is assumed to be a disk file name. The variable EXF-FILETYPES is evaluated to obtain a list of default CMS filetypes. The initial value of this variable is the list (LISP). The user-supplied filename is combined with these until a file is found and that file is used for input. If no such file is found, EXF terminates and returns a value of NIL.

The resolved filetype determines the reader syntax, files of type LISP use standard syntax, while files of type LISP370 use LISP370 syntax.

The second argument, if present, specifies the output stream for the READ-EVAL-PRINT loop. A missing argument, or one of NIL, causes the current value of CUROUTSTREAM to be used. If **arg2** is the identifier *, output is directed to the console. If **arg2** is the identifier &, any output is discarded. If **arg2** is the identifier =, output is directed to a file with filetype EXF and filename and mode the same as the input file. Any other identifier is taken as a filename, with filetype EXF and filemode A. A list of identifiers is assumed to specify filename, filetype and filemode.

If a FILE property is to be given, it must immediately follow **arg2**. When the third argument is the identifier FILE, the fourth argument, **filespec**, specifies a library which is to receive fast-loadable objects. As with **arg2**, this may be an =, specifying a library file of filetype LISPLIB with a filename the same as the input file. An identifier is interpreted as the filename for a library of filetype LISPLIB, while a list of identifiers is interpreted as the filename, filetype and filemode.

In the cases of output streams and LISPLIBs, an existing file of the same name, type and mode will be replaced by the new file.

The remaining arguments are paired and added to the current (by EXF) binding of OPTIONLIST. Note that no evaluation takes place during this binding, only constant values may be placed in OPTIONLIST as property values.

Two such properties interact with EXF. If a WIDTH property is currently in OPTIONLIST, or is added to it by the EXF argument list, its value is used to specify the buffer length for the output stream. If no such property is defined, the default value is 80 characters. If a FILE property is given, then a property of NOLINK with a value of non-NIL is added to OPTIONLIST before any of the remaining properties in the EXF argument list are added.

The effect of this is to suppress the creation of bpis in the current running system if a fast-loadable file is being created. To override this default, the call to EXF must contain the arguments NOLINK NIL following the FILE **filearg** arguments.

This function rebinds CURREADTABLE and OUTREADTABLE to their current values. As a result, any permanent side effects to these tables are visible in the environment that invoked EXF. If it is desirable to avoid these side effects, the expression (COPY-IOTABLES) should appear in the file.

EXF-FILETYPES —— *variable*

The value of this variable is a list of identifiers that specify a sequence of default CMS filetypes. This list is used by the EXF operator and the Lispedit READ and IMPORT commands to find a file specified by name only. A typical use of this variable is to add additional filetypes to it in the profile file evaluated every time LISP/VM is started up.

8.2 The Option List

OPTIONLIST —— *variable*

An a-list, holding named values which are examined by various functions in the definition facility.

The values of various properties on this augmented list control various aspects of definition, compilation and assembly. GET is used to search OPTIONLIST, thus NIL is the default value for any property not explicitly present.

Various processes effect the option list. Every definition operator may temporarily add its own options. EXF adds options which are removed when it finishes. Other operators change the current value.

Both EXF and DEFINATE re-bind OPTIONLIST, and the standard operators modify it in such a way that their effects are lost upon exit from these functions.

8.3 Properties of the Option List

BPILIST —— *key word*

(BPILIST . **boolean**)

A non-NIL value for BPILIST causes an assembly listing to be produced. This listing contains the hexadecimal System/370 machine code produced by the assembler, together with a variable amount of symbolic LAP code.

If BPILIST is non-numeric or if it is greater than 3, a full listing is produced. This includes all instructions generated by the assembler, all comments and all source instructions.

A value of 3 causes intermediate instructions to be dropped from the listing. That is, instructions generated by the assembler, which in turn resulted in the generation of further instructions rather than in object code.

A value of 2 causes comments to be dropped from the listing.

A value of 1 causes only source instructions to be printed symbolically, although the object code (hexadecimal) is printed in full.

FILE —— *key word*

(FILE . **rstrm**)

The value of FILE should be NIL or a stream. If FILE is non-NIL a loadable bpi-image will be written onto it. If FILE is NIL no action is taken.

By use of the FILE and NOLINK options programs can be compiled and/or assembled for future loading without their being defined in the running system. (See LOADVOL function.) The resulting file, when loaded, causes the same assignments to take place as would have resulted if the NOLINK option were NIL.

INITSYMTAB —— *key word*

(INITSYMTAB . **a-list**)

The value of this property should be an a-list (or NIL, which is an empty a-list) which will be searched by the assembler (LAP) for operation code values, symbolic register names, symbolic immediate operand names and symbolic literals. The values in INITSYMTAB override the build-in values of the assembler, and are overridden in turn by symbols established by EQU statements in the LAP code.

LAPLIST —— *key word*

(LAPLIST . **boolean**)

A non-NIL value for LAPLIST causes the assembly code produced by the compiler to be PRETTYPRINTed. This property does not control the printing of LAP source code, which is under the control of the SOURCELIST property. The default value is NIL.

LISTING —— *key word*

(LISTING . **stream**)

The value of LISTING should be NIL or a fast stream. If LISTING is a stream the output produced as a result of the preceding four options (SOURCELIST, TRANSLIST, LAPLIST and BPILIST) will be written onto that stream. If LISTING is NIL it defaults to the value of the fluid CUROUTSTREAM.

MACRO-APP-SD —— *key word*

(MACRO-APP-SD . **sd**)

The environment in which macros are to be expanded by the compiler. The use of MACRO-APP-SD protects from conflict between variables bound by the compiler and the bindings of macros used in expressions being compiled. If the value is NIL, the initial state (where only nil-environment bindings are present) is used.

MESSAGE —— *key word*

(MESSAGE . **stream**)

The value of MESSAGE should be NIL or a fast stream. All error and warning messages from the definition functions are written onto the MESSAGE stream. If the value of MESSAGE is NIL it defaults to CUROUTSTREAM.

NOLINK —— *key word*

(NOLINK . **boolean**)

If the NOLINK property has a non-NIL value, the result of an assembly is not made into a bpi. The default is to create a bpi and assign it to the name with which it was paired in the specification list.

NONINTERRUPTIBLE —— *key word*

(NONINTERRUPTIBLE . **boolean**)

If the NONINTERRUPTIBLE property is non-NIL, no polling for interrupts is inserted in the bpi by the assembler. Explicit POLL statements will be assembled. The default value is NIL, i. e. the bpi is interruptible.

OPTIMIZE —— *key word*

(OPTIMIZE . **sint**)

The value of the OPTIMIZE property controls the amount of code modification during pass 2 of the compiler. The default value is 4, the highest level of optimization. Level 0 lets the code emitted by the various generators stand as-is. The levels from 1 to 3 perform various operations, such as dead code elimination, code motion to eliminate branches, common code merging, etc.

NOMERGE —— *key word*

(NOMERGE . **boolean**)

A non-NIL value of the NOMERGE property suppresses the compiler's contour merging. This is an optimization in which the variables of operator LAMBDAs are promoted, that is made into non-argument variables of enclosing LAMBDAs. The resulting bpi will create fewer stack frames than the the interpretive evaluation of the original form.

For debugging purposes it may be desirable to compile a function with this merging suppressed.

OP-RECOGNITION-SD —— *key word*

(OP-RECOGNITION-SD . **sd**)

The environment in which the operators are evaluated by the compiler. The values found in this environment are used to distinguish functions, macros and special forms. Various operators exist to augment this environment, see -- Heading id 'compenv' unknown --.

QUIET —— *key word*

(QUIET . **boolean**)

If the QUIET property has a non-NIL value warning and informational messages to the console are suppressed.

NOSTOP —— *key word*

(NOSTOP . **boolean**)

If the NOSTOP property has a non-NIL value errors found during macro will not signal an error, but will print their messages on the MESSAGE and LISTING streams.

SOURCELIST —— key word

(SOURCELIST . **boolean**)

If SOURCELIST has a non-NIL value the source program (either LISP or LAP) is PRETTYPRINTed by the definition functions. The default is NIL. When running with SOURCELIST non-NIL it should be remembered that the printing by the supervisor can be controlled by the settings of the fluid variables ,ECHOSW and/or ,VALUSW.

TRANSLIST —— key word

(TRANSLIST . **boolean**)

A non-NIL value for TRANSLIST causes the output of pass one of the compiler to be PRETTYPRINTed. This is the "transformed" LISP/VM, with all macros expanded and with various other changes, which will be made into a bpi by pass two of the compiler and the assembler. A number of forms internal to the compiler appear in this listing. If definitions of these internal forms existed (unfortunately impossible in some cases) interpretation of this transformed LISP/VM would duplicate the behavior of the bpi which results from the full compilation/assembly process. The default value is NIL.

8.4 Operations on the Option List

ADDOPTIONS —— function

(ADDOPTIONS [**id item**] ...)

This operator constructs an a-list from its operands, using them alternately as keys and values. The result is appended onto the front of the current value of OPTIONLIST.

If the number of operands is odd, an error is signalled.

MAADDTEMPDEFS —— function

(MAADDTEMPDEFS **filearg**)

filearg must designate a LISPLIB. The objects read from **filearg** are added to the MACRO-APP-SD, as the values of their keys.

This allows a set of functions need by compiler macros to be available, without their interfering with the standard system functions.

MATEMPDEFINE —— function

(MATEMPDEFINE **list1** [**list2**])

Where the operands are as for DEFINE.

MATEMPDEFINE invokes DEFINATE with a requested action of REALIZE.

The results are added to MACRO-APP-SD. This allows the temporary definition of functions needed by macros, without their interfering with the normal system definitions. These will persist only as long as the current binding of OPTIONLIST.

ORADDTEMPDEFS —— function

(ORADDTEMPDEFS **filearg**)

filearg must designate a LISPLIB. The objects read from **filearg** are added to the OP-RECOGNITION-SD, as the values of their keys.

This allows a set of macro definitions to be installed during the processing of a set of compilations, by EXF for example, without their becoming a permanent part of the system.

ORTEMPDEFINE —— function

(ORTEMPDEFINE **list1** [**list2**])

Where the operands are as for DEFINE.

ORTEMPDEFINE invokes DEFINATE with a requested action of REALIZE.

The results are added to OP-RECOGNITION-SD. This allows temporary macro definitions, which will persist only as long as the current binding of OPTIONLIST, without interfering with the standard system definitions.

9.0 Predicates

A predicate is a function whose result is either NIL or non-NIL and which is usually used for testing. A built-in predicate name usually ends in P. Whenever possible, the non-NIL value returned is one that could be of use.

9.1 General

NONSTOREDP —— function and compile-time macro

(NONSTOREDP **item**)

This operator returns **item** if **item** is not a stored object. Stored objects are possibly heap resident and may be moved by the garbage collector. Small integers, gensyms, and bpis are examples of non-stored objects.

9.2 NIL and Truth Value

NULL —— built-in function

(NULL **item**)

This function returns the value non-NIL if **item** is NIL; otherwise, it returns the value NIL.

NOT —— built-in function

(NOT **item**)

This operator is exactly equivalent to NULL. It is useful for readability when the predicate in a COND is to be inverted.

9.3 Pairs and Lists

ATOM —— built-in function

(ATOM **item**)

This function returns the value NIL if **item** is a pair; otherwise, it returns the value non-NIL. In the list interpretation of data objects, ATOM is the general test for an empty list.

PAIRP —— built-in function

(PAIRP **item**)

This function returns **item** if **item** is a pair; otherwise, it returns the value NIL. In the list interpretation of pairs, PAIRP is a test for a non-empty list.

9.4 Vectors and Bpis

VECP —— built-in function

(VECP **item**)

This function returns the value **item** if **item** is a vector capable of holding arbitrary objects; otherwise, it returns the value NIL.

IVECP —— *built-in function*

(IVECP **item**)

This function returns the value of **item** if **item** is a vector of 32 bit integers; otherwise, it returns the value NIL.

RVECP —— *built-in function*

(RVECP **item**)

Returns **item** if it is a vector of 64 bit floating point numbers, else returns NIL.

CVECP —— *built-in function*

(CVECP **item**)

This function returns the value **item** if **item** is a character vector (that is, a vector of characters); otherwise, it returns the value NIL.

BVECP —— *built-in function*

(BVECP **item**)

This function returns the value **item** if **item** is a bit vector; otherwise, it returns the value NIL.

FBPIP —— *built-in function*

(FBPIP **item**)

This function returns the value **item** if **item** is a compiled function; otherwise, it returns the value NIL.

MBPIP —— *built-in function*

(MBPIP **item**)

This function returns the value **item** if **item** is a compiled macro expression; otherwise, it returns the value NIL.

BPIP —— *function*

(BPIP **item**)

This operator returns the value **item** if **item** is compiled function expression or a compiled macro expression; otherwise, it returns the value NIL.

9.5 Identifiers

IDENTP —— *built-in function*

(IDENTP **item**)

This function returns the value **item** if **item** is an identifier; otherwise, it returns the value NIL.

IDENTP will return a non-NIL value for special forms, built-in functions, identifiers and gensyms. NIL is not an identifier, and (IDENTP "NIL) = NIL.

BFP —— *built-in function*

(BFP **item**)

This function returns the value **item** if **item** is an built-in function; otherwise it returns the value NIL.

SFP —— *built-in function*

(SFP **item**)

This function returns the value **item** if **item** is a special form; otherwise it returns the value NIL.

GENSYMP —— *built-in function*

(GENSYMP **item**)

This function returns the value **item** if **item** is a gensym; otherwise, it returns the value NIL.

SIMPLEIDP —— *built-in function*

(SIMPLEIDP **item**)

This function returns the value **item** if **item** is an identifier that is not a special form, built-in function, or gensym. Otherwise the value is NIL.

CHARP —— *function and compile-time macro*

(CHARP **item**)

This operator tests whether **item** is one of the 256 identifiers which span the total range of possible single character print names. CHARP returns NIL if **item** is not one of these objects; otherwise, it returns the value **item**.

DIGITP —— *function and compile-time macro*

(DIGITP **item**)

This operator returns the value **item** if **item** is one of the 10 identifiers which span the total range of possible single digit print names (0, 1, 2, 3, 4, 5 ,6 ,7, 8, 9); otherwise, it returns the value NIL.

9.6 Place Holders

PLACEP —— *function and compile-time macro*

(PLACEP **item**)

This operator returns the value **item** if **item** is %.EOF; otherwise, it returns the value NIL.

%.EOF is the value of (READ x) when x is an empty stream, e.g. a stream at end-of-file. This is the only condition under which READ will return this value since it is an object without a readable external form.

9.7 Numbers

NUMP —— *built-in function*

(NUMP **item**)

This operator returns the value **item** if **item** is any type of number; otherwise, it returns the value NIL.

SINTP —— *built-in function*

(SINTP **item**)

This operator returns the value **item** if **item** is a small integer; otherwise, it returns the value NIL.

BINTP —— *built-in function*

(BINTP **item**)

This operator returns the value **item** if **item** is a large integer; otherwise, it returns the value NIL.

INTP —— *built-in function*

(INTP **item**)

This operator returns the value **item** if **item** is an integer number; otherwise, it returns the value NIL.

RNUMP —— *built-in function*

(RNUMP **item**)

This operator returns the value **item** if **item** is a real number; otherwise, it returns the value NIL.

9.8 Funargs

FUNARGP —— *built-in function*

(FUNARGP **item**)

This function returns the value **item** if **item** is a funarg; otherwise, it returns the value NIL.

9.9 State Descriptors

STATEP —— *built-in function*

(STATEP **item**)

This function returns the value of **item** if **item** is a state descriptor; otherwise, it returns the value NIL.

9.10 Hashtables

HASHTABLEP —— *built-in function*

(HASHTABLEP **item**)

This function returns the value of **item** if **item** is a hashtable; otherwise, it returns the value NIL.

9.11 Readtables

READTABLEP —— *built-in function*

(READTABLEP **item**)

This function returns the value of **item** if **item** is a readtable; otherwise, it returns the value NIL.

9.12 Streams

STREAMP —— *function*

(STREAMP **item**)

This function returns the value **item** if **item** could be a stream (i.e., if **item** is a pair); otherwise, it returns the value NIL. STREAMP is exactly equivalent to PAIRP.

FASTSTREAMP —— function

(FASTSTREAMP **item**)

This function returns the value non-NIL if **item** is a fast stream (i.e., a pair whose CDR is a vector of length at least 3); otherwise, it returns the value NIL.

Note: this test is completely heuristic so that it is possible for an arbitrary object to be recognized as a fast stream.

IS-CONSOLE —— function

(IS-CONSOLE **item**)

This function returns the value NIL if **item** is not a fast console stream; otherwise, it returns the value **item**.

9.13 Other Predicates

EQ —— built-in function

(EQ **item1 item2**)

EQ tests for pointer identity between its two arguments. Its value is the identifier non-NIL if **item1** and **item2** are identical pointers. This means that the pointer type codes as well as the pointer address fields are identical. If these fields are not identical, the value of EQ is NIL.

> EQ and EQUAL are equivalent for identifiers created by the Reader or by the INTERN operator. The CVEC2ID operator can create two identifiers that have the same external form but are neither EQ nor EQUAL. EQ and EQUAL are equivalent for all small integers. It is not possible to create two distinct small integers having the same value.

EQ may be used for quick tests of equivalence. If two expressions are EQ, then they are necessarily EQUAL; however, the converse is not true. Two expressions which are not EQ may nevertheless be EQUAL.

Note that two copies of a given object will not be EQ because they are stored at different locations. Similarly, it is possible to have different representations of the same numeric value which are EQUAL, but which are not EQ.

EQUAL —— function

(EQUAL **item1 item2**)

This is a generalized equality testing function applicable to any data objects, including circular objects and numeric quantities.

For numeric quantities to be EQUAL, they must represent the same value. For tests involving one or two real (floating point) numbers, a fuzz factor may be relevant. If an integer is to be compared with a real number, the integer is converted to a real value for the comparison.

Two vectors are EQUAL if they are of the same type, the same length, and their absolute parts are identical and their pointer parts are EQUAL.

For composite arguments, EQUAL implements access-equivalent equality testing. This means that two objects are EQUAL if every part of one object which can be reached by a composition of accessing functions is EQUAL to the corresponding part of the the other object reached through the same composition of accessing functions. Intuitively, two objects are EQUAL if they denote the same (possibly infinite) tree.

The value of EQUAL is either NIL or non-NIL.

UEQUAL —— *function*

(UEQUAL **item1 item2**)

This is a generalized update-equality testing function applicable to any data object in the same sense as EQUAL. It differs from EQUAL in that for two objects to be UEQUAL, not only must corresponding parts of the objects be EQUAL through the access functions, but there must be the same number of unique parts and if any of these parts were to be updated in one object and the same update operation performed on the corresponding part of the other, then the objects would still be EQUAL.

In addition, numeric values are considered UEQUAL only if they are of the same type and numerically equivalent. Bit and character vectors are UEQUAL only if they have the same capacity as well as the same type, length, and contents.

Intuitively, two objects are UEQUAL if and only if they denote equivalent rooted directed graphs, i.e. if they denote EQUAL objects which also have the same circular and non-circular sharing object.

The value of UEQUAL is either NIL or non-NIL.

UGEQUAL —— *function*

(UGEQUAL **item1 item2**)

Like UGEQUAL, this function compares two objects for structural equivalence. In addition, it keeps a table of the gensyms in each object. If **item1** and **item2** have the same structure and the same pattern of gensym occurrences, this function return a non-NIL value, otherwise the value is NIL.

Since two successive invocations of the READ function will create different gensyms in the internal form of a single external form, both EQUAL and UEQUAL will return NIL when comparing such objects. UGEQUAL can be used to test if two objects have equivalent external forms.

10.0 Control Structures

10.1 Specification of Values

Many of the data objects are constants, that is they evaluate to themselves. This is true of all numbers, strings and vectors, for example. Lists (i.e. pairs) and identifiers, however, generally do not have this property.

If, in an expression, you wish an identifier or a pair itself, rather than its value, you must QUOTE it. It is also possible (in compiled programs) to specify a constant as the value of an expression (evaluated during the compilation process). This is done by using the CONSTANT operator.

QUOTE —— special form

(QUOTE **item**)

The value of a QUOTE expression is just the operand, **item**. This allows one to mention an identifier, or a list without having it evaluated as an expression. The READER accepts the form '**item** as equivalent to (QUOTE **item**).

```
(QUOTE X)
Value = X
'ABC
VALUE = ABC
```

CONSTANT —— macro

(CONSTANT **exp**)

The CONSTANT operator evaluates **exp** and returns its value.

During compilation, CONSTANT is defined as a macro, which replaces itself by the QUOTEd value of **exp**. This allows the use of the values of expressions (as evaluated at compile time) as constants.

The bpi resulting from the compilation of

```
(LAMBDA () (CONSTANT (EXP 3)))
```

will return e^3, without having to compute it each time.

10.2 Assignment

SETQ —— macro

(SETQ **id exp**)

The assignment operator. The value of **exp** is assigned to **id**. While **exp** is evaluated, **id** is not, and must be an identifier, otherwise an error is signalled.

```
COUNT
Value = COUNT
(SETQ COUNT '100)
Value = 100
COUNT
Value = 100
```

RESETQ —— macro

(RESETQ **id** [**item**])

Assigns the value of **item** to **id**, as in SETQ. The value of this expression is the value which **id** had *before* the assignment.

item defaults to NIL.

```
(SETQ X '(1 2 3))
Value = (1 2 3)
(RESETQ X (CDR X))
Value = (1 2 3)
X
Value = (2 3)
```

SET —— built-in function

(SET **id exp**)

The other assignment operator. The value of **exp** is assigned to **id**. Unlike SETQ, both arguments are evaluated. The value of **id** must be an identifier, otherwise an error is signalled.

```
COUNT
Value = COUNT
(SETQ X 'COUNT)
Value = COUNT
(SET X '100)
Value = 100
COUNT
Value = 100
X
Value = COUNT
```

SET-ID —— function

(SET-ID **id item**)

This operator is equivalent to SET, except that it affects only FLUID or non-lambda bindings.

SET-LEX-ID —— function

(SET-LEX-ID **id item**)

Semantically equivalent to SET but faster.

SET-GLOBAL-ID —— function

(SET-GLOBAL-ID **id item**)

This operator performs an assignment in the current non-lambda environment. It does not effect any lambda bindings. If no binding for **id** is found, one is added to the head of the current non-lambda environment.

CSET-ID —— function

(CSET-ID **id item**)

This operator is the assignment operator corresponding to CEVAL-ID. It searches the control chain for a FLUID binding of **id**, and if it finds one updates it with **item**. If no binding is found in the control chain, no updating is performed and the value of the CSET-ID expression is **id**. No search of the non-lambda environment is made.

CSET-LEX-ID —— function

(CSET-LEX-ID **id item**)

The assignment operator corresponding to CEVAL-LEX-ID. Differs from CSET-ID in that it has access to those LEXical bindings which are visible in the application context.

10.3 Expression Grouping

There are three types of expression groupings: PROGN, PROG1, PROG2, which only group expressions; SEQ, which groups expressions and acts as a scope for labels; and PROG, which groups expressions and acts as a scope for bound variables and labels. Since PROG1 and PROG2 are not as commonly used, they are described at the end of this section.

PROGN —— *special form*

(PROGN [**exp1** ...])

This form groups expressions. The group of expressions is known as the ***progn-body.***

Evaluation of the PROGN expression consists of the following steps:

1. The expressions in the progn-body are evaluated from left to right.
2. In normal termination, all expressions in the progn-body are evaluated and the value of the PROGN expression is the value of the last expression in the progn-body.
3. In abnormal termination, all expressions in the progn-body are evaluated until a transfer of control expression is evaluated. The value of the PROGN expression is the value defined by the transfer of control construct.

```
(PROGN (SETQ X '(A B)) X)
Value = (A B)
X
Value = (A B)
(PROGN (SETQ X '10) (SETQ Y '20) (PLUS X Y))
Value = 30
X
Value = 10
```

SEQ —— *special form*

(SEQ [**exp** ...])

This form groups expressions and acts as a scope for labels. The group of expressions is known as a *seq-body.* Any of these expressions which are identifiers are known as *labels.*

Evaluation of the SEQ expression consists of the following steps:

1. The expressions in the seq-body, with the exception of labels, are evaluated from left to right. Labels, though not evaluated, are considered to have the value NIL.
2. In normal termination, the value of the SEQ expression is the value of the last expression in the seq-body.
3. In abnormal termination, all expressions in the seq-body are evaluated until a transfer of control expression is evaluated. The value of the SEQ expression is the value defined by the transfer of control construct.

A SEQ expression is a label-scope for labels.

Labels are used as the target of GO expressions.

PROG —— *macro*

```
(PROG
   (local-variable-list)
   {expr...} )
local-variable-list
   [{local-variable | (local-variable expr)}...]
local-variable
   id | (LEX id) | (FLUID id)
```

This form groups expressions and acts as a scope for local variables and labels. The group of expressions is known as a *prog-body.* Any of these expressions which are identifiers are known as *labels.* The &expr. in the local-variable-list is known as the *initialization expression.* A local-variable for which a binding class is not specified is given a default class of LEX.

Evaluation of the PROG expression consists of the following steps:

1. The local-variables are created, the initialization expressions are all evaluated from left to right in the environment which was in effect before the PROG expression was evaluated, and the associated variables are set to these values or to NIL if no initialization expression was specified.
2. The expressions in the prog-body, with the exception of labels, are evaluated from left to right. Labels, though not evaluated, have the value NIL.
3. In normal termination, the value of the PROG expression is NIL.
4. In abnormal termination, all expressions in the prog-body are evaluated until a transfer of control expression is evaluated. The value of the PROG expression is the value defined by the transfer of control construct.

A PROG expression is a contour, that is, a scope for bound variables. A prog-body is a label-scope for labels.

The form (LEX **id**) is only required for the identifiers FLUID or LEX.

Because the initialization expressions are evaluated in the environment which was in effect before the PROG expression was evaluated, evaluation of a RETURN expression in an initialization expression causes that previous environment to be terminated. Evaluation of a RETURN expression in any other context within the PROG expression causes the evaluation of the PROG expression to be terminated and the value of the PROG expression is the value of the RETURN expression. The following example shows the importance of evaluating initialization in this environment.

```
(SETQ X '(2 1))
Value = (2 1)
(PROG ( (X (CONS 3 X) ) (PRINT X) (RETURN (CONS 4 X)) )
(3 2 1)
Value = (4 3 2 1)
X
Value = (2 1)
```

A value of a variable outside the PROG expression is used, manipulated, but may be unaffected after the PROG expression completes. Of course, if an updating operation had been applied within the PROG expression, their effects could have been visible afterwards.

Evaluation of a PROG expression results in the creation of an implicit lambda expression, with the local variables becoming the bound variables of the lambda expression and the initial values becoming the arguments.

10.3.1 *Other Expression Groupings*

PROG1 —— function and compile-time macro

(PROG1 **exp** ...)

This form groups expressions. The group of expressions is known as the prog1-body.

Evaluation of the PROG1 expression consists of the following steps:

1. The expressions in the prog1-body are evaluated from left to right.
2. In normal termination, all expressions in the prog1-body are evaluated and the value of the PROG1 expression is the value of the first expression in the prog1-body.
3. In abnormal termination, all expressions in the prog1-body are evaluated until a transfer of control expression is evaluated. The value of the PROG1 expression is the value defined by the transfer of control construct.

   The PROG1 expression is most commonly used to evaluate an expression with side effects and returñ a value that must be evaluated before the side effects happen.

PROG2 —— function and compile-time macro

(PROG2 **exp exp** ...)

This form groups expressions. The group of expressions is known as the prog2-body.

Evaluation of the PROG2 expression consists of the following steps:

1. The expressions in the prog2-body are evaluated from left to right.
2. In normal termination, all expressions in the prog2-body are evaluated and the value of the PROG2 expression is the value of the second expression in the prog2-body.
3. In abnormal termination, all expressions in the prog2-body are evaluated until a transfer of control expression is evaluated. The value of the PROG2 expression is the value defined by the transfer of control construct.

   PROG2 is used to allow a side effect to occur before and after the evaluation of an expression whose value is returned.

10.4 Iteration

DO —— macro

```
(DO
    (local-variable-list)
    (end-test {expr...})
    {expr...} )
local-variable-list
   {local-variable | (local-variable expr)
         | (local-variable expr expr)}...
local-variable
   id | (LEX id) | (FLUID id)
```

This form is an iteration expression which provides top-of-loop testing with an "UNTIL-like" test. It provides expression grouping and a scope for both variables and labels. local-variable-list, a list consisting of a *test expression* and a group of expressions known as an *end-body*, and a group of expressions known as a *do-body*. The first expression of the local-variable-list is known as the *initialization expression* and the second expression is known as the *step expression*. A local-variable for which a binding class is not specified is given a default class of LEX.

Evaluation of the DO expression consists of the following steps:

1. The local-variables are created, the initialization expressions are all evaluated from left to right in the environment which was in effect before the DO expression was evaluated, and the associated variables are set to these values or to NIL if no initialization expression was specified.
2. The end-test test expression is evaluated. If the value is non-NIL, normal termination occurs. If the value is NIL, the expressions in the do-body are evaluated from left to right, all step expressions are evaluated and the associated local-variables modified by the step values. This set of actions is repeated until normal or abnormal termination occurs.
3. Normal termination consists of evaluating the end-body from left to right. The value of the DO is the value of the last expression.
4. In abnormal termination, all expressions in the do-body are evaluated until a transfer of control expression is evaluated. The value of the DO expression is the value defined by the transfer of control construct.

A DO expression is a contour, that is, a scope for bound variables. A do-body is a label-scope for labels.

The form (LEX **id**) is only required for the identifiers FLUID or LEX.

Because the initialization expressions are evaluated in the environment which was in effect before the DO expression was evaluated, evaluation of a RETURN expression in an initialization expression causes that previous environment to be terminated. Evaluation of a RETURN expression in any other context within the DO expression causes the evaluation of the DO expression to be terminated and the value of the DO expression is the value of the RETURN expression.

Because the do-body is a label-scope, evaluation of an EXIT expression within the do-body causes termination of an individual iteration rather than termination of the DO expression.

Evaluation of a DO expression results in the creation of an implicit lambda expression, with the local variables becoming the bound variables of the lambda expression and the initial values becoming the arguments.

Since the local variables are set after all step-expressions have been evaluated, explicit assignments to step variables in the step expression will be lost.

10.5 Transfer of Control

GO —— special form

(GO **id**)

id must be associated with a label in an enclosing label-scope. The GO expression transfers control to the associated label. Control may not transfer out of an enclosing contour or label-scope or an error is signalled.

EXIT —— special form

(EXIT [**exp**])

The EXIT expression causes control to leave the nearest enclosing label-scope or contour. If **exp** is present, the value of **exp** becomes the value of the terminated expression. Otherwise, the value of the terminated expression is NIL.

RETURN —— special form

(RETURN [**exp**])

The RETURN expression causes control to leave the nearest enclosing contour. If **exp** is present, the value of **exp** becomes the value of the terminated expression. Otherwise, the value of the terminated expression is NIL.

Control may transfer over enclosing label-scopes, causing their termination.

10.6 Conditional Evaluation

COND —— special form

(COND [**clause** ...])

This is the IF ... THEN ... ELSE of LISP. Each clause is a list of expressions of the form

```
(PREDICATE [exp ...])
```

(Any clause which is not a list is treated as a comment.)

The clauses are examined in order, and the **predicate** in each clause is evaluated. If the value is NIL, the next clause is examined.

If the value is not NIL, the remainder of the clause is examined. If there are no **exps** the value of the **predicate** is the value of the COND expression. Otherwise the list of **exps** is evaluated as if it were prefaced with PROGN. The value of the final **exp** becomes the value of the COND.

In either case, no further clauses are examined.

```
(COND ((GREATERP X 0) X)
      ('ELSE (MINUS X)))
```

returns the absolute value of X. The 'ELSE predicate can never evaluate to NIL since it is quoted.

CASEGO —— macro

(CASEGO **exp list** ...)

Where each **list** is an alternative of the form

(**item id**)

and where **id** must be associated with a label within an enclosing SEQ or PROG. CASEGO evaluates **exp** and compares the resulting value with the unevaluated **items** from successive alternatives. If one is found which is EQ to the value of **exp**, control transfers to the associated label. Control may not transfer out of an enclosing contour or an error is signalled.

```
(CASEGO
  (CAR X)
  (A L1)
  (B L2)
  (3 L2)
  (XYZ D))
```

Will transfer control to the label L1 if the CAR of the value of X is the identifier A, to the label L2 if it is the identifier B or the number 3, to the label D if it is the identifier

XYZ. If it is none of those four, control will continue following the CASEGO expression.

If control continues through the last expression of the CASEGO, the value of the CASEGO is the value of **exp**.

OR —— *macro*

(OR [**exp** ...])

This operator evaluates the **exps** from left to right, until the first **exp** whose value is non-NIL. The value of the expression is that non-NIL value, if such exists, and is NIL otherwise.

```
(OR exp1 exp2 ... expn)
```

is equivalent to

```
(COND (exp1) (exp2) ... (expn))
```

(OR) has a value of NIL.

AND —— *macro*

(AND [**exp** ...])

This operator evaluates the **exps** from left to right, until the first **exp** whose value is NIL. The value of the expression is the value of the final **exp**, or NIL, if an earlier **exp** resulted in that value.

```
(AND exp1 exp2 exp3)
```

is equivalent to

```
(COND (exp1 (COND (exp2 exp3))))
```

(AND) has a value of non-NIL.

SELECT —— *macro*

(SELECT **exp1** **list** ... **exp2**)

Where each **list** is an alternative of the form

```
(exp exp ...)
```

The SELECT operator evaluates **exp1**. It then evaluates the first **exp** of each alternative until it find one whose value is EQ to the value of **exp1**. If it finds such a one, it evaluates the remaining **exps** in that alternative as an implied PROGN, returning the value of the final **exp** as the value of the SELECT expression.

If no such **list** is found **exp2** is evaluated and its value becomes that of the SELECT expression.

SELECT is a macro which generates a function expression. A SELECT is a scope for local variables.

10.7 Multiple Level Returns, CATCH and THROW

Each application of an ordinary operator (function expression or bpi) results in the creation of a *frame* in the stack. Normally, evaluation remains within a frame until the end of evaluation of the body of a function expression or an explicit RETURN causes evaluation to revert to the immediately preceding frame.

There exists a set of operators which mark certain stack frames as *catch points*. A catch point is a frame which can receive control directly from a frame which is not its immediate successor, via the operator THROW. The frame passing control to a catch point must be a successor of the catch point, but may be many levels below it.

Certain catch points will intercept only specific THROW operations, others will intercept a wide class, or even all THROW operations.

Each THROW has two operands, the first (referred to as the *tag*) specifying the targeted catch point, the second providing an arbitrary value which is available when the (referred to as the *value*) THROW terminates. When a catch point receives control it has access to both of these values. It may then evaluate arbitrary code, continue evaluation, or propagate the THROW, with the same or modified tag.

Each catch point corresponds to a stack frame, with lambda bindings. In particular each such frame has a lexical binding of CATCH,MESS which is a "message", a distinctive value identifying the catch point. These values are found and interpreted by the & operator. See page 229 and Figure 12 on page 88.

CATCH —— macro

(CATCH **id1** **exp** [**id2** [**item** ...]])

Where neither **id1** nor **id2** are evaluated, but are used as written.

CATCH establishes a catch point and then evaluates **exp**. If no THROW occurs during the evaluation the value of the CATCH expression is the value of **exp**. If a THROW to a tag EQ to **id1** occurs, the value is the value THROWen.

If **id2**, is present, it is interpreted as a variable, and is used to indicate the mode of return. If the evaluation of **exp** completed normally, **id2** is assigned a value of NIL. If a THROW to **id1** prematurely terminated the evaluation, **id2** is assigned a value of **id1**. (Of course, if a THROW to some other tag occurs, control never returns to this CATCH.)

The evaluation of **exp** is performed in such a way as to contain the scope of EXIT and RETURN expressions within it. Either will provide a value for **exp**, but neither will cause control to leave the CATCH without its setting the value of **id2**, if present.

The **items**, if present, are make part of the value of CATCH,MESS. If **id2** is NIL **items** may follow it, but it is treated as not present.

CATCHALL —— macro

(CATCHALL **exp** [**id** [**item** ...]])

Where **id** is not evaluated, but is used as written.

CATCHALL establishes a catch point and then evaluates **exp**. If no THROW occurs during the evaluation the value of the CATCH expression is the value of **exp**. If a THROW occurs to any tag the value is the value THROWen.

If **id**, is present, it is interpreted as a variable, and is used to indicate the mode of return. If the evaluation of **exp** completed normally, **id** is assigned a value of NIL. If a THROW prematurely terminated the evaluation, **id** is assigned the tag THROWen to as its value.

The evaluation of **exp** is performed in such a way as to contain the scope of EXIT and RETURN expressions within it. Either will provide a value for **exp**, but neither will cause control to leave the CATCH without its setting the value of **id**, if present.

The **items**, if present, are make part of the value of CATCH,MESS. If **id** is NIL **items** may follow it, but it is treated as not present.

THROW —— function and compile-time macro

(THROW {**id** | **sint**} **item**)

The THROW operator terminates the current evaluation and searches backwards in the stack for a catch point. At each catch point found the value of the tag of the is examined to determine whether the destination has been reached. If so, the THROW is stopped

ERRSET[1]	
(0 . [user-item ...])	Return THROWen value.
ERRCATCH[1]	
(2 . [user-item ...]) (3 . [user-item ...])	Return THROWen value. Return THROWen value, sets FLAGVAR.
CATCH	
(4 . [user-item ...]) (5 . [user-item ...])	Return THROWen value. Return THROWen value, sets FLAGVAR.
NAMEDERRCATCH[1]	
(6 . [user-item ...])	Return THROWen value.
THROW-PROTECT	
(7 . [user-item ...])	Evaluate epilog and continue THROW.
CATCHALL	
(8 . [user-item ...]) (9 . [user-item ...])	Return THROWen value. Return THROWen value, sets FLAGVAR.
[1] marks catch points which intercept numeric tags and count them down.	

Figure 12. User Defined CATCH,MESS values, actions and interpretation.

and evaluation resumes as that catch point. If not, the THROW is continued, possibly with a modified tag, or after arbitrary clean-up code has been evaluated, see THROW-PROTECT, below.

At the catch point which stops the THROW the value of **item** is available. The value of the tag is sometimes available.

UNWIND —— function

(UNWIND [**sint** [**item**]])

The UNWIND operator evaluates a THROW with a numeric tag. **sint** defaults to 1 (one), while **item** defaults to NIL.

UNWIND is normally used to escape from the error break loop, passing control back to an error-expecting catch point.

THROW-PROTECT —— macro

(THROW-PROTECT **exp1 exp2**)

This operator evaluates its operands in sequence, establishing a catch point during the evaluation of **exp1**.

If the evaluation of **exp1** completes normally, its value is reserved, and becomes the value of the THROW-PROTECT expression, after **exp2** has been evaluated.

If the evaluation of **exp1** results in a THROW which is not caught before control passes the THROW-PROTECT, the THROW is temporarily stopped, **exp2** is evaluated, and the THROW is continued.

The evaluations of both **exp1** and **exp2** are protected against control being lost due to RETURN or EXIT expressions. A THROW in **exp2** will, however, take precedence over the resumption of the suspended THROW.

SUPERMAN[1]	
901	Call to SUPV Tries again, with existing streams.
902	EVAL of user provided expression, INITSUPV Same action as 901.
SUPV[1]	
903	Read input Returns to read with existing streams.
904	Echo-print Same action as 903.
905	EVALFUN of input Same action as 903, but ,VAL updated with caught value.
906	Value-print Same action as 903.
907	Print of message when UNWIND is caught Same action as 903.
EXF[1]	
908	SUPV call EXFTEMP LISPLIB not renamed (if it exists), files shut.
ERRORLOOP[1]	
909	Error message print Enters the read loop.
910	Read input Returns to input read.
911	EVALFUN input Same as 910.
912	Value-print Same as 910.
913	Print of message when UNWIND is caught Same as 910.
DISPATCHER	
914	Dispatch loop Stops interrupt servicer scan.
[1] marks catch points which intercept numeric tags and count them down.	

Figure 13. System Defined CATCH,MESS values, actions and interpretation.

ERRSET —— *macro*

(ERRSET **exp** [**item** ...])

The ERRSET operator establishes a catch point and evaluates **exp**. If the evaluation terminates normally the value of the ERRSET expression is a list of one element, containing the value of **exp**.

If a THROW with a numeric tag occurs during the evaluation, ERRSET examines the tag. If the tag is 0 (zero), the THROW is stopped and the value of the ERRSET expression is the value THROWen. If the tag is not 0 (zero) the THROW is continued, with the tag decremented by 1 (one).

Thus, the first operand of the THROW (or the UNWIND) controls the number of ERRSET (and ERRCATCH and NAMEDERRSET) expression to be skipped.

The **items**, if present, are make part of the value of CATCH,MESS.

NAMEDERRSET —— *macro*

(NAMEDERRSET **id exp [item** ...])

NAMEDERRSET behaves like ERRSET, but allows a tag and user messages to be specified.

This allows an UNWIND directly to a specific NAMEDERRSET, even when there may be an unknown number of ERRSETs or ERRCATCHs intervening.

The **items**, if present, are made part of the value of CATCH,MESS.

ERRCATCH —— *macro*

(ERRCATCH **exp** [**id** [**item** ...]])

ERRCATCH behaves like ERRSET with two exceptions. First, the value of the ERRCATCH expression is either the value of **exp** (not in a list), or the value of the intercepted THROW. Second, **id**, if present, is set to NIL if the evaluation of **exp** terminates normally, and to 0 (zero) if a THROW is intercepted.

Unlike ERRSET, which can be fooled by THROWing a list as value, ERRCATCH allows the return of arbitrary values, while still detecting abnormal returns.

The **items**, if present, are make part of the value of CATCH,MESS. If **id** is NIL **items** may follow it, but it is treated as not present.

10.8 Iteration over Lists and Vectors, the Mapping Operators

Mapping operators are the commonest iterative constructs.

All of the mapping operators are macros, which construct PROGs, with their functional operand explicitly placed in operator position. This eliminates the need to construct funargs, with their overhead of state descriptors, and allows many macros to be used as functional operands.

List mapping operators apply a functional operand to successive portions of one or more lists, often constructing a value from the results of these applications. In addition, operators are provided which iterate over vectors, and others which will terminate their iteration when some criterion is met.

All of the mapping operators terminate when their shortest list or vector is exhausted.

This allows their use with "infinite" lists, e.g. the value of the LOTSOF operator, as long as one of their arguments is non-cyclic.

10.8.1 List Mapping Operators

The mapping operators can be split along two axes.

- Those that apply their **app-ob** to the successive elements of their **list** operands.
  MAPC, MAPCAR, MAPCAN

- Those that apply their **app-ob** to their **list** operands, and the successive CD...Rs of those operands.
  MAP, MAPLIST, MAPCON

and

- Those that return their first **list** operand as their value.
  MAPC, MAP

- Those that construct a list of the values returned by successive applications of their **app-ob**.
  MAPCAR, MAPLIST

- Those that use NCONC to splice together the values returned by successive applications of their **app-ob**.

MAPCAN, MAPCON

MAPC —— *macro*

(MAPC **app-ob list** ...)

MAPC applies **app-ob** to the 1st, 2nd, etc. elements of the **lists**. The value of the expression is the first **list**.

```
(MAPC (LAMBDA (X Y) (PRINT (CONS X Y))) '(1 2 3 4) '(9 8 7))
(1 . 9)
(2 . 8)
(3 . 7)
Value = (1 2 3 4)
```

MAP —— *macro*

(MAP **app-ob list** ...)

MAP applies **app-ob** to the **lists**, then to the CDRs, CDDRs, etc. of the **lists**. The value of the expression is the first **list**.

```
(MAPC (LAMBDA (X) (PRINT X)) '(1 2 3 4))
(1 2 3 4)
(2 3 4)
(3 4)
(4)
Value = (1 2 3 4)
```

Note that the expression

```
(MAP PRINT '(1 2 3 4))
```

would have the same result.

These two operators are used only for their side effects, in this case, printing.

MAPCAR —— *macro*

(MAPCAR **app-ob list** ...)

MAPCAR applies **app-ob** to the 1st, 2nd, etc. elements of the **lists**. The value of the expression is a list of the values of these applications.

```
(MAPCAR (LAMBDA (X Y) (CONS X Y)) '(1 2 3 4) '(9 8 7))
Value = ((1 . 9) (2 . 8) (3 . 7))
```

Extra operands are discarded. Thus, one can use the termination condition of the mapping operators, when the shortest **list** is exhausted, in many ways.

For instance:

```
(SETQ X '(1 2 3 4))
Value = (1 2 3 4)
(MAPCAR (LAMBDA () ()) X)
Value = (() () () ())
```

Here we have constructed a list of NILs, equal in length to another list.

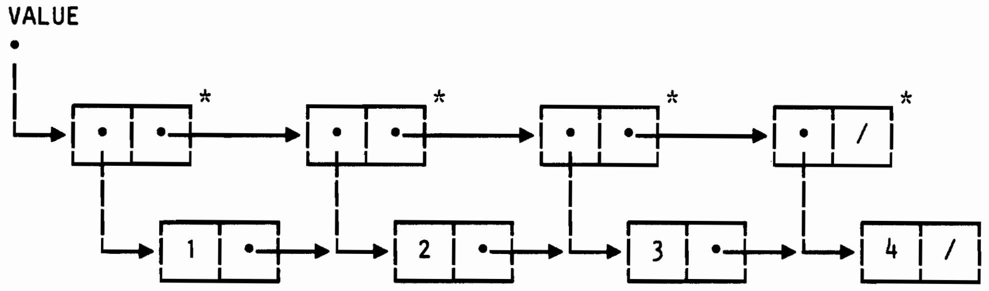
In order to facilitate such operations a group of auxiliary operators has been provided, NILFN, TRUEFN and IDENTITY. See page 95.

MAPLIST —— *macro*

(MAPLIST **app-ob list** ...)

MAPLIST applies **app-ob** to the **lists**, then to the CDRs, CDDRs, etc. of the **lists**. The value of the expression is a list of the values of these applications.

```
(MAPLIST IDENTITY '(1 2 3 4))
Value = ( (1 . %L1=(2. . %L2=(3 . %L3=(4)))) %L1 %L2 %L3)
```



(Note, the pairs marked by * are newly created.)

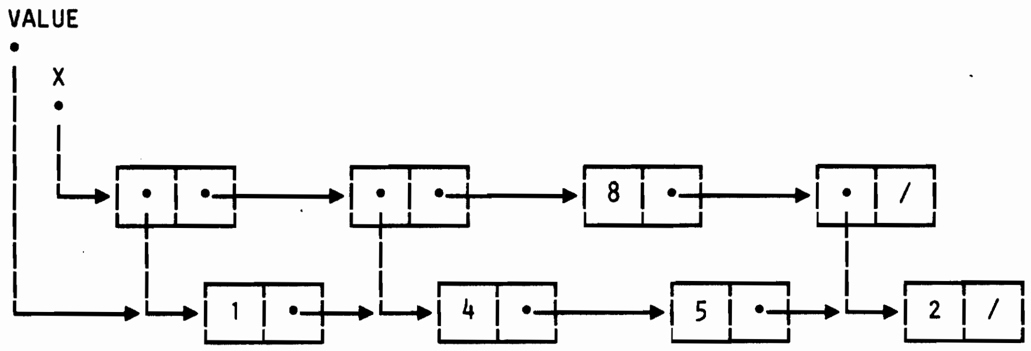
Figure 14. Result of MAPLIST.

MAPCAN —— *macro*

(MAPCAN **app-ob list** ...)

MAPCAN applies **app-ob** to the 1st, 2nd, etc. elements of the **lists**. The value of the expression produced by NCONCing the values of these applications together.

```
(MAPCAN LIST '(1 2 3) '(6 7 8 9))
Value = (1 6 2 7 3 8)
(SETQ X (LIST '(1) '(4 5) 8 '(2)))
Value = ((1) (4 5) 8 (2))
(MAPCAN IDENTITY X)
Value = (1 4 5 2)
X
Value = ((1 . %L1=(4 5 . %L2=(2))) %L1 8 %L2)
```



(Note, no new pairs are created.)

Figure 15. Result of MAPCAN.

With both MAPCAN and MAPCON side effects can be drastic. Unless you are constructing the values of the individual applications, you should be very careful.

MAPCON —— *macro*

(MAPCON **app-ob list** ...)

MAPCON applies **app-ob** to the **lists**, then to the CDRs, CDDRs, etc. of the **lists**. The value of the expression produced by NCONCing the values of these applications together.

```
(MAPCON COPY '(1 2 3 4))
Value = (1 2 3 4 2 3 4 3 4 4)
```

NMAPCAR —— *macro*

(NMAPCAR **app-ob** **list** ...)

NMAPCAR applies **app-ob** to the 1st, 2nd, etc. elements of the **lists**. The first **list** is updated, with RPLACA, with the values of the successive applications. The value of the expression is the first (updated) **list**.

```
(SETQ X '(1 2 3 4))
Value = (1 2 3 4)
(NMAPCAR ADD1 X)
Value = (2 3 4 5)
X
Value = (2 3 4 5)
(NMAPCAR CONS X (LOTSOF a b))
Value = ((2 . a) (3 . b) (4 . a) (5 . b))
X
Value = ((2 . a) (3 . b) (4 . a) (5 . b))
```

SCANOR —— *macro*

(SCANOR **app-ob** **list** ...)

SCANOR applies **app-ob** to the 1st, 2nd, etc. elements of the **lists**. The iteration terminates at the first such application which results in a non-NIL value. If all applications result in NIL values, the value of the expression is NIL, otherwise it is the non-NIL, terminating, value.

```
(SCANOR (LAMBDA (X) (GREATERP X 10)) '(5 3 13 8 23))
Value = 13
(SCANOR (LAMBDA (X) (GREATERP X 100)) '(5 3 13 8 23))
Value = ()
```

ASSQ (see page 118) could be defined as

```
(LAMBDA (I L)
  (SCANOR
    (LAMBDA (X)
      (COND
        ( (AND (PAIRP X) (EQ (CAR X) I))
          X )) )
    L))
```

SCANAND —— *macro*

(SCANAND **app-ob** **list** ...)

SCANAND applies **app-ob** to the 1st, 2nd, etc. elements of the **lists**. The iteration terminates at the first such application which results in a NIL value. If all applications result in non-NIL values, the value of the expression is the value of the last application, otherwise it is NIL.

```
(SCANAND NUMP '(1 2 3 4 5))
Value = 5
(SCANAND NUMP '(1 2 a 3 b))
Value = ()
```

10.8.2 Vector Mapping Operators

The following operators are designed primarily for iteration of vector operands. Since they use ELT to access their operands, they will work correctly on lists, as well as on vectors. Since they require an upper limit on their index value, they compute the MIN of SIZE of all of their operands. This restricts their use to non-cyclic lists. In addition, since SIZE must traverse its operand when it is a list, such operands will be traversed twice.

These operators should only be used when one or more of the arguments are vectors, and never should be used with circular lists.

MAPE —— *macro*

(MAPE **app-ob** **vec** ...)

MAPE applies **app-ob** to the 1st, 2nd, etc. elements of the **vecs**. The value of the expression is the length of the shortest **vec**, i.e., the number of iterations.

```
(MAPE PRINT "ABC")
A
B
C
Value = 3
```

MAPELT —— *macro*

(MAPELT **app-ob** **vec** ...)

MAPELT applies **app-ob** to the 1st, 2nd, etc. elements of the **vecs**. The value of the expression is a vector containing the values of the successive applications.

```
(MAPELT
  (LAMBDA (X Y) (APPEND X Y))
  "ABCD"
  <"1" "23" "456" "7890" "12345">)
Value = <"A1" "B23" "C456" "D7890">
```

NMAPELT —— *macro*

(NMAPELT **app-ob** **vec** ...)

NMAPELT applies **app-ob** to the 1st, 2nd, etc. elements of the **vecs**. The first **vec** is updated by SETELT with the successive value of the applications. The value of the expression is the first (updated) **vec**.

```
(SETQ X <1 2 3 4>)
Value = <1 2 3 4>
(NMAPELT PLUS X '(10 20 30 40))
Value = <11 22 33 44>
X
Value = <11 22 33 44>
```

These are the vector counterparts of SCANOR and SCANAND. The previous discussion applies.

VSCANOR —— *macro*

(VSCANOR **app-ob** **vec** ...)

VSCANOR applies **app-ob** to the 1st, 2nd, etc. elements of the **vecs**. The iteration terminates at the first such application which results in a non-NIL value. If all applications result in NIL values, the value of the expression is NIL, otherwise it is the non-NIL, terminating, value.

```
(SETQ X "This, is a test.")
Value = "This, is a test."
(VSCANOR (LAMBDA (X) (MEMQ X '(.  ,  ;  :))) X)
Value = (, ; :)
```

VSCANAND —— *macro*

(VSCANAND **app-ob** **vec** ...)

VSCANAND applies **app-ob** to the 1st, 2nd, etc. elements of the **vecs**. The iteration terminates at the first such application which results in a NIL value. If all applications result in non-NIL values, the value of the expression is the value of the last application, otherwise it is NIL.

```
(VSCANAND LESSP <3 6 4 8> <5 7 8 10>)
Value = non-NIL
(VSCANAND LESSP <3 6 4 8> <5 7 2 9>)
Value = NIL
```

10.8.3 Miscellaneous

MAPOBLIST —— function

(MAPOBLIST **app-ob**)

MAPOBLIST applies **app-ob** to each identifier in the object array, that is to each non-gensym, interned identifier in the system. The value of the expression is NIL, so any results must be obtained via side effects.

WRAP —— function

(WRAP **list item**)

This operator iterates over **list**, creating a new list whose elements are determined by the value of **item**.

For **list** of the form

```
(i1 i2 ...)
```

if **item** is NIL, the value will be **list**,
if **item** is not a pair, the value will be

((**item** i1) (**item** i2) ...)

if **item** is a list, its elements are match with the elements of **list**, with the value containing elements from **list** where the corresponding element of **item** is NIL, and lists of the elements from **item** and **list** otherwise.

If **item** is a list, and if **list** contains more elements than **item**, the final CDR of **item** is matched against the remaining elements of **list**.

```
(WRAP '(A B C D E) 'QUOTE)
Value = ('A 'B 'C 'D 'E)
(WRAP '(A B C D E) 'FLUID)
Value = ((FLUID A) (FLUID B) (FLUID C) (FLUID D) (FLUID E))
(WRAP '(A B C D E) '(FLUID () () . FLUID))
Value = ((FLUID A) B C (FLUID D) (FLUID E))
(WRAP '(A B C D E) '(X Y Z))
Value = ((X A) (Y B) (Z C) D E)
```

10.8.4 Auxiliary Operators

These are operators which correspond to often used **app-ob** operands. They produce more efficient code.

IDENTITY —— function and compile-time macro

(IDENTITY **item**)

The value of IDENTITY is **item**. This operator is exactly equivalent to

```
(LAMBDA (X) X)
```

TRUEFN —— function and compile-time macro

(TRUEFN)

The value of TRUEFN is non-NIL. This operator is exactly equivalent to

```
  (LAMBDA () 'non-NIL)
```

TRUEFN and NILFN ignore the value of extra operands

NILFN —— *function and compile-time macro*

(NILFN)

The value of NILFN is NIL. This operator is exactly equivalent to

```
  (LAMBDA () ())
```

11.0 Operations on Identifiers

11.1 Creation

INTERN —— function

(INTERN **cvec**)

This operator searches the object array for a pre-existing identifier with a pname EQUAL to **cvec**. If one exists, it is returned as the value of INTERN. Otherwise, a new identifier is created, with a print name EQUAL to **cvec** and added to the object array. This new identifier is returned as the value of INTERN. If **cvec** is not a character vector, an error is signalled.

CVEC2ID —— function

(CVEC2ID **cvec**)

This operator creates a new identifier, with a pname EQUAL to **cvec** This new identifier is returned as the value of CVEC2ID. It is not added to the object array. If **cvec** is not a character vector, an error is signalled.

GENSYM —— function and compile-time macro

(GENSYM)

This operator constructs a new, unique, non-stored identifier.

The mechanism used to insure unique ID's is simply to have a counter which is incremented every time a new gensym is required and to incorporate this counter's value into the print name of the identifier.

DOWNCASE —— function

(DOWNCASE **id**)

The DOWNCASE operator returns the identifier whose pname is the result of translating any upper case alphabetic characters in the pname of **id** into their lower case equivalents.

OR

(DOWNCASE **cvec**)

This operator, when applied to a character vector, translates all upper-case alphabetic character to their lower-case equivalents. This translation is done in place, updating the operand.

DOWNCASE will also accept a list of character vectors, in which case its value is a new list of the translated vectors.

UPCASE —— function

(UPCASE **id**)

The UPCASE operator returns the identifier whose pname is the result of translating any lower case alphabetic characters in the pname of **id** into their upper case equivalents.

OR

(UPCASE **cvec**)

This operator, when applied to a character vector, translates all lower-case alphabetic character to their upper-case equivalents. This translation is done in place, updating the operand. UPCASE will also accept a list of character vectors, in which case its value is a new list of the translated vectors.

11.2 Accessing

PNAME —— function

(PNAME **id**)

Returns a copy of the print name of **id**. If the value of **id** is not an identifier, an error is signalled. The print name is a character vector.

GET —— function

(GET **id item**) .

If **id** is not an identifier or a pair, value is NIL. Otherwise the property list of **id** is searched for the first occurrence of an element such that

```
(EQ item (CAR element))
```

is true. If found, the value of GET is (CDR element). If such an element is not found, the value of GET is NIL.

OR

(GET **list item**)

If **list** is not an identifier or a pair, value is NIL. If **list** is a pair, it is interpreted as an a-list, and searched for an element whose CAR is EQ to **item** If found, the value of GET is the CDR of the element, otherwise the value of GET is NIL.

Note the difference between GET and ASSQ, in that ASSQ returns the name-value pair, whereas GET returns only the value portion. Also, note the reversal of the order of arguments, GET receives the list to be search, followed by the name; while ASSQ (and the other .ASS. operators) receive the name first, followed by the list to be searched.

PROPLIST —— function

(PROPLIST **id**)

The value of **id** must be an identifier. Returns the property list associated with that identifier. The property list is an association list. The value returned is not the actual property list, but is the result of applying APPEND to the property list and NIL. Thus, while the name-value pairs may be updated, the actual property list itself is secure.

11.3 Searching and Updating

PUT —— function

(PUT **id item1 item2**)

Operator to update the property list of the identifier **id**. If the **item1** property already exists, its associated value is changed to **item2**. If the **item1** property does not currently exist, a new property with this name is put at the beginning of the property list.

The value of **id** must be an identifier or a list, but **item1** and **item2** may be any expressions.

OR

(PUT **list item1 item2**)

list is interpreted as an a-list. It is searched for a pair with CAR EQ to **item1**. If such a pair is found it is updated, with RPLACD, making **item2** its CDR. If no such pair is found a new pair, **(item1 . item2)**, is added to the *front* of **list**, by updating operations. is put at the beginning of the property list.

The value of **list** must be an identifier or list, but **item1** and **item2** may be any expressions.

DEFLIST —— function

(DEFLIST **list item**)

This function expects **list** to be a list of pairs whose CARs are identifiers and whose CADRs are arbitrary values. For each of these pairs, (PUT ID **item** VALUE) is performed, assigning the CADR value from the pair as the value of the **item** property in the identifier's property list.

The value of DEFLIST is a new list containing the property values (the CADRs of the elements of **list**).

To put NUM properties on each of the identifiers A through F, with values 10 through 15, one would write

```
(DEFLIST
  '( (A 10) (B 11) (C 12) (D 13) (E 14) (F 15) )
  'NUM)
Value = (10 11 12 13 14 15)
```

REMPROP —— function

(REMPROP **id item**)

The **item** property of the identifier **id** is removed from the property list of **id**. The value of REMPROP is NIL if there is no **item** property. If the property exists, the value of REMPROP is the value associated with that property.

(REMPROP **list item**)

The **item** property of the identifier **list** is removed from the property list of **list**. The value of REMPROP is NIL if there is no **item** property. If the property exists, the value of REMPROP is the value associated with that property.

The effect is that of NREMOVE, given the name-value pair as an operand.

11.4 Updating

REMALLPROPS —— function

(REMALLPROPS **id**)

All properties on the property list of **id** are removed. If **id** is not a stored identifier the value of REMALLPROPS is NIL, otherwise the value is **id**.

11.5 Object Array

OBARRAY —— function

(OBARRAY)

Returns as value a copy of the current object array. This is a vector containing elements which are identifiers (INTERN'ed variables).

Because the value of OBARRAY is a copy of the actual object array, it may be modified in any way by the user.

12.0 Operations on Numbers

The majority of operators which expect numbers as operands will accept either integer or real numbers. Unless otherwise stated, any operator having mixed-type operands will return a real number as the result, even if the value is a whole number. When an operator returns an integer result, the choice of type for that result, small or large integer, will not be determined by the types of the operands. See "Numbers" on page 31.

12.1 Conversion

FLOAT —— *function*

(FLOAT **num**)

INT2RNUM —— *function*

(INT2RNUM **num**)

The operand value must be numeric or an error is signalled. The FLOAT and INT2RNUM functions convert the operand value to a real number which is returned. If the operand value is an integer, the converted value is the closest real approximation to that integer, unless the integer exceeds the range of real numbers (approximately 10^{75}), in which case an error is signalled. If the operand value is already a real number, that value is returned.

FIX —— *function*

(FIX **num**)

RNUM2INT —— *function*

(RNUM2INT **num**)

The operand value must be numeric or an error is signalled. The FIX and RNUM2INT functions convert the operand value to an integer which is returned. If the operand value is a real number, the converted value is the integral part of that operand value. If the operand value is already an integer, that value is returned.

12.2 Predicates

PLUSP —— *function*

(PLUSP **item**)

The PLUSP function tests whether the operand value is positive, zero, negative or non-numeric. If it is positive or zero, the operand value is returned. If it is negative or non-numeric, NIL is returned.

QSPLUSP —— *macro*

(QSPLUSP **sint**)

The non-checking counterpart of PLUSP. If the operand value is not a small integer, the value returned is unspecified.

ZEROP —— *function*

(ZEROP **item**)

The ZEROP function tests whether the operand value is zero, non-zero or non-numeric. If it is zero, the operand value is returned. If it is non-zero or non-numeric, NIL is returned.

QSZEROP —— macro

(QSZEROP **sint**)

The non-checking counterpart of ZEROP. If the operand value is not a small integer, the value is unspecified.

MINUSP —— function

(MINUSP **item**)

The MINUSP function tests whether the operand value is negative, positive, zero, or non-numeric. If it is negative, the operand value is returned. If it is positive, zero, or non-numeric, NIL is returned.

QSMINUSP —— macro

(QSMINUSP **sint**)

The non-checking counterpart of MINUSP. If the operand value is not a small integer, the value is unspecified.

ODDP —— function

(ODDP **item**)

The ODDP function tests whether the operand value is an odd or even integer. If it is odd, the operand value is returned. Otherwise, NIL is returned.

QSODDP —— macro

(QSODDP **sint**)

The non-checking counterpart of ODDP. If the operand value is not a small integer, the value is unspecified.

GREATERP —— function

(GREATERP **num1 num2**)

Both operand values must be numeric or an error is signalled. The GREATERP function compares the operand values. If the first operand value is greater than the second, then non-NIL is returned. Otherwise, NIL is returned. If either operand is a real number, the real fuzz factor may affect the comparison. See page 31

QSGREATERP —— macro

(QSGREATERP **sint1 sint2**)

The non-checking counterpart of GREATERP. If both operand values are not small integers, the value is unspecified.

LESSP —— function

(LESSP **num1 num2**)

Both operand values must be numeric or an error is signalled. The LESSP function compares the operand values. If the first operand value is less than the first, then non-NIL is returned. Otherwise, NIL is returned. If either operand is a real number, the real fuzz factor may affect the comparison. See page 31

QSLESSP —— *macro*

(QSLESSP **sint1 sint2**)

The non-checking counterpart of LESSP. If both operand values are not small integers, the value is unspecified.

12.3 Computation

+ —— *function and compile-time macro*

(+ **num** ...)

PLUS —— *function and compile-time macro*

(PLUS **num** ...)

All operand values must be numeric or an error is signalled. The + and PLUS functions compute and return the sum of all operand values. If all operand values are integers, the sum is an integer. Otherwise, the sum is real. The + and PLUS macros cause an equivalent result.

- —— *function and compile-time macro*

(- [**num1 num2** ...])

DIFFERENCE —— *function and compile-time macro*

(DIFFERENCE [**num1 num2** ...])

If all arguments are omitted, the value is 0; if any arguments are supplied, they must all be numeric or an error is signalled.

If only one argument is given, the value is (MINUS **num1**).

When two arguments are supplied, the - and DIFFERENCE functions subtract the second operand value from the first and return the difference. If both operand values are integers, the difference is an integer. Otherwise, the difference is real. The - and DIFFERENCE macros cause an equivalent result.

If more than two arguments are supplied, the third argument is subtracted from the difference of the first two, and so on.

* —— *function and compile-time macro*

(* **num** ...)

TIMES —— *function and compile-time macro*

(TIMES **num** ...)

All operand values must be numeric or an error is signalled. The * and TIMES functions compute and return the product of all operand values. If all operand values are integers, the product is an integer. Otherwise, the product is real. The * and TIMES macros cause an equivalent result.

/ —— *function*

(/ [**num1 num2** ...])

QUOTIENT —— *function*

(QUOTIENT [**num1 num2** ...])

If all arguments are omitted, the value is 1; if any arguments are supplied, they must all be numeric or an error is signalled.

If only one argument is given, the value is (/ 1 **num1**).

When two arguments are supplied,

num1 is the dividend and **num2** is the divisor. The / and QUOTIENT functions divide the dividend by the divisor and return the quotient. If the dividend and divisor are integers, the quotient is an integer. Otherwise, the quotient is real. If the divisor is zero, an error is signalled.

If more than two arguments are supplied, the quotient of the first two is divided by the third, and so on.

** —— *function*

(** **num1 num2**)

EXPT —— *function*

(EXPT **num1 num2**)

num1 is the base and **num2** is the exponent. The base and exponent must be numeric or an error is signalled. The ** and EXPT functions raise the base to the power of the exponent and return the result. If the base is an integer and the exponent is a positive integer, the result is an integer. Otherwise, the result is real. Since imaginary numbers do not exist in LISP/VM, the base cannot be negative when the exponent is not a positive integer.

MAX —— *function and compile-time macro*

(MAX **num** ...)

All operand values must be numeric or an error is signalled. The MAX function finds and returns the algebraically largest operand value. The MAX macro causes and equivalent result.

QSMAX —— *function and compile-time macro*

(QSMAX **sint1 sint2**)

The non-checking counterpart of MAX, defined for only two operands. If both operand values are not small integers, the value returned is unspecified.

MIN —— *function and compile-time macro*

(MIN **num** ...)

All operand values must be numeric or an error is signalled. The MIN function finds and returns the algebraically smallest operand value. The MIN macro causes an equivalent result.

QSMIN —— *function and compile-time macro*

(QSMIN **sint1 sint2**)

The non-checking counterpart of MIN, defined for only two operands. If both operand values are not small integers, the value returned is unspecified.

ABSVAL —— *function*

(ABSVAL **num**)

The operand value must be numeric or an error is signalled. The ABSVAL function returns the absolute value of the operand value.

QSABSVAL —— *function and compile-time macro*

(QSABSVAL **sint**)

The non-checking counterpart of ABSVAL. If the operand value is not a small integer, the value is unspecified.

MINUS —— *function*

(MINUS **num**)

The operand value must be numeric or an error is signalled. The MINUS function applies unary minus to the operand value and returns the result.

QSMINUS —— *function and compile-time macro*

(QSMINUS **sint**)

The non-checking counterpart of MINUS. If the operand value is not a small integer, the value is unspecified.

QSPLUS —— *function and compile-time macro*

(QSPLUS **sint1 sint2**)

The non-checking counterpart of PLUS, defined for two operands. If both the operand values are not small integers, or if the sum is outside the small integer range, the value will be an unspecified small integer.

ADD1 —— *function*

(ADD1 **num**)

The operand value must be numeric or an error is signalled. The ADD1 function adds 1 to the operand value. If the operand value is an integer, the sum returned is an integer. Otherwise, the sum returned is a real number.

QSADD1 —— *function and compile-time macro*

(QSADD1 **sint**)

The non-checking counterpart of ADD1. If the operand value is not a small integer, the value is unspecified.

QSINC1 —— *macro*

(QSINC1 **sint**)

This function is equivalent to QSADD1, except when its operand value is -1. In that case, its value is unspecified and possibly fatal.

The QSINC1 function compiles to a single machine instruction, and is used for index arithmetic in many internal functions. The QSINC1 macro causes an equivalent result.

QSDIFFERENCE —— *function and compile-time macro*

(QSDIFFERENCE **sint1 sint2**)

The non-checking counterpart of DIFFERENCE, defined for small integers. If both the operand values are not small integers, or if the difference is outside the small integer range, the value will be an unspecified small integer.

SUB1 —— *function*

(SUB1 **num**)

The operand value must be numeric or an error is signalled. The SUB1 function subtracts 1 from the operand value. If the operand value is an integer, the difference returned is an integer. Otherwise, the difference returned is a real number.

QSSUB1 —— *function and compile-time macro*

(QSSUB1 **sint**)

The non-checking counterpart of SUB1. If the operand value is not a small integer, the value is unspecified.

QSDEC1 —— *macro*

(QSDEC1 **sint**)

This function is equivalent to QSSUB1, except when its operand value is zero. In that case, its value is unspecified and possibly fatal.

QSDEC1 compiles to a single machine instruction, and is used for index arithmetic in many internal functions. The QSSUB1 macro causes an equivalent result.

QSTIMES —— *function and compile-time macro*

(QSTIMES **sint1 sint2**)

The non-checking counterpart of TIMES, defined for two operands. If both operand values are not small integers, or if the product is outside the small integer range, the value is an unspecified small integer.

DIVIDE —— *function*

(DIVIDE **num1 num2**)

num1 is the dividend and **num2** is the divisor. The dividend and divisor must be numeric or an error is signalled. The DIVIDE function divides the dividend by the divisor and returns a list whose first element is the quotient and whose second element is the remainder. If both operand values are integers, the quotient and remainder will be integers. Otherwise, the quotient and remainder will be real numbers. In this case, the remainder is computed by multiplying the divisor by the quotient and subtracting it from the dividend. If the divisor is zero, an error is signalled. THe DIVIDE macro causes an equivalent result.

QSQUOTIENT —— *function and compile-time macro*

(QSQUOTIENT **sint1 sint2**)

The non-checking counterpart of QUOTIENT. If both operand values are not small integers, or if the quotient is outside the small integer range, the value is an unspecified small integer.

REMAINDER —— *function and compile-time macro*

(REMAINDER **num1 num2**)

num1 is the dividend and **num2** is the divisor. The dividend and divisor must be numeric or an error is signalled. The REMAINDER function divides the dividend by the divisor and returns the remainder. If both operand values are integers, the remainder will be an integer. Otherwise, the remainder will be a real number which is computed by multiplying the divisor by the quotient and subtracting it from the dividend. If the divisor is zero, an error is signalled. The REMAINDER macro causes an equivalent result.

QSREMAINDER —— *function and compile-time macro*

(QSREMAINDER **sint1 sint2**)

The non-checking counterpart of REMAINDER. If both the operand values are not small integers, or if the remainder is outside the small integer range, the value will be an unspecified small integer.

LEFTSHIFT —— *function*

(LEFTSHIFT **num sint**)

num must be a small integer or one word large integer and &sint. must be a numeric value or an error is signalled. The first operand value is the field to be shifted and the second operand value is the shift-size, the number of bits to be shifted. If the shift-size is positive, the LEFTSHIFT function applies a binary left shift of the shift-size to the field and the shifted value is returned. If the shift-size is negative, the field is shifted right and returned. Any bits shifted outside of a 32 bit word are lost, and zero bits are supplied as needed.

QSLEFTSHIFT —— *function and compile-time macro*

(QSLEFTSHIFT **sint1 sint2**)

The non-checking counterpart of LEFTSHIFT. If both the operand values are not small integers, or if the result outside the small integer range, the value will be an unspecified small integer.

RIGHTSHIFT —— *function and compile-time macro*

(RIGHTSHIFT **num sint**)

num must be a small integer or one word large integer and &sint. must be a numeric value or an error is signalled. The first operand value is the field to be shifted and the second operand value is the shift-size, the number of bits to be shifted. If the shift-size is positive, the RIGHTSHIFT function applies a binary right shift of the shift-size to the field and the shifted value is returned. If the shift-size is negative, the field is shifted left and returned. Any bits shifted outside of a 32 bit word are lost, and zero bits are supplied as needed. The RIGHTSHIFT macro causes an equivalent result.

ALINE —— *function -- FOR THE EXPERT LISP/VM USER*

(ALINE **sint1 sint2**)

Value is the small integer **sint1** rounded up to the nearest multiple of the small integer **sint2**, where **sint2** is a power of 2. If **sint1** or **sint2** are not small integers or are negative, an error is signalled. If **sint1** is zero or one, returns **sint1** unchanged. If **sint2** is not a power of 2, result is unspecified.

```
(ALINE 15 4) = 16.
```

QSNOT —— *function and compile-time macro*

(QSNOT **sint**)

The QSNOT function returns the bitwise complement of the numeric value of the operand value, as a small integer. If the operand value is not a small integer, the value is unspecified. The QSNOT macro causes an equivalent result.

QSAND —— *function and compile-time macro*

(QSAND **sint1 sint2**)

The QSAND function returns the bitwise AND of the numeric values of the operand values. small integer. If both operand values are not small integers, the value is unspecified. The QSAND macro causes an equivalent result.

QSOR —— *function and compile-time macro*

(QSOR **sint1 sint2**)

The QSOR function returns the bitwise OR of the numeric values of both operand values as a small integer. If both operand values are not small integers, the value is unspecified. The QSOR macro causes an equivalent result.

QSXOR —— *function and compile-time macro*

(QSXOR **sint1** **sint2**)

The QSXOR function returns the bitwise exclusive OR of the numeric values of both operand values as a small integer. If the operand values are not small integers, the value is unspecified. The QSXOR macro causes an equivalent result.

EXP —— *function*

(EXP **num**)

The EXP function returns the value of e raised to the power of the operand value.

If the operand value is greater than 174.66, an error is signalled.

LN —— *function*

(LN **num**)

num must be a numeric value within the range of a real number or an error is signalled. The LN function computes and returns the natural logarithm of the operand value. The result is a real number.

LOG —— *function*

(LOG **num**)

num must be a numeric value within the range of a real number or an error is signalled. The LOG function computes and returns the base ten logarithm of the operand value. The result is a real number.

LOG2 —— *function*

(LOG2 **num**)

num must be a numeric value within the range of a real number or an error is signalled. The LOG2 function computes and returns the base two logarithm of the operand value. The result is a real number.

SIN —— *function*

(SIN **num**)

num must be a numeric value less than approximately $3.5*10^{15}$, or an error is signalled. The SIN function computes and returns the sine of the operand value, the angle in radians. The sine is a real number.

COS —— *function*

(COS **num**)

num must be a numeric value less than approximately $3.5*10^{15}$, or an error is signalled. The COS function computes and returns the cosine of the operand value, the angle in radians. The cosine is a real number.

13.0 Operations on Pairs

These operators treat pairs as pairs, not imposing any interpretation, such as lists, on the contents.

13.1 Creation

CONS —— built-in function

(CONS **item1 item2**)

CONS is the basic pair-creating function. Its value is the new pair constructed with **item1** as its CAR component and **item2** as its CDR component.

Of particular interest is the case where **item2** is a pair. Then the value of CONS is the list formed by adding **item1** to the beginning of the list **item2**.

CONSFN —— function

Do several CONSes with one call.

`(CONSFN` **item1 [item2 . . .] itemn**`)`

The result of this operation is a pair where the CAR is **item1** and the CDR is **itemn** or a pair where the CAR is **item2**. If **itemn** is NIL, this operator is equivalent to LIST.

13.2 Accessing

CAR —— built-in function

(CAR **item**)

One of the two basic access functions defined on pairs. Its value is the CAR component of the pair **item**.

In its list interpretation, the value of (CAR **item**) is the first element of the list **item**.

If **item** is not a pair, an error is signalled.

CDR —— built-in function

(CDR **item**)

One of the two basic access functions defined on pairs. Its value is the CDR component of the pair **item**. If **item** is not a pair, an error is signalled.

The value of (CDR **item**) is usually the list containing all but the first element of the list **item**; however, this is not the case when **item** is the terminating pair of a list. In that case, (CDR **item**) is the terminating atom, usually NIL.

C...A | D...R —— function and compile-time macro

(C...A | D...R **item**)

There are twenty eight macros defined which give meaning to operators of this form, where ... designates any sequence of one to four As or Ds. For example,

`(CADDR` **item**`)`

is equivalent to

`(CAR (CDR (CDR` **item**`)))`

and so on. The interpretations of these macros are performed efficiently. If any of the successive CAR and CDR operations applies to a non-pair, an error is signalled.

OR

(CA[D...]R **list**)

Any operator of this form, that is zero or more CDRs followed by a single CAR, may be though of as an accessing function for a list. CAR gives the first element, CADR the second, etc.

CD...R —— *function and compile-time macro*

(CD...R **list**)

These operations return the "tail" of their argument. CDR gives the list, less its first element, CDDR, less its first and second, etc.

QC...A | D...R —— *function and compile-time macro*

(QC...A | D...R **item**)

Each of the C...R operators (including CAR and CDR) has a corresponding QC...R operator. These operator perform the same action as the C...R operators, without first checking their argument for type. Thus, if applied to anything but pairs their action is unpredictable.

In compiled programs these operation are performed by in-line code.

QCA[D...]R —— *function and compile-time macro*

(QCA[D...]R **list**)

The non-checking versions of the preceding operators.

QCD...R —— *function and compile-time macro*

(QCD...R **list**)

The non-checking versions of the preceding operators.

IFCAR —— *function and compile-time macro*

(IFCAR **item**)

This is a conditional CAR operator. If **item** is a pair, it is equivalent to (CAR **item**), otherwise it has a value of NIL.

XORCAR —— *function and compile-time macro*

(XORCAR **item**)

This is another conditional CAR operator. If **item** is a pair, it is equivalent to (CAR **item**), otherwise the argument, **item**, is returned as the value.

IFCDR —— *function and compile-time macro*

(IFCDR **item**)

This is a conditional CDR operator. If **item** is a pair, it is equivalent to (CDR **item**), otherwise it has a value of NIL.

XORCDR —— *function and compile-time macro*

(XORCDR **item**)

This is another conditional CDR operator. If **item** is a pair, it is equivalent to (CDR **item**), otherwise the argument, **item**, is returned as the value.

13.3 Updating

RPLACA —— *built-in function*

(RPLACA **pair item**)

This is one of the two basic functions for updating pairs. Its value is the updated pair which results when the CAR component of the pair **pair** is replaced by **item**. If **pair** is not a pair, an error is signalled.

QRPLACA —— *macro*

(QRPLACA **pair item**)

Equivalent to RPLACA, with no type checking of arguments.

OR

(QRPLACA **list item**)

Equivalent to RPLACA, with no type checking of arguments.

RPLACD —— *built-in function*

(RPLACD **pair item**)

This is the other basic function which updates pairs. Its value is the updated pair which results when the CDR component of the pair **pair** is replaced by **item**. If **pair** is not a pair, an error is signalled.

QRPLACD —— *macro*

(QRPLACD **pair item**)

Equivalent to RPLACD, with no type checking of arguments.

OR

(QRPLACD **list item**)

Equivalent to RPLACD, with no type checking of arguments.

RPLPAIR —— *function*

(RPLPAIR **pair1 pair2**)

This function is equivalent to

```
(RPLACA pair1 (CAR pair2))
(RPLACD pair1 (CDR pair2))
```

Its value is the updated pair, **pair1** If either argument is not a pair, an error is signalled.

QRPLPAIR —— *macro*

(QRPLPAIR **pair1 pair2**)

Equivalent to RPLPAIR, with no type checking of arguments.

NCONS —— *function*

(NCONS **pair item1 item2**)

This function is equivalent to

```
(RPLACA pair item1)
(RPLACD pair item2)
```

Its value is the updated pair. If the argument, **pair**, is not a pair, an error is signalled.

QNCONS —— *macro*

(QNCONS **pair item1 item2**)

Equivalent to NCONS, with no type checking of arguments.

14.0 Operations on Lists

These operations assume the list interpretation of pairs, see "Lists" on page 33. The value of the final, non-pair, CDR of an argument is generally ignored once its non-pairness has been discovered. Similarly, an atomic (non-pair) argument is in most cases treated as an empty list.

14.1 Creation

LIST —— function and compile-time macro

(LIST [**item** ...])

Returns as its value a list of n elements, the first element being the value of the expression item1, et cetera.

```
(LIST item1 item2 ... itemn)
```

is equivalent to

```
(CONS item1 (CONS item2 ... (CONS item2 NIL)))
```

LOTSOF —— function

(LOTSOF **item** ...)

LOTSOF returns an infinite list of its arguments.

```
(LOTSOF 2 'A '(8 . 9))
```

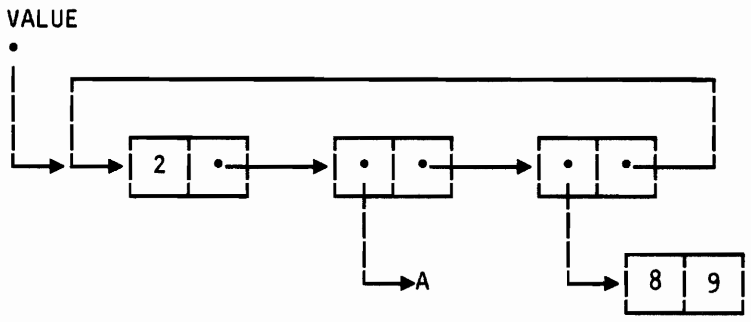


Figure 16. Result of LOTSOF.

CONS —— built-in function

(CONS **item list**)

When its second argument is a list, CONS can be considered a list creating operator. It is used to add new items to the beginning of a list. See also page 109.

APPEND —— function

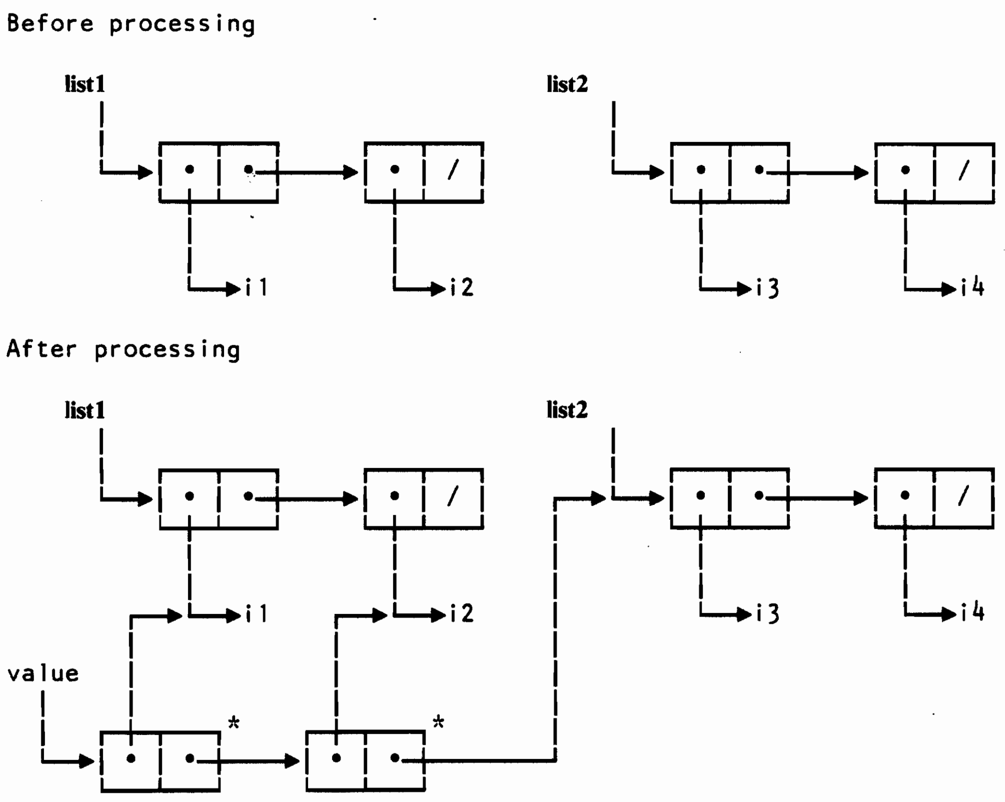
CONC —— function

(APPEND [**list1** ... **listn**])

If APPEND is invoked with no arguments, the value is NIL. If APPEND is invoked with one argument, the value is precisely that argument. If APPEND is invoked with several arguments, and all but the last are atoms (ie. not pairs, or equivalently, empty lists), the value is again the last argument.

In all other cases, the value of an invokation of APPEND is a list which has a length equal to the sum of the lengths of the arguments. The pairs that form the top-level list structure of all but the last argument are copied, and the last argument becomes the CDR of the last copied pair. If any argument but the last is a circular list, an error is signalled.

There is no copying of the structure below the top level, which would be accessed by descending the elements of **list1**. Consider (APPEND '(i1 i2) '(i3 i4)). The structure before, and after the call would be:



(Note, the pairs marked by * are newly created.)

Figure 17. Result of APPEND.

APPEND2 —— function

(APPEND2 **list1** &list2)

A special case of APPEND with exactly two arguments.

CONCAT —— function

(CONCAT [**item1** ... **itemn**)

If **item1** is NIL or a pair, the effect of CONCAT is identical to the effect of APPEND. If **item1** is a vector or a specific vector, then CONCAT builds a new vector that holds all the elements of the arguments.

REVERSE —— function

(REVERSE **list**)

This operator returns as its value a new, top-level copy of the list **list** where the elements of this new list are in the inverse order of their occurrence in **list**.

```
(REVERSE '(1 2 3 4))
Value = (4 3 2 1)
```

REMOVE —— *function*

(REMOVE **list item** { **arg** })

If **arg** is omitted, this function copies the top-level list structure of **list** until a list element EQUAL to **item** is found. That element is skipped, and the rest of **list** becomes the CDR of the last copied pair. In effect, this operator builds a list equivalent to **list** with the first occurrence of **item** deleted.

If **arg** is a positive integer, contine the above process to remove the first **arg** occurrences of **item** If **arg** is anything else, remove all occurrences of **item**.

REMOVEQ —— *function*

(REMOVEQ **item list** { **arg** })

Equivalent to REMOVE, but uses EQ rather than EQUAL.

VEC2LIST —— *function*

(VEC2LIST **vec**)

This operator constructs a new list, containing the elements of the vector, **vec**. See page 126.

UNION —— *function*

(UNION **list1 list2**)

This operator constructs a new list contains all the elements appearing in either of the lists **list1** and **list2**. Each element appears only once in the value list. MEMBER, which in turn uses EQUAL, is used to detect whether an element appears in a list. Any elements in the union found in **list1** will precede those found only in **list2**. The order will match that of the lists of origin. Thus:

```
(UNION '(T H I S I S A T E S T) '(O F T H E U N I O N))
```

results in

```
(T H I S A E O F U N)
```

UNIONQ —— *function*

(UNIONQ **list1 list2**)

This operator differs from UNION only in using MEMQ, and hence EQ, to test for membership in its arguments.

INTERSECTION —— *function*

(INTERSECTION **list1 list2**

Constructs a new list containing only those elements appearing (at the top level) in both **list1** and **list2**. MEMBER, which in turn uses EQUAL, is used to detect whether an element appears in a list. The elements in this new list are in the same order as their occurrence in **list1**. If either argument is not a pair, it is treated as if it were the only element in a list of length one.

INTERSECTIONQ —— *function*

(INTERSECTIONQ **list1 list2**)

This operator differs from INTERSECTION only in using MEMQ, and hence EQ, to test for membership in its arguments.

SETDIFFERENCE —— function

(SETDIFFERENCE **list1 list2**)

Constructs a new list containing only those elements appearing (at the top level) in **list1** but not in **list2**. MEMBER, which in turn uses EQUAL, is used to detect whether an element appears in a list. The elements in this new list are in the same order as their occurrence in **list1**. If either argument is not a pair, it is treated as if it were the only element in a list of length one.

SETDIFFERENCEQ —— function

(SETDIFFERENCEQ **list1 list2**)

This operator differs from SETDIFFERENCE only in using MEMQ, and hence EQ, to test for membership in its arguments.

MTON —— function

(MTON **sint1 sint2**)

This operator returns a list of the integers from **sint1** to **sint2**, inclusive. It does not check its arguments, and if either is not a small integer, or if **sint2** is less than **sint1** the results will be unpredictable, and possibly fatal.

14.2 Accessing

CA[D...]R —— function and compile-time macro

(CA[D...]R **list**)

These operators return elements of a list. CAR returns the first, CADR the second, CADDR the third, and CADDR the fourth. See page 110.

QCA[D...]R —— function and compile-time macro

(QCA[D...]R **list**)

The non-checking versions of the preceding operators. See page 110.

CD...R —— function and compile-time macro

(CD...R **list**)

These operations return the "tail" of their argument. CDR gives the list, less its first element, CDDR, less its first and second, etc. See page 110.

QCD...R —— function and compile-time macro

(QCD...R **list**)

The non-checking versions of the preceding operators. See page 110.

ELT —— function and compile-time macro

(ELT **list sint**)

ELT returns the **sint** element of **list** See page 125.

LAST —— function

(LAST **list**)

Returns as value the last element of **list** (i.e. the CAR of the last pair comprising **list**).

If **list** is a non-pair or is circular an error is signalled.

LASTPAIR —— *function*

(LASTPAIR **list**)

This operator returns as its value the last pair forming the list **list**. If **list** is a non-pair or is circular an error is signalled.

```
(LASTPAIR '(1 2 3 4))
Value = (4)
```

This value (as is true of the CD...R forms) is not a copy. Updating operations applied to it will effect the original list.

14.3 Searching

MEMBER —— *function*

(MEMBER **item list**)

This operator searches the list **list** for the object **item**, using EQUAL testing for identity. It **item** is not found, or if **list** is not a pair, the value of MEMBER is NIL. If **item** is found, the value of MEMBER is that portion of **list** beginning with **item**. **list** is searched on the top level only.

UMEMBER —— *function*

(UMEMBER **item list**)

This operator is similar to MEMBER, except that it uses UEQUAL testing for identity instead of EQUAL testing.

MEMQ —— *function*

(MEMQ **item list**)

This operator is similar to MEMBER, except that it uses EQ testing for identity instead of EQUAL testing.

QMEMQ —— *macro*

(QMEMQ **item list**)

When compiled, this operator produces inline code equivalent to the MEMQ operator. This operator does not check for circular lists, and will loop indefinitly.

TAILP —— *function*

(TAILP **item list**)

This operator searches **list** for a CDR EQ to **item**. If such is found the value is non-NIL, otherwise it is NIL.

The case of **item** EQ to **list** is considered a success. Since EQ is used,

```
(TAILP '(3 4) '(1 2 3 4))
Value = NIL
```

will occur, as the two lists are separate. On the other hand,

```
(SETQ X '(3 4))
Value = (3 4)
(SETQ Y (APPEND '(1 2) X))
Value = (1 2 3 4)
(TAILP X Y)
Value = non-NIL
```

because of the sharing in the value of APPEND.

ASSOC —— function

(ASSOC **item a-list**)

a-list is a list of pairs, ((name1 . value1) (name2 . value2) ...). ASSOC compares **item** with name1, then name2, ..., using EQUAL to perform the comparison. Any elements of **list** which are not pairs are skipped. If **a-list** is not a list, or if **item** is not found in **a-list**, the value of ASSOC is NIL. Otherwise, the value of ASSOC is the first pair (name . value) such that (EQUAL **item** 'name) is true. If there are elements of **a-list** which are not pairs, they are skipped and the next element of **a-list** is examined.

UASSOC —— function

(UASSOC **item a-list**)

This operator is identical to ASSOC except that UEQUAL, rather than EQUAL, is used for the comparison of **item** with the **a-list**.

ASSQ —— function

(ASSQ **item a-list**)

This operator is similar to ASSOC, except it uses EQ rather than EQUAL to compare **item** with the CARs of elements in **a-list**.

QASSQ —— macro

(QASSQ **item list**)

When compiled, this operator produces inline code equivalent to the ASSQ operator. This operator does not check for circular lists, and will loop indefinitly.

SASSOC —— function

(SASSOC **item a-list app-ob**)

This operator is similar to ASSQ, but requires three arguments and if **item** is not matched, it returns the result of applying **app-ob** to no arguments, instead of the NIL value returned by ASSOC.

GET —— function

(GET **list item**)

GET returns the value of a name-value pair of an association list.

See page 98.

QGET —— macro

(QGET **list item**)

When compiled, this operator produces inline code equivalent to the GET operator. This operator does not check for cyclic lists, and may loop indefinitely.

14.4 Searching and Updating

ADDTOLIST —— function

(ADDTOLIST **list item**)

If **list** is not a pair, the value returned is (LIST **item**). Otherwise this operator searches **list** for an instance of **item**, using EQ. If no instance of **item** is found, **item** is added to the tail of the list, using RPLACD. In effect,

```
  (NCONC list (LIST item))
```

The value returned is the list **list**, whether or not **item** was initially present.

PUT —— *function*

(PUT **list item1 item2**)

PUT is used to update or add a name-value pair to an association list. See page 98.

REMPROP —— *function*

(REMPROP **list item**)

REMPROP removes a name-value pair from an association list. See page undefined

14.5 Updating

RPLACA —— *built-in function*

(RPLACA **list item**)

When its first argument is interpreted as a list, RPLACA replaces the first element of that list.

```
(SETQ X '(1 2 3 4))
Value = (1 2 3 4)
(RPLACA X '10)
Value = (10 2 3 4)
X
Value = (10 2 3 4)
```

See page 111.

QRPLACA —— *macro*

(QRPLACA **list item**)

Equivalent to RPLACA, with no type checking of arguments. See page 111.

RPLACD —— *built-in function*

(RPLACD **pair item**)

When its first argument is interpreted as a list, RPLACD replaces the tail of that list.

```
(SETQ X '(1 2 3 4))
Value = (1 2 3 4)
(RPLACD X '(20 30 40))
Value = (1 20 30 40)
X
Value = (1 20 30 40)
```

See page 111.

QRPLACD —— *macro*

(QRPLACD **pair item**)

Equivalent to RPLACD, with no type checking of arguments. See page 111.

SETELT —— *function and compile-time macro*

(SETELT **list sint item**)

SETELT updates the **sint** element of **list**. See page undefined.

NCONC —— *function*

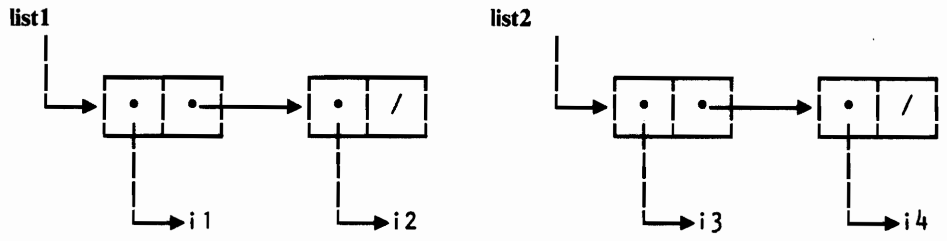
(NCONC [**list1** ... **listn**])

If NCONC is invoked with no arguments, the value is NIL. If NCONC is invoked with one argument, the value is precisely that argument. If NCONC is invoked with several arguments, and all but the last are atoms (ie. not pairs, or equivalently, empty lists), the value is again the last argument.

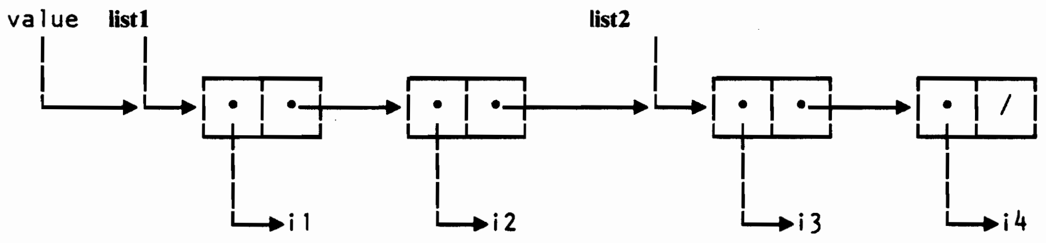
In all other cases, the value of an invokation of NCONC is EQ to the first argument that is a pair. Each argument that is a pair is scanned up to the final pair, and the next argument that is a pair becomes the CDR of that pair. If any argument but the last is a circular list, an error is signalled.

Consider (NCONC '(i1 i2) '(i3 i4)). The structure before, and after the call would be:

Before processing



After processing



(Note, no new pairs are created.)

Figure 18. Effect of NCONC

NCONC2 —— function

(NCONC2 **list1** **list2**)

A special case of NCONC with exactly two arguments.

NREVERSE —— function

(NREVERSE **list**)

This function shuffles the CDR components of the pairs making up **list** so that the CDR of the last pair in **list** points to the next-to-last pair, et cetera. The CDR of the first pair is made NIL, and the value of NREVERSE is the last pair of **list**, which is now the first pair of a list containing exactly the same elements as **list**, but in reversed order. See also REVERSE, page 114, which constructs a new list in reversed order instead of reusing the pairs in the original list.

Consider (NREVERSE '(i1 i2 i3 i4)). The structure before, and after the call would be:

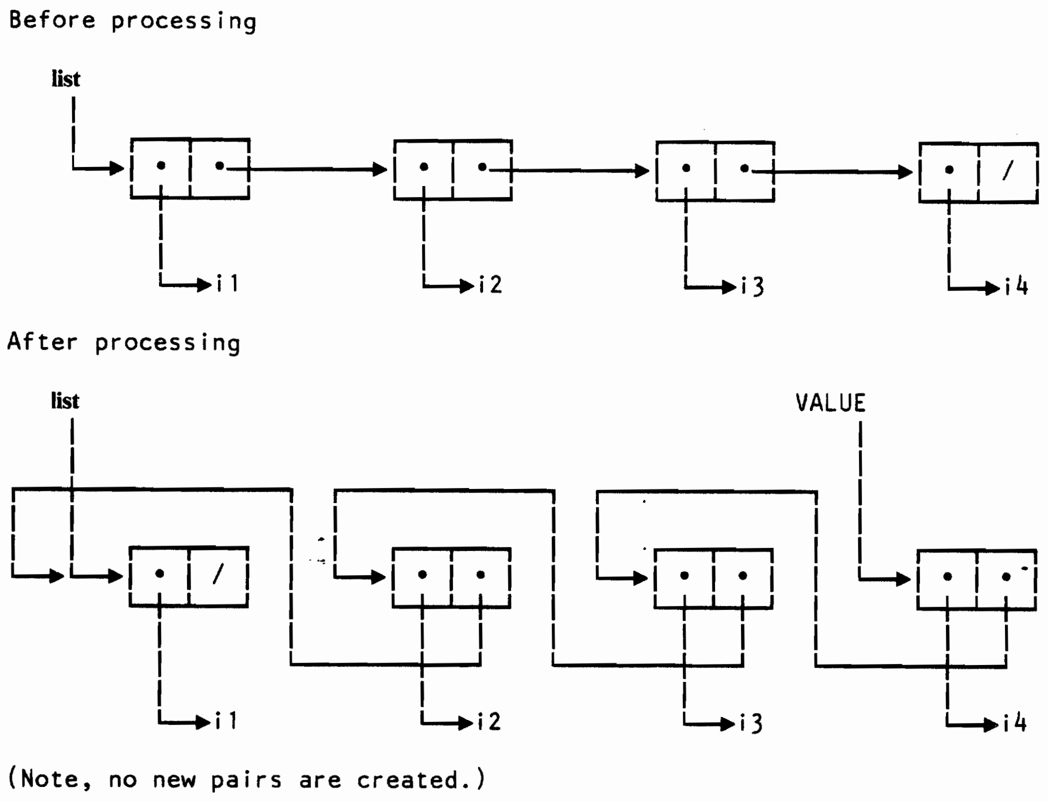


Figure 19. Effect of NREVERSE

NREMOVE —— function

(NREMOVE **list item** { **num** })

Uses EQUAL to search **list** for the first **num** occurrences of **item**. If **item** is not found, or if **list** is atomic, returns **list** as its value. If **item** is found, it is removed up to **num** times.

If the first element of the list is not EQUAL to **item**, the list will be updated, so that **item** will be removed from any objects containing **list** If the first element of the list is EQUAL to **item** then NREMOVE will return the result of NREMOVE applied to the CDR of the list, with a count of one less.

If the **num** is not given as an argument, it defaults to one, hence only the first instance of **item** will be removed. If the third argument is NIL, all instances of **item** will be replaced.

NREMOVEQ —— function

(NREMOVEQ **item list** { **num** })

Equivalent to NREMOVE, but uses EQ rather than EQUAL.

14.6 Miscellaneous

LENGTH —— function

(LENGTH **list**)

If **list** is a pair, LENGTH returns the number of elements in the list beginning with that pair. If **list** is not a pair, the value of LENGTH is zero.

If **list** is a circular list an error break is taken.

QLENGTH —— *macro*

(QLENGTH **list**)

When compiled, this operator produces inline code equivalent to the LENGTH operator. This operator does not check for circular lists, and may loop indefinitely.

SIZE —— *function*

(SIZE **list**)

SIZE returns the number of elements in **list**. See page 124.

SORT —— *function*

(SORT **list**)

SORT returns a newly CONSed list, containing the element of **list**, ordered by the operator SORTGREATERP. SORTGREATERP is initially defined as equivalent to GGREATERP, page undefined, but may be redefined at the users pleasure.

The quicksort algorithm is used.

SORTBY —— *function*

(SORTBY **app-ob list**)

This operator sort its second argument, **list**. Where SORT compares the elements of the list, SORTBY compares the values of (**app-ob** element). Thus where

```
(SORT '((2 g) (1 n) (3 a)))
Value = ((1 n) (2 g) (3 a))
```

SORTBY produces

```
(SORTBY 'CADR '((2 g) (1 n) (3 a)))
Value = ((3 a) (2 g) (1 n))
```

15.0 Operations on Vectors

The set of data objects grouped under the title vector includes bpis although they are mentioned only once. Those operators which effect only character vectors or bit vectors are defined in their own section.

15.1 Creation

MAKE-VEC ——function

(MAKE-VEC **sint**)

Returns as value a new vector containing **sint** elements. The initial value of each element is NIL. This is the basic allocating operator for vectors that are capable of holding arbitrary objects.

MAKE-RVEC ——function

(MAKE-RVEC **sint**)

Allocates and returns as value a real vector containing **sint** elements. Each of the floating point values (elements) are initialized to zero.

MAKE-IVEC ——function

(MAKE-IVEC **sint**)

Allocates and returns as value an integer vector containing **sint** elements. The elements of the allocated vector are not initialized. See also GETZEROVEC.

GETZEROVEC ——function

(GETZEROVEC **sint**)

Like MAKE-IVEC, except the elements of the integer vector are initialized to zero.

VECTOR ——function and compile-time macro

(VECTOR [**item** ...])

This operator is the vector analogue of LIST. Its value is a vector containing its operands.

```
(VECTOR 1 2 3 4)
Value = <1 2 3 4>
```

LIST2VEC ——function

(LIST2VEC **list**)

This operator constructs a new vector from the elements of **list**. If **list** is non-pair, the value of LIST2VEC is a vec with zero elements. If **list** is a circular list, the operator loops. Otherwise, the value is a vector of the form:

```
<(CAR LIST) (CADR LIST) ... (CAD.R LIST)>
```

LIST2RVEC ——function

(LIST2RVEC **list**)

This operator is similar to LIST2VEC but for the fact that the resulting vector is a vector of floating point (real) numbers. The elements of **list** may or may not already be real numbers. If they are not real, they are floated. If any element of **list** is not a number,

STANDARD VECTOR OPERATORS					
A tabular index to primitive vector operators					
Type of Vector	Predicate	Value of element	Specific Selection Function	Specific Updating Function	Allocating Function
VEC	VECP	Anything	ELT	SETELT	MAKE-VEC
Character	CVECP	Identifier	ELT	SETELT	MAKE-CVEC
Bit	BVECP	NIL or ¬NIL	ELT	SETELT	MAKE-BVEC
Word	IVECP	Integer	ELT	SETELT	MAKE-IVEC
Real	RVECP	Real	ELT	SETELT	MAKE-RVEC
Analogous operators for pairs are listed below for comparison					
Pair	PAIRP	.	CAR CDR	RPLACA RPLACD	CONS
List	LISTP	Anything	ELT	SETELT	LIST

Figure 20. Standard Vector Operators

or cannot be converted into a floating point number, the INT2RNUM operator which is called by LIST2RVEC will signal an error.

LIST2IVEC —— function

(LIST2IVEC **list**)

This operator is similar to LIST2VEC but for the facts that its value is an integer vector and the elements of **list** must be integers within the range acceptable for integer vectors. This range is $2^{31}-1 > \text{value} \geq -2^{31}$.

15.2 Accessing

SIZE —— function

(SIZE **vec**)

SIZE returns as its value the number of elements in its argument -- that is, one more than the maximum valid index which may be used to address an element of this vector. SIZE may also be applied to lists, in which case it performs exactly as does LENGTH. See page 122. If SIZE is given an argument other than a pair or vector, it returns the value zero.

Note that string vectors (either character or bit) may have a capacity larger than the number of elements currently in the vector.

> You may note that while SIZE (and LENGTH) are lumped as "miscellaneous" operators when applied to lists, they are included among the access operators for vectors. This is because the number of element is a vector is explicitly recorded in the data structure, while the number of elements in a list can only be found by counting.

OR

(SIZE **list**)

When applied to lists, SIZE is equivalent to LENGTH.

MAXINDEX —— function

(MAXINDEX **vec**)

Returns as value the maximum allowed index for the given vector. This operator is defined as (SUB1 (SIZE **vec**)), so while its principal use concerns vectors, it could be applied to other objects as well.

QVMAXINDEX —— function and compile-time macro

(QVMAXINDEX **vec**)

This is the non-checking counterpart of MAXINDEX. If its operand is a vec it returns the desired value, otherwise its value is unpredictable.

> Note, that QVMAXINDEX will produce the correct value when applied to an integer vector.

QVSIZE —— function and compile-time macro

(QVSIZE **vec**)

This is the non-checking counterpart of SIZE. If its operand is a vec it returns the desired value, otherwise its value is unpredictable.

> Note, that QVSIZE will produce the correct value when applied to an integer vector.

ELT —— function and compile-time macro

(ELT **item sint**)

This is the general selection operator for vectors and lists. Its value is the **sint** element of **item**, where the type of object returned by ELT is indicated in Figure 20 on page 124 for various types of **items**.

If **sint** is not within the bounds of **item**, an error is indicated. If **item** is not a vector or a list, an error is indicated.

The first element of a vector or a list is selected by using zero as the **sint** value.

If **item** is a vector of reals, ELT will allocate a new real number into which the value of the accessed element is copied, and return this new real to the caller. This is necessary (although not very efficient) because pointers pointing inside a vector are not allowed (they confuse the garbage collector). Thus, if a collection of real numbers are to be assembled into a vector, it is better to have a vector when access to these reals is made on an individual basis using ELT. The vector of reals exists for applications where the user has implemented arithmetic processes requiring the contiguous storage of real data in order to evaluate efficiently.

If **item** is an integer vector, ELT may have to build a new large integer and return it as the value of ELT for certain values in the integer vector. Any value within the range of a small integer will be returned as a small integer, and will not require allocation of heap space. Values outside the range of small integers must be converted by ELT into large integers.

If **item** is a character vector ELT will return one of the 256 identifiers with one byte pnames.

For bit strings ELT returns a value of NIL or ¬NIL. ELT is the only access operator for bit strings.

OR

(ELT **list sint**)

The value of ELT is the **sint** element of **list**, where the first element is designated by zero (0), the second by one (1), etc.

If **sint** is not within the bounds of **list**, an error is signalled. If **list** is not a vector or a list, an error is indicated.

Note: ELT applied to NIL will always produce a bounds error, as NIL is interpreted as the empty list.

QVELT —— macro

(QVELT **vector sint**)

This is the non-checking counterpart of ELT. It is only defined when its first operand, **vector** is a vector, and its second operand, **sint**, is a small integer and is in the correct range. In that case its value is the same as ELTs. Otherwise its value is unpredictable, and possible fatal.

VEC2LIST —— function

(VEC2LIST **vec**)

This operator constructs a new list, containing the elements of the vector, **vec**. The elements of the resulting list will correspond to the values of ELT for the specific vector from which they are drawn.

15.3 Searching

VMEMQ —— function

(VMEMQ **item vec**)

Searches **vec** for an element EQ to **item**. If such an element is found the value of VMEMQ is its index. Otherwise the value is NIL.

```
(SETQ V '<A 3 8 B X>)
Value = <A 3 8 B X>
(VMEMQ 'B V)
Value = 3
(VMEMQ 'L V)
Value = NIL
```

15.4 Updating

SETELT —— function and compile-time macro

(SETELT **item1 sint item2**)

This is the inverse operator of ELT -- it updates the **sint** element of **item1** to be **item2**. The nature of **item2** for various types of **item1** is indicated in Figure 20 on page 124.

SETELT will signal an error if **item1** is not updatable, if **sint** is out of range, or if **item2** is not compatible with the type of **item1**.

SETELT may be used to update the **sint** element of a list.

The value of SETELT is **item2**, the value used in updating the specified object.

```
(SETELT LIST 3 VALUE)
```

is equivalent to:

```
(RPLACA (CDR (CDR (CDR LIST))) VALUE)
```

For character strings, **item** must be an identifier. For bit strings it may have any value, with NIL and anything else being distinguished.

QSETVELT —— macro

(QSETVELT **vector sint item**)

This is the non-checking counterpart of SETELT. It is only defined when its first operand, **vec** is a vector, and its second operand, **sint**, is a small integer and is in the correct range. In that case its value is the same as SETELTs. Otherwise its value is unpredictable, and possible fatal.

MOVEVEC —— *function*

(MOVEVEC **vec1 vec2**)

Copies the contents of vector **vec1** into vector **vec2**. Both arguments must have the same capacity, and they must be both vectors or both binary (character, bit, real or integer) vectors.

16.0 Operations on Character and Bit Vectors

Two kinds of vectors, character vectors and bit vectors, permit operations not allowed on other vectors. Unless otherwise indicated, the following operators apply to character vectors.

16.1 Creation

MAKE-CVEC —— function

(MAKE-CVEC **sint**)

Creates a character vector with a capacity of at least **sint** characters. The new vector is returned as the value of MAKE-CVEC. Vectors are allocated in increments of fullwords: for a character vector, the first word includes only the first character of the string, prefaced by a 3-byte current length field. Therefore, the actual capacity of the vector is defined by

(((**sint** + 6) / 4) * 4) - 3 characters.	
sint	Maximum capacity of allocated string
1	1
2-5	5
6-9	9
.	.

Figure 21. Character string allocation

Zero or negative numbers are invalid values for **sint** and will cause an error to be signalled. The character string returned by MAKE-CVEC is initialized to the null string, that is, its contents length is zero, and it prints as "".

MAKE-FULL-CVEC —— function

(MAKE-FULL-CVEC **sint** [{**id** | **cvec** | **sint**}])

Similar to MAKE-CVEC in that a new character vector is allocated and returned as the value of MAKE-FULL-CVEC. The new string, however, contains **sint** instead of zero characters. The **id** argument is optional. If it is specified as an identifier, the new string will be initialized so that each character is the initial letter of the P-name of **id**. If it is a small-integer, the low order eight bits become the fill character. If it is a string, the leftmost character is used. If **id** is not specified, the string will be initialized to binary zero characters.

MAKE-BVEC —— function

(MAKE-BVEC **sint**)

Allocates a bit vector with a capacity of at least **sint** bits. The new vector is returned as the value of MAKE-BVEC. Vectors are allocated in increments of full words: for a bit vector, the first word includes only the first 8 bits of the string, prefaced by a 3-byte current length field. Therefore, the actual capacity of the vector is defined by

(((**sint** + (31 +(3*8))) / 32) * 32) - 24 bits	
sint	Maximum capacity of allocated string
1-8	8
9-40	40
41-72	72
.	.

Figure 22. Bit string allocation

CONCAT —— function

(CONCAT **cvec** ...)

When its first argument is a character vector, this operator returns as its value a new vector made by concatenating all of the vectors **cvec**, ... Because it is frequently used, a special case has been made so that **cvecn** may be either a character vector or a stored identifier (not a gensym). In this case, the print name of the identifier is concatenated into the result string. If the first argument to the operator is a character vector and if **cvecn** any of the other arguments are neither a character vector or stored identifier, an error is signalled.

PRIN2CVEC —— function

(PRIN2CVEC **item**)

This operator creates a character string containing the external form of **item**.

For example, to obtain the character equivalent of an integer, one can write

```
  INTEGER
  Value = 1352
  (PRIN2CVEC INTEGER)
  Value = "1352"
```

item may be any value.

CVEC2BVEC —— function

(CVEC2BVEC **cvec**)

This operator copies its operand, **cvec**, and converts it into a bit vector. The contents of the vector are unchanged, only the type of the pointer to it, and the contents length are modified.

This operator allows direct access to the bit pattern of a character string.

BVEC2CVEC —— function

(BVEC2CVEC **bvec**)

This operator copies its operand, **bvec**, and converts it into a character vector. The contents of the vector are unchanged, only the type of the pointer to it, and the contents length are modified.

PNAME —— function

(PNAME **id**)

PNAME creates a character vector which is a copy of the print name of **id**.

ANDBIT —— *function*

(ANDBIT **bvec** ...)

ANDs **bvec** ... and returns as value the resultant vector. None of the argument strings is changed by the operation.

An error is indicated if any operand is not a bit vector or if the vectors are not equal in length.

ORBIT —— *function*

(ORBIT **bvec** ...)

ORs **bvec** ... and returns as value the resultant vector. None of the argument vectors is changed by the operation.

An error is indicated if any operand is not a bit vector or if the vectors are not equal in length.

XORBIT —— *function*

(XORBIT **bvec** ...)

Exclusive ORs **bvec** ... and returns as value the resultant vector. None of the argument vectors is changed by the operation.

An error is indicated if any operand is not a bit vector or if the vectors are not equal in length.

MAKETRTTABLE —— *function*

(MAKETRTTABLE **table-definition item)**

This operator creates a S/370 translate-and-test table. Operands are a collection of characters-of-interest, **table-definition** and an inversion flag, **item**. The first operand can be either a character vector or a list. If a character vector it is converted to a list of identifiers using VEC2LIST.

The list elements may be small integers, character vectors or non-gensym identifiers, or pairs with one of the above as CAR and a small-integer as CDR. The table, a newly created character vector of length 256, is initialize with x00 (if **item** is NIL), or xFF (if **item** is non-NIL). Then the positions corresponding to the list elements (or their CARs, if pairs) are set to xFF or x00 (opposite of the initial value), or to the CDR value for pairs.

The resulting vector is a suitable operand for STRPOS, STRPOSL, and STRTRT.

MAKESTRING —— *function and compile-time macro*

(MAKESTRING **cvec)**

cvec must be a constant which is a character vector. It will not be evaluated. When used in compiled code, the **cvec** will be placed in the bpi, not in the heap. This is done to free the more valuable, garbage collected, space. This operator returns a copy of its argument.
-- FOR THE EXPERT LISP/VM USER

16.2 Accessing

SIZE —— *function*

(SIZE **bORcvec)**

This operator returns as its value the current length of the character or bit vector **bORcvec**

As with non-string vectors, the size of a string is an intrinsic part of the object. Again, note that this is the size of the contents of the string, not its capacity.

QCSIZE —— *function and compile-time macro*

(QCSIZE **cvec**)

This is a non-checking counter part of SIZE, which assumes that its argument will be a character vector. If **cvec** is a character vector, it returns its contents length. If **cvec** is not a character vector its result is unpredictable.

CAPACITY —— *function*

(CAPACITY **bORcvec**)

This function returns the maximup possible size of a bit or character vector.

QLENGTHCODE —— *function and compile-time macro*

(QLENGTHCODE **bORcvec**)

This is the non-checking counter part of CAPACITY for character vectors. For bit vectors it returns the number of potential bytes in the vector, in other words the number of bits divided by eight.

ELT —— *function and compile-time macro*

(ELT **cvec sint**)

Returns as value the character object (identifier whose print name is a single character) corresponding to the **sint**th element of the character vector **cvec**. An error is signaled if **sint** is negative or exceeds the current length of the string **cvec**.

ELT —— *function and compile-time macro*

(ELT **bvec sint**)

ELT may be thought of as a predicate returning nil or ¬ nil depending on the **sint** element of the **bvec**

SUBSTRING —— *function*

(SUBSTRING **cvec sint1 sint2**)

This macro returns a copy of part (or all) of **cvec**. The returned value starts with the **sint1** (index) character of **cvec** (remember, index zero is the first character) and is **sint2** (length) characters long. If **sint2** is specified as (), that designates the end of the string.

STRING2ID-N —— *function*

(STRING2ID-N **cvec sint**)

This operator treats its first operand, **cvec**, as a collection of tokens, where the tokens are substrings, separated by one or more blanks. (Leading blanks are ignored.) The **sint**th token is extracted and INTERNed (See page 97) and the resulting identifier is returned as the value. If there are less than **sint** tokens in **cvec**, the value is zero (0).
-- FOR THE EXPERT LISP/VM USER

STRING2PINT-N —— *function*

(STRING2PINT-N **cvec sint**)

This operator treats its first operand, **cvec**, as a collection of tokens, where the tokens are substrings, separated by one or more blanks. (Leading blanks are ignored.) The **sint**th token is extracted and, if it consists wholly of digits, the corresponding positive

small integer is returned as the value. If there are less than **sint** tokens in **cvec** or the **sint**th token contains non-digit characters, the value is NIL.
-- FOR THE EXPERT LISP/VM USER

VEC2LIST —— function

(VEC2LIST **bORcvec**)

VEC2LIST converts a vector (including character or bit vectors) to a list.

16.3 Searching

STRPOS —— function

(STRPOS **cvec1 cvec2 sint item**)

This operator searches **cvec2** for a substring, **cvec1**, which may contain don't care characters. **sint** is the an index, indicating the starting position for the search. **item** is NIL or the don't care character. An error is signalled if **item** is not a small integer, identifier, string or NIL.

The value is NIL if the substring is not found, and the index of the first character of the substring if found.

```
X
Value = "This is a test of those operations."
(STRPOS "h*s" X 0 "*")
Value = 1
(STRPOS "h*s" X 2 "*")
Value = 19
(STRPOS "h*s" X 20 "*")
Value = NIL
```

STRPOSL —— function

(STRPOSL **table cvec sint item**)

This operand searches a character vector for a character from a given set. **table** should be a translate-and-test table, as built by MAKETRTTABLE, or a valid first argument for that function. **cvec** is the vector to be searched. **sint** is an index, indicating the point in the vector at which the search is to start. **item** is a flag, if NIL search ends at first character specified by **table**, otherwise search ends at first character not specified by **table**. The value is NIL, if the search fails, or the index of the found character.

```
X
Value = "This is a test of those operations."
(STRPOSL "aeiou" X 0 'T)
Value = 2
(STRPOSL "aeiou" X 0 NIL)
Value = 0
(STRPOSL "aeiou" X 3 'T)
Value = 5
(STRPOSL "aeiou" X 6 'T)
Value = 8
```

STRTRT —— function

(STRTRT **table cvec pair**)

This operator searches **cvec** for a specified character or characters. It is similar to STRPOSL, but without the negation flag, and the position argument (**pair**) is an updatable pair, whose CAR is the starting position.

If the desired character is not found the value is NIL. If it is found, the value of the pair is updated, with its CAR being set to the position of the found character and its CDR being set to the **table** entry. If **pair** is not a pair, one is created and initialized with zero.

```
X
Value = "This is a test of those operations."
(SETQ Y (CONS 0 0))
Value = (0 . 0)
(STRPOSL '((a . 1) (e . 2) (i . 3) (o . 4) (u . 5)) X Y)
Value = (2 . 3)
(RPLACA Y (ADD1 (CAR Y)))
Value = (3 . 3)
(STRPOSL '((a . 1) (e . 2) (i . 3) (o . 4) (u . 5)) X Y)
Value = (5 . 3)
(RPLACA Y (ADD1 (CAR Y)))
Value = (6 . 3)
(STRPOSL '((a . 1) (e . 2) (i . 3) (o . 4) (u . 5)) X Y)
Value = (8 . 1)
```

Note that in the above example it would be move efficient to call MAKETRTTABLE once with the first operand, and then pass that as the value to STRPOSL. The test in STRPOSL to see if a ready made TRT table has been provided is a heuristic one. If the first operand is a string containing 256 characters, it is used as a TRT table, otherwise it is passed to MAKETRTTABLE.

16.4 Updating operators

SETSIZE —— function and compile-time macro

(SETSIZE **bORcvec sint**)

The length of the character or bit vector **bORcvec** is updated to be the value **sint**. An error is indicated if **bORcvec** is not a character or bit string, or if the value of **sint** exceeds the maximum potential length of **bORcvec**. The value of SETSIZE is the vector **bORcvec** with its new length. If the size is increased, unspecified values will be added at the end of the vector.

QSETSIZE —— macro

(QSETSIZE **cvec sint**)

This is the non-checking counterpart to SETSIZE for character vectors. If **cvec** is a character string, and **sint** is a small integer ≤ the capacity of **cvec**, then the result is equivalent to SETSIZE. Otherwise the result is unpredictable, and possible fatal.

TRIMSTRING —— function

(TRIMSTRING **cvec**)

This operator updates the capacity of its argument, to produce a character vector with the minimum capacity which will hold the current contents. The portion of the vector which is discarded (if any) becomes inaccessible. There are no operations to increase the capacity of a vector in place.
-- FOR THE EXPERT LISP/VM USER

SUFFIX —— function and compile-time macro

(SUFFIX **id cvec**)

Updates the character vector **cvec** by adding the first character of the print name of the stored identifier **id** to the end of the vector. **id** is usually a character object, but may be any stored identifier. This function increments the length of the character vector by one, or causes an error to be signalled if there is not sufficient space in the vector **cvec** for the additional character.

SETELT —— *function and compile-time macro*

(SETELT **bORcvec sint item**)

SETELT updates the **sint** element of **bORcvec** to be **item**. If **bORcvec** is a character vector, **item** is assumed to be an id with a one character print name, if not, the first character of the print name is used. See page undefined.

RPLACSTR —— *function and compile-time macro*

(RPLACSTR **cvec1 sint1 sint2 cvec2** [**sint3** [**sint4**]])

This is a generalized string modification routine. It can replace any part of **cvec1** with any part of **cvec2**, making any n necessary adjustment in the length of **cvec1** because the replacement characters from **cvec2** are greater or fewer than the characters being replaced in **cvec1**. Furthermore, it can insert **cvec2**, or some specified substring of **cvec2**, into **cvec1**.

cvec1 must be a character vector, else an error is indicated. **cvec2** may be either a character vector, or it may be a stored identifier (not a gensym). In the latter case, the print name of the identifier (which is a character string) will be used as **cvec2**, and **sint3** and **sint4** refer to this string. If **cvec1** or **cvec2** are not ad described, an error is indicated.

sint1 specifies the index of the first character in **cvec1** to be replaced. **sint2** specifies the number of consecutive characters, beginning with the **sint1** character, to be replaced. **sint3** and **sint4** specify the location and number of characters from **cvec2** which are to replace the designated characters in **cvec1**.

sint1, **sint3**, **sint2** and **sint4** may be either integer values or NIL; an error is indicated if they are not.

In general, an INDEX may vary from zero to the current length-1. If NIL is specified for an index, the numeric value zero is used. If **sint1** is equal to the current length, **sint2** must be zero: by use of this convention, **cvec2** can be appended to the end of **cvec1**.

If zero is specified for **sint2**, **cvec2** is inserted in **cvec1** before the position specified by **sint1**. If NIL is specified for a length, all of the characters from the related index value to the end of the string are used. In effect, using NIL for the value of LENx is an efficient way of specifying the value:

```
(DIFFERENCE (SIZE STRx) INDEXx)
```

sint4 and **sint3** are optional arguments. If they are not specified, a value of NIL will be assumed.

> Whenever possible, RPLACSTR will update the original **cvec1**, and return as its value the updated string. However, if **sint4** is greater than **sint2**, it is possible that **cvec1** does not have sufficient space for the result string. In this case, a new string is constructed and this new string is returned as the value of RPLACSTR.
>
> The user may test whether the updated string is the original **cvec1** or a copy by an expression such as:
>
> ```
> (EQ cvec1 (SETQ TEMP (RPLACSTR cvec1 ... )))
> ```
>
> which will be true if **cvec1** has been updated in place, and false if a new string had to be created. The purpose of the SETQ operation is to preserve the value of RPLACSTR in case a new string was created.

DOWNCASE —— *function*

(DOWNCASE **cvec**)

DOWNCASE returns the lower-case equivalent of a character vector. This function also accepts a list of charcter vectors. See page 97.

UPCASE —— function

(UPCASE **cvec**)

UPCASE returns the upper case equivalent of a character vector. This function also accepts a list of character strings. See page 97.

16.5 Comparing operators

CGREATERP —— function

(CGREATERP **cvec1 cvec2**)

This functions compares two character vectors and returns true if **cvec1** is greater than **bORcvec2**, otherwise it returns NIL. The comparison is done using the S/370 CLCL instruction.

If the two strings are of unequal length, the shorter string is considered to be padded on the right with binary zeros for purposes of comparison. If an argument is not a character vector, an error is signalled.

BGREATERP —— function

(BGREATERP **bvec1 bvecn**)

Compares two bit vectors, and returns true if **bvec1** is greater than **bvecn**. If the vectors are unequal in length, the shorter vector is considered to be padded on the right with zeros for purposes of comparison. The argument vectors are not changed by this function, even the bits beyond the current length of the strings are preserved.

If **bvec1** or **bvecn** is not a bit vector, an error is signalled.

17.0 Operations on Hashtables

17.1 Creation

MAKE-HASHTABLE —— *function*

(MAKE-HASHTABLE **id1** [**id2**])

This operator return as its value a new, empty, hashtable. **id1** is used to specify the class of key for this hashtable, and must have a value of EQ, UEQUAL, ID or CVEC, where:

```
EQ     - Key may be any Lisp object, EQ used for search
UEQUAL - Key may be any Lisp object, UEQUAL used for search
ID     - Key must be an identifier, EQ used for search
CVEC   - Key must be a character vector, EQUAL used for search
```

The second argument, **id2**,.is optional. It must have a value of WEAK or STRONG, and defaults to STRONG. It specifies whether the hashtable should use "weak" or "strong" links to its keys. In the former case, key-value pairs in which the only reference to the key is from a weak hashtable will be dropped during garbage collection.

17.2 Accessing

HGET —— *function*

(HGET **hashtable item1** [**item2**])

This operator searches **hashtable** for a value associated with the key **item1**. If one is found it is returned, otherwise if **item2** is present, it is returned. If **item2** is not given the value is NIL.

If **item1** and **hashtable** are nonconformal an error is signalled.

HGETPROP —— *function*

(HGETPROP **hashtable item1 item2**)

This operator searches **hashtable** for a property list associated with **item1**. If one is found it is search for a value associated with **item2**, which is returned. If either search fails the value is NIL.

If **item1** and **hashtable** are nonconformal, an error is signalled.

HKEYS —— *function*

(HKEYS **hashtable**)

This operator returns a list of all the keys in **hashtable**, in arbitrary order. It does not follow chain fields.

If **hashtable** is not a hashtable an error is signalled.

HASHTABLE-CLASS —— *function*

(HASHTABLE-CLASS **hashtable**)

If **hashtable** is a hashtable, returns the key class (EQ, UEQUAL, etc., see MAKE-HASHTABLE, this page), otherwise returns NIL.

HCOUNT —— *function*

(HCOUNT **hashtable**)

Returns the number of key/value pairs in **hashtable**.

If the argument is not a hashtable an error is signalled.

17.3 Accessing or Updating

HCHAIN —— *function*

(HCHAIN **hashtable1** [**hashtable2**])

When the optional second argument is absent, returns the contents of the chain field of **hashtable**. If **hashtable1** is not a hashtable an error is signalled.

When **hashtable2** is specified this operator sets the chain field of **hashtable1** to point to **hashtable2**. This causes searches which fail in **hashtable1** to be continued in **hashtable2**, but with new entries only added to **hashtable1**.

A mismatch of the key classes of **hashtable1** and **hashtable2** will result in an error being signalled.

HGFACTS —— *function*

(HGFACTS **hashtable** [**num1 num2**])

When the optional second and third arguments are absent, returns a list of the shrinkage and growth count/size ratios for **hashtable**. Due to the representation of the ratios in the hashtable the values returned may differ slightly from those provided in a previous three argument call to HGFACTS.

When **num1** and **num2** are specified this operator sets the count/size ratios which control the shrinkage and growth of **hashtable**. The MIN of each number an 64 is taken and the results are stored in **hashtable**.

If the ratio of number of entries in **hashtable** to the number of buckets falls below **num1** **hashtable** is shrunk to the smallest size which would not cause immediate growth. If the count/size ratio exceeds **num2** **hashtable** will be grown to the largest size which would not cause immediate shrinkage.

The relative values of **num1** and **num2** are not checked, and the specification of **num1** > **num2** could cause oscillations.

17.4 Searching and Updating

HPUT —— *function*

(HPUT **hashtable item1 item2**)

Operator to update the hashtable **hashtable**. If the key **item1** already has a value in **hashtable**, its value us changed to **item2**. Otherwise it is added to the hashtable with a value of **item2**.

If **item1** and **hashtable** are not conformal, an error is signalled.

*HPUT** —— *function*

(HPUT* **hashtable** &a-list.)

For each name-value pair in &a-list. a HPUT is done into **hashtable**. If any of the names are non-conformal with **hashtable** an error is signalled without any of the HPUTs being done.

HREM —— *function*

(HREM **hashtable item**)

This operator searches **hashtable** for a key/value pair associated with **item**. If one is found it is deleted from **hashtable**. The value returned is the key/value pair, if such existed, otherwise it is NIL.

If **hashtable** is not a hashtable, or if **item** and **hashtable** are nonconformal, an error is signalled.

HPUTPROP —— *function*

(HPUTPROP **hashtable item1 item2 item3**)

This operator adds (or changes) an item on a property list associated with **item1** in **hashtable**. The property **item2** is given a value of **item3**.

If **item1** and **hashtable** are nonconformal an error is signalled.

HREMPROP —— *function*

(HREMPROP **hashtable item1 item2**)

Searches **hashtable** for a property list associated with **item1**. If such is found it is searched for a property **item2**, and the property and its value are deleted from the list. If the searches are successful the value returned is the (deleted) property value. Otherwise the value is NIL.

If **hashtable** is not a hashtable an error is signalled.

17.5 Updating

HCLEAR —— *function*

(HCLEAR **hashtable**)

This operator removes all key/value pairs from **hashtable**. The count is reset to zero and the bucket array is reduced to a single element vector.

If **hashtable** is not a hashtable an error is signalled.

Operator to set the chain field of **hashtable1** to point to **hashtable2**. This causes searches which fail in **hashtable1** to be continued in **hashtable2**, but with new entries only added to **hashtable1**.

A mismatch of the key classes of **hashtable1** and **hashtable2** will result in an error is signalled.

17.6 Miscellaneous

MAPHASH —— *function*

(MAPHASH **app-ob hashtable**)

This operator applies **app-ob** to each key and value in **hashtable**, in arbitrary order. **app-ob** must be a function of two arguments, the first being a key from **hashtable** and the second the corresponding value. The results of the applications are discarded, and the value of the MAPHASH expression is NIL.

If the second argument is not a hashtable an error is signalled.

HASHEQ —— *function*

(HASHEQ **item**)

For any argument this operator returns a positive small integer hash code in the range 0 to 2^{26}-1.

The hash codes for two EQ arguments will be equal.

HASHUEQUAL —— *function*

(HASHUEQUAL **item**)

This operator returns a positive small integer hash code for any lisp object, in the range 0 - 2^{26}-1. The hash codes for objects which are UEQUAL will be equal.

HASHCVEC —— *function*

(HASHCVEC **cvec**)

This operator returns a positive, small integer hash code for any character vector, **cvec**, in the range 0 - 2^{26}-1.

The hash codes for two EQUAL character vectors will be equal. If &cvec. is not an cvec an error is signaled.

HASHID —— *function*

(HASHID **id**)

This operator returns a positive small integer hash code for any identifier, **id.**, in the range 0 to 2^{26}-1. The hash codes for identifiers with EQUAL pnames will be equal.

If **id** is not an identifier an error is signaled.

18.0 Operations on Arbitrary Objects

18.1 Creation

COPY —— *function*

(COPY item)

This operator creates a distinct object which is UEQUAL to the original. Numbers, identifiers, and bpis are not copied. The exact structure, including all cycles and sharing, of **item** is reproduced. The purpose of COPY is to allow operations on either the COPY or the original without affecting the other. Only those parts of the object which can be modified are copied.

```
(SETQ X 5)
(SETQ Y (COPY X))
(EQ Y X)
VALUE T
(UEQUAL Y X)
VALUE T
(SETQ X (CONS A B))
(SETQ Y (COPY X))
(EQ Y X)
VALUE NON-NIL
(UEQUAL Y X)
VALUE T
(SETQ X (CONS A B))
(RPLACD X X)                                  make circular
(SETQ Y (COPY X))
(EQ Y X)
VALUE NON-NIL
(UEQUAL Y X)
VALUE T
X
VALUE = %L1=(A . %L1)
Y
VALUE = %L1=(A . %L1)
```

SUBST —— *function*

(SUBST item1 item2 item3)

This operator creates a copy of **item3** in which all instances of **item2** are replaced by **item3**. EQUAL is used in searching for instances of **item2**. The resulting uses of **item1** will all be EQ, and are not themselves examined for instances of **item2**.

This operator will fill all space if **item3** contains a cycle.

```
(SUBST 'X '(Y Z) '(A (X Y Z) Y Z Y Z))
Value = (A (X . X) Y Z . X)
(SUBST '(A) 'X '(U <X V X> (W . X) X))
Value = (U <%L1=(A) V %L1> (W . %L1) %L1)
(SUBST 6 2 '(4 3 2 1 0))
Value = (4 3 6 1 0)
```

SUBSTQ —— *function*

(SUBSTQ item1 item2 item3)

This operator differs from SUBST in using EQ rather than EQUAL.

MSUBST —— *function*

(MSUBST **item1 item2 item3**)

This operator produces a value which is EQUAL to that produced by SUBST for the same arguments. It differs in copying only those parts of **item3** which contain changes. EQUAL is used in searching for instances of **item2**. The resulting uses of **item1** will all be EQ, and are not themselves examined for instances of **item2**.

This operator will fill all space if **item3** contains a cycle.

```
(SETQ V '(A (X Y Z) Y Z Y))
Value = (A (X Y Z) Y Z Y)
(SETQ W (MSUBST 'X '(Y Z) V))
Value = (A (X . X) Y Z Y)
(LIST V W)
Value = ((A (X . X) . %L1=(Y Z Y)) (A (X Y Z) . %L1))
```

MSUBSTQ —— *function*

(MSUBSTQ **item1 item2 item3**)

This operator differs from MSUBST in using EQ rather than EQUAL.

EQSUBSTLIST —— *function*

(EQSUBSTLIST **list1 list2 item**)

This operator is a multi-item version of SUBSTQ. A copy of **item** is created, with every instance of a component which is EQ to the nth element of **list2** replaced by the nth element of **list1**. As in SUBSTQ the substituted objects are not examined for further substitution.

Cycles in **item** will cause space to be filled.

```
(EQSUBSTLIST '(A B C) '(1 2 3) '(3 (4 <1 (2 . 1) 5>) 3))
Value = (C (4 <A (B . A) 5> C)
(EQSUBSTLIST '((A B) (B B)) '(A B) '(X (B A) (A . B) C))
Value = (X ((B B) (A B)) ((A B) B B) C)
```

EQSUBSTVEC —— *function*

(EQSUBSTVEC **r-vec1 r-vec2 item**)

Differs from EQSUBSTLIST in requiring vectors for its first and second operands.

```
(EQSUBSTVEC '<A B C> '<1 2 3> '(3 (4 <1 (2 . 1) 5>) 3))
Value = (C (4 <A (B . A) 5> C)
(EQSUBSTVEC '<(A B) (B B)> '<A B> '(X (B A) (A . B) C))
Value = (X ((B B) (A B)) ((A B) B B) C)
```

The reason for the existence of EQSUBSTVEC beside EQSUBSTLIST is pragmatic. A QUOTEd vector uses less space in the heap and in the SHLISPWS file than the equivalent QUOTEd list. Similarly, the VECTOR operator generates less code than does the LIST operator.

18.2 Accessing

SETQP —— *macro*

(SETQP **item1 item2**)

The SETQP operator performs a traversal of its two operands, in parallel. **item1** is used as written, as if it had been QUOTEd, while **item2** is first evaluated, and its value used.

When ever an ID is encountered in **item1** the corresponding component of **item2** is assigned to that ID as its value. If a pair is encountered in **item1** for which no corresponding pair exists in **item2** the traversal stops and the value of the SETQP expression is NIL. (See the following paragraph for the sole exception.) Similarly, if a vector is encountered in **item1** without a corresponding vector in **item2**, of at least the same length, the traversal stops, with a value of NIL.

If **item1** contains a list of three elements, the first of which is an =, the second an ID and the third an arbitrary object, the corresponding component of **item2** is assigned to the ID and is then descended in parallel with the structure, the third element of the list. This is the exception alluded to in the previous paragraph. Note that the third element may be simply another ID, resulting in a multiple assignment.

If the entire structure of **item1** is traversed the value of the SETQP expression is non-NIL.

```
(SETQ X '(1 2 3 4))
Value = (1 2 3 4)
(SETQP (Y . X) X)
Value = non-NIL
Y
Value = 1
X
Value = (2 3 4)
(SETQP (() Z) X)
Value = non-NIL
Z
Value = 3
(SETQP (() () Y X) X)
Value = NIL
(SETQ X '(A . <FOO <"This" 0 2> BAR>))
Value = (A . <FOO <"This" 0 2> BAR>)
(SETQP (C . <() <S () L>>) X)
Value = non-NIL
C
Value = A
S
Value = "This"
L
Value = 2
(SETQP (C . <() (= B <S () L>)>) X)
Value = non-NIL
B
Value = <"This" 0 2>
C
Value = A
S
Value = "This"
L
Value = 2
(SETQ X '(1))
Value = (1)
(COND ((SETQP (Y . X) X) (PRINT Y)) ('T (PRINT 'Empty)))
1
Value = 1
(COND ((SETQP (Y . X) X) (PRINT Y)) ('T (PRINT 'Empty)))
Empty
Value = Empty
```

In the cases where SETQP returns NIL, some of the assignments may have been done before the failure was discovered. The exact order in which the assignments and testing is done can not be easily predicted.

QSETQ —— *macro*

(QSETQ **item1 item2**)

The QSETQ operator is the non-checking equivalent of SETQP. The programmer *must* be certain that the value of **item2** is conformal with **item1**. The value of the QSETQ operator is unpredictable.

18.3 Searching

EQQ —— *macro*

(EQQ **item1 item2**)

The EQQ operator traverses **item1** and the value of **item2**, in the manner of SETQP.

When an ID is encountered in **item1** its value is compared (using EQ) with the corresponding component of **item2**. Similarly, when a component of the form (QUOTE item) is encountered in **item1** item is compared with the corresponding component of **item2**.

If at any time such a comparison fails, or if **item1** and **item2** prove to be non-conformal, the value of the EQQ expression is NIL. Otherwise, it is non-NIL.

```
(SETQ X '(1 2 3 4))
Value = (1 2 3 4))
(SETQ Y 3)
Value = 3
(EQQ (() () Y) X)
Value = non-NIL
(EQQ ('1 () () '4) X)
Value = non-NIL
(EQQ (Y) X)
Value = NIL
(EQQ (() () () () ()) X)
Value = NIL
```

QEQQ —— *macro*

(QEQQ **item1 item2**)

This is the non-checking version of EQQ. The same comparisons are made, but **item2** is not checked for conformity.

CYCLES —— *function*

(CYCLES **item**)

This operator searches its operand, **item**, for any cycles. If none are found a value of NIL is returned. If any exist a vector is returned.

This vector contains twice as many elements as the number of cycles found. The even elements (index 0, 2, etc.) contain pointers to an element of each of the cycles found. The remaining, odd, elements are initialized to zero.

```
(CYCLES '(A %L1=(1 2 . %L1) B . %L2=(C D E . %L2)))
Value = <%L1=(1 2 . %L1) 0 %L2=(C D E . %L2) 0>
(CYCLES '%L1=(1 2 %L2=(3 4 . %L3=(5 %L2 . %L3)) %L1))
<%L1=(1 2 %L2=(3 4 . %L3=(5 %L2 . %L3)) %L1) 0 %L2 0 %L3 0>
```

This, and SHAREDITEMS, produce a vector of special components of their operand. This vector is used by the PRINT operators to identify the components of a object which should be labeled. The "extra" (odd indexed) elements are used for bookkeeping, to record which components have been printed in full.

The PRINT operators apply the value of the identifier S,TO-BE-LABELED to their operands. In the initially created system this identifier has SHAREDITEMS as its value. This guarantees UEQUALity over printing and subsequent reading. The user may redefine S,TO-BE-LABELED, however any definition which does not detect cycles may cause the PRINT operators to loop.

SHAREDITEMS —— function

(SHAREDITEMS **item**)

This function examines the arbitrary object **item** for shared substructure. If there is only one possible path from the root, **item**, to every part of **item**, then the value NIL is returned. If there is shared substructure, the value of SHAREDITEMS is a vector whose even (0, 2, etc.) elements are pointers to the shared nodes (either vectors or list cells), and whose odd (1, 3, 5, etc.) elements are initialized to zero.

Since any cycle must be shared, this function subsumes CYCLES, which is useful when only circular object is of interest.

See comments on the relationship between CYCLES, SHAREDITEMS and PRINT in the description of CYCLES.

18.4 Updating

NSUBST —— function

(NSUBST **item1 item2 item3**)

The NSUBST operator searches its third operand, **item3**, for any components EQUAL to **item1**. If any such are found they are replaced (using RPLACA, RPLACD or SETELT) by **item2**. All such instances are EQ, and are not themselves examined for instances of **item1**.

Unless **item3** itself is EQUAL to **item1** the value of NSUBST will be EQ to **item3**.

```
(SETQ X '(A (B C) D))
Value = (A (B C) D)
(SETQ Y (NSUBST '(A C) '(C) X))
Value = (A (B A C) D)
(EQ X Y)
Value = non-NIL
(SETQ Y (NSUBST '(Z) 'A Y))
Value = (%L1=(Z) (B %L1 C) D)
(EQ X Y)
Value = non-NIL
```

NSUBSTQ —— function

(NSUBSTQ **item1 item2 item3**)

The NSUBSTQ operator differs from NSUBST in using EQ rather than EQUAL.

RPLQ —— macro

(RPLQ **item1 item2**)

The RPLQ operator is the inverse of the SETQP operator. It updates the value of **item1** with the values of any identifiers or quoted objects it finds in **item2** (un-evaluated).

In the same way as SETQP does, RPLQ returns NIL if the value of **item1** is not conformal with **item2**.

```
(SETQ X '(1 2 3))
Value = (1 2 3)
(SETQ Y "xyz")
Value = "xyz"
(RPLQ X (Y () 'ANY))
Value = non-NIL
X
Value = ("xyz" 2 ANY)
```

item2 may consist of any object of pairs and vectors.

QRPLQ —— macro

(QRPLQ **item1 item2**)

This is the non-checking version of RPLQ. If **item2** and the value of **item2** are not conformal damage to the system may result.

18.5 Miscellaneous

GT —— function

(GT **item1 item2**)

where **item1** and **item2** are either both numbers, both **cvecs**, or both **bvecs**, or signals an error. Compares the two and has a non-NIL value if the first is greater than the second, otherwise returns NIL.

LT —— function

(LT **item1 item2**)

where **item1** and **item2** are either both numbers, both **cvecs**, or both **bvecs**, or signals an error. Compares the two and has a non-NIL value if the first is less than the second. Otherwise returns NIL.

GE —— function

(GE **item1 item2**)

where **item1** and **item2** are either both numbers, both **cvecs**, or both **bvecs**, or signals an error. Compares the two and has a non-NIL value if the first is greater than or equal to the second. Otherwise returns NIL.

LE —— function

(LE **item1 item2**)

where **item1** and **item2** are either both numbers, both **cvecs**, or both **bvecs**, or signals an error. Compares the two and has a non-NIL value if the first is less than or equal to the second. Otherwise returns NIL.

GGREATERP —— function

(GGREATERP **item1 item2**)

Compares **item1** and **item2**. If **item1** is greater than **item2** then has a non-NIL value. An arbitrary order has been imposed on different types, so that if the two **items** have different types, then one will be greater than the other.
Warning: This function may not halt on cyclic structures.

19.0 Structured Access to Data Objects

The structured definition facility allows users to define collections of access functions for data objects in a convenient centralized manner.

Consider a list of three elements in which the second element is a vector of two elements. The expression

```
    (DEFINE '(SAMPLE (STRDEF SAMPLE (A1 , B1 <C1 , C2> , D))))
```

defines SAMPLE to be an access macro for the above object. Let the variable X be bound to the object (1 <2 3> 4), then the expressions below will evaluate as shown:

```
  (SAMPLE A1 X) = 1
  (SAMPLE B1 X) = <2 3>
  (SAMPLE D X) = 4
  (SAMPLE B1 C1 X) = (SAMPLE C1 X) = 2
  (SAMPLE B1 C2 X) = (SAMPLE C1 X) = 3
```

The expression

```
  (SET-S (SAMPLE C1 X) (ADD1 17))
```

will update the object bound to X to (1 <18 3> 4). The expression

```
  (SHOW-S SAMPLE X)
```

will evaluate to the expression

```
  (SAMPLE (A1 = 1 , B1 = <C1 = 2 , C2 = 3> , D = 4))
```

19.1 Syntax of Structured Definitions

A structured definition is an expression of the form

```
  (STRDEF field-def)
```

It is a macro call that expands to a macro definition. Structure definitions can be defined, compiled and filed like function and macro expressions.

> When structured definitions are included in temporary compiler environments, they must be included in both the operation recognition environment and the macro application environment. The function and disposition TEMDEFINE is used for this purpose.

19.1.1 Field Definitions

A field definition describes the structure of a component of a data object. It may assign one or more names to that component. It may also associate other attributes with that component

Unnamed field with defined sub-structure

```
shape [attr val] ...
```

This form defines an unnamed field with defined sub-structure. This field cannot be extracted or modified, but any named components can be.

Named field with defined sub-structure

```
name ... [shape [attr val] ... ]
```

This form defines a named field with sub-structure. A field can have any number of names that can be used interchangably. Accessing or setting a structured sub-field causes a compiler warning message.

Named field with no further structure

```
name ... [= init-expr [attr val] ... ]
```

This form defines a named field with no defined sub-structure. Whenever an instance of the object is created, this field is initialized to NIL or init-expr is evaluated to produce an initial value.

19.1.2 Defining the Structure of a Field

The shape component of a field-def describes the structure of the field.

List or pair

```
(field-def [, field-def] ... [; field-def])
```

defines a sub-list. The field-def following the ';' describes the remainder of the list treated as a single field.

Vector

```
<field-def [, field-def] ... >
```

defines a vector.

Association List

```
& prop-name ... &
```

defines a sub-structure that is an association list. Each item in the alist is retrieved as a named field.

Named Flags and Sets of Flags

```
$ {    flag-name            } ... $
   (set-name flag-name ... )
```

defines a field that can contain one of the flag-name identifiers specified in the declaration. Each set-name must be an identifier and denotes any of the flag values following; the latter may be identifiers or positive small integers. An integer value can be defined as a named flag by including it in a set of one element.

Set names can only be used in access expressions; update expressions must use a specific flag name or the name of a set containing only one integer. An access expression that mentions a flag-name or set-name is a predicate that test the setting of the field. An update expression is written without a value-expression and updates the field to the specified flag setting.

Bit-level Fields

```
          bitnum
: name { (bitnum [width [init]]) } ... :
          (name2 val)
```

defines a field that contains an integer value. Individual groups of bits can be interrogated and set by name. If only bitnum is specified, then the field is one bit wide. If a width is specified, it must not extend the field beyond the end of the machine word. Bitnum must be an integer from 6 to 31. If name2 is specified, name defines a name for value val in bitfield name.

An access expression for a bit field extracts the contents of the field as an integer value. An access expression for a named bit-field value is a predicate that test for that value.

An update expression for a bit field sets the bit field to the low-order width bits of the value expression. An update expression for a named bit-field value has no value expression and sets the bit field to the associated value.

19.1.3 Attributes of Field Definitions

Each field definition can be followed by one or more attribute/value pairs.

Display of Field Values

```
SHOW fn
```

This attribute interacts with the SHOW-S form. If fn is NIL, SHOW-S does not display the value of that field. If fn is non-NIL it must be a function of one argument. SHOW-S displays the value of the function applied to the value of the field.

Speed of Generated Code

```
QFORM xx
```

This attribute is recognized only in the top-level field-def of STRDEF. If xx is NIL, safe access code is generated, so that if an instance expression evaluates to an inappropriate object, an error may be signalled. If xx is non-NIL fast, non-checking access code is generated.

Comments

```
* comment
```

This attribute allows comments to be inserted in structured definitions.

19.1.4 A Note on the Syntax of Structured Definitions

The structured definition facility gives the appearance of syntax by using some reserved identifiers as delimiters. Note that these delimiters must be entered as identifiers in order to be recognized. The shape expression (A,B) defines a sub-structure that contains only one sub-field with A,B as the name of the sub-field. To define a list of two fields named A and B respectively, you must enter the separating blanks, as in (A , B).

Future extensions of the structured definition facility will require additional keywords and delimiters. In order to remain compatible with these extensions, do not use field names that consist of a single non-alphabetic character.

19.2 Operations With Defined Structures

Creating-expression —— macro

A creating-expression evaluates to a new instance of a defined object.

```
(object-name) = new-instance-of-object
```

Each field in the new object is initialized to the value of the initial expression specified in the definition. These expressions are evaluated in the lexical context where the creating expression occurs.

Access-expression —— macro

An access-expression evaluates to the current setting of a field in a particular instance of a defined object.

```
(object-name access-path instance-expression)
```

The name of a object is the first name mentioned in the top level field definition used with STRDEF. An access path is a sequence of field names that uniquely identifies a sub-structure. Intermediate object names need not be specified if the final sub-field names are unique. An access expression used as an expression evaluates to the value of the field in the object that is the value of the instance expression.

SET-S —— *macro*

The SET-S form performs an assignment to a sub-field of a object.

```
(SET-S access-expression [value-expression])
```

The effect of this expression is to update the field specified in the access expression with the value of value-expression. If value-expression is omitted, the sub-field is set to the initial value or to a named value denoted by the access expression.

The value of a SET-S expression is not defined.

TEST-S —— *macro*

The TEST-S form is a predicate that evaluates to a non-NIL value if the specified field is present in the given instance of the object.

```
(TEST-S object-name access-path instance-expression)
```

The value is NIL if the specified field is not present in the value of instance-expression. The value is non-NIL (but unpredictable) if the field is present.

SHOW-S —— *macro*

The SHOW-S form expands an instance of a defined data object to build an expression that contains field names as well as values.

```
(SHOW-S object-name access-path instance-expression)
                   = exploded-object-instance
```

19.3 Cross-Reference Facility

If you compile object uses stored in a Lispedit indexed file, you can get a cross-reference of object and field uses by function. The cross-reference facility is activated by adding the STRUCT-XREF property to a file with the Lispedit command

```
SETDP filename ( STRUCT-XREF xrefmem
```

The argument xrefmem is the name of a new member in the indexed file. Once this command is evaluated, the VALUE property of xrefmem will be updated every time the file is compiled with the DDIR command. The information stored is as follows:

```
( (object-name-1
      (field1
          (Access m1 m2 ... )
          (Update m1 m2 ... )
          (Create m1 m2 ... )
          (Test  m1 m2 ... ) )
      (field2  ... )
      ... )
  (object-name-2 ... )
  ... )
```

where m1 m2 ... are names of members in the file that refer to a object in the specified way.

In addition, every time DMEM or DDIR is used on a particular member of the file, the property STRUCT-USES is updated with a more detailed summary of object usage in the particular member function:

```
( (object-name-1
     (field1
         (Access refvar1 refvar2 ... )
         (Update (refvar1 valvar1) (refvar2 valvar2) ... )
         (Test refvar1 refvar2 ... ) )
     (field2 ... )
     ... )
  (object-name-2 ... )
  ... )
```

where refvar1, refvar2, ... are the variables used to reference a object and valvar1, valvar2, ... are the variables from which a value is stored into a field.

THE LISP/VM EDITOR

20.0 Introduction

Lispedit is an interactive program development tool that allows a user to compose, edit, test and file programs in a unified environment. The main components of the system are a structural editor for expressions, a file management facility, and a display interface. A screen oriented debugging tool allows interactive execution of functions in their normal run-time environment.

21.0 The Lispedit User Interface

21.1 Lispedit Screen Format

The display screen is divided into 5 major areas by Lispedit. The Top Display normally fills most of the screen; the Fence Line serves as a divider between the top display and the Message Area; it also shows current status information; the Message Area shows the sequence of inputs and messages in the current edit environment; the PF Area shows the current meaning of the Program Function keys; the Command Area is only part of the screen where input is allowed.

21.1.1 The Top Display Area

The top region of the screen normally contains a highlighted display of the current focus and several levels of containing expression. The focus is the term we use for the current position of the editor in a data or program object. Whereas the current position of a text editor is a line in a file, the current position of Lispedit is a sub-expression of the Top Expression. The focus is the implicit or default argument to many commands.

Commands that generate a large amount of information use the top display area for output. You can edit the value of ,VAL to examine these messages in more detail.

21.1.2 The Fence Line

This line is always displayed to separate the top display from the message area. It contains the current recursion level of Lispedit, usually an indication of the current input state, and a description of the object currently being edited.

21.1.3 The Message Area

The size of this area varies with the amount of data to be shown. Simple Lispedit commands normally do not produce messages and therefore show only an echo in the message area. When a command produces messages, they appear in the message area following the echo of the command. In Message Review Mode, this area can be scrolled independently to examine the previous commands and messages.

Note that all values shown in the message area are truncated to fit on one line. The most recent value shown in the message area is also stored in the variable ,VAL. To examine a truncated value in detail, edit the value of ,VAL.

21.1.4 The PF-key Area

This one-line area displays a brief description of each Program Function Key that is currently defined. Since PF keys can be re-defined by the user, be sure to specify short descriptions to allow all the descriptions to fit on one line.

21.1.5 The Command Area

The meaning of an input line is determined by the current input state shown in the fence line. Certain expressions and previous commands can be brought back to the Command Area to be modified and/or re-entered.

21.1.6 Condensed Display Formats

In many cases, Lispedit displays *elided* or *truncated* expressions and values. Ellipsis is used in the top display in order to show a prettyprinted and syntactically correct picture when the text will not fit on the screen. When the focus and the immediate context of the focus cannot be fully shown on the screen, parts of the text are displayed as '...' to indicate one or more expressions and as '&' to indicate exactly one expression. These ellipses can occur both in the context and the focus. In order to view the elided text, you must shift the focus to a position closer to the ellipsis.

Values in the message area are normally truncated to one line. The editor can be invoked recursively to edit these values if you want to see the missing components.

21.2 Lispedit Input States

The meaning of user inputs varies depending on the input state of Lispedit. The current input state is always identified by the text of the fence line or the format of the message area.

21.2.1 Command Mode

The normal state of Lispedit is *command mode*. In this mode, a typical input is a Lispedit command token. Many Lispedit command tokens are a single letter, and all command tokens are identifiers. Since many commands need no arguments, or have implicit default arguments, a normal Lispedit command consists of typing a command token and pressing the ENTER key. When the ENTER key is pressed, the command is decoded, and the necessary action is performed. At the end of the cycle, the display is updated to reflect the effect of the command on the top display; the message area shows the command that was performed, followed by system messages, if any were produced. At this point, the system is ready to accept a new command.

It is also possible to enter additional commands before the end of the cycle; in this case the update of the display may be delayed until all pending commands have been executed. When the display is updated, the message area will show all the commands that were executed in that cycle.

When arguments are included in a command, the command line consists of the command token followed by one or more expressions in normal syntax. The one exeception to normal syntax is that trailing closing parentheses may be omitted; Lispedit always supplies the necessary number of closing delimiters to balance an input line. Arguments to Lispedit commands are normally taken as entered; they are not evaluated. Thus the command 'I (PLUS 1 1)' inserts the expression (PLUS 1 1), not the number 2.

If an input line does not begin with a Lispedit command token, it is evaluated as an expression and the input followed by the value are displayed in the message area.

Certain special characters are reserved by Lispedit to implement special interface features.

21.2.2 Input/Replace Mode

This mode is identified by the text '(Input)' or '(Replace)' on the fence line.

In input mode, all user input is treated as a continuous stream of atoms and delimiters. The ENTER key simply transmits the data on the input line to Lispedit and clears the Command Area for more text. The effect of input mode is similar to continuous text input in a text editor. The difference is that the top expression display remains pretty-printed. After each ENTER, the focus shifts to the last balanced expression mentioned in the input line.

In Input mode, the first expression on the input line is inserted to the right of the current focus. In Replace mode, the first expression on the input line replaces the current focus. In both Input and Replace mode, additional text is inserted as required by the syntax. Replace mode occurs when Input mode is invoked in situations where insertion is not possible. These situations occur if the focus is the top expression, a vector element or the atomic CDR of a pair. Replace mode reverts to Insert mode after the first ENTER.

Input or Replace mode ends when you enter a null line.

21.2.3 Interactive Interpreter Mode

Interactive Interpreter Mode is identified by the text '(Heval)' on the fence line.

In this mode, the expression displayed in the top display is being executed. The focus is normally the next sub-expression to be evaluated. All the facilities of command mode are available to view and modify the expression being evaluated. In addition, several interpreter commands are also activated.

These let you control the progress of the execution. After each interpreter command is entered, the focus shifts to the next step in execution and the message area shows the result of the last step.

21.2.4 Message Review Mode

Message Review Mode is identified by index numbers preceding each line in the message area.

In this mode you can examine the last 50 lines of input and messages. If the message shows a truncated or elided value, you can recall that value to edit the full form. You can transfer the text of a message or input to the Command Area to repeat or modify a previous entry.

Message review mode ends as soon as you enter a blank line, a Lispedit command or an expression.

21.2.5 Null and Blank Lines

A fine distinction is made in the system between a *null line* and a *blank line*. A null line is generated if the ENTER key is depressed when the Command Area is clear of all user input. A null line is normally ignored; it is also the signal that ends input mode. A blank line contains at least one space character but must contain only space characters. The effect of a blank line is to recompute the display screen and to clear the message area.

21.3 The Lispedit Command Interface

There are three main classes of input recognized by Lispedit:

- A command line is a line that begins with the name of a Lispedit command.
- A special command line is a line that begins with one of the special characters detected by Lispedit.
- If a line does not begin with a Lispedit command or special character, it is treated as an expression to be evaluated.

21.3.1 Command Lines

A command line is an input line that begins with the name of a Lispedit command. The name may be preceded by any number of spaces; it is terminated by any non-alphabetic character. The remainder of the line is scanned for arguments to the command. Each argument to a command must be an expression. All of the following inputs

```
CHANGE 3 V4
CHANGE3 V4
                    CHANGE              3               V4
```

represent the same command. The command name is CHANGE, the first argument is the number 3, and the second argument is the identifier V4. Since a command name consists entirely of alphabetic characters, a numeric argument may be entered immediately following the command name. Arguments on the other hand are arbitrary expressions and must therefore be separated by spaces or appropriate delimiters.

The last expression on an input line does not need trailing closing parentheses. For example, the two commands

```
INSERT (SETQ X '(A B C))
INSERT (SETQ X '(A B C
```

are equivalent. Lispedit will insert any missing closing parentheses, vector brackets or string quotes.

21.3.2 Special Lines

Special lines begin with a special character; the remainder of the line may be an argument or may be ignored. Since special commands typically refer to and manipulate the current environment history, they are not themselves included in that history.

"	causes the last input to be repeated
$	causes the most recently entered Lispedit command to be displayed in the Command Area. In input mode (see INM command) a line beginning with $ is treated as a command, in the normal read loop leading $ characters are removed and ignored.

The following special commands initiate **review mode**. In this mode, recent commands, results and messages can be examined and recalled to the command line.

In review mode, the message area is normally expanded and each line is prefixed with a sequence number increasing with age. Review mode ends when a blank line or any non-review command is entered.

?	(in command mode) recall the most recent input to the command line and begin review mode. (in review mode) recall to the command line the most recent input from the current position in the console history. In both cases the console history is positioned at the line immediately preceding the recalled line.
?n	position line n of the console history at the bottom of the screen.
?Um	scrolls m lines back in time.
?Nm	scrolls m lines forward in time.
??	places the text of the bottom line of the message are in the Command Area where it can be modified and re-entered.
??n	places the text of message n in the Command Area.

21.3.3 Selecting the Focus with the Cursor

SELECT is a command that is valid only when defined as a PF key setting. When a key defined as SELECT is pressed,

- If the cursor is in the program area of the screen, the effect of this command is to select as the new focus the sub-expression closest to the cursor.
- If the cursor is anywhere else, the effect of this command is to move the cursor to the first character of the current focus.

21.3.4 Expressions

An expression entered in the Lispedit Command Area is evaluated and the value is shown in a message of the form

```
  expression = value
```

where expression is compressed if necessary to fit in the left half of the message. If the expression you want to evaluate is an identifier that may be confused with a Lispedit command you must use the VALUES command. The argument to the VALUES command is always evaluated.

If an error is signalled during the execution of the expression, a message of the form

```
  expression RESULTED IN ERROR message
```

is shown instead. If you want to enter the error break loop in the event of an error, use the VB command.

21.3.4.1 Gensyms in Lispedit

Since each mention of a gensym in a Lispedit input line will create a reference to a brand new internal gensym, it is not possible to mention in a command gensyms displayed in the top display.

One way to create a program containing gensyms is to build the program using a unique dummy identifier (say XXX) for each desired gensym. When the program is completed, the command

```
    EV C 'XXX (GENSYM) '*
```

can be used to change all the dummy occurrences to the same gensym.

To manipulate gensyms in the current expression, position the focus on some occurrence and evaluate (SETQ FOO (CURS)). You can then use the EV command to mention the gensym in other commands. For example,

```
   EV LOC FOO
```

will locate an occurrence of the gensym.

22.0 Where Things Are Stored: Lisp, Lispedit and Files

It is useful to consider the Lispedit environment, the LISP/VM environment, and the CMS environment as three distinct collections of named objects.

- The LISP/VM environment consists of variables bound to values. Assignment is the normal means of moving values in this environment.
- The Lispedit name space consists of one or more concurrent edit environments, each with its own edit status and command history.
- The Lispedit file environment is a subset of the CMS file environment. Each object in it is a three level hierarchy of indexed file, member and property names.

Since a very common activity of a Lisp user is to write and test programs, there is a collection of commands that support the view that a file is a collection of function and macro expressions destined to be assigned to identifiers in the Lisp environment. These commands are listed by the HELP KNOWNID command.

When more detailed control is needed over file operations, there are commands and operators that transfer data between the various possible pairs of environments:

- The EDIT command transfers data from the LISP/VM environment to an edit environment. Since editing operations modify the object immediately, there is no need for a command to transfer an object from and edit environment to the LISP/VM environment.
- GET and PUT commands transfer data between an Indexed file and a Lispedit environment.
- DXM and DXF transfer data from Indexed Files into the LISP/VM environment. STORE transfers data from the LISP/VM environment to an indexed file.

22.1 Lispedit Indexed Files

The purpose of the Lispedit file system is to allow expressions to be saved in CMS files for later retrieval and editing. An indexed file is a list of named property lists. There are commands to retrieve the entire list, a single property list or a single property value from a given property list. The entire file object will be referred to below as an *indexed file*. An indexed file is uniquely denoted by a CMS file name and mode letter combination. Individual property lists will be called *members*. A member in an indexed file is uniquely denoted by an identifier other than a special character. A property is an identifier other than a special character. A property value in an indexed file is uniquely determined by the member name and property.

From the CMS point of view, an indexed file is a set of files with a common file name or file type. They can be manipulated with CMS file operations if some care is exercized. Normally these files are manipulated by the appropriate Lispedit commands and file system functions.

Properties in an indexed file are classified as *immediate* or *pushdown*. When the value of a pushdown property is updated, the previous value is preserved and can be retrieved as long as the contents of the indexed file are not explicitly garbage collected. When the value of an immediate property is updated, the previous value is lost.

22.1.1.1 Specifying Indexed File Objects

All commands that specify an objects in indexed files use the following syntax to denote that object:

```
xf
xf mem
xf mem prop
xf mem prop age
xob
```

In each of these, xf specifies an indexed file, mem is a member name, prop is a property name. The age parameter specifies the current value of a property or an older value retained in the file. The notation xob denotes any object from an indexed file. Each component of the combination xf mem prop must be an identifier if specified. The identifiers =, * and & have a special significance. The purpose of * is to collect several parts of of a file.

id1 id2 id3 denotes the value of property id3 for member id2 in indexed file id1.

id1 id2 * denotes a list of property/value pairs for member id2 in indexed file id1.

id1 * *id3* denotes a list of the form ((mem (id3 . value)) ...) where mem ranges over all members in indexed file id1.

id1 * * denotes a list of the form ((mem (prop . val) ...) ...) where prop ranges over all properties of each member and mem ranges over all members in xf id1.

* *p2 p3* denotes a list of objects produced by the arguments p1 p2 p3 where p1 ranges over all indexed files currently in the heap. This form is not permitted in most commands.

The entry = for any parameter denotes the corresponding current indexed file default. These defaults are displayed by all commands that update them, by HELP ENV, and by SXD with no arguments.

The entry & for an xf, mem, or prop parameter denotes the first identifier to be found in the current focus. This form is handy when editing the result of an LX command: while focussed on a list describing a member, GET xf & will fetch that member. Another use is in Heval: while looking at an expression, TRAP xf & will trap the function that is called by the expression.

The argument & denotes the focus, or the first identifier within the focus. The forms &1 &2 and &3 denote the first, second, or third identifier within the focus.

The effect of omitted arguments depends on the command. The default values are described for each command separately.

The age parameter is omitted or 1 to denote the most recent value or current value of a property; *, to denote the oldest; or a number n, to denote the n-th most recent value.

The identifier that specifies xf may have one of these forms:

```
xfn
xfn*modeletter
```

The first form specifies the first file by that name in the CMS search sequence. The second form identifies a file on a particular mini-disk. All Lispedit output messages use the form filename*modeletter.

22.1.2 Dispositions

A common reason for filing expressions is to build a library of function definitions. Lispedit supports this activity with specialized *disposition functions* that interact with a set of distinguished properties to allow functions to be defined or compiled, to memory or to a LISPLIB file. Lispedit also maintains appropriate macro states associated with an indexed file, so a set of macros can be defined for use with all the compilations done from a given file.

22.1.3 Disposing of an Entire Indexed File

When an indexed file is loaded, a wide variety of operations may be performed. An object may be loaded as a compiled function and bound to an identifier, or it may be simply evaluated for its side effects. It may be ignored entirely, or bound to an identifier without any additional processing. We call the normal action to be taken at load time the **disposition** of a member.

The disposition of a member is the value of the DISPOSITION property of the member, or of the file. Typical disposition are DEFINE (the ultimate default), COMPILE, SETQ, EVAL and NONE. For each member in the indexed file, in the order defined by the file, a disposing expression is built and evaluated. Parameters to the operation can override the filed default disposition and define a transformation for atomic member dispositions. It is also possible to disable the inclusion of file and member updates to OPTIONLIST.

22.1.4 Disposing of Individual Members

In contrast to the EXF function which can only evaluate an entire file of definitions, Lispedit allows individual functions to be retrieved, modified and defined in arbitrary order. When disposing of an individual member, more flexibility is possible and allowed than when disposing of the entire indexed file. In addition to the parameters allowed in the above section, it is possible to override entirely the member disposition determined by the indexed file. It is also possible to supply an alternate function name and a property other than VALUE to be used in building the disposition expression.

In order to interact correctly with temporary macro definitions in an indexed file when an individual member is disposed of, Lispedit makes an macro application environment and passes it to the compiler. This macro application environment contains bindings created by all members that precede the given member in the file order and that claim to affect the macro application environment. When an macro application environment is made, it is remembered and re-used, if possible, during the disposition of another member. Normally, if all the macros local to an indexed file precede any members that use them, a macro application environment will be made exactly once during an edit session.

22.1.5 Options in Disposition Commands

The commands DXM and DXF transform the source data stored in an indexed file into executable code, fast-loadable files of data (LISPLIBs), side-effects on the current environment, and so on and on.

The effect of these commands is controlled by the value of the DISPOSITION property on a member and by a variety of options specified in the command.

Keyword Options Recognized by Both DXM and DXF:

APPEND	Add listing to an existing file
ERASE	Erase current listing file
ADD	Add to existing fast-loadable (LISPLIB) file
NEW	Replace existing LISPLIB file
LINK	
NOLINK	Do or do not modify the current LISP/VM environment (applies only to dispositions SETQ, EVAL, and those that invoke DEFINE, COMPILE or ASSEMBLE)
LIST	
NOLIST	Do or do not make a listing file (default: NOLIST)
FILE	
NOFILE	Do or do not produce a LISPLIB file.
MEMOPT	
NOMEMOPT	Use or ignore OPTIONS property of each member.

XFOPT

NOXFOPT Use or ignore OPTIONS property of indexed file.

NEWE to re-create the macro environenment associated with an indexed file. This option must be used if one of the members that affect the macro environment has been modified. Members that affect the macro environment are those with dispositions of TEMPDEFINE, ORTEMDEFINE, MATEMPDEFINE, TEMPSETQ, ORTEMPSETQ, MATEMPSETQ, ADDTEMPDEFS, ORADDTEMPDEFS, MAADDTEMPDEFS.

USEE to use the current macro environment.

The atoms x = NONE, PRINT, EVAL, SETQ, TEMPDEFINE, DEFINE and COMPILE have different effects in DXM and DXF. In DXM, x specifies a disposition that overrides the normal computation for effective disposition. In DXF, x defines the default disposition for the entire file. All other atomic options are considered errors.

Pair options recognized by both DXM and DXF:

(LIST . x) (LISTING . x)

(LISTING . NONE) Suppress Lispedit listing logic

(LISTING . NIL) Suppress listing

(LISTING . T) Print listing to file xfn PRINT A

(LISTING fn [ft [fm]]) Print listing to the given file. Defaults are PRINT and A

(FILE . NONE) Suppress Lispedit LISPLIB logic

(FILE . NIL) Do not write to any LISPLIB file

(FILE . T) Write to the LISPLIB file associated with the indexed file.

(FILE fn [ft [fm]]) Write to the specified LISPLIB; note that if the normal LISPLIB is specified, it is not recognized. Defaults are LISPLIB and A

(LISTFLAG . x)

(FILEFLAG . x) If x is ADD, add to the listing or LISPLIB file, if x is NEW replace it.

(MEMOPT . x)

(XFOPT . x) If x is NIL, ignore the OPTIONS prop. on member or xf, otherwise use it

(OPTIONLIST . x) Use x instead of the current binding of OPTIONLIST

(LISPOPTS x y ...) Add x y ... to OPTIONLIST without examining

(DISPOSITION . x) Override the file DISPOSITION property

(DMAP (x . y) ...) Translate the member disposition x to y, x must be an atom.

(NOLINK . x) Applies to DEFINE, COMPILE, EVAL and SETQ dispositions

Pairs of the form (x . y) where x is NONE, PRINT, EVAL, SETQ, DEF, DEFINE, COMPILE, ASSEMBLE or TEMPDEFINE, are added to the disposition map.

Pair options recognized only by DXM:

(OVERDISP . x) Defines x as the effective disposition

(NAME . id) If a single function is being defined, define it as id instead of the name specified by the file.

(USE . x) Use property x instead of VALUE

All other pair options are added to OPTIONLIST.

Some general rules:

- Duplicate options are considered errors
- Unless an explicit option is present, DXM assumes ADD and DXF assumes NEW for both LISTFLAG and FILEFLAG.
- Unless an explicit option is present, (FILE . nonnil) implies (NOLINK . T). If the FILE option is missing, assume (NOLINK . NIL)

22.1.6 Some Notes on Filed Objects

22.1.6.1 Gensyms

Since objects in indexed files are stored and fetched with the Printer and Reader some cautions are in order when these objects contain gensyms.

Each property value is read and written with an independent operation. As a result, it is not possible to have two property values that contain the same gensym. In fact, previous values of the same property (retrieved by mentioning an age parameter) will exhibit distinct gensyms.

In order for two function definitions to share a gensym, they must be both be stored as the value of a single property. This can be accomplished by setting the FORMAT property to PAIRS for the member in question.

22.1.6.2 The FORMAT Property and the Format of VALUEs

The normal format of the VALUE property is

```
    (VALUE . (expr))  normally shown as (VALUE expr)
```

where expr is the expression used by the disposition functions.

This format assumption can be modified by adding a FORMAT property to a member. If this property is missing or its value is NIL, the default value is LIST.

FORMAT property value:	Expected VALUE property format:
IMM	(VALUE . expr)
LIST	(VALUE expr)
PAIRS	(VALUE . (id expr)) or (VALUE (id1 expr1) ...) or (VALUE ((id1 expr1) ...))
LINK	(VALUE . member2) The effect of this format is to go to member2 for the actual value.

The PAIRS format can be used to define a function with a name different from the member name. It also allows several functions to be defined in one member (a useful facility if the functions share a gensym or are mutually recursive macros.)

23.0 Lispedit Commands by Various Categories

23.1 Frequently used commands.

/ [expr [name]]

to shift the focus to the next occurrence of expr.

BIGGER [n]

to shift the focus to a containing expression.

CHANGE from to [count [scope]]

to substitue one expression for another.

CMS to pass a command to CMS.

COMPILE [[xf] id]

to compile an operator definition.

COPY [count [to__tag [from__tag]]]

to copy the focus.

DECLARE to insert vars in the bvlist of a PROG.

DEFINE [[xf] id]

to cause an operator def. to be interpreted.

DEL [count]

to delete one or more items in a list.

EDIT expr

to edit the value of expr in a fresh environment.

EVAL cmd

to do a Lispedit command with evaluated arguments.

FILEXOB [[xf] id]

to file a definition in an indexed files.

FLATTEN to remove one level of list structure from the focus.

HELP displays descriptive information.

INSERT expr ...

to insert expr ... after the focus

LOADXOB [[xf] id]

to load objects stored in an indexed file.

MOVE [count [to__tag [from__tag]]]

to copy and delete the focus.

NEXT [n]

to shift the focus toward the tail of a list or vector.

PREVIOUS [n]

to shift the focus toward the head of a list or vector.

QUERY [[xf] id]

to show status of LOADed objects.

QUIT [expr]

to discard an edit environment [and return the value of expr].

REPLACE e1 e2 ...

to replace the focus with e1 e2 ...

RUN to finish evaluating the focus and step to the next expression.

SAVEXOB [[xf] id]

to save a definition in an indexed files.

SMALLER n1 ...

to descend to a sub-expression of the focus.

STEP to advance the EP.

STOP to terminate interactive evaluation.

TO to insert a copied or moved expression.

TOP to shift the focus to the top of the current expression.

TRAP [[xf] id]

to enable interactive evaluation of a function.

UNTRAP id

to disable interactive evaluation of a function.

VALUES expr ...

to evaluate one or more expressions.

WRAP [n] e1 e2 ...

to wrap and prefix the focus with e1 e2 ...

23.2 Commands that shift the focus.

/ [expr [name]]

to shift the focus to the next occurrence of expr.

BACK to return to a previous screen (does not undo updates).

BIGGER [n]

to shift the focus to a containing expression.

FORTH the opposite of BACK.

FORWARD [num]

to shift the focus in current or bigger expression.

GO tag

to shift the focus to a TAGged place.

LOCPAT pattern [name]

to find the next occurrence of a pattern.

LOCPRED [pred [name]]

to shift the focus to the next occurrence of pred.

Num to shift the focus forward or to a smaller expression.

NEXT [n]

to shift the focus toward the tail of a list or vector.

PREVIOUS [n]

to shift the focus toward the head of a list or vector.

SMALLER n1 ...

to descend to a sub-expression of the focus.

TOP to shift the focus to the top of the current expression.

23.3 Commands that modify the current expression.

ALTREP to convert lists to vectors etc ...

CHANGE from to [count [scope]]

to substitue one expression for another.

DECLARE to insert vars in the bvlist of a PROG.

DEL [count]

to delete one or more items in a list.

FLATTEN to remove one level of list structure from the focus.

INSERT expr ...

to insert expr ... after the focus

INSERTBEFORE expr ...

to insert expressions before the focus.

INSERTMODE to switch to input mode.

LIFT [n]

to replace a containing expression with the focus.

MOVE [count [to__tag [from__tag]]]

to copy and delete the focus.

PREF e1 e2 ...

to prefix the focus with e1 e2 ...

REPLACE e1 e2 ...

to replace the focus with e1 e2 ...

RESTORE to reverse the effect of the last update command.

SORT [scope] [access__fn] [U] [R]

to reorder items in a list.

TO to insert a copied or moved expression.

WRAP [n] e1 e2 ...

to wrap and prefix the focus with e1 e2 ...

23.4 Debugging and interactive evaluation commands.

CHECK expr ...

to display values during interactive evaluation.

COME to direct interactive evaluation.

COMP to complete a function without interactions.

CONT to reset the EP or current value.

EVAL cmd

to do a Lispedit command with evaluated arguments.

EVALPOINTER to shift the focus to the EP.

FIN to finish evaluating the focus.

HEVAL [arg] ...

to begin interactive evaluation of the focus.

RUN to finish evaluating the focus and step to the next expression.

STEP to advance the EP.

STEPF	to advance the EP in fine detail.
STOP	to terminate interactive evaluation.
TEST expr ...	to evaluate for effect only.
TRAP [[xf] id]	to enable interactive evaluation of a function.
UNCHECK expr ...	to stop displaying values.
UNTRAP id	to disable interactive evaluation of a function.
VALUES expr ...	to evaluate one or more expressions.
VALUESANDBREAK expr	to evaluate and take error breaks.

23.5 Commands that create or use known identifiers.

COMPILE [[xf] id]	to compile an operator definition.
DEFINE [[xf] id]	to cause an operator def. to be interpreted.
EDIT expr	to edit the value of expr in a fresh environment.
FILEXOB [[xf] id]	to file a definition in an indexed files.
LOADXOB [[xf] id]	to load objects stored in an indexed file.
QUERY [[xf] id]	to show status of LOADed objects.
SAVEXOB [[xf] id]	to save a definition in an indexed files.
TRAP [[xf] id]	to enable interactive evaluation of a function.
UNTRAP id	

to disable interactive evaluation of a function.

XF [xf [mem [prop]]]

to set the current file defaults.

23.6 Descriptions of concepts and parameters.

HELP displays descriptive information.

23.7 Commands that search the top expression.

/ [expr [name]]

to shift the focus to the next occurrence of expr.

ALLEXPR expr [name [scope]]

to locate all occurrences of an expression.

ALLPAT pattern [name [scope]]

to locate all occurrences of a pattern.

ALLPRED pred [name [scope]]

to locate all exprs that satisfy a pred.

LOCPAT pattern [name]

to find the next occurrence of a pattern.

LOCPRED [pred [name]]

to shift the focus to the next occurrence of pred.

23.8 Commands that affect the edit environment.

ADDCOM to define new commands.

BACK to return to a previous screen (does not undo updates).

CASE to control input case conversion.

COMSEARCH to query or modify the command search order.

COPY [count [to__tag [from__tag]]]

to copy the focus.

DOCOMMANDS to perform several commands as one.

EDIT expr

to edit the value of expr in a fresh environment.

EDITHERE expr

to edit the value of expr in this environment.

FLUSH	to flush console history.
FORTH	the opposite of BACK.
GETHERE xf mem prop	to fetch an object from an indexed file.
GETMEM mem prop	to edit a file object in a fresh environment.
GETMEMHERE mem prop	to edit a file object in this environment.
GETXOB xf mem prop	to fetch a file object into a fresh environment.
GO tag	to shift the focus to a TAGged place.
MAP	to iterate a command over a list or vector.
MOVE [count [to__tag [from__tag]]]	to copy and delete the focus.
NAME name	to name (and thus retain) a Lispedit environment.
PF	to display and modify PF-key settings.
QUIT [expr]	to discard an edit environment [and return the value of expr].
READ fn [ft [fm]]	to read from a file into an edit environment.
RESOLVE cmd	to locate the filed definition of a command.
RESUME	to return to the last discarded edit environment.
SHOW	to modify the screen format parameters.
TAG id__or__num	to tag the focus.
TO	to insert a copied or moved expression.
XF [xf [mem [prop]]]	to set the current file defaults.

23.9 Commands that affect the indexed file environment.

ARCHIVE to create a compact (single) file from an indexed file.

COMPILE [[xf] id]
to compile an operator definition.

COMPX xf1 xf2 ...
to compare names and dates.

COPYX xf mem prop (TO xf mem prop
to copy indexed files.

DEFINE [[xf] id]
to cause an operator def. to be interpreted.

DISPOSITION xf mem (disp
to query or set the disposition of a file or member.

DISPXFILE xf (opt ...
to process all the members in an indexed file.

DISPXLOCALS xf
to load temporary macro defs. etc.

DISPXMEM xf mem (opt ...
to process a single member from a file.

ERASEX xf mem prop
to erase part or all of an indexed file.

EXHUME to re-create an indexed file from the archived form.

EXPORT xf
to create a sequential file from an indexed file.

FILEXOB [[xf] id]
to file a definition in an indexed files.

GETHERE xf mem prop
to fetch an object from an indexed file.

GETMEM mem prop
to edit a file object in a fresh environment.

GETMEMHERE mem prop
to edit a file object in this environment.

GETXOB xf mem prop

to fetch a file object into a fresh environment.

IMPORT to transform a sequential file into an indexed file.

LINKX desc [anc] ...

to link files in descendent/ancestor order.

LISTX xf mem prop (opt ...

to display names, dates, etc.

LOADXOB [[xf] id]

to load objects stored in an indexed file.

MDEF [xf [mem]]

to macro-expand the focus.

ORDERX xf mem (mem1 mem2 ...

to re-order members in a file.

PRINTX xf (opt ...

to list all or part of an indexed file.

PUTFOCUS xf mem prop

to file the focus.

PUTXOB xf mem prop

to save an object in an indexed file..

QUERY [[xf] id]

to show status of LOADed objects.

RECLAIMX xf ([ERASE]

to discard obsolete data in an indexed file.

SAVEXOB [[xf] id]

to save a definition in an indexed files.

SETXPROP xf (name value ...

to associate a prop. with a file.

XF [xf [mem [prop]]]

to set the current file defaults.

23.10 Commands that interact with the operating system.

CMS to pass a command to CMS.

EXITLISP [rc]

to leave the LISP/VM environment.

FILELISP [fn [ft [fm]]]

to save the workspace and exit.

NAMELISP [fn [ft [fm]]]

to query or rename the current workspace.

READ fn [ft [fm]]]

to read from a file into an edit environment.

SAVELISP [fn [ft [fm]]]

to save the workspace.

WRITE fn [ft [fm]]

to prettyprint the top expression into a file.

23.11 Commands that affect the Lisp environment.

COMPILE [[xf] id]

to compile an operator definition.

DEFINE [[xf] id]

to cause an operator def. to be interpreted.

DISPOSITION xf mem (disp

to query or set the disposition of a file or member.

DISPXFILE xf (opt ...

to process all the members in an indexed file.

DISPXLOCALS xf

to load temporary macro defs. etc.

DISPXMEM xf mem (opt ...

to process a single member from a file.

EVAL cmd

to do a Lispedit command with evaluated arguments.

EXITLISP [rc]

to leave the LISP/VM environment.

FILELISP [fn [ft [fm]]]

to save the workspace and exit.

FILEXOB [[xf] id]

to file a definition in an indexed files.

LOADXOB [[xf] id]

to load objects stored in an indexed file.

MDEF [xf [mem]]

to macro-expand the focus.

NAMELISP [fn [ft [fm]]]

to query or rename the current workspace.

QUERY [[xf] id]

to show status of LOADed objects.

SAVELISP [fn [ft [fm]]]

to save the workspace.

SAVEXOB [[xf] id]

to save a definition in an indexed files.

TEST expr ...

to evaluate for effect only.

VALUES expr ...

to evaluate one or more expressions.

VALUESANDBREAK expr

to evaluate and take error breaks.

23.12 Commands that affect or describe the screen format.

SHOW to modify the screen format parameters.

24.0 Lispedit Command Descriptions

24.1 How to Read Command Descriptions

Each command description in the following pages consists of some or all of the following lines:

A —— Lispedit Command

Command name(s): A AB ABR ABRIDGE

ABRridge xxx yyy zzz --- Use this command to replace xxx yyy zzz with x y z.

The first line gives all the synonyms by which the command can be invoked. A Lispedit command is an input line that begins with a command name. When part of a command name is shown in lowercase, that part can be omitted. The arguments to the command are usually atoms separated by from the command name and from each other by blanks. If an argument is a list or vector, it must be closed properly so that it will not capture a following argument. Lispedit will supply closing parentheses or brackets for the last argument.

The text following the header lines is a description of the effect of the command and of various argument options.

24.2 Alphabetic List of Commands

/ —— Lispedit Command

Command name(s): /, LOC, LEX, LOCEXPR

/expr -- to shift the focus to the next occurrence of an expression EQUAL to expr. The search begins with the current focus, and continues depth-first left-to-right, until a match is found or until the end of the top expression is reached.

/ -- as above, using the most recent search argument.

/expr - -- to search for expr starting at the focus and proceeding to previous expressions.

/expr 0 -- to make expr the default search argument without changing the current focus.

/expr name [0 or -]

When the name argument is given, the name is defined as a command such that:

name [-]	continues the search forward [or backward]. The first step in the search is to descend the current focus.
name STEP [-]	continues the search and begins by moving forward [or backward].
name DESC [-]	continues the search and begins by descending the current focus
name TEST [-]	continues the search and begins by testing the current focus.
name LAST	shifts the focus to the most recent success.

name FIRST shifts the focus to the place where the command was defined.

ADDCOM —— Lispedit Command

Command name(s): ADDCOM, ADD

ADD name DO cmd ... -- to define name as a new command that performs the sequence of commands cmd ...

ADD name NIL -- to remove a user-defined command.

ADD name fn [comment] -- to define name as a command that invokes the function fn.

In this last case, fn must be a function expression or a quoted function name, since otherwise it will be interpreted as a command name. The function to be invoked must accept an indefinite number of arguments.

Example: ADD LX DO (LO X) (REP Z)
defines LX as a command that locates X and if found replaces it with Z.

ALLEXPR —— Lispedit Command

Command name(s): ALLEXPR, AEX

AXP expr [name [scope]] -- to locate one or all occurrences of expressions EQUAL to expr in the specified scope.

If the scope argument is omitted or 1, search only the current focus.

If scope is n, then search the current focus and the next n-1 expressions. If scope is * search the current focus and all the expressions up to the end of the current list or vector (including a non-NIL CDR).

If scope is S, search only the sub-expressions of the current focus. If scope is T, search the top expression.

If name is omitted or NIL, shift the focus to the first match, other matches cannot be reached. If name is an identifier, set it up as a command that will step through all the matches.

ALLPAT —— Lispedit Command

Command name(s): ALLPAT, APA

APA pattern [name [scope]] -- to locate one or all occurrences of a pattern in one or more expressions.

If the scope argument is omitted or 1, search only the current focus.

If scope is n, then search the current focus and and the next n-1 expressions. If scope is * search the current focus and all expressions up to the end of the current list or vector (including a non-NIL CDR).

If scope is S, search only the sub-expressions of the current focus. If scope is T, search the top expression.

If name is omitted or NIL, shift the focus to the first match, other matches cannot be reached. If name is an identifier, set it up as a command that will step through all the matches.

ALLPRED —— Lispedit Command

Command name(s): ALLPRED, APR

APR pred [name [scope]] -- to locate one or all occurrences of expressions that satisfy pred in the specified scope.

If the scope argument is omitted or 1, search only the current focus.

If scope is n, then search the current focus and the next n-1 expressions. If scope is * search the current focus and all expressions up to the end of the current list or vector (including a non-NIL CDR).

If scope is S, search only the sub-expressions of the current focus. If scope is T, search the top expression.

If name is omitted or NIL, shift the focus to the first match, other matches cannot be reached. If name is an identifier, set it up as a command that will step through all the matches.

ALTREP —— Lispedit Command

Command name(s): ALTREP, ALT, VL, LV

ALT -- to replace the focus with an alternate representation of the focus. Replace a list with a vector of the same elements; a vector with a list of its elements; and any other atom with a list of the characters in its external form.

If the focus is a list, the following forms may be used to get other representations:

ALT NUM -- to convert a list of digits into a number.

ALT CVEC -- to convert a list of identifiers and cvecs into one cvec.

ALT ID -- to convert a list of identifiers and cvecs into a single identifier.

ARCHIVE —— Lispedit Command

Command name(s): ARCHIVE

ARCHIVE [xf] [(opt ...] -- to dump the contents of an indexed file into a single sequential file. The EXHUME command can be used to recreate the indexed file from the archival file. Several indexed files can be archived in the same sequential file.

Options:

(ARCH fn [ft [fm]])	to specify the output file parameters. The normal default values are xfn LSDIRARC A.
ERASE	to erase an existing output file. If this option is omitted, the new output is added to an existing file.
RECLAIM	to dump only the most recent value of each property. This option dumps the file as if it were garbage-collected, but does not affect the file itself. If this option is omitted all the values of each property are dumped.

BACK —— Lispedit Command

Command name(s): BACK

BACK [n] -- to shift to the previous (or n-th previous) focus.

This command is particularly useful when another command has just shifted the focus to a surprising place and lost your place in the current expression.

See also FORTH. These commands can be used to review the recent history of the display.

BIGGER —— Lispedit Command

Command name(s): BIGGER, B, BIG

Bigger [n] -- to shift the focus to the nearest [or n-th] expression containing the current focus.

Bigger expr -- to shift the focus to an expression such that it contains the current focus and also begins with expr.

Bigger * -- to shift the focus to the top expression.

CASE —— Lispedit Command

Command name(s): CASE

CASE [U or M] ---- to cause or suppress uppercase conversion of all input. The normal mode of Lispedit is to accept all input as typed.

CASE [U or M] G ---- to specify the current conversion state and also change the global default used to initialize a new edit level.

CASE ---- without arguments, to display the current conversion state.

When the CASE-SHIFT readtable flag is in effect, the Reader folds all identifiers to upper case; in this situation CASE M is the most useful setting since it allows the contents of cvecs to be entered in mixed case. When the CASE-KEEP readtable flag is in effect, CASE U may be useful, for otherwise all the standard operators have to be entered in uppercase characters.

When uppercase conversion is suppressed, certain inputs are still converted to uppercase even when CASE-KEEP is in effect:
- the command token in a Lispedit command line
- SOME keywords in Lispedit command arguments (if a command refuses to recognize an argument or option, try again in uppercase)
- the entire argument string passed to CP/CMS by the CMS command.

CHANGE —— *Lispedit Command*

Command name(s): CHANGE, C, CH, CNG

C oldexpr newexpr -- to change the first occurrence of oldexpr in the current focus into newexpr. The search proceeds in a depth-first fashion.

C oldexpr newexpr count [scope] -- to change more than one occurrence of oldexpr, and to search the focus and expressions following the focus.

The count is specified as a positive integer, or as * to change all occurrences. The default is 1.

The scope of the change command is specified as * to include the focus and all expressions up to the end of the current list; or as a positive integer n, to include the focus and the next n-1 expressions.

Only list elements, vector elements and non-NIL atomic CDRs are tested during the search. A non-NIL CDR will be included in the scope only if it is specified as *.

CHECK —— *Lispedit Command*

Command name(s): CHECK

This command is effective only when the interactive interpreter (Heval) is in control.

CHECK e1 e2 e3 ... -- to add e1 e2 e3 ... to the list of checked expressions. All checked expressions are evaluated and displayed at the end of each evaluation step.

UNCHECK -- to turn off all checking.

UNCHECK e1 e2 e3 ...   -- to remove e1 e2 e3 ... from the list of checked expressions.

NOTE: The list of checked expressions is associated with a trapped function as long as trapping remains in effect.

CMS —— Lispedit Command

Command name(s): CMS

CMS x y z ...   -- to treat the string x y z ... as a CMS command. The entire string is converted to upper case.

CMS   -- if the command is issued with no argument, control is passed to the CMS SUBSET read-loop. The subset command RETURN will return control to Lispedit.

Since this function gets the CMS command string directly from the Lispedit input stream, it will work correctly only if is called by the command interpreter. Use the function OBEY to execute CMS commands from Lisp programs.

NOTE: Executing arbitrary CMS commands from the Lisp environment is a facility that may vary from one installation to another.

COME —— Lispedit Command

Command name(s): COME

This command is effective only when the interactive interpreter (Heval) is in control.

The effect of this command is to initiate continuous evaluation until the evaluation pointer (EP) is about to enter the current focus.

COME FAST   -- to suppress interactive execution of trapped functions during this step.

COMP —— Lispedit Command

Command name(s): COMP

This command is effective only when the interactive interpreter (Heval) is in control.

The effect of this command is to resume continuous evaluation until control leaves the current function. Heval will not stop at GO, EXIT, or RETURN statements; nor will there be an interaction before a value is returned.

COMP FAST   -- to suppress interactive evaluation of trapped functions until the next ineraction.

COMPILE —— Lispedit Command

Command name(s): COMPILE

COMPILE [spec] [(opt] -- to compile all the specified definitions.

The argument of the COMPILE command specifies one or more identifiers. For each id, if the value is a function or macro expression, the expression is preserved in a system table and the value of id is replaced with the compiled form of the expression. Subsequent EDIT commands will continue to show the uncompiled form.

SPEC	ID or IDS specified
=	the identifier whose value is the current top expression (this is the default)
id	the specified id
*	all the ids that have been previously mentioned in commands that deal with known ids (see HELP KNOWNID)
xf mem	all the ids defined by the specified member of an indexed file
xf *	all the ids defined in an indexed file The option (ONLY mem1 mem2 ...) can be used to specify only the members mem1 mem2 ...

COMPX —— Lispedit Command

Command name(s): COMPX

COMPX xf1 xf2 ... [(opt ...] -- to compare two or more indexed files. The result of this command is an information display that can be edited subsequently. ONLY the member names and the update dates of member properties are compared in this operation.

COMSEARCH —— Lispedit Command

Command name(s): COMSEARCH, CS

CS	- to display the current command search order
CS SET xf1 xf2 ...	- to redefine the command search order
CS ADD xf1 xf2 ...	- to add to the command search order
CS DROP xf1 xf2 ...	- to remove files from the current command search order
CS BASIC xf	- to respecify the basic command file

xf, xf1, xf2 ... are indexed file specs. (Read HELP XOB).

A command token in an input line is recognized by searching in the following order:
(1) the list of local commands (defined by ADDCOM)
(2) a list of command files that is set or modified by CS SET, CS ADD or CS DROP
(3) the basic command file defined by CS BASIC.

In CS ADD, if xf1 denotes a command file that already exists in the current search order, then xf2 ... are inserted in the search order following xf1. Otherwise, xf1 ... are inserted ahead of the current command files. Any inserted command file that already appears in the current search order is deleted before insertion.

CONT —— *Lispedit Command*

Command name(s): CONT

This command is effective only when the interactive interpreter (Heval) is in control.

Use CONT with no arguments to change the course of evaluation. The effect of this command to set the evaluation pointer (EP) at the initial step for the current focus. You CANNOT set the EP at arbitrary locations in a function. In the general, you can point to any expression containing the current EP; you can point to an expression that would be a valid GO target (if it had a label); you can also point to a previously evaluated argument of a function that has not yet been applied.

Use   CONT expr   to substitute expr for the current EP. This form can be used to avoid evaluating an expression that you know will not work; it can also be used to substitute a correct value after an incorrect value has been computed. Since expr is evaluated, dont forget to QUOTE constants. This form can also be used to alter the flow of control, since expr may be a GO, EXIT or RETURN expression.

COPY —— *Lispedit Command*

Command name(s): COPY

COPY [count [to__tag [from__tag]]]   -- to copy and remember one
   [or count] expressions starting with the focus. The copy can
   be inserted subsequently with the TO command.

count may be a positive or negative integer, *, or -*.

If to__tag is given, shift the focus to to__tag and insert the copied expression(s) at the tagged place.

from__tag is used to tag the place where the copy command was called.

COPYX —— *Lispedit Command*

Command name(s): COPYX

COPYX [xf1 [mem1 [prop1]]] (kind xf2 [mem2 [prop2]] -- to copy an indexed file object from one file to another.

The defaults for xf1, xf2, mem2 and prop2 are all =
The defaults for mem1 and prop1 are * .
(Read HELP XOB for details).

kind is TO or REP

The combination xf mem prop denotes a source object, and xf2 mem2 prop2 denotes a destination object. Note that = in the source object denotes the corresponding file system default, while = in the destination object denotes the corresponding source parameter. The TO form assumes the destination object does not exist; the REP form erases an existing destination object before making the copy.

FORM	EFFECT
xf * *	To copy an entire file. If REP is specified, the destination file is erased; otherwise xf2 must specify a new file. mem2 and prop2 must be omitted or =.
xf mem *	To copy a member. If REP is specified, the destination member is erased; otherwise xf2 mem2 must specify a new member. prop2 must be = or omitted.
xf mem prop	To copy a property. REP and TO behave identically in this case.
xf * prop	To copy a property for all members in xf. If a member has the specified prop in xf, it is copied to xf2; if the source property is missing, then if REP is specified, a value of NIL is copied, otherwise the destination member is not changed or created.

DECLARE —— Lispedit Command

Command name(s): DECLARE, DEC, DCL

DEC item ... -- to modify the bv list of the nearest PROG expression that contains the focus. The items in the argument list are processed from left to right to produce a cumulative effect.

ITEMS:

id	to add the id to the bv list if it is not already there.
(DROP x y ...)	to remove x y ... from the bv list.
(DROP)	to replace the bv list with NIL.
(LEX x y ...)	If x ... is in the bv list, make sure it is declared lexical, if not add to the bv list.
(LEX)	to make all the vars in the bv list lexical.
(FLUID x y ...)	if x ... is in the bv list, make sure it is declared fluid, otherwise add (FLUID x) to the bv list.
(FLUID)	make all the vars in the bv list fluid.

DEFINE —— *Lispedit Command*

Command name(s): DEFINE

DEFINE [spec] [(opt] -- to cause all the specified operator definitions to be interpreted.

The argument of the DEFINE command specifies one or more identifiers. For each id, if the value is a compiled function or macro, and if the expression form of the operator can be found, the compiled definition is discarded, and replaced by the interpreted expression form.

SPEC	ID or IDS specified
=	the identifier whose value is the current top expression (this is the default)
id	the specified id
*	all the ids that have been previously mentioned in commands that deal with known ids (see HELP KNOWNID)
xf mem	all the ids defined by the specified member of an indexed file
xf *	all the ids defined in an indexed file The option (ONLY mem1 mem2 ...) can be used to specify only the members mem1 mem2 ...

DEL —— *Lispedit Command*

Command name(s): DEL, DELETE

DEL [n] -- to delete 1 [or n] list elements starting at the focus. The new focus is the Next item after the last item deleted.

DEL -n -- to delete n list elements before the focus, starting with the focus. The new focus is the item Previous to the last item deleted.

DEL * -- to delete the focus and all list elements to the end of the current list.

DEL -* -- to delete the focus and all list elements to beginning of the current list.

The RESTORE command can be used to recover from the most recent DEL.

DISPOSITION —— *Lispedit Command*

Command name(s): DISPOSITION, DI, DIS, DISP

DIsposition [xf [mem]] -- to query the effective disposition of a member in an indexed file.

DIsposition xf * -- to query the default disposition of all members in an indexed file.

DIsposition [xf [mem]] (newdisp -- to define the

disposition of a member in an indexed file, or to
define the default disposition of all members.

The disposition of a member in an indexed file specifies what must be done when the member is brought into the Lisp environment. Typical dispositions for function definitions are DEFINE and COMPILE. A disposition of SETQ allows a value from a file to be assigned to a variable. A disposition of EVAL causes the value stored in the file to be evaluated when loaded. A disposition of NONE causes the filed value to be ignored.

There are many more dispositions with more subtle effects. In the absence of any other specification, the disposition of every member in a file is DEFINE.

If the indexed file has a value for the property DISPOSITION then that value is the default disposition for every member in the file.

```
     This value is queried with:  DI xf *
     It is modified with:  DI xf * ( newvalue
```

If a member has a value for the property DISPOSITION, then that value is the disposition for that member.

```
     This value is queried with:  DI xf mem
     It is modified with:  DI xf mem ( newvalue
```

All of the above decisions can be overridden by options in the commands DXM and DXF.

DISPXFILE —— Lispedit Command

Command name(s): DISPXFILE, DXF

DXF [xf] [(option ...]   -- to perform the appropriate disposition operation for each member in an indexed file. The default disposition is to assume that the VALUE property of each member is a function definition that is made effective in the current Lisp environment.

The file argument default is =. Read HELP XOB for details.

DXF [xf] (COMPILE [FILE]   -- is a common form. The effect of this form is to compile all definitions into the current Lisp environment. The FILE option directs the output of the compiler into an intermediate file that can be subsequently loaded with the   LOAD xf   command.

Read HELP DOPT for a full description of available options.

DISPXLOCALS —— Lispedit Command

Command name(s): DISPXLOCALS, DXL

DXL [xf] -- to define in the global environment all the macro definitions local to the given indexed file.

Local macro definitions are members with dispositions of TEMPDEFINE, SETQ, ADDTEMPDEFS and the OR... and MA... variants of these. These definitions affect only the compilation environment for other members in the file; they normally do not appear in the normal Lisp environment or in the default compile environment.

This command is useful if the local macros are needed in the global environment, for example if Heval is evaluating a definition that uses these macros.

DISPXMEM —— Lispedit Command

Command name(s): DISPXMEM, DXM, DMEM

DXM [xf [mem]] [(opt ...] -- to perform the appropriate disposition operation for one member in an indexed file. The default disposition assumes that the VALUE property of the given member is a function definition that is made effective in the current Lisp environment.

The file argument defaults are = = respectively.
Read HELP XOB for a complete description.

DXM [xf [mem]] (COMPILE [FILE] -- is a common alternate form. The effect is to compile the function definition into the current Lisp environment. The FILE option directs the output of the compiler to update an intermediate file.

Read HELP DOPT for a full description of available options.

If mem is given as *, DXM recognizes the option (ONLY n1 n2 ...) to dispose of members n1, n2 ... in that order.

If the top expression is a single property value, DXM recognizes the option LOCAL. The effect is to use the top expression to build the disposition expression, without updating or referencing the value stored in the file.

DOCOMMANDS —— Lispedit Command

Command name(s): DOCOMMANDS, DO

DO c1 c2 c3 ... -- to perform Lispedit commands c1, c2, c3 ...

Each command ci may be of the form:
- cmnd-token for commands with no arguments
- (cmnd arg1 arg2 ...) for commands with arguments.

The commands are executed only as long as they succeed. The first failure terminates the DO command. As a result, the following recursive example is useful:

```
ADDCOM X DO (LO ABC) (I DEF) X
```

This command will scan from the current focus to the end of the top expression, and insert the atom DEF after every occurrence of ABC.

The advantage of this command over stacked sequence of commands is that the entire sequence is remembered as the previous input. It is also useful in conjunction with the MAP command.

EDIT —— Lispedit Command

Command name(s): EDIT, E, ED, EDI

Edit expr -- to invoke Lispedit recursively and generate a new edit environment with a new top expression and focus and a new set of environment parameters. The value of expr becomes the top expression in the new environment.

Edit & -- to edit the value of the current focus in a new environment.

Use the QUIT command to return to the invoking environment. After a QUIT is executed, the form below can be used to resume the discarded edit environment.

EDIT -- to resume at the most recent use of QUIT, or to enter a new and empty edit environment if QUIT has not been used yet. Read HELP RESUME for some more details on resuming edit environment. An alternate way to preserve an edit environment is to invoke the NAME command in that environment.

When EDIT is invoked to edit the value of an identifier, the indexed file parameters in the new environment are set to the parameters associated with that identifier by a previous EDIT or LOAD command. When an id is edited for the first time, the indexed file named in DEFAULT-INDEXED-FILE is associated with the id. The id is used as the member name and the property name is set to VALUE. The id is subsequently known to Lispedit and the associated file parameters and expression are shown whenever that identifier is edited, even when the value is compiled.

EDIT xf mem -- to edit the source expression form of the identifier associated with the specified object in an indexed file.

If EDIT is invoked with an arbitrary expression, the new environment inherits the indexed file parameters of the invoking environment.

(EDIT [expr]) is a Lisp form that will invoke Lispedit from any environment.

HEVAL NOTE: During interactive evaluation, the variables ,EXP and ,VAL refer to the most recent expression and value generated by Lispedit if they are mentioned in an expression entered by the user. In the user program these variables evaluate to the corresponding value in the program. The expression (CURS) always evaluates to the focus in the user program; in trapped functions this is never the current focus on the screen.

EDITHERE —— Lispedit Command

Command name(s): EDITHERE, EH

EH expr -- to replace the current top expression with the value of expr.

EH xf mem -- to replace the current top expression with the source form of the value of the identifier LOADed from the specified file.

If expr is an identifier, and the top expression is replaced subsequently, the binding of the identifier will be updated. Note also that the object being edited is the actual value of expr, not a copy. As a result any changes made to the expression will be made in place to that object.

An abreviated form of expr is displayed in the fence line. If expr is not an identifier, and the top expression is subsequently replaced, expr is removed from the fence line since the pointer to the original expression has been lost.

ERASEX —— Lispedit Command

Command name(s): ERASEX, ERX

ERX xf [* [*]]	- to erase an indexed file entirely.
ERX xf mem [*]	- to erase a member from a file. The member name will remain in the file, with no properties until RECLAIMX removes it.
ERX xf mem prop	- to erase a property from a member.
ERX xf * prop	- to erase a property from all members in a file.

Read HELP XOB for details of indexed file object names.

Note, that like all update operations, this command does not update ancestor files. In order to delete an inherited property, it is necessary to PUT a value of NIL to mask the inherited value.

EVAL —— Lispedit Command

Command name(s): EVAL, EVA, EV

EVal cmnd arg1 [arg2] ... -- to do a Lispedit command with evaluated arguments. cmnd must be a valid Lispedit command token. arg1, arg2, ... may be any Lisp expressions.

Example:

EV C (CADR (CURS)) "FOO "* will change all occurrences of the CADR of the current focus to FOO. This may be convenient if the expression to be changed is too big to enter in the command line.

HEVAL NOTE: During interactive execution, all evaluations are done in the user program state; this environment does NOT include the current edit environment, therefore references to edit parameters such (CURS) will yield surprising results.

EVALPOINTER —— Lispedit Command

Command name(s): EVALPOINTER, EP

This command is effective only when the interactive interpreter (Heval) is in control.

The effect of this command is to position the focus at the evaluation pointer (EP). Sometimes the EP is at a place that cannot be shown in the source program; in that case, the focus points to the smallest source expression that contains the EP.

In a normal edit session, there is only one focus that is selected by the user. During interactive evaluation, on the other hand, there may be two expressions of interest: one is the focus selected by the user edit commands, another is the EP that points to the expression that is of current interest to the Heval.

During interactive evaluation, the left-hand text on the fence line identifies the meaning of the brightened expression on the screen:

At next expr.	- this message means that the focus is the next expression that the interpreter will evaluate.
At last expr.	- this message means that the focus is the last expression that has been evaluated by the interpreter. The next step is either the completion of a larger expression or the beginning of another.
Not at next exp	- this message means that the focus is not the EP, and that the next step in evaluation is to start a new expression.
Not at last exp	- this message means that the focus is not the EP, and that the last step in evaluation was to finish evaluating an expression.
Above next expr	- this message means that the interpreter is

about to start an expression that was derived from the focus but cannot be displayed in the source pgm.

Above last expr - this message means that the interpreter has just finished an expression as above.

EXHUME —— Lispedit Command

Command name(s): EXHUME

EXHUME opt ... ---- to recreate indexed files from a dump file created with the ARCHIVE command.

Options:

(FILE fn [ft [fm]]])	to specify the input file. This option is REQUIRED. The normal defaults are LSDIRARC *.
n	to specify the number of files to be restored from the file. The default is one (1).

The name of each file is archived by the dump command. EXHUME creates a file by that name on the A-disk. An existing indexed file with the same name will be DESTROYED.

EXITLISP —— Lispedit Command

Command name(s): EXITLISP, EXIT

EXITlisp [rc] -- to leave Lisp and return to the environment that invoked Lisp. This command normally return to CMS and discards the current Lisp session unless it has been saved explicitly. The argument if given must be a small positive integer; this number will be the CMS return code of the invokation of Lisp.

EXPORT —— Lispedit Command

Command name(s): EXPORT

EXPORT [xf] [(opt ...] -- to create a sequential file from an indexed file or from a subset of one. The resulting file can be executed with the EXF function to obtain a result similar to DXF. This feature can be used to maintain Lisp functions with Lispedit and transmit the source definitions to a Lisp system without Lispedit.

Options:

(ARCH fn [ft [fm]])	to specify the output file parameters. The normal default values are xfn LISP A. The syntax of the output file is controlled by the file type and the current value of FILE-SYNTAX.
ERASE	to erase an existing output file. If this option is omitted, the new output is added to an existing file.
NPP	to suppress PRETTYPRINTing to the output file.
(ONLY m1 m2 ...)	to output only members m1 m2 ... in that order.

FILELISP —— Lispedit Command

Command name(s): FILELISP

FILELISP [fn [ft [fm]]]xxx-- to save the current workspace on disk and return to the environment that invoked LISP/VM.

The default values for the arguments are those used with the most recently saved workspace. They can be interrogated and changed with the NAMELISP command. See also SAVELISP.

FILEXOB —— Lispedit Command

Command name(s): FILEXOB, FILEX, FILE

FILE [=] -- to SAVE the current top expression in an indexed file and LOAD the saved value into the Lisp environment.

If this operation is completed successfully, exit the current edit environment.

This command is effective only when the top expression is an object from an indexed file (xob) or the value of an identifier.

FIN —— Lispedit Command

Command name(s): FIN

This command is effective only when the interactive interpreter (Heval) is in control.

The effect of this command is to start continuous evaluation until the evaluation pointer is about to leave the current focus. A typical use may be when the evaluation pointer is within the operator of a MAPCAR expression and you would like to see the result of the MAPCAR without going through all the intermediate steps.

FIN FAST -- to suppress interactive evaluation of trapped functions until the next interaction.

FLATTEN —— Lispedit Command

Command name(s): FLATTEN, FL

FLatten -- to remove one level of list structure from the focus.

This command is effective only when the focus is a sub-list of another list. The list elements of the focus become successive elements of the list that contained the focus. The new focus is the first element of the flattened list.

The focus is shown in capitals in the following example:

Before	(a b (X Y Z) c d)

After FL (a b X y z c d)

FLUSH —— *Lispedit Command*

Command name(s): FLUSH

FLUSH [n] -- to clear the message history and reset the capacity to n lines of messages.

Lispedit maintains a history of the most recent messages shown in the message area. This history can be examined using REVIEW mode and values shown partially in the message can be retrieved and examined in their full form.

It is sometimes desirable to discard this history in order to release some heap or stack space or in order to discard an invalid pointer. It may also be desirable to retain more or fewer lines and values than the 50 that Lispedit normally retains. The parameter n when given must be a number from 1 to 100, otherwise the value 50 is used.

FORTH —— *Lispedit Command*

Command name(s): FORTH

FORTH [n] -- to scroll forward (in time) trough previously displayed foci.

See also BACK. These commands can be used to review the recent history of the display.

FORWARD —— *Lispedit Command*

Command name(s): FORWARD, FWD, F

FWD [n] ---- to shift the focus to the next [or n-th] following expression in any list or vector containing the current focus.

This command is useful when the focus is near the end of several levels of list structure and you want to shift to the next expression in the nearest surrounding list.

GETHERE —— *Lispedit Command*

Command name(s): GETHERE, GETH, GH

GH [xf [mem [prop [age]]] -- to replace the current top expression with a value fetched from an indexed file. The default arguments are = = VALUE and NIL respectively, (see HELP XOB for details).

GH xf * [prop [age]] ((ONLY m1 m2 ... -- to replace the top expression with a list that consists of the property lists of members m1 m2 ... .

GH xob (KXD -- to keep the current indexed file defaults if they are conformal.

This command changes the current indexed file defaults and the description of the current top expression. If the specified object does not exist, the top expression is usually NIL. You can use REP and other edit operations to create a suitable object. When the top expression is subsequently filed with PUT, all the necessary file components will be created.

GETMEM —— Lispedit Command

Command name(s): GETMEM, GM

GM [mem [prop [age]]] [(opt ...] -- to edit recursively the specified member from the current default indexed file.

This command is equivalent to:

GET = [mem [prop [age]]] [(opt ..]

GETMEMHERE —— Lispedit Command

Command name(s): GETMEMHERE, GMH

GMH [mem [prop [age]]] [(opt ...] -- to replace the top expression with the specified member from the current default indexed file.

This command is equivalent to:

GH = [mem [prop [age]]] [(opt ..]

GETXOB —— Lispedit Command

Command name(s): GETXOB, GET

GET [xf [mem [prop [age]]] -- to create a new Lispedit environment where the specified object is the top expression. Each time this command is used, a new copy of the object is fetched from the file. The default parameters are =, =, VALUE, and NIL respectively, (see HELP XOB for details).

GO —— Lispedit Command

Command name(s): GO

GO xxx -- to shift the current focus to a previously TAGged expression. xxx may be an identifier or an integer.

HELP —— Lispedit Command

Command name(s): HELP, H, HE, HEL

The HELP command provides a variety of on-line documentation services. The information is presented in a display that temporarily overlays the normal Lispedit screen. Press ENTER or any PF key but 1 to continue; if you press ENTER and you have entered a command on the command line, then the command will be executed. Press PF1 to edit the text on the screen if it contains ellipses.

HELP KEY	displays a list of keywords that classify Lispedit commands into sub-groups
HELP BASIC	displays a short description of the most frequently used commands

HELP xxx displays information about xxx

if xxx is one of the keywords mentioned above, display a short description of all commands in the sub-group

if xxx is Lispedit command, display a description of the command

if all else fails, display some information about the Lisp meaning of xxx

More Specialized Forms of HELP:

HELP ENV	displays the settings of Lispedit environment parameters
HELP KEY keywd	displays a short description of the commands associated with keywd
HELP COM cmnd	displays a description of the command cmnd
HELP LISP id	displays information about the Lisp meaning of id
HELP WHAT	displays a list of all the commands available in Lispedit
HELP	displays a short description of the Lispedit interface

HEVAL —— Lispedit Command

Command name(s): HEVAL

HEVAL e1 e2 e3 ... -- to evaluate the focus interactively.

If the focus is a function expression, it is applied to the values of e1, e2, e3, ...

If the focus is an macro expression, it is applied to e1, and the result is the value of the macro application.

During interactive evaluation, the focus is shifted in step with the evaluation pointer (EP). The commands STEP, RUN, FIN, COME and STEPF control the progress of evaluation. The CONT command changes the EP or substitutes an alternate expression for the current one. STOP aborts evaluation, QUIT aborts with a value. VALUE displays values of variables, EP resets the focus to the current EP.

The TRAP command changes a function definition so that it is evaluated interactively every time it is called.

Other Lispedit commands are available as in the normal environment.

IMPORT —— Lispedit Command

Command name(s): IMPORT

IMPORT opt ...   -- to create or update a indexed file from a sequential Lisp input file.

Options:

(FILE fn [ft [fm]]) specifies the input file to be used. This option is REQUIRED. Defaults are are taken from the value of EXF-FILETYPES. The syntax of the input file is controlled by the value of FILE-SYNTAX.

EXPAND to create a member for each function in a multi-function DEFINE or COMPILE expression.

NOEX or NOEXPAND to suppress the expansion of DEFINE and COMPILE arguments. This is the default setting.

(XF . SAME) to use the file name as the indexed file name. This is is the default setting.

(XF xf) to specify the indexed file explicitly.

NEW to create a new file. This is the default setting.

ADD to add to an existing indexed file.

(RMAP (key namefun) ...) If an expression is not recognized by IMPORT, it is filed under a generated member name with disposition EVAL. If this option is given, expressions of the form (key . x) are filed with a member name supplied by namefun applied to the input expression.

(xxx namefun) options of this form that are not one of the above, are added to the RMAP option.

INSERT —— Lispedit Command

Command name(s): INSERT, IN, INS, I

I e1 [e2] ...   -- to insert one or more expressions to the right of the current focus. The new focus is the last inserted expression.

I -- with no arguments, this command changes the state of the Lispedit input area. Subsequent inputs are not treated as commands or expressions to be executed. Instead, all inputs are treated as data to be inserted in the top-expression. After each line of input, the new focus is the last complete expression on that line. Unballanced closing parentheses or brackets cause the focus to shift up. A null line restores Lispedit to command mode. If the first character on a line is the special character $ then the remainder of the line is executed as a command or expression.

Read HELP INM for additional options available in input mode.

INSERTBEFORE —— *Lispedit Command*

Command name(s): INSERTBEFORE, INB

INB a b c ... -- to insert a b c ... before the current focus and leave the focus unchanged.

INSERTMODE —— *Lispedit Command*

Command name(s): INSERTMODE, INM

INM -- to change the state of the Lispedit input area.
Subsequent inputs are not executed as commands or expressions.
Instead, they are scanned and used to modify the top expression.

- a complete s-expression is inserted to the right of the current focus or replaces the current focus.
- a closing parenthesis or bracket shifts the focus up one level.
- an opening parenthesis or bracket causes the expression (]focus[) to be inserted following the current focus or to replace the current focus. The new focus is the CAR of that expression, and the input state becomes (Replace).

When the fence line shows (Insert) the next expression is inserted to the right of the focus. When it shows (Replace) the next expression will replace the focus.

A null line restores Lispedit to command mode. In (Insert) or (Replace) mode, if the first character on a line is $, then the remainder of the line is executed as a command or expression. When INM is invoked and the focus is the top expression, and atomic CDR or a vector element, the initial mode is (Replace); otherwise the initial mode is (Input).

INM REP -- to force (Replace) as the initial mode.

INM [REP] (expr -- to shift the focus to expr and then switch to (Input) or (Replace) mode.

Note: Vectors will show as lists until the closing bracket is entered. Identifiers, numbers and constant vectors cannot be longer than one line.

LIFT —— *Lispedit Command*

Command name(s): LIFT

LIFT [arg] -- to replace a higher expression with the current focus.

If arg is omitted (or a number), the focus is "lifted" one (or arg) levels up in the current list structure. If arg is not a number, the focus is lifted up to an expression that has a CAR that is EQUAL to arg.

The focus is underscored in the following examples:

```
Before              (PLUS 1 (COND ((ATOM X)  (FOO X))('T 17)))
                    -------
After LIFT COND     (PLUS 1     (FOO X))
                                -------
```

LINKX —— *Lispedit Command*

Command name(s): LINKX

LINKX xf1 xf2 xf3 ... -- to establish a parent/child relation among the specified indexed files. Xf1 is made a child of xf2, xf2 a child of xf3, etc.

LINKX xf -- to cut the parent link in an indexed file.

For retrieval purposes, a child inherits the contents of all the parent files. If several files contain an entry then the entry in the nearest parent wins. Update operations are always done to the explicitly named file.

The order of inherited member names is as follows:

    Names in oldest parent
    Names in next oldest parent but not in oldest parent
    ...
    Names in root child but not in any parent file.

Thus it is not possible to insert a new name in a child to precede a name in a parent.

When a list of files is specified in the LINKX command, each name may denote an existing file or a new file. If a name denotes a new file, a file with that name is created. In all cases, the name must be unique in all visible disks and it is recommended that parents in general have unique names. As a result, a mode spec. is superfluous for an existing file. A mode spec. may be given for a new file in order to create it on a given disk. A mode spec. should never be used to disambiguate between files of the same name.

LISTX —— *Lispedit Command*

Command name(s): LISTX, LX

LX [xf [mem [prop]]] [(option ...] -- to display data about one or more indexed files without affecting the file parameters in the current edit environment.

Indexed file defaults are = * * (Read HELP XOB for details).

The options XF MEM PROP VAL XFUP MEMUP determine the depth of detail in the information displayed.
A number in the list of options determines the depth to which ancestor files are examined for members and properties.
Normally, only files on the A-disk are examined. The DISK option extend the search to other disks.

DETAILED DESCRIPTION OF OPTIONS:

Option	To show:
XF	only indexed file name
XFUP	file name and dates of most recent updates
MEM	member name only
MEMUP	member name and dates of updates to member
PROP	names of existing properties
VAL	names and values of properties
XFPROPS	names of file properties
(XFPROPS p1 p2 ...)	names and values of file properties p1 ..

When xf is specified as * the indexed files on the disk(s) specified by the DISK option are searched:

omitted	(DISK A)
DISK	(DISK A)
(DISK *)	all linked disks are searched
(DISK m1 ...)	only the specified disks are searched in alphabetic order.

When searching disk files for a particular member, the default action is to stop the search as soon as the member is found. The ALL option forces a scan of all files.

If mem is * then (ONLY m1 m2 ...) can be used to specify the members to be displayed.
If prop is * then (PROPS p1 p2 ...) can be used to specify the propeties to be displayed.

ARG PATTERN	DEFAULT DEPTH OPTION
* * *	XF
not* * *	MEM
any not* any	PROP
any any not*	PROP

If the MEM option is specified, inherited members are displayed
as (mem Inherited__from: depth . xf)
or (mem . Not__found) if their depth exceeds the given depth limit.
If the PROP option is given, a similar form is used for properties.

LOADXOB —— Lispedit Command

Command name(s): LOADXOB, LOADX, LOAD

LOAD xf * [((ONLY mem1 mem2 ...]
or
LOAD xf mem -- to load all or part of a Lispedit indexed file into the current workspace. An indexed file normally contains function and value definitions associated with an identifier and a disposition.

Once an object is loaded, the file parameters of the LOAD operation are associated with the identifier to which the object was assigned. Whenever the value of this identifier is edited, the same parameters become the default file parameters of that edit environment.

LOAD id -- to reload the value of an identifier from the indexed file associated with id by a previous LOAD or SX command.

LOAD [=] -- to reload the current edit object.

The identifiers used as keys in the file are called members, and the disposition determines how the value associated with a member is to be loaded. Typical dispositions are DEFINE, COMPILE, SETQ and EVAL.

The LOAD command performs the appriopriate disposition for each member specified. This process maintains and uses an intermediate file that contains a current, fast loading, form of the value.

The commands DXF and DXM and the Lisp functions LOADVOL and SUBLOAD should be used if more detailed control of disposition and loading is necessary.

LOCPAT —— Lispedit Command

Command name(s): LOCPAT, LPA

LPA pattern -- to shift the focus to the next expression that matches the pattern. The search starts with the current focus and continues depth-first and left-to-right.

LPA [pattern [name] [-or 0]] -- read HELP LEX for a description of all the options on search commands.

Pattern syntax:

?	- matches any expression
(???)	- matches zero or more expressions in a list
<???>	- matches zero or more expressions in a vector
?xxx	- any identifier beginning with ? is either a pattern variable or a keyword,

All other expressions are constants that match EQUAL expressions.

LOCPRED —— Lispedit Command

Command name(s): LOCPRED, LPR

LPR pred -- to shift the focus to the next occurrence of an expression that satisfies the predicate pred. The search begins with the current focus and continues depth-first left-to-right.

LPR [pred [name] [0 or -]] -- the options available with LPR are the same as those described for LEX.

MAP —— *Lispedit Command*

Command name(s): MAP

MAP [limit] cmd -- to perform cmd at the current focus, shift to the right and repeat, until limit is reached. If limit is a number, repeat that many times. If limit is omitted or *, repeat until the focus is the last element of a list or vector.

MAP [limit] DO cmd1 cmd2 ... -- to repeat a sequence of commands.

In all cases, iteration stops if any command fails. This feature can be used to repeat commands until a condition is satisfied by combining search commands and the TEST command in appropriate ways.

MDEF —— *Lispedit Command*

Command name(s): MDEF

MDEF -- to macro-expand the focus in the normal environment of compilation. The result is shown in the message area.

MDEF xf [mem] [(opt] -- to expand the current focus in the macro environment defined by xf and mem. The default for mem is =.

The only allowed option is EXPAN, NOEXPAN or (EXPAN . opt) as defined for the DXM command.

MOVE —— *Lispedit Command*

Command name(s): MOVE, MV

MOVE [count [to_tag [from_tag]]] ---- to copy and delete one [or count] expressions starting with the focus. The copy can be inserted subsequently with the TO command.

count may be a positive or negative integer, *, or -*.

If to_tag is given, shift the focus to to_tag and insert the copied expression(s) at the tagged place.

from_tag is used to tag the place where the copy command was called.

Num —— *Lispedit Command*

Command name(s): Num, NUMBER

An input line beginning with a positive or negative small integer is interpreted by Lispedit as a general traversal command.

A positive number moves the focus forward and to smaller expressions.

A negative number moves the focus to previous and bigger expressions.

Steps are counted and attempted in this order:

+n	Next, Smaller, Bigger and Next. The NIL at the end of a list is not counted. For large n, traversal stops on the rightmost non-nil atom in the top expression.
-n	Prev, Bigger.

NAME —— Lispedit Command

Command name(s): NAME

NAME name -- to name an edit environment so that it can be resumed after a QUIT. The argument is defined as a command that resumes the named environment. The new name will also show in the fence line.

NAME NIL -- to erase the name of the current edit environment. Un-named Lispedit environments show as an integer recursion level on the fence line. They can be resumed after a QUIT with the RESUME command if no intervening calls to Lispedit have been made.

NAMELISP —— Lispedit Command

Command name(s): NAMELISP

NAMELISP -- to query the file parameters associated with the current workspace.

NAMELISP fn [ft [fm]] -- to modify the file parameters associated with the current workspace. These parameters are used by the SAVELISP and FILELISP commands and functions.

NEXT —— Lispedit Command

Command name(s): NEXT, N, NE, NEX

Next [n] ---- to shift the focus one [or n] expressions closer to the tail of the list or vector containing the focus.

Next * ---- to shift the focus to the last expression in the list or vector containg the focus.

When the focus is the last element of a list, NEXT 1 will shift the focus to the atomic CDR of the list.

ORDERX —— Lispedit Command

Command name(s): ORDERX

ORDERX [xf [mem]] (mem1 mem2 mem3 ... ---- to re-order members in an indexed file. Ordering is significant when the disposition of one member may interact with that of another, as in the case of TEMPDEFINE and COMPILE.

The file argument defaults are = * respectively.

If mem is given as *, move members mem1 mem2 mem3 ... to the head of list of members in the indexed file.

If mem is not given as *, it must be an existing member. The file is re-ordered so that mem1 mem2 mem3 ... are immediate successors of mem in the ordering.

In both cases, if memi is invalid, it is ignored. If memi specifies a new member, a new member is created with no properties.

PF —— Lispedit Command

Command name(s): PF, PFKEYS

PF --- to display all PF-key settings.

PF mode --- to display the PF-key settings associated with the specified mode.

mode is one of INPUT HEVAL REVIEW or NORMAL

PF num [mode] [def [doc]] [G] --- to define a PF-key.

num is a number from 1 to 12 that specifies the key, or * to specify all undefined keys.
mode is as above to specify when the key should be effective. The default mode is NORMAL.
def is NIL, a character string, or the name of a special PF-key command.
NIL indicates that the definition should not change.
A string specifies the new setting of the key.
Pressing the key is equivalent to typing the string on the command line.
A name denotes a special PF-key action.
SELECT is used to point with the cursor
doc is a short string that is used to document the command setting on the line above the command area.
G specifies that the PF-key setting should affect all edit sessions that are created subsequently. Normally, a change to a PF-key affects only the current edit session.

PF RESET --- to set the PF-keys in the current session from the current global definition. This call is necessary if you want to use in the current session a PF-key defined with the G option.

PF RESET G --- to reset the global PF-key settings to the built-in default values.

This command is a common candidate for a Lispedit profile, read HELP PROFILE for details.

PREF —— *Lispedit Command*

Command name(s): PREF, PREFI, PREFIX

PREFIX a b ... c -- to replace the focus d with the expression
(a b ... c . d)

If the focus is the list (d e f) the the focus after PREFIX a b c
will be the list (a b c d e f)

NOTE that this command will behave somewhat inconsistently if the focus is the top expression or an atomic CDR.

If the focus is the top expression (d e f) then the focus after
PREFIX a b c will be the NEW top expression (a b c d e f)

If the focus is the atomic top expression d then the focus after
PREFIX a will be the NEW top expression (a . d)

If the focus is the atomic CDR d then the focus after
PREFIX a b c will be the expression a in the updated
tail of list (a b c . d)

PREVIOUS —— *Lispedit Command*

Command name(s): PREVIOUS, P, PR, PRE, PREV

Prev [n] -- to shift the focus one [or n] expressions closer to the head of the list or vector containing the focus.

Prev * -- to shift the focus to the first (leftmost) expression in the list or vector containing the current focus.

PRINTX —— *Lispedit Command*

Command name(s): PRINTX, PX

PRINTX [xf] [(opt1 opt2 ...] -- to produce a listing file from a Lispedit indexed file. If the command is called with no arguments, make a listing file from the current default file; the listing file will show all properties of all members in the file; a header line separates the members and an index is included at the end. The output replaces the file xfn PRINT A.

Options may be specified to alter the format and contents of the file.

Option:	Effect:
(LISTing fn ft fm)	to direct the output to the specified file; the defaults are PRINT for ft and * or A for fm.
ERASE or APPEND	to erase an existing output file or to add to it.
HEADS or NOHeads	enables or disables the printing of header lines

INDEX or NOIndex	enables or disables the printing of an index.
(PAGE n)	The index is produced on the assumption that the page length is 55 lines. This option changes the page length to n.
n	a number specifies the depth to which ancestor files are searched.
(ONLY mem1 mem2 ...)	can be used to print only members mem1, mem2, ... in the order given.
(IF pred)	can be used to control which members are printed. The predicate pred must be a function of two arguments. If the value of (pred xf mem) is non-NIL, mem is included.
(PROPS p1 p2 ...)	can be used to print only properties p1, p2, ... for each member.

A sequence of PRINTX commands can be entered to produce a listing file with a cumulative index to all the members printed in the file. The index is sorted by member name and shows the file and page for each member. The following sequence will print files FOO, BAR and BAZ in a file BIG MYLIST T1, and add a cumulative index:

```
PRINTX FOO (BIGINDEX (LIST BIG MYLIST T1
PRINTX BAR (BIGINDEX
PRINTX BAZ (BIGINDEX
PRINTX (CLOSE
```

PUTFOCUS —— Lispedit Command

Command name(s): PUTFOCUS, PUTF

PUTF [xf [mem [prop]]] [(opt ...] -- to store the current focus in a Lispedit indexed file with the given file parameters.

The indexed file argument defaults are = = = respectively. Read HELP XOBnfor a more complete description.

Options:
The REP option is the same as for PUT.
The WRAP option files the value (focus). This is convenient when the focus is a function expression and you want to file it as a normal VALUE property value.

PUTXOB —— Lispedit Command

Command name(s): PUTXOB, PUT

PUT [xf [mem [prop]]] [(REP] -- to store the current top expression in a Lispedit indexed file.

The default indexed file arguments are = = = respectively. Read HELP XOB for a full description of indexed file arguments.

In contrast with the SAVE and FILE commands, this comnmand affects only the specified indexed file. It does not affect the current edit environment or the Lisp environment.

The indexed file object description must match the top expression description, ie. if the top expression is of the form FOO BAR *, the PUT arguments must be FOO1 BAR1 * and not FOO1 * * or FOO1 * PROP.

The effect of the REP option is to replace the file object instead of updating it. If this option is omitted, properties and/or members that do not appear in the top expression are left unchanged.

QUERY —— Lispedit Command

Command name(s): QUERY

QUERY [spec] [(opt ...] -- to display status information about the specified identifiers.

The argument of the QUERY command specifies one or more identifiers. For each id the status information is shown as a list of the form

```
(id xf mem sflag tflag)
```

where sflag is SAVED UNSAVED FBPI MBPI or ???
and tflag is TRAPPED COMPILED DEFINED or OTHER

SPEC	ID or IDS specified
=	the identifier whose value is the current top expression (this is the default)
id	the specified id
*	all the ids that have been previously mentioned in commands that deal with known ids (see HELP KNOWNID)
xf mem	all the ids defined by the specified member of an indexed file
xf *	all the ids defined in an indexed file The option (ONLY mem1 mem2 ...) can be used to specify only the members mem1 mem2 ...

The flag values may also be used as options on the QUERY command in order to limit the scopw of the query. For exampe, the command

```
QUERY ABC * ( UNSAVED TRAPPED
```

will display a list of all the functions in file ABC which are both unsaved and trapped.

QUIT —— *Lispedit Command*

Command name(s): QUIT, Q, QU, QUI

QUIT -- to leave the current Lispedit recursion level and return to whoever called it. QUIT from level 0 returns control to the Lisp supervisor. To return to the CMS environment directly, enter (RET).

QUIT expr -- to quit and return a value to the calling environment. The argument is evaluated in the current environment. The default value is NIL.

The RESUME command will return control to the Lispedit level from which the most recent QUIT was executed.

READ —— *Lispedit Command*

Command name(s): READ

READ [fn [ft [fm]]] [opt] ... -- to read an expressison from a file and make it the new top expression. If no file arguments are given, use the last file mentioned in a READ command.

OPTIONS:	
0	- to re-read the last expression read.
-n	- to read the n-th previous expression in the file.
+n	- to read the n-th following expression.
(IF pred)	- to read the file until an expression that satisfies pred is found. The count option applies to the number of expressions that satisfy pred.

Filetype defaults are taken from EXF-FILETYPES.

Note that each READ is followed by TEREAD, as in EXF. The current READ file is global to the current Lispedit workspace.

RECLAIMX —— *Lispedit Command*

Command name(s): RECLAIMX

RECLAIMX [xf] [(opt ...] -- to remove obsolete values and members from an indexed file. Since most update operations to an indexed file are not destructive, but cumulative, the files associated with it will grow with time. When the backup values are no longer needed, the RECLAIMX command can be used to recover some file space.

The indexed file default is = ; read HELP XOB for details.

The options are one or more of the keywords BUCKETS, HEAD, or LISPLIB. If no options are given, all the components of the file are reclaimed, otherwise only the mentioned components are.

Deleted objects are added to the file: xfn GARBAGE xfmode. You should archive or erase this file to recover the disk space. The option ERASE will erase the file of deleted objects.

REPLACE —— *Lispedit Command*

Command name(s): REPLACE, REP

REP e1 [e2 e3 ...] -- to replace the focus with expression e1.
If expressions e2 e3 ... are given, insert them to the right
and leave the focus on the last inserted expression.
NOTE: multiple arguments are not accepted when the focus is the
top expression, a vector element or an atomic CDR.

REP -- with no arguments to put the focal expression in the
input area for textual modification. If the focus does not fit
in the input area, it may be descended and edited part by part,
or Lispedit may be called recursively to edit the current
focus.

RESOLVE —— *Lispedit Command*

Command name(s): RESOLVE, RES

RES token -- if token is a Lispedit command, the result will be
to place in the input area a command that will retrieve the
definition of that command.

RESTORE —— *Lispedit Command*

Command name(s): RESTORE, REST

RESTORE -- to restore data lost by the most recent delete or
replace operation.

If the focus is currently positioned at the place where the
delete or replace operation ended, the effect of the command is
to undo the operation.

Otherwise, the lost data is stored in the variable ,VAL and a
message is displayed. The data can then be viewed by entering
(EDIT ,VAL) or it can be inserted in the current expression by
entering the commands:

EVAL INS ,VAL   or   EVAL REP ,VAL

RESUME —— *Lispedit Command*

Command name(s): RESUME

RESUME -- this command is equivalent to the command EDIT called
with no arguments, or to the expression (EDIT).

The effect of this command is to resume the most recently active
Lispedit environment.

When a Lispedit environment is given a name with the NAME
command, that name is also defined as a command that will resume
the named environment. If that command is deleted or redefined,
that environment is naturally lost.

If you end up in Lisp as a result of a QUIT from Level 0, evaluating the expression (EDIT) will return control to Lispedit at the point QUIT was called. If you end up in Lisp by a jump to NILSD or an UNWIND from an unusual state, evaluating (EDIT) will return control to the most recently active Lispedit read-loop. Normally, resuming a Lispedit read-loop restores only the Lispedit environment, but not the previous call chain.

As a result of this facility, it should be difficult to lose inadvertantly the contents of an edit session.

RUN —— Lispedit Command

Command name(s): RUN

This command is effective only when the interactive interpreter (Heval) is in control.

The effect of this command is to complete the evaluation of the current expression, display the result, and advance the evaluation pointer to the next expression.

RUN FAST -- will suppress interactive evaluation of any functions called during the evaluation of this step.

SAVELISP —— Lispedit Command

Command name(s): SAVELISP

SAVELISP [fn [ft [fm]]] -- to save the current workspace on disk.

The default values for the arguments are those used with the most recent workspacsave. They can be interrogated and changed with the NAMELISP command. See also FILELISP.

SAVEXOB —— Lispedit Command

Command name(s): SAVEXOB, SAVEX, SAVE

SAVE [spec] [(opt] -- to save the definitions of the specified operators in the appropriate indexed files.

The argument of the SAVE command specifies one or more identifiers. For each id, if there exist an unsaved expression form of the definition, it is saved in the indexed file associated with that identifier.

SPEC	ID or IDS specified
=	the identifier whose value is the current top expression (this is the default)
id	the specified id
*	all the ids that have been previously mentioned in commands that deal with known ids (see HELP KNOWNID)
xf mem	all the ids defined by the specified member of an indexed file
xf *	all the ids defined in an indexed file

The option (ONLY mem1 mem2 ...) can be used to specify only the members mem1 mem2 ...

When the top expression is the value of an identifier, all changes are reflected immediately in the Lisp environment. In this case, the effect of the SAVE command is to update the filed copy.

When the top expression is an xob, all the changes to it exist only in the current edit environment. In this case, the SAVE command updates both the filed copy of the object and the current Lisp definition.

SETXPROP —— Lispedit Command

Command name(s): SETXPROP, SXP

SXP [xf] ([prop val] ... [prop EV expr] ... ---- to set the value of one or more file properties in an indexed file. File properties affect the handling of each member in the file. For example, the command
```
SXP FOO (DISPOSITION COMPILE
```
will cause all members in file FOO to be compiled if DXF or DXM is called subsequently.

The default file argument is = . Read HELP XOB for details.

prop val	Set file property prop to value val
prop EV expr	Set file property prop to the value of expression expr

Some interesting file properties:

DISPOSITION	disposition of members without a DISPOSITION prop.
FILEPROPS	member properties in this list are pushdown props, the value of these properties is not lost when replaced by a PUT. Previous values can be retrieved by using the age parameter.
OPTIONS	the value of this property is added at the head of OPTIONLIST during disposition operations.

Certain file properties are reserved and SXP will complain if you try to change them.

SHOW —— Lispedit Command

Command name(s): SHOW

SHOW [option] ... -- to modify the parameters that control the format of the display.

OPTIONS:

none	to reset the screen format from the global defaults
G ...	to modify the global defaults instead of the current environment. (You will need to call SHOW with no args. to reflect these changes in any currently active envs.)

G to reset the global defaults with builtin values.
(MSG 0) to vary the size of the message area with the number of messages to accumulated since the last display update.
(MSG n) to always show only the most recent n-1 Lispedit messages. If more messages need to be displayed, put the scope in MORE state and type the messages to the screen.
WIDE to display the focus breadth first.
DEEP to display the focus depth first.
(FOCUS n) or n to use at most n lines to display the focus.
(LEFT n) to display at most n expressions to the left of the focus in the list immediately containing the focus.
(OUTER n) to display at most n expressions to the left of any outer list containing the focus.
(UP n) to show at most n expressions above the focus.
(SCREEN n) to use n lines of the screen for all Lispedit output.
(ORG n) to position Lispedit output starting on line n (the to line of the screen is line 0).

(FMARK [start [stop]]) may be used to control the highlighting of the focus. If start and stop are omitted, brightening is used. Otherwise, start (and stop) must be short character vectors used to bracket the focus on the screen when brightening is not desired or possible.

Normal settings: WIDE (FOCUS 10) (LEFT 5) (OUTER 1) (UP 4)
(SCREEN 22) (ORG 0) (MSG 0)

SMALLER —— Lispedit Command

Command name(s): SMALLER, S, SM, SMA, SML

Sml [n] ... -- Shift the focus to the first [or n-th] sub-expression of the current focus. If n is *, shift the focus to the last sub-expression. If n is -1, shift to the last sub-expression; if -2, then to the next to last, etc ...

If arguments n1 n2 ... are given, the focus shifts to the n1-th son of the initial expression, then to the n2-th son of the resulting expression, and so on until the argument list is exhausted or a non-descendable expression is reached.

NOTE that the first element of each descended list or vector is 1 (not 0 as in the ELT function).

SORT —— Lispedit Command

Command name(s): SORT, SO, SOR

SORT [extent] [sortfield] -- to replace a sequence of list elements with the same elements in some order. The extent argument may be n, +n, -n, *, +*, or -*; the default is *. The sortfield argument must be a function of one argument that can be applied to each of the items to be sorted.

EXAMPLE: Assume the focus is the expression (X . 3) in the list

((X . 6) (X . 1) (X . 3) (X . 5) (X . 2) (X . 0))

then the command SORT 3 CDR will result in the list

((X . 6) (X . 1) (X . 2) (X . 3) (X . 5) (X . 0))

STEP —— Lispedit Command

Command name(s): STEP

This command is effective only when the interactive interpreter (Heval) is in control.

The effect of this command is to take one coarse step in the evaluation of an expression. The step size is determined by the current position of the evaluation pointer. A coarse step is one of the following sequences of steps:

- recognize the operator of the current expression and advance the evaluation pointer to the first argument
- evaluate the argument, display the value, and advance the evaluation pointer to the next argument
- when all the arguments are evaluated, apply the operator, display the value, and advance the evaluation pointer to the next expression.

STEP FAST -- to suppress interactive evaluation of trapped functions during this step.

The command STEPF allows smaller steps to be taken during the evaluation of an expression.

STEPF —— Lispedit Command

Command name(s): STEPF

This command is effective only when the interactive interpreter (Heval) is in control.

The effect of this command is to advance evaluation by one step. These steps are smaller than those taken by the STEP command and allow the evaluation of an expression to be examined in very fine detail.

Each fine step of evaluation is one of the following:

- position the evaluation pointer on the operator
- evaluate the operator
- recognize the application of a macro
- evaluate the body of the macro to generate a new expression. (these two steps may be repeated several times)
- position the evaluation pointer on the next argument
- evaluate the next argument
- when all the arguments have been evaluated, prepare to apply the operator of this expression

- produce the value of the application
- advance the evaluation pointer to the next expression.

STEPF FAST -- to suppress the interactive evaluation of trapped functions during this step.

STOP —— *Lispedit Command*

Command name(s): STOP

This command is effective only when the interactive interpreter (Heval) is in control.

The effect is to terminate the evaluation of the initial expression given to the interpreter.

QUIT expr -- will terminate the evaluation of the current function, but will continue the current call chain. The value of expr will be the value of the current function. Note that expr is evaluated in the Heval environment, not in the edit environment.

TAG —— *Lispedit Command*

Command name(s): TAG

TAG xxx -- to tag the current focus with the identifier or integer xxx. The command GO xxx can subsequently be used to return to the current focus from anywhere in the current expression.

This command is also useful in conjunction with the MOVE command.

TEST —— *Lispedit Command*

Command name(s): TEST

TEST expr1 expr2 ... -- to evaluate one or more expressions. The expressions are evaluated for effect only, and the values are not shown in the message area.

The argument expressions are evaluated in order as long as they produce non-NIL values. The first value of NIL terminates the command and signals a command failure. The effect of this signal is to stop any DO or MAP commands currently in effect.
The main purpose of this command is to allow sequences of commands to be repeated as long as some arbitrary expression yields a non-NIL value.

HEVAL NOTE: During interactive evaluation, all evaluations are done in the user program state; this environment does NOT include the current edit environment, therefore references to edit parameters such (CURS) will yield surprising results.

Use the form TEST & to evaluate the current focus.

TO —— Lispedit Command

Command name(s): TO

TO -- to insert at (or after) the current focus, one or more expressions saved by a preceding MOVE or COPY command. Note that TO can be called more than once for each call to MOVE or COPY.

TO After	-- same as above.
TO Before	-- to insert before the current focus.
TO Rep	-- to replace the current focus.

TOP —— Lispedit Command

Command name(s): TOP, T

TOP -- to shift the focus to the top expression. The top expression is the outermost Lisp object for which Lispedit was called.

When the focus is the top expression, certain operations are not allowed. For example, it does not make sense to insert a lisp expression to the right or left of the top expression. The top expression is associated with a source that is identified on the fence line. The sources that Lispedit recognizes are:

- the value of a Lisp identifier. In this case, a REPlace operation on the top expression will also update the value of the identifier.
- the value of a Lisp expression.
- an expression from a sequential file.
- an item from a Lispedit Indexed File (xob).

TRAP —— Lispedit Command

Command name(s): TRAP, TR, TRA

TRAP spec ---- to modify the function(s) specified by spec in such a way that when they are called, the interactive interpreter (Heval) will display their evaluation. See below for a description of spec.

UNTRAP spec -- to turn off trapping for the specified object(s).

During the evaluation and display of a trapped function, the command QUIT expr can be used to continue evaluation with the value of expr used as the value of the aborted call. The command STOP will abort the entire call chain that invoked the trapped function.

--- HOW TO SPECIFY WHAT TO TRAP ---

TRAP function_name	This form is used to trap a DEFINED Lisp function or to trap any LOADed function.

TRAP xf mem	This form is used to trap a compiled function for which the source form is available in the specified indexed file.
TRAP xf *	This form is used to trap all the functions stored in the specified file.
TRAP *	This form is used to trap all LOADed functions.
TRAP =	this form has two meanings:

(1) if you are editing the value of identifier xxx
it means TRAP xxx
(2) if you are editing a filed definition, this
form means TRAP = =

UNCHECK —— *Lispedit Command*

Command name(s): UNCHECK

This command is effective only when the interactive interpreter (Heval) is in control.

CHECK e1 e2 e3 ... -- to add e1 e2 e3 ... to the list of checked expressions. All checked expressions are evaluated and displayed at the end of each evaluation step.

UNCHECK -- to turn off all checking.

UNCHECK e1 e2 e3 ... -- to remove e1 e2 e3 ... from the list of checked expressions.

NOTE: The list of checked expressions is associated with a trapped function as long as trapping remains in effect.

UNTRAP —— *Lispedit Command*

Command name(s): UNTRAP, UNT, UNTR, UNTRA

UNTRAP spec -- to turn off evaluation trapping for the specified function(s). See HELP TRAP for a description of spec.

Trapping is enabled by the TRAP command. A trapped function or macro is executed by the Lispedit interactive interpreter (Heval).

VALUES —— *Lispedit Command*

Command name(s): VALUES, V, VA, VAL, VALU, VALUE

Values expr1 expr2 ... -- to evaluate one or more expressions and display the values in the message area. This command can be used to evaluate identifiers that conflict with Lispedit commands, or simply to evaluate several expressions in one interaction.

HEVAL NOTE: During interactive evaluation, all evaluations are done in the user program state; this environment does NOT include the current edit environment, therefore references to edit parameters such (CURS) will yield surprising results.

Use the form   VALUES &   to evaluate the current focus.

VALUESANDBREAK —— Lispedit Command

Command name(s): VALUESANDBREAK,   VB

This command has the same form and effect as the VALUES command. The only difference is that it allows error breaks whereas VALUES simply returns with a message.

WRAP —— Lispedit Command

Command name(s): WRAP,   W,   WR,   FOF

WR [n] [a b c ...]   -- to wrap one [or n] items into a list with
  a b c ... inserted at the head of the list.

If n is negative items are collected to the left of the focus.
If n is * the focus and all the items to the right in the current list are collected. If n is -* the focus and all the items to the left are collected.

The count n may be omitted if only the focus is to be wrapped and the first item to be inserted is not *, -* or a number.

Example: If FOO is the focus in the expression
    (a b c FOO bar d e)
then the result of WR 2 PLUS 1   will be the expression
    (a b c (PLUS 1 FOO bar) d e)

WRITE —— Lispedit Command

Command name(s): WRITE

WRITE [fn [ft [fm]]] [([width] [NEW or APPEND]]

to prettyprint the current top expression into a file. The file parameters default to those used in the most recent WRITE command if there was one, otherwise fn is required and ft fm default to LISP and A1 respectively.

The width parameter applies only to a new file.

The NEW option erases an existing file.

The APPEND option appends to an existing file. This is the default.

Command name(s): XF, SETXDEF, SXD, SX

XF -- to display information about the current top expression and about the indexed file parameters associated with the current edit environment.

The top expression may be the value of an id, the value of some arbitrary expression, or an object from an indexed file.

When the top expression is the value of a filed property, this command indicates whether this value is saved or unsaved (modified).

XF xf [mem [prop [age]]] -- to set the indexed file default parameters to new values. Unspecified parameters are not changed.

When the top expression is the value of an identifier, the file parameters in the edit environment are also associated with the identifier so that whenever the editor is looking at the value of this identifier, the same file parameters are always in effect.

The indexed file default settings are used whenever a parameter is given as = in a command that operates on a objects in indexed files.

If the current top expression is not an object from an indexed file, any new setting is accepted. Otherwise, the new setting must be similar to current top expression description. You may not change a specific setting to *, or a * setting to a specific value.

DEBUGGING AIDS

25.0 Interactive Evaluation

The Lispedit environment includes a powerful debugging facility, called Heval. Heval allows interactive evaluation of user programs. Heval uses the normal Lispedit display to show the user program, and the brightened focus to indicate the current *evaluation pointer*. The evaluation pointer (*EP*) may be at the next expression to be evaluated, or it may be at the expression that was evaluated most recently. As expressions are evaluated, Heval shows the values in a message area.

It is also possible to monitor the values of arbitrary variables and expressions while evaluation progresses.

The progress of evaluation is controlled by user commands that determine the size of the step Heval will take next. In effect, evaluation progresses in discrete and indivisible units: when Heval is expecting a command from the user, evaluation is suspended; the user enters a command that causes evaluation to proceed to some point; evaluation is again suspended, and Heval allows the next user interaction. While evaluation is suspended, the full Lispedit command set can be used to examine and modify the current program.

25.1 Using Heval to Evaluate a Single Expression

25.1.1 Starting Evaluation

One way to initiate interactive evaluation is to position the Lispedit focus on an expression and to enter the command HEVAL. The effect of the command is to start interactive evaluation of the current focus. Interactive evaluation continues until the value of the selected expression is computed, or until the STOP command is entered.

The focus can be any expression when the HEVAL command is entered. When the focus is a function expression, Lispedit makes a special case: the arguments of the HEVAL command are evaluated and their values are bound to the variables declared in the function expression.

During interactive evaluation, the special commands described below are used to control the progress of evaluation. Heval uses the focus to display the current evaluation pointer. A status message on the screen indicates whether the EP is about to enter or about to leave that expression. In some situations, because of macro transformations, it is not possible to show the EP in the source expression; in that case, Heval shows the smallest source expression that contains the EP.

Normally, Heval forces the focus and EP to coincide. If editing commands are used to shift the focus away from the EP, the status line indicates that the EP is not at the current focus.

25.1.2 Coarse Control of Evaluation

The two most commonly used Heval commands are STEP and RUN. The effect of the STEP command is to advance the EP to the next expression to be evaluated. Any values produced in this step are displayed in the message area. If the EP is about to enter a non-atomic expression, the effect of the STEP command is to advance the EP to the first sub-expression. If the EP is at an atomic expression, the effect of STEP is to display the value of the expression and advance the EP to the next expression. When the EP is at the last sub-expression of a composite expression, the effect of STEP is to display the value of the sub-expression and the value of the containing expression. For example, if the EP is at Z in the following expression

```
        (PLUS 1 (TIMES 2 (MINUS Z)))
```

then a STEP command will advance the EP to the next expression and display the value messages

```
Z = 17
(MINUS Z) = -17
(TIMES 2 (MINUS Z)) = -34
(PLUS 1 (TIMES 2 ...)) = -33
```

Since value messages are always truncated to a single line, some values will not show in full. In this case, Lispedit can be called recursively to show the full value of an expression. Whenever Lispedit displays a message of the form xxx = yyy the variable * is set to the expression on the left of the equal sign, and the variable = is set to the right hand side. Thus, the command E * will display the original expression, and the command E = will display the result in full.

The effect of the RUN command is to complete the evaluation of the expression at the EP without interacting with the user, display the value and advance the EP to the next expression. Thus, if the EP was at (MINUS Z) in the above example, a RUN command would produce the messages

```
(MINUS Z) = -17
(TIMES 2 (MINUS Z)) = -34
(PLUS 1 (TIMES 2 ...)) = -33
```

Note that the STEP and RUN commands are performed with respect to the current EP, not the current focus. When one of these command is entered, the current focus is ignored, evaluation is advanced the required amount, and then the focus is positioned at the current EP. Thus, the focus before STEP or RUN is entered is ignored; after one of these commands, the focus is always identical to the EP. If the focus is not at the current EP, the effect of a STEP or RUN command may be surprising. The NH command may be used to position the focus on the current EP in order to see what the immediate effect of a STEP or RUN command will be.

25.1.3 How to Skip Uninteresting Sections of Evaluation

The FINISH command can be used to complete the evaluation of an expression that currently contains the EP. The effect of the command is to resume evaluation without interactions until the EP is about to leave the current focus. For example, you may have used several STEP commands to view the evaluation of several arguments in a complex expression, but are not interested in the details of evaluation of the remaining arguments. Another common situation is one in which the EP is inside a loop where you are not interested in the remaining iterations. In both cases, you can position the focus on the expression you want to complete, and issue the FINISH command.

Another way to evaluate parts of a program without interactions is with the COME command. The effect of the COME command is to resume uninterrupted evaluation until the EP is about to enter the current focus. For example, by positioning the focus on the first expression in a loop and using repeated COME commands, you can view the state of the program as each iteration is about to begin.

The effect of these command, in contrast to the effect of STEP and RUN, is to evaluate continuously until the EP reaches the current focus.

It is possible to select a focus such that the EP will never reach the selected expression. In this case Heval will pause just before the EP leaves the outermost expression that was given to Heval initially.

Note that the FINISH command does not require that the EP be within the selected focus.

25.1.4 How to Modify the Course of Evaluation

In many cases it is desirable during debugging to repeat the evaluation of some part of the program. For example, you used a RUN command to rush past an expression that yields a surprising value. You can repeat the evaluation of an expression by positioning the focus on that expression and issuing the CONTINUE command. This time you have the opportunity to see the evaluation in detail.

It is important to note that CONTINUE is simply an imperative command that shifts the EP from one place to another in the program. Thus, even if the EP is moved back to a previous location in the program, all the effects of evaluation remain. CONTINUE does not undo any evaluation, it simply continues evaluation from a new place.

The CONTINUE command can also be used with an argument. In this case the argument is an expression that is evaluated instead of the expression at the EP. When the EP is about to enter an expression, this form of CONTINUE will substitute the expression in the command for the expression about to be evaluated. When the EP is about to leave an expression, the value of the expression in the command replaces the value computed by the program.

25.2 Miscellaneous Issues

25.2.1 Immediate Evaluation

In the normal Lispedit environment, any input that is not a valid Lispedit command is treated as an expression to be evaluated. The same effect takes place in the Heval environment, but the input expression is evaluated in the context of the current user program evaluation. Thus to see the value of the variable X, you simply enter X; to see the value of some expression, you simply enter the expression. If you enter the expression (SETQ X 17)and X is a variable in your program, then the value of X will be 17 when evaluation continues.

If the name of a variable looks like a Lispedit command, the EV command must be used. If a displayed value is truncated, it can be seen by editing the value of the variable =.

25.2.2 Interrupting Heval

The named interrupt // (enter precisely the two characters //) interrupts continuous evaluation initiated by a Heval command and returns control to the user at the next opportunity. The EP after an interrupt will be positioned at the next expression to be evaluated.

A similar situation holds if an error break occurs during evaluation and the user response in the break is (UNWIND 1).

25.2.3 Detailed Control of Evaluation

The STEPFINE command advances evaluation in a more detailed way than the STEP command. With STEPFINE you can see the details of macro expansion and argument evaluation.

The CHECK command lets you monitor the values of variables and expressions as evaluation progresses. If the CHECK command is used with arguments, the argument expressions are added to the list of checked expressions. Every time evaluation is advanced one step, the values of all the checked expressions are displayed in the message area. The UNCHECK command turns off checking entirely or for selected expressions.

25.2.4 What Happens When there is an Error in the Source Program

Many errors are detected by Heval and simply announced by an error message. Some errors during evaluation can only be caught by the run-time environment. The latter cause an error break. (UNWIND 1) is normally an appropriate exit from the break-loop. In both cases, the evaluation pointer will be positioned on entry to the expression that contained the error. At this point you can skip or replace the expression with CONTINUE, or change the values of variables, or cancel evaluation with the STOP command.

25.2.5 Some Fine Points

There are several subtle points to be kept in mind when the function that is evaluated interactively is a function that interacts directly with the Lispedit environment. Such a function could be an existing Lispedit command for example.

When Lispedit is stepping through a trapped function, there are two environments of interest to the user. One is the edit environment in which the trapped function is shown; the other is the evaluation environment in which the trapped function was called and in which it is currently evaluating. Normally, the distinction between these two environments does not intrude in a session. But if the trapped function refers to an edit environment, then the distinction is very important.

All references to an edit environment in a trapped function are resolved to the edit environment in which the trapped function was called, not to the environment in which it is being shown.

All references to an edit environment in an expression entered at the console are resolved in the same way. In other words, an input expression is evaluated as if it were a part of the trapped function. There are three exceptions to this rule. The function call (CURS) evaluates to the current visible focus in the trapped function if is entered as part of a command or input expression. The variables * and = are resolved to the current edit environment when entered as part of a command or input expression. If you want to refer to the value of the variable * in the trapped environment, the command CONTINUE * will cause * to be evaluated in the evaluation environment.

25.3 Using Heval to Evaluate Defined Functions

The TRAP command modifies a function definition in such a way that every time the function is called, it is evaluated interactively. Any number of functions may be *trapped* at one time. By this means, you can evaluate interactively those components of an application that require scrutiny while tested components are evaluated normally. The interactive evaluation takes place in the proper application context with non-lambda variables and arguments appropriately set. The UNTRAP command disables trapping either globally or selectively.

Every time a trapped function is called, a new edit environment is created to show the evaluation of that particular invocation of the function. If the function is modified during evaluation, the changes are effective in all invocations of that function. In addition, checked expressions mentioned during one invocation are associated with that specific function and are effective only when control is in the function.

25.3.1 Trapping Compiled Functions

Heval interacts with the Lispedit file system to allow interactive evaluation of compiled functions. Since interactive evaluation can only be performed on the source form of a program, compiled functions require special attention. If the source form of a compiled function can be found, the TRAP command saves the compiled definition and replaces it with the modified source to allow interactive evaluation. The UNTRAP command restores the initial compiled form.

If a trapped compiled function is modified during interactive evaluation, the changes are effective as long as the function remains trapped. Since the UNTRAP command restores the previous compiled form, the changes will be lost. In order to make the changes permanent, you must store the modified source in the file and recompile. If a trapped function is redefined, the function is automatically untrapped and the pre-trap definition is discarded. Any currently evaluating trapped invocations remain in interactive mode until control leaves them.

25.3.2 Trapped Functions and Continuous Evaluation

It is possible to disable the interactive evaluation of trapped functions while evaluating some interactive step. There are two way of achieving this effect.

The FAST option prevents interactive evaluation of trapped functions and also disables Heval interrupts.

The INT option prevents interactive evaluation of trapped functions but allows Heval interrupts.

The COMPLETE command can be used to run the current invocation of a function without further interactions.

26.0 Break-Loop Facilities

When an error occurs in a program it is possible to examine the values of variables. When the error occurs while the interpreter is working, the state of the interpreter is available.

& —— macro

(& **[id] [item1 [item2 ...]]**)

The & operator controls the examination of stack frames created by the evaluation process. It is possible to look at arguments of functions, to determine bindings of variables, and to display variable values.

id, if present, specifies a command to the operator, the remaining arguments may be in arbitrary order.

id The COMMAND, an identifier or list specifying the action, i.e., the kind of stack examination desired. The possibilities are:

INDEX Display a series of stack frame identifications sequentially indexed with a small integer. These appear in a LIFO order (last in evaluation, first in listing). The form of each frame identification is:

1. index, or frame number
2. frame name, or NIL
3. frame type
4. contour level, if not outermost.

FULL Stack frame identification followed by

1. argument names, if any arg
2. all variables, with their been bound at this frame.

(list of identifiers) For every identifier there is an indication of lexical or fluid binding, stack frame identification, and value.

If the identifier refers to a generated symbol, i.e. one in the form %Gn, only the numeric part, i.e. n, should appear in the list.

FLUID Stack frame identification for only those frames at which fluid binding information exists; following each identification are all the variables, and their values, with FLUID bindings at that frame.

LEX Stack frame identification for only those frames at which lexical binding information exists; following each identification are all the variables, and their values, bound lexically at that frame.

UNWIND Display information for each stack frame which has is catch point. Those which are known to the & operator are annotated. See UNWIND, page 88, and Figure 12 on page 88.

BIND Stack frame identification for only those frames at which some binding information exists; following each identification are all the variables bound at that frame.

ARGS Stack frame identification for only those frames representing functions to which arguments have been passed; following each identification are all the argument names and their values.

SECD Stack frame identification for only those frames associated with SECD (interpretive) evaluation and which also have some elements on the SECD control or stack; there follows the elements of the control and stack.

The remaining arguments may be given in arbitrary order. The only caveat is with respect to the START and STOP points, where the relative order of appropriate values (*, numbers) defines their meaning.

chain The search control.

A Indicates that the environment chain is to be examined. If not present, the control chain is examined.

print The print control. The default is an elided display, one line per variable.

PP Indicates that PRETTYPRINTing is to be done. This option is useful when a variable must be examined in detail.

PR Indicates that the normal PRINT function is to be used. This will show it existence of shared sub-structure.

(PR operator) Indicates the the supplied **operator** is to be used for printing.

floor Reference frame specification.

The frame numbers displayed by the & operator and used in the **start** and **stop** operands are relative to a "floor" frame. Normally this is the nearest frame on the control chain which is in use by the operator EVALFUN. This choice prevents the internal workings of the & operator from distracting the user.

It is possible to specify another frame as the "floor" frame.

(FL s-int) use the frame **sint** above the one being used by the display function as the floor. The display function itself is frame 0.

(FL id) where **id** has a bpi as its value. Use the nearest frame (in the chain specified by **chain**) which is in use by the bpi which is the value of **id.**

start Start control.

s-int1 Specifies the first stack frame which is to be examined.

start defaults to 1.

stop Stop control.

s-int2 Specifies the last stack frame which is to examined.

* Specifies that the stack examination should continue through to the highest level.

stop defaults to:

start + 15 when **id** is INDEX and **start** was specified.

10 when **id** is FULL and **start** was not specified.

* when **id** is UNWIND and any other case when **start** was specified.

start in any other case when **start** was specified.

value request for value

Must be used with when a list of identifiers is specified as the command.

VAL Returns the value of the first variable found.

```
(EDIT (& (X) VAL 5 14))
This will cause Lispedit to examine the value
of X in the first frame between the fifth and
fourteenth in which X is defined.
```

VALLIST Returns a list of all values of all variables found.

(&), with no operands is equivalent to (& index 1 14).

We will compile a version of the factorial function which forces an error break when its argument is zero.

```
e fact
i
(LAMBDA (N)
   (COND ( (EQ N 0) (BREAK)) ( 'T (TIMES N (FACT (SUB1 N)))))))
■press enter to leave input mode¬
COMPILE
Macro-expanding FACT
Compiling FACT
Assembling FACT
Linking FACT
```

We now call the new function, using the VB Lispedit command. Typing an expression or an expression preceeded by the V command, usually returns control to Lispedit when an error occurs while an expression is being evaluated. (The V command is useful when evaluating identifiers, which also are Lispedit commands). It is used because one often types in simple mistakes. The VB command causes the Break-Loop to be activated for all errors.

```
VB (FACT 5)
Error: forced break,
Break taken,
 FIN with any value, otherwise UNWIND to exit break loop.
(&)
      ?ARGS?  = NIL
1 %.FBPI.CONDERR
2 %.FBPI.BREAK
3 %.FBPI.FACT
4 %.FBPI.FACT
5 %.FBPI.FACT
6 %.FBPI.FACT
7 %.FBPI.FACT
8 %.FBPI.FACT
9 %.FBPI.SECD2
10 %.FBPI.SECD1 = Contour: 1
11 %.FBPI.LE,USER-EVAL
12 %.FBPI.LE,USER-CATCH = Contour: 1
13 %.FBPI.LE,USER-CATCH
14 %.FBPI.LE,VALUE

NIL
```

Note that (&) displays the value of the variable ?ARGS?, which is sometimes used to pass information about the error (see below).

We now ask for more information about four of the stack frames.

```
(& full 3 5)
3 %.FBPI.FACT
    1  Argument(s)
       N
    1 Lexical binding(s)
       N   0

4 %.FBPI.FACT
    1  Argument(s)
       N
    1 Lexical binding(s)
       N   1

5 %.FBPI.FACT
    1  Argument(s)
       N
    1 Lexical binding(s)
       N   2
```

We request a complete back-trace of the stack.

```
(& index)
1 %.FBPI.CONDERR
2 %.FBPI.BREAK
3 %.FBPI.FACT
4 %.FBPI.FACT
5 %.FBPI.FACT
6 %.FBPI.FACT
7 %.FBPI.FACT
8 %.FBPI.FACT
9 %.FBPI.SECD2
10 %.FBPI.SECD1 = Contour: 1
11 %.FBPI.LE,USER-EVAL
12 %.FBPI.LE,USER~~-CATC~~H = ~~Cont~~our: 1
13 %.FBPI.LE,USER-CATCH
14 %.FBPI.LE,VALUE
15 %.FBPI.LE-VALUES
16 %.FBPI.SECD1 = Contour: 1
17 %.FBPI.LE-EVAL-AND-BREAK
18 %.FBPI.SECD1 = Contour: 1
19 %.FBPI.LE,COMMAND-ARGS
20 %.FBPI.LE,READ-LOOP
21 %.FBPI.%G1466 = Contour: 1
22 %.FBPI.LE,APPLY-LEVEL
23 %.FBPI.LE,EDIT-FUN
24 %.FBPI.LE-EDIT-RECURSIVE
25 %.FBPI.SECD1 = Contour: 1
26 %.FBPI.LE,COMMAND-ARGS
27 %.FBPI.LE,READ-LOOP
28 %.FBPI.%G1466 = Contour: 1
29 %.FBPI.LE,APPLY-LEVEL
30 %.FBPI.LE,EDIT-FUN
31 %.FBPI.SECD2
32 %.FBPI.SECD1 = Contour: 2
33 %.FBPI.SUPV = Contour: 1
34 %.FBPI.SUPV
35 %.FBPI.SUPERMAN = Contour: 2
36 %.FBPI.SUPERMAN = Contour: 1
37 %.FBPI.SUPERMAN
38 %.FBPI.HIGHLORD

NIL
```

We request information about all bindings of N.

```
(& (n)
)
N
   Lexical 3 %.FBPI.FACT
            0
   Lexical 4 %.FBPI.FACT
            1
   Lexical 5 %.FBPI.FACT
            2
   Lexical 6 %.FBPI.FACT
            3
   Lexical 7 %.FBPI.FACT
            4
   Lexical 8 %.FBPI.FACT
            5

NIL
```

We request information about catch-points in the control chain.

```
(& unwind )
2 %.FBPI.S,ERRORLOOP = Contour: 1
      "Lisp UNWIND point, value ignored, continue in read loop."
15 %.FBPI.LE,USER-CATCH = Contour: 1
      "Lispedit UNWIND point, value ignored, continue in read loop."
23 %.FBPI.LE,READ-LOOP
      "Lispedit UNWIND point, value ignored, continue in read loop."
24 %.FBPI.%G1466 = Contour: 1
      "Lispedit QUIT point."
30 %.FBPI.LE,READ-LOOP
      "Lispedit UNWIND point, value ignored, continue in read loop."
31 %.FBPI.%G1466 = Contour: 1
      "Lispedit QUIT point."
36 %.FBPI.SUPV = Contour: 1
      "Lisp UNWIND point, value ignored, continue in read loop."
38 %.FBPI.SUPERMAN = Contour: 2
      "Lisp UNWIND point, value ignored, supervisor re-started."

NIL
```

We leave the break loop, providing a value for the call to BREAK.

```
(fin 1)
Value = 120
```

Now we will force some system detected errors.

```
VB (CAR ())
Error: FR domain,
 FIN with any value, otherwise UNWIND to exit break loop.
(&)
      ?ARGS?  = (NIL CAR)
(LAST ?ARGS?) = CAR
1 %.FBPI.CONDERR
2 SECD
3 %.FBPI.SECD2
4 %.FBPI.SECD1 = Contour: 1
5 %.FBPI.LE,USER-EVAL
6 %.FBPI.LE,USER-CATCH = Contour: 1
7 %.FBPI.LE,USER-CATCH
8 %.FBPI.LE,VALUE
9 %.FBPI.LE-VALUES
10 %.FBPI.SECD1 = Contour: 1
11 %.FBPI.LE-EVAL-AND-BREAK
12 %.FBPI.SECD1 = Contour: 1
13 %.FBPI.LE,COMMAND-ARGS
14 %.FBPI.LE,READ-LOOP

NIL
```

Here, ?ARGS? holds the operator (CAR) and the invalid operand (NIL). Most Lispedit function are named LE,name. Many of the stack frame names are Lispedit functions.

We request information about an interpreter frame. Interpreter frames always have the phrase SECD in them.

```
(& full 2)
2 SECD
    1 Lexical binding(s)
       ?ARGS?   (NIL CAR)
    9 elements in SECD control
       NIL
       META_STMT2
       (ERR1 2)
       META_FORM
       (CONDERR 2 NIL ?ARGS? 'TRUEFN)
       META_FORM
       %.FBPI.CONDERR
       NIL
       META_APP2
    4 elements in SECD stack
       2
       NIL
       (NIL CAR)
       TRUEFN


NIL
```

We will UNWIND to the supervisor and force another error.

```
(UNWIND 1)

(foo x)
■YError: operator value is not a function, the operator was:■UFOO
(FOO X)■Yresulted in error■U(6 (X FOO) ("Error:  FIN with any val..."))
PF: 1 Recall  2 Rept  3 QUIT  4 Big  5 Sml  7 Prev  8 Fwd  10 Sel  11 Back
```

This time we did not use the VB command, so we did not end up in the break loop, instead Lispedit tells us some of the information which is available the break loop. The error was caused because foo has not been assigned a function as a value.

? —— function

(? [**item1**. [**item2** ...]])

The ? operator is the functional equivalent of the & operator. It differs only in having its operands evaluated before it is called, rather than receiving them unevaluated, as & does.

CALLEDBYP —— function

(CALLEDBYP **bpi**)

The CALLEBY? operator searches the control chain for a frame in use by **bpi**. If none is found the value is NIL, otherwise it is the distance, in frames, up the stack to that frame.

This number will differ by a constant from the frame number used by the & and ? operators.

BOUNDEDBYP —— function

(BOUNDEDBYP **bpi**)

The BOUNDEDBYP operator searches the environment chain for a frame in use by **bpi**. If none is found the value is NIL, otherwise it is the distance, in frames, up the stack to that frame.

This number will differ by a constant from the frame number used by the & and ? operators.

27.0 Call Tracing

TRACE —— *macro*

(TRACE **exp id** ...)

exp is evaluated, with all calls to the operators which are the values of **id** ... traced.

```
(TRACE
  (PLUS (TIMES 12 20) (QUOTIENT 13 4.3))
  PLUS
  TIMES
  QUOTIENT)
TIMES Called by EVALFUN
   Args = (12 20)
   Value = 240
TIMES Returned
QUOTIENT Called by EVALFUN
   Args = (13 4.3)
   Value = 3.023255813953
QUOTIENT Returned
PLUS Called by EVALFUN
   Args = (240 3.023255813953)
   Value = 243.023255814
PLUS Returned
Value = 243.023255814
```

```
(TRACE (MEMBER 'C '(A B C D E)) EQUAL)
EQUAL Called by MEMBER
   Args = (A C)
   Value = NIL
EQUAL Returned
EQUAL Called by MEMBER
   Args = (B C)
   Value = NIL
EQUAL Returned
EQUAL Called by MEMBER
   Args = (C C)
   Value = *T*
EQUAL Returned
Value = (C D E)
```

MONITOR —— *function*

(MONITOR **id [list1 [list2]]**)

MONITOR is a simple call tracing operator. The first argument is the **id**, the value of which is the function or macro to be traced. Once MONITOR has been evaluated, all calls to **id** will be intercepted and the values of the arguments, followed by the value of the call to **id**, or the expansion of **id** if it is a macro, will be printed on CUROUTSTREAM. If **id** is MONITORed in compiled code, the **id** of the calling program will also be printed.

The two optional arguments are lists of IDs. If they are present and non-NIL the FLUID bindings of the IDs in the **list1** will be printed before the MONITORed function is actually called, while those in the **list2** will be printed after the function returns.

The effect of MONITOR is removed by (UNEMBED **id**).

```
(COMPILE '(
(STEP-X
  (LAMBDA (N)
    (SETQ X
      (COND ( (NUMP X) (PLUS X N)) ( 'T N)))))
))
Macro-expanding STEP-X
X occurs free
Compiling STEP-X
Assembling STEP-X
Linking STEP-X
Value = (STEP-X)
(MONITOR 'STEP-X '(X) '(X))
Value = STEP-X
(STEP-X 4)
X  = X
STEP-X Called by EVALFUN
   Args = (4)
   Value = 4
STEP-X Returned
X  = 4
Value = 4
(STEP-X 4)
X  = 4
STEP-X Called by EVALFUN
   Args = (4)
   Value = 8
STEP-X Returned
X  = 8
Value = 8
(STEP-X 4)
X  = 8
STEP-X Called by EVALFUN
   Args = (4)
   Value = 12
STEP-X Returned
X  = 12
Value = 12
```

```
(MONITOR 'PROG)
Value = PROG
(PROG (A B)
  (SETQ A 1)
  (SETQ B 2)
  (SETQ A (CONS A B))
  (RETURN A))
PROG Being MACRO-expanded
   Args = (PROG (A B)
            (SETQ A 1)
            (SETQ B 2)
            (SETQ A (CONS A B))
            (RETURN A))
   Expansion = ( (LAMBDA (A B)
                   (SEQ
                     (SETQ A 1)
                     (SETQ B 2)
                     (SETQ A (CONS A B))
                     (RETURN A)) )
                 NIL
                 NIL )
PROG Returned
Value = (1 . 2)
(UNEMBED 'PROG)
Value = PROG
```

UNEMBED —— *function*

(UNEMBED **id**)

Removes the effect of an EMBED of the value of **id**. In particular, cancels the effect of a use of MONITOR on **id**.

EMBED —— *function*

(EMBED **id exp**)

Where the value of **id** is normally an operator, **exp**, and **exp** is a function or macro expression.

If **exp** does not contain instances of **id**, then EMBED is effectively a reversible assignment of **exp** to **id**.

If **exp** contains instances of **id**, then **exp** is modified in such a way as to cause those instances to evaluate to **exp**, the previous value of **id**. All other uses of **id** (exterior to **exp**) will evaluate to this modified expression.

```
(DEFINE
  '((FACT (LAMBDA (N)
      (COND
        ((EQ N 0) 1)
        ('T (TIMES N (FACT (SUB1 N)))))))))
Value = (FACT)
(FACT 6)
Value = 720
(EMBED
  'FACT
  '(LAMBDA (X) (PRINT "Hi there!") (FACT X)))
Value = FACT
(FACT 6)
"Hi there!"
"Hi there!"
"Hi there!"
"Hi there!"
"Hi there!"
"Hi there!"
"Hi there!"
Value = 720
```

EMBEDDED —— function

(EMBEDDED)

Returns a list of all ids which currently have EMBEDed definitions.

28.0 Performance Monitoring and Measurements

BYTES —— function

Addup the storage used by an object.

```
(BYTES item) = (heap-bytes identifiers state-descriptors)
```

This function measures the number of bytes consumed by an object. The counts for identifiers and state descriptors include all occurrences, thus we get (BYTES "(a . a)) = (8 2 0). Shared and looping sub-structures are measured correctly.

COUNT —— macro

(COUNT expr f1 f2 ...) = ((f1 count1) ...)

This function evaluates expr in an environment in which all calls to functions f1 f2 ... are counted. After expr is evaluated, the environment is cleaned up and the value is returned. If one of the given functions is never called, associated count is NIL.

COUNT-CALLS —— macro

(COUNT-CALLS expr) = (count . value)

The argument expr is evaluated in an environment in which most function calls and contour entries are counted. The resulting count is returned in a list together with the value of expr. This is a useful way of measuring the potential efficiency that can be gained by integrating procedures or flattening contours in an application.

COUNT-PAIRS —— macro

```
(COUNT-PAIRS expr) = (t (t1 f1 (f11 t11) (f12 t12) ... )
                        (t2 f2 (f21 t21) (f22 t22) ... )
                        ... )
```

This function monitors all function calls during the evaluation of expr and returns a cross-reference table of call frequencies. The values in the table are as follows:

```
t    total number of calls counted
ti   total number of times function fi was called
tij  total number of times function fij called function fi.
```

LIST-PAIRS —— macro

```
(LIST-PAIRS expr) = message
```

This function monitors all function calls during the evaluation of expr and produces a listing of call frequencies in the file CALL PAIRS A. The format of this file is illustrated in the following example. The line numbers have been added for reference purposes.

```
 1.|
 2.|  (PRINT (CURS))
 3.|
 4.|  Total number of calls: 55
 5.|
 6.|  23 S,PRINTEXPPNAME      ( (,PRIN1 23))
 7.|  23 ,PRIN1              ( (,PRINO 23))
 8.|   2 S,LISPOTOUT         ( (,TERPRI 1) (S,PRINTEXPPNAME 1))
 9.|   1 PRINT               ( (NIL 1))
10.|   1 LE,P-FOCUS          ( (LE-CURS 1))
```

Lines 1-3 show the expression that was monitored. Line 4 lists the total number of calls. The remaining lines show the call frequencies. For example, line 7 states that ,PRIN1 was called 23 times and that the only caller was ,PRINO.

SEESWHAT —— *macro*

(SEESWHAT id1 id2 ...) = ((id1 . what1) ...)

where whati is a list of items of the form (legend item1 item2 ...) SEESWHAT examines bpis and reveals what functions are called, what variables are used free, and what objects are quoted.

SHOW-CALLS —— *macro*

(SHOW-CALLS **exp**)

This function monitors all function calls and contour entries during the evaluation of expr and dumps the information in the file SHOW CALLS A1. Since every function call and contour entry adds a line to the file, it is likely to get very large. The format of the file is illustrated below.

```
1.|  LE,NORMAL  <-- NIL <-- EVALFUN
2.|  LE,SPECTOK  <-- LE,NORMAL  <-- NIL
3.|  PNAME  <-- LE,SPECTOK  <-- LE,NORMAL
4.|  LE,LOCAL  <-- LE,NORMAL  <-- NIL
```

Line 1 indicates that EVALFUN called the interpreter, and that the interpreter then called LE,NORMAL. Line 2 states that LE,NORMAL then called LE,SPECTOK.

TT —— *macro*

(TT **exp**) = (VT= vtime TT= ttime)

This is a coarse measurment tool for expressions that take several milliseconds to evaluate. Vtime is virtual machine time in milliseconds, and ttime is total cpu time in milliseconds. Total time includes paging and other CP overheads.

WHOSEES —— *macro*

(WHOSEES id1 id2 ...) = ((id1 . who1) (id2 . who2) ...)

where whoi is a list of entries of the form (legend f1 f2 ...). Each of the identifiers f1 f2 ... reference idi in the manner described by legend. WHOSEES identifies the following kinds of usage:

- function call from a bpi
- free variable in a bpi
- inside a quoted object of a bpi
- inside the value of a non-lambda binding

NOTE: This function examines only non-lambda bindings, and bypasses current bindings on the stack.

INPUT/OUTPUT

29.0 Internal and External Forms

The Reader is a program that translates a character stream into internal expressions. The Printer is a program that translates expressions into a character stream that can be shown on a display device or stored in a file. An important consideration in LISP/VM has been to define precisely the equivalence between stored expressions and their external representation as sequences of characters.

A simple statement of this equivalence is:

> Let E be an internal expression If we print E to some output medium and then read that representation as E', E and E' will be EQUAL with a few specific exceptions.

These exceptions will be discussed in detail later.

It is possible (and even necessary) for the Printer to output notations that cannot be read back into LISP/VM by the Reader.

The actions of the Reader and Printer are controlled to a large extent by a Readtable. There are several predefined readtables. These can be copied and modified to redefine or extend the syntax.

29.1 Standard Syntax

Expressions consist of identifiers, numbers and constant vectors gathered together into the allowed composite objects. Character vectors are isolated from the remaining text by a special character that denotes the beginning and end of the string of characters; lists and vectors are denoted by various other special characters.

Both identifiers and numbers are extended tokens (ie. they may consist of more than one character) separated from each other by space characters or by special characters that denote structures. An extended token is a number if it satisfies the rules for a number, otherwise it is an identifier. Normally the end of a line signals the end of a token, although the end-of-line condition is not a character in typical input situations.

Each character in an identifier is converted to upper-case unless that character is preceded by an escape character. The first escaped lower-case letter in an identifier stops upper-case conversion in the remainder of the identifier. The text between string delimiters is stored exactly as entered.

29.1.1 Syntax Common to Input and Output

All the macro characters are symbol break characters with the exception of '.', ' | ', and ':'.

(marks the beginning of a list. The Reader collects expressions until the closing parenthesis is found. Parts of a list may be specified in dotted-pair notation (see below).

) marks the end of a list or dotted pair.

. is an infix operator that denotes a dotted pair. It is recognized in this function only if surrounded by delimiter characters. The pair dot must be followed by exactly one expression and a closing parenthesis.

< begins a vector.

> ends a vector.

' is the quoting symbol. The combination 'expr gets the internal representation (QUOTE expr).

| is the symbol escape prefix. The character immediately following the | is treated as an alphabetic character. In a character vector, you must use | to include | or " as characters in the string. In any other token, | forces the token to be an identifier.

" is the string delimiter. In the representation of character vectors, it delimits sequences of characters. In the representation of boolean vectors, it delimits strings of hexadecimal digits that encode boolean values.

xxx/ is a notation that specifies that the identifier or number beginning with xxx is continued on the following line. Any text following the / is ignored.

: is reserved for future use as the package marker in identifiers.

% is a dispatching character. The effect depends on the characters that follow. If the modifying character is a letter, then the upper or lower case form can be used interchangably.

%Gnnn	denotes a gensym. Two occurrences of gensyms with the same nnn in one expression denote the same (EQ) internal gensym. Two expressions read at different times can never share a gensym.
%I< ... >	denotes an integer vector of as many elements as specified.
%nnI< ... >	denotes an integer vector of nn elements. Unspecified elements are set to 0.
%F< ... >	denotes a vector of floating point values.
%nnF< ... >	denotes a vector with room for nn floating point values. Unspecified elements are set to 0.
%nn(...)	denotes a list of nn elements. Unspecified elements are set to NIL.
%nn< ... >	denotes a vector of nn elements. Unspecified elements are set to NIL.
%nn"... "	denotes a character vector with a capacity of nn bytes. Unspecified bytes are unpredictable.
%B"... "	denotes a bit vector. The contents of the bit vector is described by the hexadecimal digits. The current length of the bit vector is determined by the rightmost one-bit in the data.
%Bkk"... "	denotes a bit vector with a capacity of at least kk bits. The actual capacity is adjusted upward to the next word boundary. The current length of the vector is determined by the data.
%Bkk:cc"... "	denotes a bit vector with capacity at least kk bits and a current length of cc bits.
%B:cc"... "	denotes a bit vector with a current length of cc bits. The capacity is determined by the next word boundary.
%Lnn=x	is an input label definition. Labels are used to read expressions with shared and circular structure.
%Lnn	is a label occurrence. Each label occurrence denotes an EQ occurrence of the expression at the label definition. Definitions and uses can be in any order.
*%nn*x*	is equivalent to x repeated nn times in the current list or vector. Note that all the occurrences will be EQ.

*%*x*	is meaningful only within fixed length lists and vectors. The effect is to insert enough occurrences of x to fill the list or vector.
%.xxx	is defined as an error signal, to prevent the input of output-only notations.
%)	causes the current incomplete expression to be discarded. The read operation is re-started at the following character in the input stream.

29.1.2 Syntax Seen Only in Input

%Xxxxx	denotes an integer value in base 16.
% "..."	denotes a comment
% % ...	marks a comment that runs to the end of the current line

29.1.3 Syntax Seen Only in Output

The following constructs appear in output to indicate the occurrence of various internal objects. Since these objects can only be created by an evaluation their occurrence in input data causes an error to be signalled.

%.SD.xxxx	denotes a state descriptor at address xxxx.
%.SUBR.xxx	denotes a compiled function with name xxx.
%.MSUBR.xxx	denotes a compiled macro with name xxx.
%.FUNARG.(bd . sd)	denotes a funarg with body bd and state descriptor sd.
%.EOF	denotes an end-of-file mark. This is the only value that satisfies the PLACEP predicate.
%.READTABLE.xxxx	denotes a readtable at address xxxx.
%.?.xxx	denotes a pointer that has no print representation at all.

29.1.4 Some Differences Between Input and Output Syntax

Since the input syntax allows many objects to be specified in several ways, the printed representation of those objects will not be identical to the input form. The Printer must make a fixed choice in each case where the user has several options.

A more significant difference occurs with gensyms. The reader creates a brand new gensym each time a new input gensym is encountered. As a result, reading the same expression twice will cause two expressions with different gensyms.

30.0 Reading

Reading is the process of translating the external form of an expression into the corresponding internal form. The external form is normally passed to the Reader through a stream object. The stream may fetch data from a console or file record or from internal objects. Whenever an optional stream argument is omitted, the value of CURINSTREAM is used. The syntax of the external form is defined by an object, called a readtable. The Reader assumes that the value of CURREADTABLE is a readtable. The previous section describes the syntax defined by the Standard readtable. A following section describes how the user can build new readtables.

READ —— function and compile-time macro

The value of this function is one expression from the input stream.

`(READ` **[stream]**`) = expr`

The stream is left positioned immediately following the last character of the input expression. If the stream is empty, the value is %.EOF, the end-of-file marker that is recognized by the PLACEP predicate. The stream argument defaults to the value of CURINSTREAM.

Each invocation of READ defines a new context for gensyms and data labels. Within one READ operations, gensyms and data labels with the same external form denote EQ internal objects. READ also establishes a catch-point that decrements and stops UNWINDs. If the user UNWINDs to a READ, the read operation is re-started from the current position in the stream.

READ-CHAR —— function and compile-time macro

The value of this function is one character from the input stream.

`(READ-CHAR` **[stream]**`) = chrid  or` **stream**

This operator extracts one character from the stream **stream** and returns it as the value. At the same time, the stream is advanced so that the extracted character is gone from the stream. If the stream is empty, the value is the stream argument (which can normally be distiguished from an identifier.) This function discards line end indications, so the caller will see only actual data values from **stream.**

The operand, **stream,** is optional, and defaults to CURINSTREAM if omitted.

PUTBACK —— function

Undo the effect of one READ-CHAR call.

`(PUTBACK` **item stream**`)`

The effect of this operator is to return **item** to the stream argument **stream** in such a way that the next use of READ-CHAR will yield **item** as the value.

It is good programming practice to use PUTBACK only to return a single character to a stream, and only the same character that was extracted by READ-CHAR. Multiple PUTBACKs or returning alternate characters may work in some situations but not in others.

TEREAD —— function and compile-time macro

Discard the remainder of an input record in a fast stream.

`(TEREAD` **[stream]**`)`

This operator forces an end of line condition in the fast stream **stream.** Any characters left in the current input buffer are lost. A second application of TEREAD with no intervening NEXTs will have no effect.

If **stream** is omitted it defaults to CURINSTREAM.

> TEREAD does not refresh the buffer in a stream, it simply puts the steam into the end-of-line condition, so that the next application of NEXT will refresh the buffer.

READ-NEXT-LINE —— function

Advance the stream so the next input will be from the next file or console line.

```
(READ-NEXT-LINE [stream]) = NIL or 1
```

The stream defaults to CURINSTREAM. If the next line is empty, the value is NIL otherwise the value is 1.

READ-ONE-LINE —— function

Read an expression from a stream but do not reach beyond the end of the current file or console record.

```
(READ-ONE-LINE [stream]) = expr
```

If stream is omitted, CURINSTREAM is used. If end-of-line is reached before all sequences in the input expression are closed, they are closed automatically. If there is no data on the current line, the function returns %.EOF.

READ-LINE —— function

Fetch one record from a file or the console as a character vector.

(READ-LINE **stream**) = cvec or %.EOF

This operator reads one line from **stream**. This is a single record, from a DASD file, or the contents of the input area, for a console stream.

If **stream** is in the end-of-file condition the value is a read-place-holder, otherwise it is a string.

NEXT —— function and compile-time macro

(NEXT **stream**)

This operator advances **stream** to the next character and returns the updated **stream** as its value; the CAR of **stream** is this next character.

If the last character in the buffer of **stream** has been extracted, it is put into the end-of-line configuration. This is a condition in which the CAR of **stream** is EQ to **stream**. The stream is returned in this configuration.

If **stream** is in the end-of-line configuration, an attempt is made to re-fill the buffer. If the buffer can be refilled the first character is extracted from it, otherwise the stream is put into the empty configuration. This is a condition in which the CAR of **stream** and the CDR of **stream** are both EQ to **stream**. (See "Streams" on page 36).

NEXT does not check **stream** to verify that it is an input stream.

UNWIND —— function

(UNWIND 1)

evaluated in a error break caused by a syntax error will cause all input collected to this point to be discarded and the read operation is re-started from the current position in the stream. This is a convenient error response to an input error in data typed at the console, since the program reading the data is unaware of the error and does not need to check for it.

(UNWIND 2)

evaluated in an error break caused by a syntax error cuses the call to the Reader to be aborted. This is the typical response to a syntax error in input to a read/eval loop.

31.0 Printing

All output functions use the readtable bound to OUTREADTABLE to define the output syntax. When an optional stream argument is omitted, the value of CUROUTSTREAM is used.

PRINT —— function and compile-time macro

This function prints the argument to a stream and advances to the next output line.

```
(PRINT expr [stream]) = expr
```

If stream is omitted, the value of CUROUTSTREAM is used.

PRIN0 —— function and compile-time macro

This function prints the argument to an output stream and leaves the stream positioned at the character following the last output character.

```
(PRIN0 expr [stream]) = expr
```

Identifiers and numbers are always followed by a blank character. If stream is omitted the value of CUROUTSTREAM is used.

PRIN2CVEC —— function

This function creates a character vector that contains the external form of the argument.

```
(PRIN2CVEC expr) = character_vector
```

Note that this function always builds a new character vector. The last character in the vector will always be non-blank unless expr is an identifier that has a pname ending with a blank.

PRINM —— function and compile-time macro

(PRINM **item** **[stream]**)

A specialized print operator which expects **item** to be a pair (otherwise **item** is CONSed with NIL and this pair is treated as **item**) and prints each element of the list **item**, with blanks between the elements. There are no top level parentheses printed, and if any element of the list **item** is a character vector, it is printed by PRINTEXP instead of the normal, PRIN0, routine, so that its delimiting characters are not printed.

If **stream** is omitted it defaults to CUROUTSTREAM.

PRETTYPRINT —— function and compile-time macro

(PRETTYPRINT **item** **[stream]**)

Similar to PRINT, except a more complicated program is invoked which understands many of the more common forms of symbolic expressions and prints them in a structured format. Unlike PRINT, PRETTYPRINT will not try to print structures with cycles in them. If a cycle exists in **item**, the regular PRINT function is invoked. PRETTYPRINT does not attempt to evidence shared structures as does PRINT, but prints each shared substructure in full. If it is not provided, the current value of the variable CUROUTSTREAM is used. PRETTYPRINT uses a free variable, PRETTYWIDTH, to define for it the maximum length of output lines.

If **stream** is omitted it defaults to CUROUTSTREAM.

PRETTYPRIN0 —— function and compile-time macro

(PRETTYPRIN0 **item** **[stream]**)

This is a subfunction of PRETTYPRINT which takes the same arguments but does not perform a TERPRI after **item** has been printed.

The following operator does not use a stream.

CONSOLEPRINT —— function and compile-time macro

(CONSOLEPRINT **cvec**)

This operator displays its operand directly on the console device without going through a stream. It is useful in situations where the overhead involved in the PRINT machinery is unnecessary or undesirable.

PRINTEXP —— function and compile-time macro

(PRINTEXP **cvec [stream]**)

This operator writes the characters comprising **cvec** into **stream**, with no string delimiters, and no escape characters.

If **stream** is omitted it defaults to CUROUTSTREAM.

PRIN1 —— function and compile-time macro

(PRIN1 **item [stream]**)

This operator writes the characters making up the external form of **item** to **stream**. No formatting is performed and no TERPRI is done. No blank is printed after **item**. Thus the next write to **stream** (if no TERPRIs intervene) will place characters immediately following the results of this operation.

This operator will print only non-descendable objects (i.e. neither pairs nor vectors). The value of PRIN1 is **item**.

If **stream** is omitted it defaults to CUROUTSTREAM.

PRIN1B —— function and compile-time macro

(PRIN1B **item [stream]**)

This operand is similar to PRIN1, but a blank character is written into **stream** after **item** is printed.

If **stream** is omitted it defaults to CUROUTSTREAM.

TERPRI —— function and compile-time macro

(TERPRI **[stream]**)

This operator forces output of the current line in STREAM. A second application of TERPRI, with no intervening PRINT-CHARs, will result in a blank line. Remember, that all operators that do output call PRINT-CHAR.

If **stream** is omitted it defaults to CUROUTSTREAM.

WRITE-LINE —— function

(WRITE-LINE **cvec stream**)

Writes **cvec** onto **stream**, as a single line. Only the contents of **cvec** are written, with no string delimiters or escape characters.

Any data written to the stream since the previous TERPRI (explicit or implicit) is lost.

SKIP —— function and compile-time macro

(SKIP **sint [stream]**)

This operator issues **sint** TERPRI's to **stream**. The value of SKIP is NIL.

If **stream** is omitted it defaults to CUROUTSTREAM.

TAB —— function and compile-time macro

(TAB **sint** [**stream**])

This operator causes sufficient blanks to be written into **stream** so that the last character in the stream output buffer is a byte position **sint**. If there are already more than **sint** bytes in the current output buffer, a TERPRI is performed, and **sint** blanks inserted into an empty output buffer.

The effect of (TAB **n stream**) is to cause the next character written to **stream** to be placed in position **n**+1 in its buffer.

If **stream** is omitted it defaults to CUROUTSTREAM.

PRINT-CHAR —— function and compile-time macro

(PRINT-CHAR **character stream**)

This operator places **character** into **stream**

If the buffer is full it is written onto the output device, and **character** becomes the first character of the new buffer. (See "Streams" on page 36).

PRINT-CHAR does not check **stream** to verify that it is an output stream.

PUT-EXPR —— function

This function stores the external form of the argument in a character vector supplied by the caller.

```
(PUT-EXPR expr maxlen charvec) = newlen or NIL
```

This function stores the external form of expr in charvec. The character vector is updated starting at the character position following the current length of the vector. The length of the output vector is the lesser of maxlen or the capacity of the vector.

If the entire external form fits within the maximum length, the value of the function is the current length of the vector. If the external form does not fit in the vector, the value is NIL and the vector is left updated with the partial result. The length is the maximum allowed in this case.

Identifiers and numbers are followed by a blank.

COUNT-EXPR —— function

This function determines the number of characters in the external form of the argument.

```
(COUNT-EXPR expr) = number
```

If the value is positive, it is the actual number of characters in the external form of expr, and the last character that would be printed is an integral part of the output. If the value is negative, the absolute value is the number of characters in the external form, but it does not include the blank that is normally printed following identifiers and numbers.

The following output functions do not check their arguments or the validity of the environment. They may be used with caution in situations where speed is important, since the use of these functions bypasses a call to SHAREDITEMS.

,PRINT-SUBEXPR —— function

Print a sub-expression and leave the stream positioned at the next output character.

```
(,PRINT-SUBEXPR expr shrvec stream) = NIL or expr
```

The value is NIL if expr is an identifier or number that should normally be followed by a space or delimiter.

,PUT-SUBEXPR —— *function*

Insert the external form of a sub-expression into a character vector.

```
  (,PUT-SUBEXPR expr shrvec maxlen charvec) = NIL  or  newlen
```

The value is NIL if the external form did not fit within the capacity of charvec; in this case charvec is filled to the capacity with the truncated external form. The value is negative if expr is an identifier or number that must be followed by a blank or delimiter.

,COUNT-SUBEXPR —— *function* Measure the length of the external form of a sub-expression.

```
  (,COUNT-SUBEXPR expr shrvec) = length
```

The length is negative if expr is an identifier or number that should be followed by a blank or delimiter.

32.0 Readtables

32.1 User-Defined Syntax

The readtables used by the Reader and Printer can be modified by the user to allow a great variety of special syntactic forms.

32.1.1 Readtable and Character Attributes

Each readtable contains a number of flags that define the general behavior of the reader. In addition, there are a number of flags associated with each character in the character set.

32.1.1.1 Readtable Style Attributes

The flags are associated with each readtable. The flags are identified by the following keywords:

NUM-NAME specifies that a symbol that does not qualify as a number becomes an identifier. Thus, the sequence '123ABC' denotes the identifier 123ABC.

NUM-STOP specifies that in a symbol that starts as a number, the first character that fails to be collected as part of the number is a symbol delimiter. Thus, the sequence '123ABC' is equivalent to '123 ABC'.

POINT-REAL specifies that an integer followed by a decimal point is read as a floating-point value.

POINT-INT specifies that an integer followed by a decimal point is read as an integer value.

EOL-BREAK specifies that end-of-line acts like a symbol break. This flag affects both the reading and printing of identifiers and numbers.

EOL-SKIP specifies that end-of-line is invisible when a symbol is gathered.

EOF-ERR specifies that end-of-file before the end of a sequence is an error.

EOF-CLOSE specifies that all open sequences are implicitly closed by end-of-file.

CASE-SHIFT specifies that all (un-escaped) characters in an identifier are converted to upper-case during input.

CASE-KEEP specifies that all identifiers are stored exactly as read.

32.1.1.2 Character Classes

Flags are associated with each character in a readtable. The character class determines whether the character is a symbol break, one of the special characters recognized by the reader, or a user-defined macro character. The flags are identified by the following keywords:

CONSTITUENT

Characters in this class are normal identifier components. This is the default class for most characters.

IGNORE Characters in this class are ignored entirely in the input stream. They do not cause a symbol break and are not included in the text of an identifier. This class is normally empty.

WHITE Characters in this class are ignored if encountered between tokens. They cause a symbol break.

ESCAPE Characters in this class are escape marks used to indicate that the following character is to be treated as a simple constituent character. Normally this class contains only one character.

PACKAGE Characters in this class are used to separate the components of an identifier into a package name and a print-name. Normally, this class contains only one character.

CONTINUE Characters in this class are continuation marks. Continuation marks are effective only when EOL-BREAK is in effect. They are recognized only within the text of identifiers and numbers. The effect of a continuation mark is to skip any remaining characters on the current line and to continue the identifier or number on the following line. Normally this class contains only one character.

MACRO-CONST, DISP-CONST

Characters in this class are user defined macro characters that do not cause a symbol break. As a result, these macro characters are recognized only when they occur after a break character.

MACRO-BREAK, DISP-BREAK

Characters in this class are user-defined macro characters that cause a symbol break and are therefore recognized anywhere in the input stream.

32.1.2 Built-in Character Roles

Read macros and character classes allow a many-to-one mapping of input text into expressions. The Printer must choose one specific character for each of the various output constructs. The following character roles define the characters used by the printer to map expressions into text.

All the input constructs that are not defined by a character role are built into the Printer and cannot be changed unless the code of the Printer is changed.

Some of the character roles are also used to define input constructs that must be associated with a unique character, or that cannot be defined as character macros.

The character role names defined below are used as arguments to the DEFINE-SYNTAX function.

QUOTE-SYM —— *output role*

This character is used to abbreviate expressions of the form (QUOTE expr).

Note that the input behavior of this character must be defined separately as a read macro. The built-in function ,RMAC-QUOTE is used in the pre-defined readtables.

SYM-ESCAPE —— *output role*

The character immediately following this character is treated as a symbol constituent with no further translation or processing. An extended token containing SYM-ESCAPE is automatically an identifier. In character vectors, it allows the string delimiter to be a character in the string.

In input, any character in the ESCAPE character class causes this behavior. This attribute must be defined separately.

STRING-FENCE —— *input and output role*

The character that delimits character and boolean vectors. The input behavior of this character must be defined separately as a read macro. The predefined readtables use the function ,RMAC-STRING.

SYS-DISPATCH —— output role

Identifies the required dispatch character. The input behavior must be defined separately as a dispatch character.

SYM-CONT —— output role

This is the continuation character for identifiers or numbers if EOL-BREAK is in effect.

During input, characters in the CONTINUE class are recognized in this capacity. The character class must be set separately.

PKG-SYM —— output role

(Not yet supported) Identifies the package name part of an identifier.

In input, any character in the PACKAGE character class causes this behavior. The input character class must be defined separately.

OUT-MARK —— output role

Identifies output-only constructs. This character is always used in combination with the SYS-DISPATCH character.

32.1.3 Manipulating Readtables

32.1.3.1 Creating Pre-defined Readtables

STANDARD-READTABLE —— function

This function creates a readtable that defines the Standard Syntax described in the previous section.

```
(STANDARD-READTABLE [kind]) = readtable
```

Kind may be IN, OUT, or INOUT to specify an input-only, output-only, or invertible readtable. An invertible readtable supports PRINT/READ equivalence. The default is IN.

LISP370-READTABLE —— function

This function creates a readtable that defines the syntax used by older (and internal) versions of LISP/VM.

```
(LISP370-READTABLE [kind]) = readtable
```

Kind may be IN, OUT, or INOUT to specify an input-only, output-only, or invertible readtable. An invertible readtable supports PRINT/READ equivalence. The default is IN.

COMMON-READTABLE —— function

This function creates a readtable that defines a subset of the syntax specified by the Common Lisp Definition.

```
(COMMON-READTABLE [kind]) = readtable
```

Kind may be IN, OUT, or INOUT to specify an input-only, output-only, or invertible readtable. An invertible readtable supports PRINT/READ equivalence. The default is IN. The input readtable for true Common Lisp is an input-only table because the Common Lisp convention for vectors is different from the output format of LISP/VM vectors. The Common Lisp invertible readtable reserves < and > as vector delimiters and the notations #I<, #F<, #" and #B" for integer, real, character and bit vectors.

RAW-READTABLE —— function

This function creates a readtable with no macro or dispatch characters.

```
(RAW-READTABLE [kind]) = readtable
```

Kind may be IN, OUT, or INOUT to specify an input-only, output-only, or invertible readtable. An invertible readtable supports PRINT/READ equivalence. The default is IN.

PPRINT-TABLE —— function

This function copies the style flags of the current readtable and sets CASE-KEEP and NUM-NAME.

```
(PPRINT-TABLE [kind]) = readtable
```

If kind is OUT, OUTREADTABLE is copied, otherwise CURREADTABLE is copied.

32.1.3.2 Testing and Copying Readtables

READTABLEP —— built-in function

This function is the type predicate for readtables

```
(READTABLEP x) = x  or  NIL
```

The value of the function is EQ to the argument if the argument is a readtable, otherwise the value is NIL.

COPY-READTABLE —— function

This function makes a complete copy of a readtable.

```
(COPY-READTABLE [readtable]) = new-readtable
```

If the argument is a readtable, this function makes a new copy of the argument and returns it as the value. If the argument is omitted, the value of CURREADTABLE is copied. If the argument is not a readtable, an error is signalled.

COPY-READSTYLE —— function

This function makes a partial copy of a readtable. The new readtable shares all the character-related definitions with the input readtable.

```
(COPY-READSTYLE [readtable]) = new-readtable
```

If the argument is a readtable, this function makes a new copy of the argument and returns it as the value. If the argument is omitted, the value of CURREADTABLE is copied. If the argument is not a readtable, an error is signalled. Only the style attributes of the input readtable are copied. All the components of the readtable that depend on each character code are shared with the input readtable.

One situation in which this function is used is when a readtable is temporarily re-bound to suppress case folding or end-of-line recognition.

32.1.3.3 Modifying Readtables

SET-READTABLE-FLAGS —— function

This functions sets the style flags in a readtable.

```
(SET-READTABLE-FLAGS flagname ... [readtable])
(SET-READTABLE-FLAGS flaglist [readtable])
```

The effect of this function is to set the specified readtable flags. The flag names are described in a previous section; the readtable defaults to CURREADTABLE.

The flags are set in the order they are mentioned in the argument list. Since some of the flags are mutually exclusive, the last one in the list will stick.

GET-READTABLE-FLAGS —— function

This function extracts the settings of the readtable style flags.

```
(GET-READTABLE-FLAGS [readtable]) = flaglist
```

The value of this function call is a list of the names of the flags in effect in the specified readtable.

DEFINE-SYNTAX —— function

This function specifies the character to be used for the named character roles.

```
(DEFINE-SYNTAX rolename charspec [readtable]) = charnum or NIL
```

The effect of this function is to define a character to perform the specified role. Rolenames are defined in the previous section. If the character is specified as NIL, we use the corresponding character in the Standard Readtable. The character may also be specified as an integer, a character vector or an identifier. The readtable argument defaults to the value of CURREADTABLE.

GET-SYNTAX-CHAR —— function

This function extracts the character associated with a specified role in a readtable.

```
(GET-SYNTAX-CHAR role-name [readtable]) = charnum or NIL
```

The value of this function is the EBCDIC character code for the specified readtable character role.

The following functions modify a readtable beyond the standard definitions created by MAKE-READTABLE and DEFINE-SYNTAX. in all cases, if the readtable argument is omitted, the value of CURREADTABLE is used.

The character involved may be specified as an integer, a character vector, or an identifier. In the latter two cases, the first character of the vector or pname is used.

DEFINE-CHAR-CLASS —— function

This function modifies the character class of a character in the specified readtable.

```
(DEFINE-CHAR-CLASS charspec class [readtable])
```

The class argument may be one of the character class names defined above. If the class argument is MACRO-CONST or MACRO-BREAK, the character must already be defined as a read macro character. If the class argument is DISP-CONST or DISP-BREAK, the character must already be defined as a dispatch character.

GET-CHAR-CLASS —— function

This function returieves the name of the character class to which a character belongs.

```
(GET-CHAR-CLASS charspec [readtable]) = class
```

The value is NIL if charspec or readtable are invalid. Otherwise, the value is one of the character class names.

DEFINE-READ-MACRO —— function

This function defines a character as a read macro in the specified readtable.

```
(DEFINE-READ-MACRO charspec fnspec [class] [readtable])
```

Fnspec must be a function of at most two arguments. The function is called with two arguments:

```
(fn stream charnum)
```

both of which may be ignored. If the value is NIL the reader assumes that no changes were made to the current expression.

The class argument may be one of the keywords MACRO-CONST or MACRO-BREAK. If the class argument is omitted, the character class defaults to MACRO-BREAK.

DEFINE-DISPATCH-CHAR —— *function*

This function defines a character as a dispatch character in the specified readtable.

```
(DEFINE-DISPATCH-CHAR charspec [def] [class] [readtable])
```

The dispatch character is followed by an optional integer and then a character defined by the following function. There may be any number of dispatch characters, each with its own dispatch macro table.

A dispatch character is disabled by redefining the character as another macro or redefining the character class.

The class argument may be DISP-CONST or DISP-BREAK to specify the character class of the dispatch character. The default class is DISP-BREAK.

The def argument may be NEW to specify a new dispatch character with a fresh dispatch table, SYS to specify the dispatch table associated with the SYS-DISPATCH character, or a charspec to specify the dispatch table associated with that character. Note that in the last two cases, several characters may share the same dispatch table.

DEFINE-DISPATCH-MACRO —— *function*

This function defines a dispatch macro.

```
(DEFINE-DISPATCH-MACRO dchar char fn [readtable])
```

If dchar is NIL the dispatch table associated with SYS-DISPATCH is updated. If fn is NIL, the dispatch-error function is used.

The function is called with four arguments

```
(fn stream 2ndchar numarg 1stchar)
```

some or all of which may be ignored. If the value of the function is NIL, the reader assumes that the function caused no effect on the current input expression.

32.1.3.4 Non-lambda Variables and Named Readtables

CURREADTABLE —— *variable*

The value of this variable is the default readtable used by all input operations.

OUTREADTABLE —— *variable*

The value of this variable is the default readtable used by all output operations.

NAMED-SYNTAX —— *variable*

The value of this variable is a list that defines a set of named readtables. Each item in the list is a list of the form:

```
(name indef [outdef])
```

where indef and outdef may be readtables or functions of one argument. When outdef is omitted, indef must be an invertible readtable, or a function that produces an invertible table when called with the argument INOUT. When outdef is specified, indef must be an input readtable and outdef an output readtable, or functions that return such values when called with the argument IN or OUT.

Pre-defined syntax names are:

STANDARD	names Standard syntax.
LISP370	names Lisp/370 Syntax.
COMMON	names Common Lisp Syntax.
PPRINT	names a variation of the current syntax in which fewer escape characters are printed. The most common use is for prettyprinting into files that do not have to be read back into Lisp.

CURRENT-SYNTAX —— *variable*

The value of this variable is the name of the most recent syntax installed by READ-INIT.

SUPV-SYNTAX —— *variable*

The value of this variable is used to setup the read-loop syntax used by supervisors and error break loops.

FILE-SYNTAX —— *variable*

The value of this variable is a list that correlates CMS file-types to named readtables. Each entry in the list is of the form:

```
(file-type [syntax-name])
```

If the syntax-name is omitted, it is the same as the file-type.

Known file types:

LISP	Standard syntax
LISP370	Lisp/370 Syntax
PRINT	Modify current syntax with PPRINT-TABLE

READ-INIT —— *function*

This function sets CURREADTABLE, OUTREADTABLE and CURRENT-SYNTAX as specified by the argument.

```
(READ-INIT [syntax-name]) = syntax-name
```

If the argument is omitted, or if the argument is NIL, then the value of SUPV-SYNTAX is used. If the specified syntax has an INOUT table, the input readtable is EQ to the output table.

COPY-IOTABLES —— *function*

This function copies CURREADTABLE and OUTREADTABLE.

```
(COPY-IOTABLES) = syntax-name
```

This function resets CURREADTABLE and OUTREADTABLE to copies of their current values. Once this function is called, changes to the readtables will not be reflected in the named prototypes that may have been used to establish the current syntax.

If the initial values of CUUREADTABLE and OUTREADTABLE are EQ and are suitable for output, then the new values will also be EQ.

GET-READTABLE —— function

This function uses the information in NAMED-SYNTAX to extract or create a readtable.

```
  (GET-READTABLE [name [mode]]) = readtable or NIL
```

If the mode argument is omitted, IN is used. If the name argument is omitted, the value of CURRENT-SYNTAX is used. It is good practice to use COPY-READTABLE on the result.

INPUT-SYNTAX —— function

This function selects an input readtable on the basis of a CMS file-type.

```
  (INPUT-SYNTAX file-type) = readtable  or  NIL
```

This function looks up file-type in the list bound to FILE-SYNTAX and extracts or creates the associated input readtable.

OUTPUT-SYNTAX —— function

This function selects an output readtable on the basis of a CMS file-type.

```
  (OUTPUT-SYNTAX file-type) = readtable  or  NIL
```

This function looks up file-type in the list bound to FILE-SYNTAX and extracts or creates the associated output readtable.

SYM-BREAK-TABLE —— variable

ESC-FIRST-TABLE —— variable

ESC-BODY-TABLE —— variable

These non-lambda variables define the mapping from character class to various read and print properties of the associated characters. They are mentioned here mainly to reserve their names.

32.2 How to Write Input Macros

Input macros are functions called by the reader when certain characters are encountered in the input stream. These functions are specified by calling DEFINE-READ-MACRO, DEFINE-DISPATCH-CHAR and DEFINE-DISPATCH-MACRO.

A Read Macro is triggered by a single character which may or may not be a symbol break. It must be a function of at most two arguments as follows:

```
  (read-macro stream charnum) = NIL or non-NIL
```

The stream is positioned immediately following the character that triggered the call to the read-macro and charnum is normally the numeric EBCDIC code for that character. Since the macro character is an argument to the macro function, the same function may be defined for several characters.

A Dispatch Macro is triggered by a pair of characters. The first character is a Dispatch Character that may or may not be a symbol break. The second character may be preceded by a numeric argument. The dispatch macro function is called with four arguments:

```
  (dispatch-macro stream charnum1 numarg charnum2) = NIL or non-NIL
```

Charnum2 is the numeric EBCDIC code of the dipatch character. Numarg is NIL if the dispatch macro character follows the dispatch character directly; it is a positive small integer if the dispatch macro character is separated from the macro character by a numeric argument. Charnum1 is the numeric EBCDIC code of the macro character. Stream is positioned immediately following the macro character.

Input macro functions are called within a Read Environment that may be updated or examined with tenvironment and constructor functions below.

Every input macro function returns a value that is a signal to the Reader. A value of NIL indicates that no changes were made to the input expression; a non-NIL value indicates that something was inserted.

32.2.1 Input Utilities

The following functions are internal reader functions that should not be used casually by application programs. They are intended to be used by functions that implement read macros.

32.2.1.1 Sub-expressions Utilities

,READ-EXPRESSION —— function

This is the top-level input function; commonly called by the name READ.

```
(,READ-EXPRESSION stream) = expr
```

This function creates a read environment and calls ,READ-SUBEXPRESSION. Two different expressions returned by this function will not share any sub-structure or gensyms, unless this sharing was introduced by read macros that interact with persistent data.

This function also establishes an un-named CATCH point that restarts the read operation if it catches an UNWIND directly to it.

,READ-SUBEXPRESSION —— function

This is the input function normally called by all read macros to gather components that should share a common read environment.

```
(,READ-SUBEXPRESSION stream) = expr
```

The value of this function is an expression that shares gensyms and labelled sub-expressions with the containing expression. The stream is left positioned immediately following the last input character used in building expr.

,READ-SEQUENCE —— function

This function reads expressions up to and including the specified end character.

```
(,READ-SEQUENCE stream endchar stg) = list  or  stg
```

The endchar argument must be a number from 0 to 255. A sequence can also be terminated prematurely if the function ,READ-INSERT-CLOSE is called before endchar is found.

If the stg argument is NIL, the expressions are collected into a list that is returned as the value of the function call.

If the stg argument is a vector, the items in the sequence are stored in successive positions of this vector. In the case of an integer or real vector, only integers or numbers are allowed in the input stream.

This function can also be called to collect items into a pre-consed list. In this case, the stg argument must be passed as a circular list of the desired length. The list is returned with the cycle cut.

,READ-SYM-OR-NUM —— function

This function reads an identifier or a number from the input stream.

```
(,READ-SYM-OR-NUM stream charnum) = identifier, number, or NIL
```

This function scans the characters in the input stream to build a identifier or number or the special symbol NIL. The first character used in this process is the argument charnum; the second and subsequent characters are extracted from the input stream. The scan is stopped by a symbol break character and the stream is left positioned so that the next character to be fetched from the stream will be the break character.

The argument character is always included in the output token and is not checked for the break property. The type of the output token is determined by the syntax definition in the current readtable.

If the charnum argument is the current SYM-ESCAPE character, the output token will be forced to be an identifier.

,READ-CHARS —— function

This function reads characters up to and including the end character.

```
(,READ-CHARS stream endchar stg) = charvec
```

The endchar argument must be a number from 0 to 255. The characters up to endchar are collected into a character vector. If the stg argument is a character vector, it is used, otherwise a new character vector is created.

Escape characters are recognized during the collection and end-of-line conditions are ignored.

,READ-PSMINT —— function

This function reads a positive small integer from the stream, or return the value NIL.

```
(,READ-PSMINT stream) = psmint  or  NIL
```

Digits are collected until a non-digit character is found, or until the maximum number of digits for a guaranteed small integer are collected (in the current implementation, the limit is 7 digits). If EOL-SKIP is in effect, end-of-line conditions are ignored; if EOL-BREAK is in effect, end-of-line stops the number unless a continuation character is present.

32.2.1.2 Character Utilities

,READ-CHAR —— function

This function returns the next character from the stream as a positive small integer.

```
(,READ-CHAR stream) = charnum, -1 or -2
```

If the value is negative, the stream is at end-of-line or at end-of-stream.

,READ-NEXT —— function

This function returns the next character in the stream, ignoring end-of-line conditions.

```
(,READ-NEXT stream) = charnum or -2
```

The value is negative at end-of-stream.

,READ-ONE —— function

This function reads at most one character from the stream if the character can be part of a current symbol.

```
(,READ-ONE stream) = charnum, -1 or -2
```

The value is

-1 if the stream is at EOL and EOL is a symbol break

-2 if the stream is at EOF

charnum otherwise.

(,PEEK-CHAR stream) —— *function*

This function is like ,READ-CHAR but the stream is not advanced, so multiple calls to ,PEEK-CHAR will return the same value.

(,PEEK-NEXT stream) —— *function*

The value is

-1 if the stream is at EOL and EOL is symbol break in the current readtable.

-2 if the stream is at end-of-stream.

charnum if ,READ-NEXT will return that character when called.

(,PEEK-BREAK stream) —— *function*

Like ,PEEK-NEXT but returns NIL if the character is not a symbol break.

(,UNREAD-CHAR char stream) —— *function*

This function resets stream in such a way that all the above character oriented functions will return char as the next character. This function can be called repeatedly.

32.2.1.3 The Input Environment

(,READ-GENSYM psmint) = gensym —— *function*

If this function is called twice with the same argument during one read, the values will be the same gensym.

(,READ-DEF-LABEL psmint expr) —— *function*

Associate psmint with expr in the current read environment. Duplicate definitions cause an error to be signalled.

(,READ-USE-LABEL psmint) = expr —— *function*

Retrieve the definition of a data label in the current read environment. If the label is not defined an error is signalled.

(,READ-ASSQ key) = (key . val) —— *function*

This function retrieves an arbitrary name/value pair from the current read environment. If the key is not found, the value is NIL. Each call to ,READ-EXPRESSION starts with an empty association list.

(,READ-MAKEPROP key val) = (key . val) —— *function*

Update the current read environment to contain the given name/value pair.

32.2.1.4 Constructors

A read macro is called when the defining character combination is encountered in the input stream. The macro can read further data from the stream using the above functions. When the macro has composed a suitable value, it inserts it into the current context using one of the functions described below. Thus, in LISP/VM we do not have unit, splicing, and infix read macros as defined in other Lisp systems. Each macro is free to decide how the associated value is to be placed into a containing expression.

(,READ-INSERT-ONE expr) —— function

Insert one expression into the current read context.

(,READ-INSERT-LIST list) —— function

Insert each item in the list into the current read context.

(,READ-INSERT-CDR expr) —— function

Insert the expression as the final CDR of the current read context. If the insert operation is not possible an error is signalled. Note that any further insertions into that read context will also cause an error.

(,READ-INSERT-CLOSE) —— function

This function is effective only when called within the context of a call to ,READ-SEQUENCE. The effect is to close the sequence without reading the ending character.

,READ-SUBST-LABELS —— function

This function establishes the shared and cyclic structure specified by the data labels in the current expression.

```
(,READ-SUBST-LABELS expr)
```

This function may be useful in building certain kinds of input macros but it must be used with caution. Note that even before this function is called, certain shared and cyclic structures may already exist, because the reader builds this structure on the fly if possible.

32.2.2 Built-in Character Macro Functions

The functions described in this section are used to build the pre-defined readtables defined in LISP/VM. They can be used to build variations of these readtables or called by new readmacros to perform desired actions.

32.2.2.1 Dual Purpose Functions

These dispatch macro functions ignore the third and fourth argument. As a result they can be used as read macros or as dispatch macros.

,RMAC-CLOSE —— read or dispatch macro function

The macro character or the dispatch combination acts as the closing delimiter of the current sequence no matter what the sequence was expecting.

,RMAC-COMMENT —— read or dispatch macro function

Discards the remainder of the current line and return NIL.

32.2.2.2 Read Macro Functions

,RMAC-BACK-QUOTE —— read macro function

This macro function must be used in conjunction with another macro character defined by ,RMAC-UNQUOTE to define the backquote construct of Maclisp and Comon Lisp. The backquoted object shares gensyms with the containing expression, but any shared structure causes an error to be signalled.

,RMAC-CANNOT —— read macro function

Prevents input of output-only constructs such as state descriptors.

,RMAC-COMMON-ID —— read macro function

Defines the Common Lisp |xxx| notation for identifiers.

,RMAC-DOT —— read macro function

Defines dotted pair notation inside a list.

,RMAC-ERROR —— read macro function

Defines a macro character that marks the end of a sequence. Since the ending character is normally detected by ,READ-SEQUENCE its occurrence outside of the scope of a sequence is an error.

,RMAC-LIST —— read macro function

Read a list.

,RMAC-QUOTE —— read macro function

Read a quoted expression.

,RMAC-STRING —— read macro function

Read a simple character vector.

,RMAC-SUPER-CLOSE —— read macro function

Similar to ,RMAC-CLOSE but places the macro character back into the stream. If this function is defined as a read macro, the associated character will thus close all sequences up to and including one that expects to be closed by that particular character.

,RMAC-UNQUOTE —— read macro function

This macro function must be defined in conjunction with ,RMAC-BACK-QUOTE.

,RMAC-VECTOR —— read macro function

Read a simple vector of expressions.

32.2.2.3 Dispatch Macro Functions

,RMAC-BVEC —— dispatch macro function

Defines bit vector syntax.

,RMAC-COMMON-VEC —— dispatch macro function

Defines Common Lisp vector syntax.

,RMAC-DERROR —— dispatch macro function

Default definition of undefined dispatch macro characters.

,RMAC-FVEC —— dispatch macro function

Defines a vector of floating point numbers.

,RMAC-GENSYM —— dispatch macro function

Read a gensym.

,RMAC-HEXNUM —— dispatch macro function

Read an integer in hexadecimal notation.

,RMAC-IVEC —— dispatch macro function

Read an integer vector.

,RMAC-LABEL —— dispatch macro function

Read a data label.

,RMAC-LIST2 —— dispatch macro function

Read a fixed-length list.

,RMAC-REPEAT —— dispatch macro function

Repeat an expression within a sequence.

,RMAC-STRING2 —— dispatch macro function

Read a fixed length character vector.

,RMAC-VECTOR2 —— dispatch macro function

Read a fixed length vector.

32.3 Pre-Defined Readtables

32.3.1 The Raw Readtable

This readtable contains no macro definitions at all and all characters are in the CONSTITUENT class.

The components of the number representation ara defined and are the same in all readtables.

Readtable Flags:

NUM-NAME POINT-REAL EOL-BREAK CASE-SHIFT EOF-ERR

32.3.2 *Standard LISP/VM Readtable*

The standard LISP/VM readtable is created by calling the function STANDARD-READTABLE with argument IN, OUT, or INOUT. The properties of this readtable are described in the beginnin of this chapter.

Readtable Flags:

NUM-NAME POINT-REAL EOL-BREAK CASE-SHIFT EOF-ERR

Character Roles:

'	QUOTE-SYM
|	SYM-ESCAPE
"	STRING-FENCE
%	SYS-DISPATCH
:	PKG-SYM
/	SYM-CONT
.	OUT-MARK

Character Classes:

IGNORE: empty
WHITE: <space>
ESCAPE: |
PACKAGE: :
CONTINUE: /
Dispatch Char: %

Dispatch Macros: (< " % I F R B

Macro Characters:

Constituent: .

Break: (< ' ") >

32.3.3 *Lisp/370 Readtable*

The readtable created by the function LISP370-READTABLE differs from the standard readtable in the following respects:

- The QUOTE-SYM character is the double quote (EBCDIC 127)
- The STRING-FENCE character is the single quote (EBCDIC 125)
- The SYS-DISPATCH, QUOTE-SYM, STRING-FENCE and SYM-ESCAPE characters are not symbol breaks
- A sequence such as 123ABC reads as a number followed by an identifier
- End-of-line is not a symbol break

This readtable can be created with argument IN, OUT, or INOUT.

Readtable Flags:

NUM-STOP POINT-REAL EOL-SKIP CASE-KEEP EOF-ERR

Character Roles:

"	QUOTE-SYM
|	SYM-ESCAPE
'	STRING-FENCE
%	SYS-DISPATCH
:	PKG-SYM
.	OUT-MARK

Character Classes:

IGNORE: empty
WHITE: <space>
ESCAPE: |
PACKAGE: :
CONTINUE: empty
Dispatch Char: %

Dispatch Macros: (< ' % I F R B

Macro Characters:

Constituent: .

Break: (< ' ") >

32.3.4 Common Lisp Readtable

The readtable created by the function COMMON-READTABLE allows input similar to Common Lisp input. The argument IN produces a read-only readtable. The argument INOUT produces a variant syntax in which <...> still represents a vector. The number representation allows D, F, S and L as exponent markers in addition to E. Common Lisp syntax is very similar to Maclisp syntax but not indentical.

Readtable Flags:

NUM-NAME POINT-INT EOL-BREAK CASE-SHIFT EOF-ERR

Character Roles:

'	QUOTE-SYM
<backslash>	SYM-ESCAPE
"	STRING-FENCE
#	SYS-DISPATCH
:	PKG-SYM
.	OUT-MARK

Character Classes:

IGNORE: empty
WHITE: <space>
ESCAPE: <backslash>
PACKAGE: :
Dispatch Char: #

Dispatch Macros: (

Macro Characters:

Constituent: .

Break: (' " ; | <backquote> ,)

33.0 File Considerations

While most input and output can be performed without concern about the operating environment of LISP/VM, the operations that deal with files directly will depend on the operating system. The functions in this section assume a CMS environment where files are identified by two alphanumeric names of one to eight characters each, and a third qualifier that identifies a mini-disk by a single letter possibly modified by a single digit. We call these designators filename, filetype and filemode respectively. In all the functions in this sections, we use the following notation:

```
filespec --> filename                       = (filename *        *)
         --> (filename)                     = (filename *        *)
         --> (filename filetype)            = (filename filetype *)
         --> (filename filetype filemode)

filearg  --> filespec
         --> filename filetype
         --> filename filetype filemode
```

When a function expects a **filearg**, it may be called with one, two or three arguments. If the first argument is a list, the remaining arguments are ignored and the first argument is treated as a **filespec**; otherwise, the first, second and third arguments are treated as filename, filetype and filemode respectively. A NIL is equivalent to * for any file argument and matches the first file that matches any other argument exactly.

$FILEP —— CMS interface function

Convert file arguments to normal form.

`($FILEP` **filearg**`) = NIL   or   (fn ft fm)`

This function converts the several possible forms of **filearg** into a 3-list where each item is a name or *. Note that it does not check whether the file or mini-disk exist. If any component is invalid, the value is NIL.

$INFILEP —— CMS interface function

Determine if a file is available for input.

`($INFILEP` **filearg**`) = NIL   or   (fn ft fm)`

If the specified file exists, the value is a 3-list that contains the full filename, filetype and filemode parameters of the file. Otherwise the value is NIL.

$OUTFILEP —— CMS interface function

Determine if it is possible to write to a file.

`($OUTFILEP` **filearg**`) = NIL   or   (fn ft fm)`

If the specified file can be used for output, the value is a 3-list that contains the full filename, filetype and filemode parameters. Otherwise the value is NIL. Both the filename and the filetype must be specified. Note that this function does not distinguish between a new and an existing file.

$OUTOLDP —— CMS interface function

Determine if it is possible to write to a file.

`($OUTOLDP` **filearg**`) = NIL   or   (fn ft fm)`

If the specified file axists, and it can be used for output, the value is a 3-list that contains the full filename, filetype and filemode parameters. Otherwise the value is NIL. Both the filename and the filetype must be specified.

$FINDFILE —— *CMS interface function*

Find a partially specified file using a list of default filetypes.

```
($FINDFILE filespec list) = NIL  or  (fn ft fm)
```

The second argument is a list of filetypes. If the filetype is not specified in the first argument, the filetypes in the second argument are tried in turn. The first file that is found is returned as the value.

$LISTFILE —— *CMS interface function*

Make a list of filespecs that match a pattern.

```
($LISTFILE filearg) = list-of-filespecs
```

Make a list of **filespec** 3-lists for all the files that match the argument. Do the same kind of matching that the CMS command LISTFILE performs, except that filemode defaults to *

$CHECKMODE —— *CMS interface function*

$CHECKMODE is used to verify a CMS mode letter specification.

```
($CHECKMODE item) = modeletter  or  NIL
```

The argument to this function may be NIL, a character vector, or an identifier. The value is NIL or one of the character atoms *, or A through Z. If the argument is NIL, the value is *; otherwise the value is the first character in the vector or pname if it is suitable.

$MODELETTER —— *CMS interface function*

```
($MODELETTER item) = modeletter  or  NIL
```

Like $CHECKMODE but map NIL and * to A.

$FINDLINK —— *CMS interface function*

```
($FINDLINK item) = modeletter  or  NIL
```

Like $MODELETTER but make sure the specified modeletter denotes a linked mini-disk.

$EST —— *CMS interface function*

$EST test whether a CMS mini-disk exists.

```
($EST item) = 0, 1, 2, or 3
```

The argument is treated as in $MODELETTER and the resulting mini-disk is queried. The value is as follows:

```
0 - Read/Write mini-disk
1 - Read-only mini-disk
2 - Valid modeletter, but not linked
3 - Invalid argument
```

$ERASE —— *CMS interface function*

Erase a CMS file.

```
($ERASE filearg) = sint
```

Erase the specified CMS file. The value is the CMS return code.

$FCOPY —— *CMS interface function*

Make a copy of a CMS file.

`($FCOPY` **filespec1** **filespec2** `[APPEND]) = result`

Make a copy of a CMS file. If the second file already exists, and the APPEND option is specified then the data in the first file is appended to the data in the second file. The result is 0 if the operation completed, otherwise the result is a non-zero numeric return code from the CMS operation that failed.

$FILEDATE —— CMS interface function

Retrieve the date and time when the file was created.

`($FILEDATE` **filearg**`) = NIL or "mm/dd/yy hh:mm"`

If the file does not exist, the value is NIL.

$FILEREC —— CMS interface function

Fetch a record from a CMS file.

`($FILEREC` **filespec** `num) = cvec  or  NIL`

Fetch the specified record from a CMS file. If the num argument points beyond the end of the file, the value is NIL. if **filespec** is a number then the record is fetched from the console device. If 1 is specified, the entire input record is converted to uppercase.

$FILESIZE —— CMS interface function

Determine the size of a file in bytes.

`($FILESIZE` **filearg**`) = NIL  or  num`

If the file is not found, the value is NIL. Note that the size is a conservative estimate in the case of V-format files.

$REPLACE —— CMS interface function

Replace one file with another.

`($REPLACE` **filespec1** **filespec2**`) = result`

At the completion of the operation, the file named by **filespec1** contains the data that was in the file named by **filespec2**. The file named by **filespec2** is gone. The operation is carried out in such a way that a system failure in the middle of the operation will not cause data to be lost.

The result is 0 if the operation was completed, NIL if either argument is invalid, or a non-zero return code from the CMS sub-operation that failed.

34.0 Stream I/O

LISP/VM supports two major styles of I/O, stream I/O to and from disk files and terminals, and key addressed I/O to specially formatted disk files. This section covers the former, while the "Key addressed I/O" on page 283 covers the latter.

At its simplest, a stream can be considered a source of, or recipient of a sequence of characters. In practice, the essentially line (or record) oriented devices show through to a greater or lesser extent. See "Streams" on page 36.

There are six pre-constructed streams. These are:

CURINSTREAM —— *variable*

The currently active input stream. Each invocation of SUPV results in a new binding of this variable. This may be either a console (terminal) stream or a file (disk) stream.

CUROUTSTREAM —— *variable*

The currently active output stream. This may be either a console stream or a file stream. Also bound by SUPV.

ERRORINSTREAM —— *variable*

The input stream used by the break loop. Normally a console stream. Distinguished from CURINSTREAM in case an error is dètected attempting to read from CURINSTREAM.

ERROROUTSTREAM —— *variable*

The output stream used by the break loop. Normally a console stream.

NULLOUTSTREAM —— *variable*

A data sink. This stream simply discards anything written to it.

STACKLIFO —— *variable*

An output stream which places lines in the console stack, so that they will be seen by any program reading from the console.

34.1 Creation

MAKE-INSTREAM —— *function*

Make a stream suitable for input operations.

```
(MAKE-INSTREAM [num]) = console-stream

(MAKE-INSTREAM filespec [recnum]) = file-stream

(MAKE-INSTREAM NIL [cvec ... ]) = internal-stream

(MAKE-INSTREAM NIL internal-outstream) = internal-stream
```

This function creates an input stream for the most common cases. In all cases, the value is NIL if the arguments are invalid or the requested stream cannot be created. Otherwise, the value is an input stream, ie. a stream acceptable to the function READ and its subsidiaries.

If the arguments are omitted or if there is just one numeric argument, the value is a console stream. An argument of 1 forces uppercase folding of all input, all other numeric arguments create a normal console stream.

> Note that the input case folding performed by the stream is much more extensive than that performed by the CASE-SHIFT readtable flag. The readtable flag affects only identifiers and can be overcome temporarily by the escape character. The case folding in the stream affects all input and is unaffected by escape characters.

If the first argument is a filespec, then the value is a stream if the specified file exists. Otherwise the value is NIL. If the second argument is given, it specifies the record number in the file where reading is to begin; the default is to start reading at the first record in the file.

If the first argument is NIL, then an internal stream is created. This kind of stream can be used for reading without reference to an external device. The second and subsequent arguments must be character vectors or lists of cvecs. These cvecs and list are combined into a master list which is then treated as a sequence of file records during subsequent read operations. If the second argument is an internal output stream, then the output from that stream becomes available as input in the input stream.

MAKE-OUTSTREAM —— function

Make a stream suitable for output.

```
(MAKE-OUTSTREAM [num [width]]) = console-stream

(MAKE-OUTSTREAM filespec [width [recnum]]) = file-stream

(MAKE-OUTSTREAM NIL [width [internal-instream]]) = internal-stream
```

The value is NIL in all cases if the arguments are invalid or if the desired stream cannot be created. Otherwise, the value is an output stream, ie. a stream acceptable to the function PRINT and its subsidiaries.

If the first argument is omitted or 0, the value is a console stream ready for output. An argument of 2, causes complete output lines to be stacked FIFO into the CMS program stack. An argument of 3 causes complete output records to be stacked LIFO.

If the first argument is a filespec, then the value is a stream if the file is suitable for output, otherwise the value is NIL. The width argument specifies the maximum record length for a new file. The recnum argument specifies where in the file output will begin. If recnum is omitted, NIL or 0, new output is appended to the file, otherwise, the specified record and subsequent records are overlaid by any output to the stream.

If the first argument is NIL, an internal output stream is created. Output to an internal stream is collected into a list of cvecs of length at most width. If an internal-instream argument is specified, items PRINTed to the outstream can be READ from the instream..

DEFIOSTREAM —— function

(DEFIOSTREAM **list sint1 sint2**)

DEFIOSTREAM builds an object that is used by Reader and Printer functions to keep track of the buffers and other data associated with input/output operations. The arguments are not checked for validity by DEFIOSTREAM. The first use of the stream will typically involve initialization and verification of the data in the structure according to the needs of the using function. Thus, DEFIOSTREAM will create an input stream for a non-existent file, and no error will be indicated until an attempt is made to read from the resulting stream.

list is an association list which defines some of the characteristics of the stream being created. **sint1** is an integer defining the length of the lines for which buffer space is to be provided. This represents a maximum length; shorter lines may be produced by using TERPRI for output, and shorter lines may be emitted by whatever source an input stream uses. A line buffer is not allocated by DEFIOSTREAM, but will be allocated the first time the stream is used by READ or PRINT. **sint2** designates a particular record within a data set. A value of zero means use the first record if an input stream, or the next record if an output stream.

The optional argument **sint3** specifies the size of the descriptor vector contained in the stream. The normal and default size is 5. If an argument smaller than 5 is specified, unrecoverable system errors may occur at a later time. If an argument larger than five is specified, the descriptor vector positions beyond the first 5 are available to the user.

Streams may be defined by this function based either on a disk file or console as an input/output device. The **list** value is examined to determine which of these devices is to be used, and an appropriate program is selected and stored as the value of rfn in the stream structure.

In order to implement commonly-needed default stream attributes, DEFIOSTREAM makes the following modifications to **list**:

- If no MODE property is already part of **list**, (MODE . INPUT) is added to **list** by nondestructive CONSing.
- If (MODE . I) or (MODE . O) is specified, I or O is RPLACDed by INPUT or OUTPUT, respectively.

Following is a description of the properties and meanings which are most commonly used for **list**. Additional properties are ignored by the standard stream processing functions, but may be added to provide additional information to be used by the user's programs.

- (DEVICE . CONSOLE) is specified to indicate this stream is defined on the user's console.
- (FILE filename filetype) or (FILE filename filetype filemode) is specified to indicate this stream is defined on the designated disk file. Values for filename and filetype may be specified either as identifiers or character vectors.
- (MODE . INPUT) or (MODE . OUTPUT) designates an input or output stream, respectively. (This attribute was discussed above.)
- (RECFM . V) or (RECFM . F) designates whether a disk file referenced in an output stream is to have fixed or varying length records in it. Varying length records is the default assumption.
- (QUAL . {S | T | U | V | X}) designates the type of CMS read operation to be used in obtaining records from the console for this stream. The letters have the following meanings:

  S = pad records with blanks to 120 characters.
  T = read a logical line (the default operation).
  U = pad with blanks and translate to upper case.
  V = translate to upper case.
  X = read a physical line.
- (QUAL . {LIFO | FIFO | NOEDIT}) specifies for console output files that no editing is to be performed on output lines (i.e. for typewriter consoles, trailing blanks are not deleted and a carriage return is not automatically appended to the line). LIFO and FIFO designate that output lines are to be placed into the console input stack, rather than be written to the console.

SHUT —— function

(SHUT **stream**)

This operator invokes a system-dependent routine to close any file related to the argument **stream**. If no file is actually in need of closing, the action of SHUT is effectively a no-operation. The value of SHUT is -1 if **stream** is ill formed, 0 if the stream is successfully shut or un-shutable (a console stream, for example), and the return code from the system dependent shut routine otherwise.

Note that, for an output stream, if a partial buffer of data exists, an explicit TERPRI must be done before SHUT is applied, otherwise the data will be lost.

34.2 Accessing Components

All streams are pairs. A simple list may be treated as a stream, however the majority of streams have a complex structure, and are referred to as *fast streams*.

Fast streams are not a distinct data type, they are an interpretation of a particular structure of pairs and vectors. See "Streams" on page 36. Operators are provided for accessing and updating the components of streams. These should be used in preference to C...R and ELT, as they will give protection against any change in the format of streams.

It is possible for a stream to become empty. This occurs, for example, if a DASD file reaches end-of-file. The normal indication for this condition is that READ return the value %.EOF and the stream is updated into the end-of-file configuration.

Since the end-of-file configuration of a stream discards some valuable data in a stream it may be worthwhile in some situations to preserve this information. This is done by saving the value of the function STREAM-DESCRIPTOR.

For example, if an internal input stream is linked to an internal output stream, and READ attempts to read beyond the available data, the link between the two streams will be lost. It can be restored by saving the stream descriptor of the input stream and restoring it whenever it is lost.

CAR —— built-in function

(CAR **stream**)

The value of CAR, when applied to a stream, is the current character. That is, for an output stream, the last character written into the stream, for an input stream, the last character extracted from its buffer.

CAR of a stream which is at end-of-line is EQ to the stream itself.

STREAM-A-LIST —— function

(STREAM-A-LIST **stream**)

Returns the a-list of **stream**. This corresponds to the (possibly augmented) first operand of the DEFIOSTREAM invocation which originally created **stream**. See page 278.

STREAM-BUFFER —— function

(STREAM-BUFFER **stream**)

Returns the I/O buffer of **stream**. This is normally a character vector, containing the last line read (for input streams), of the next line to be written (for output streams).

The TERPRI operator empties the buffer, using SETSIZE to transform it into an empty string. A stream which has not been activated will have a buffer of NIL. Once a PRINT-CHAR or NEXT has been performed on the stream, the buffer will be created, with the proper length.

STREAM-DESCRIPTOR —— function

(STREAM-DESCRIPTOR **stream**)

Returns the descriptor from **stream**. This is a vector (see "Streams" on page 36) containing various sub-components of the stream.

STREAM-P-LIST —— function

(STREAM-P-LIST **stream**)

Returns the system dependent control block from **stream**. This p-list is an object (a string or integer vector) which is created, updated and used by the system dependent routines. It contains arbitrary data, not amenable to manipulation by operators. In an unactivated stream, the p-list will be NIL.

34.3 Updating Components

SET-STREAM-A-LIST —— function

(SET-STREAM-A-LIST **stream list**)

Replaces the current a-list of **stream** by **list**.

SET-STREAM-BUFFER —— function

(SET-STREAM-BUFFER **stream item**)

Replaces the current buffer in **stream** by **item**.

SET-STREAM-P-LIST —— function

(SET-STREAM-P-LIST **stream item**)

Replaces the current p-list of **stream** by **item**.

35.0 Key addressed I/O

This section describes the operators used to manipulate specialized DASD files referred to as *libraries* or LISPLIBs. These files are similar to OS partitioned data sets. They contain *members*, which are representations of data objects, in a form which can be loaded quickly (though writing them is often slow).

Each member in a library is addressed by means of a key, a character vector, and has a class number (between 0 and 255) associated with it.

When a library is opened (by RDEFIOSTREAM), its directory is read and made part of the stream. If the library is opened for output, the file is so marked, and may not be opened for input until it has been shut. Output operations *only* modify the stream's directory, not that in the file, and the data is appended to the end of the file, not overwritten upon old data. Only when the stream is closed is the directory in the file updated.

Some of the following operators act directly on named libraries, while others act indirectly, through streams.

35.1 Creation

RDEFIOSTREAM —— function

(RDEFIOSTREAM **list**)

This operator creates a stream which can be used to access or update a library.

list has the same general form and meaning as the corresponding operand of DEFIOSTREAM, page 278, however the only meaningful properties are MODE and FILE.

35.2 Input

RREAD —— function

(RREAD **cvec rstrm**)

This function reads the value component of a member, identified by **cvec** from the library specified by **rstrm**. **cvec** must be a character vector. **rstrm** must have have been defined by RDEFIOSTREAM. If output has been performed to the library after the creation of **rstrm**, its effects will not be seen by RREAD. Such changes are only evident in streams which were created after the writing process has shut the library. This is so because the directory in the file is only updated upon shutting.

The value of RREAD is the object read.

An error break is taken if **cvec** is not a valid key for the library.

RCLASS —— function

(RCLASS **cvec rstrm**)

This operator returns the class value for the item designated by **cvec** in the library accessed by **rstrm** The value is a small integer, in the range 0 to 255.

An error break is taken if **cvec** is not a valid key for the library.

RKEYIDS —— function

(RKEYIDS **filearg**)

This operator returns a list of identifiers corresponding to the keys in the library named by **filearg**. These identifiers are produced by INTERNing the actual keys, that is the identifier ABC represents the key "ABC".

35.3 Output

RWRITE —— function

(RWRITE **cvec item rstrm**)

This operator add, or replaces, an object representing **item** associated with the key **cvec** to the library accessed by **rstrm**

If there already exists such an object the class is unchanged, if this is a new key, the class is set to zero.

Neither state descriptors nor bpis can be written directly to a library. State descriptors have no representation in libraries. Bpis are represented, but the representation must be created directly by the LAP assembler. This is accomplished by appropriate use of the FILE option in the definition option list.

RSETCLASS —— function

(RSETCLASS **cvec sint rstrm**)

This operator sets the class component for a member of a library. **sint** must be in the range 0 to 255.

The operation is only valid if the key already has been added to the library, by an RWRITE, and an error break is taken otherwise. The effect is only visible after the library is shut.

RSHUT —— function

(RSHUT **rstrm**)

For input streams the RSHUT operator simply performs a CMS FINIS.

For output streams the RSHUT operator writes the current library directory into the file, removes the "open for output" flag and performs a CMS FINIS.

A library which has been opened for output *must* be shut before it can be read again. Failure to do so will result in an unreadable file.

35.4 Library Management

RPACKFILE —— function

(RPACKFILE **filearg**)

This operator reformats a library, discarding all inaccessible data.

RWRITE, it should be remembered, never overwrites old data representations, but rather appends new data to the end of the existing library. RPACKFILE removes the superseded data, shrinking the library to its minimal size.

If any errors occur during the processing, the original file is left unharmed.

RCOPYITEMS —— function

(RCOPYITEMS **filearg1 filearg2 list**)

This operator copies selected members from **filearg1** to **filearg2**.

list is expected to be composed of identifiers, the pnames of which are used as keys to select members from **filearg1**. These members are written into **filearg2**, replacing existing members with the same keys. Any element of **list** which does not correspond to an existing member of **filearg1** is ignored.

If errors occur during the processing **filearg2** is unchanged.

RDROPITEMS —— function

(RDROPITEMS **filearg list**)

This operator removes selected members from the library named by **filearg**.

list is interpreted in the same manner as in RCOPYITEMS, and designates the members to be dropped from the library.

Only the directory is changed, to recover the DASD space occupied by the data the operator RPACKFILE must be used.

35.5 Libraries as TXTLIBs

This set of operators is used to load collections of operator definitions from librariess. They also contain facilities for "load time" evaluation of arbitrary expressions, allowing such actions as the setting of property values.

All of these operators are controlled by the class value of the members of a library. At the moment only the values 0 (zero) and 1 (one) are used.

A member with class 0 will be assigned as the value of the identifier corresponding to the key of that member.

A member with class 1 will be passed to the interpreter for evaluation.

LOADVOL —— function

(LOADVOL **filearg**)

This operator reads all the members of the library named by **filearg**, and based on their class, assigns or evaluates them.

The value of the operator is a list of the identifiers corresponding to the keys in the library. This is equivalent to the value of RKEYIDS when applied to the same library.

SUBLOAD —— function

(SUBLOAD **filespec** {**id** | **list**})

This operator attempts to read the members with keys corresponding to **id** or the elements of **list** from the library named by **filespec**. Each such existing member is treated as in LOADVOL.

The value is a two element list, (l1 l2), where l1 is a list of all elements of **list** (or **id**) which were found in **filespec**, and l2 is a list of all those not so found.

LOADCOND —— function

(LOADCOND **filearg**)

This operator operates like LOADVOL with two differences. First, only members with class 0 are loaded, no load-time evaluations are done. Second, the identifiers corresponding to the keys in the library are evaluated and any which currently have operators as values, (function expressions or bpis), are ignored. All others are loaded and assigned.

This allows various libraries to be constructed for different applications, with overlapping sets of operators. Suppose two such libraries exist, for example a cross reference analyzer for symbolic code and for compiled code. There could be certain operators needed by both packages. These operators can be replicated in the two libraries. Then, by using LOADCOND, both packages can be loaded without loading multiple copies of the common operators.

The restriction to class 0 members is required, as no simple test can be made to determine if a class 1 member has already been evaluated.

The value is a list of the identifiers corresponding to the members loaded.

DEPLOAD —— *function*

(DEPLOAD {**filespec** | **list1**} {**id** | **list2**})

The first operand of DEPLOAD is either a library name or a list of library names. The second is either an identifier or a list of identifiers.

All of the libraries named in the first operand are opened and searched for members corresponding to the identifiers in the second argument. All those found are loaded.

Then, every such loaded member is examined. If it is a bpi, the list of identifiers which it uses as operators is extracted from it. These are evaluated and, if their value is not an operator (function expression or bpi), the libraries are searched for a member by that name. If one is found, it is loaded and its operator usage is similarly checked.

The value is a list, (l1 l2), where l1 is a list of identifiers corresponding to the members loaded and l2 is a list of operator names which did not have operator values and which could not be found in the set of libraries.

36.0 Miscellaneous CMS Interactions

SAVELISP —— *CMS interface function*

Save the current workspace as a CMS file.

```
(SAVELISP filearg)
```

This function saves the status of the entire LISP/VM environment as a CMS file. This saved workspace can be restarted at a later time to continue as if control had just returned from the SAVELISP expression. If any arguments are omitted, they are filled in from the most recent previous use of SAVELISP. The initial defaults are normally LISP SHLISPWS If an argument is specified as =, the default value is also used in that position.

The value is a cvec showing the full file parameters when control returns to the caller following a filing invocation. The value is a number when control returns to the caller after a restart of the workspace.

FILELISP —— *CMS interface function*

Save the current workspace as a CMS file AND RETURN TO cms.

```
(FILELISP filearg) = RESULT
```

This function saves the status of the entire LISP/VM environment as a CMS file and returns to CMS.

This saved workspace can be restarted at a later time to continue as if control had just returned from the FILELISP expression. If any arguments are omitted, they are filled in from the most recent previous use of SAVELISP. The initial defaults are normally LISP SHLISPWS If an argument is specified as =, the default value is also used in that position.

The value is always a number since control never returns to the caller on the first use.

SAVELISP-FILE —— *variable*

File parameters of the most recent use of SAVELISP or FILELISP.

EXITLISP —— *CMS interface function*

Discard the current workspace and return to CMS.

```
(EXITLISP &s-int.)
```

Return to the CMS environment. If an argument is specified, it must be a small integer. This integer, times 4, will be the CMS return code from Lisp.

OBEY —— *CMS interface function*

(OBEY **cvec**)

The argument is passed to CMS to be evaluated as a CMS command. The behavior of this function may vary from one installation to another. The value is the numeric code returned by the command. Zero normally denotes successful completion.

PARAMETERS —— *CMS interface function*

(PARAMETERS)

The value of this function is always a cvec containing any parameters passed to LISP/VM from the CMS environment. If LISP/VM was invoked with the LISP exec, the cvec returned contains all the text that follows the PARAM option. If LISP/VM was restarted with the LISPSTRT exec, the cvec contains all the text that follows the PROFILE option.

$TIMESTAMP —— CMS interface function

($TIMESTAMP)

The value is the current date and time as a cvec of the form "mm/dd/yy hh:mm".

$READFLAG —— CMS interface function

($READFLAG)

The value of this function is non-NIL if the next attempt to read from the terminal will return an input line that has already been supplied by the user, or by another program. If the value is NIL, the next attempt to read from the terminal will suspend evaluation until the user enters some data.

$CLEAR —— CMS interface function

($CLEAR)

Clear the screen if the console is a 32xx. This function is useful in applications that do not use the Lispedit full-screen interface.

$SCREENSIZE —— CMS interface function

($SCREENSIZE)

If the current console is a 32xx display terminal, the value is a list (rows cols) or a list (rows cols REMOTE). The second form is returned if the terminal is connected remotely or via PVM.

$T-TIME —— CMS interface function

($T-TIME)

Total cpu time in milliseconds since last logon.

$T-DELTA —— CMS interface function

($T-DELTA **num**)

The value of this function is the milliseconds of total cpu time since the time specified by **num**

$V-TIME —— CMS interface function

($V-TIME)

Value is virtual cpu time in milliseconds since last logon.

$V-DELTA —— CMS interface function

($V-DELTA **num**)

Value is virtual cpu time in milliseconds since **num**.

TIME2NUM —— function

(TIME2NUM **cvec**)

Where **cvec** is a time/date, as returned from CURRENTTIME, e.g. Encode time-date string as small integer

SYSTEM CAPABILITIES

37.0 CALLBELOW, The Operating System Interface

All of LISP/VM's interactions with the underlaying operating system are performed by the CALLBELOW operator. CALLBELOW, in turn, uses an independently assembled module, LISPCMS, which is loaded by the LISP/VM loader and which communicates with VM/SP.

CALLBELOW —— function and compile-time macro

(CALLBELOW {**cvec** | **sint**} [**sysdep-area** ...])

The CALLBELOW operator reformats its operands to match the requirements of the system dependent module and calls it.

The operands must be either small integers, character strings or integer vectors. The first operand is restricted to character strings or small integers. The value returned is a small integer, a return code from the module. Any additional results are passed by updating strings or vectors.

The first operand is a command. Normally this is a string, such as "PANICMSG", or "FILEIN". However, each command has a corresponding number, which can be determined by use of the "?" command. The use of a numeric command will avoid a search during command interpretation.

For some of the commands, a data area (string or vector) must be supplied in which the module can construct control blocks (PLISTs in VM/SP). Since it is anticipated that the supporting operating system may change from time to time there is no fixed size for these data areas. If an area passed to the module proves to be too small the module (and hence the CALLBELOW operator) will return a negative value (return code), N, such that -N is the size of the required data area. E.g., see GFIPLIST, page 295.

The following section describes all the commands currently accepted by the system dependent module. Certain of these are capable of irreparable damage to the system, and should only be used by system programmers.

In this section, the term "operand" will be used to refer to the items following the command. Thus the first operand of the command will be the second operand of the CALLBELOW.

37.1 Inquiry

? —— command to system-dependent interface

(CALLBELOW "?" **cvec**)
(CALLBELOW "?" **sint** [**cvec**])

In the first form, with the first operand a character string, the string is checked for validity as a command. The return code is the numeric value which is equivalent to **cvec**, as a command. If **cvec** does not correspond to any valid command the value is -1.

In the second form, with the first operand a small integer, the number is checked for validity as a command. The return code is:

0: Argument is valid.
-1: Argument is within valid range, but command not
implemented.
-2: Argument is outside valid range.
(The -1 value could result if a command were deleted from the module. For example, the RELPAGES command might be meaningless in certain operating systems.)

When the first operand is valid the second operand, if present and of sufficient size, will be updated with the name of the corresponding command.

```
(CALLBELOW "?" "PANICMSG")
Value = 25
(SETQ FOO 'XXXXXXXX')
Value = "XXXXXXXX"
(CALLBELOW "?" 20 FOO)
Value = 0
FOO
Value = "UNSHARE"
```

37.2 Console I/O

TOULL —— *command to system-dependent interface*

(CALLBELOW "**TOULL**")

The TOULL command returns the line length for the currently attached output console.

WARNING!

In the VM/SP system, this is the line length returned by the diagnose 24. It is **not** the screen width, but is the value set by the CP TERMINAL LINESIZE command. Anyone requiring the actual characteristics of the console should use the 32XXPLST command, if they can decode the value.

If the diagnose 24 fails, (i.e. if there is no console), a value of 120 is returned.

TINLL —— *command to system-dependent interface*

(CALLBELOW "**TINLL**")

The TINLL command nominally returns the console input line length. In VM/SP it defaults to 255.

NCONSTKD —— *command to system-dependent interface*

(CALLBELOW "**NCONSTKD**")

The NCONSTKD command tests for the presence of completed console reads. If the value is not zero one or more lines have been read from the console and stacked.

NSTACKED —— *command to system-dependent interface*

(CALLBELOW "**NSTACKED**")

The NSTACKED command returns the number of lines in the console stack. These are lines resulting from other programs, e.g., STACK commands in an editor, rather than lines entered from the keyboard.

PANICMSG —— *command to system-dependent interface*

(CALLBELOW "**PANICMSG**" **cvec**)

The PANICMSG command displays text, the contents of **cvec**, on the console device. No streams are used or effected by the operation.

This command is designed for use in various desperate situations, where there is no certainty that the stream facility is operable. The value of the CALLBELOW is always zero.

PANICRD —— *command to system-dependent interface*

(CALLBELOW "**PANICRD**" **cvec**)

The PANICRD command reads a line from the console device into **cvec**. The number of characters read is limited by the length of **cvec**.

As in PANICMSG, this operation is independent of any stream I/O, as is meant for use in crisis situations.

GCNPLIST —— *command to system-dependent interface*

(CALLBELOW **"GCNPLIST" cvec** {"INPUT" | "OUTPUT"} **sysdep-area**)

This command constructs a control block for I/O with the virtual console. The control block is built in **sysdep-area**. The first argument, **cvec**, is interpreted differently for INPUT and OUTPUT.

In an input control block **cvec** must be a one byte string which specifies the treatment of the data from the console. The contents must be one of S, T, U, V, W, X, Z or *.

code	case	padding	log/phys	length
S	mixed	blanks	logical	130
T	upper	X'00's	logical	130
U	upper	blanks	logical	130
V	mixed	X'00's	logical	130
W	mixed	none	logical	as requested
X	mixed	X'00's	physical	130
Z	upper	none	logical	as requested
*	mixed	none	physical	as requested

(Note: 'physical' means that, if the line was read from the console (as opposed to from the stack), it is not broken up into logical lines by the character X'15'. Lines read from the stack are not inspected, hence X'15' is treated like any other character.)

For output control blocks, **cvec** can have values of NIL, LIFO, FIFO or any other string. If LIFO or FIFO are specified a control block is created which will cause lines to placed in the console stack for reading. Any other string results in a control block which performs output to the virtual console, with NIL specifying that trailing blanks are to be deleted, and a carriage return appended.

A negative value signals the need for a larger **sysdep-area** for the control block, and the usual convention, that the negation of the value is the required size, is followed.

A value of 1 indicates that the second argument was neither "INPUT" nor A value of 1 indicates that the second argument was neither "INPUT" nor "OUTPUT".

RDLINE —— *command to system-dependent interface*

(CALLBELOW **"RDLINE" sysdep-area1 sysdep-area2**)

This command reads a line from the virtual console, using a control block, **sysdep-area2**, which was created by an earlier call to GCNPLIST. The data read is placed in **sysdep-area1**. Data beyond the capacity of **sysdep-area1** is lost.

The value of the command is the return code from the system read operation.

TPLINE —— *command to system-dependent interface*

(CALLBELOW **"TPLINE" sysdep-area1 sysdep-area2**)

This command writes data to the virtual console using a control block, **sysdep-area2**, defined by an earlier call to GCNPLIST. The form of the control block specifies whether the data is to be displayed or if it is to be placed in the console input stack.

32XXPLST —— *command to system-dependent interface*

(CALLBELOW **32XXPLST sint sysdep-area**)

In CMS this command evaluates a Diagnose x'24' and stores the three words of data returned into **sysdep-area**. It also attempts to determine the type of console, on the assumption that **sint** is -1, and returns what it thinks is the number of lines on the console.

See the VM/SP System Programmer's Guide.

32XXWRIT —— *command to system-dependent interface*

(CALLBELOW **"32XXWRIT" sint1 sysdep-area sint2 sint3**)

This command uses the CMS Diagnose x'58' to write multiple lines, including attribute bytes to the virtual console. It does not use the full screen support, but only issues CCWCODE x'19's. See the VM/SP System Programmer's Guide for the full details of this operation.

sysdep-area contains the data to be written to the console. **sint3** specifies the number of bytes to be written, and **sint2** specifies the index (zero indexing) of the first byte from **sysdep-area** which is to be written. **sint1** is used to set the CTL field of the CCW. Reasonable values for **sint2** are:

```
x'ff'          255        Clear the screen.  Other arguments
                                ignored.
x'fe'          254        Revert to normal CP/CMS screen.
x'nn'          1 to n     Start writing at line nn.
x'nn'+x'80'    nn+128     Start writing at line nn, with
                                ..MORE
```

37.3 File I/O

RESOLVEE —— *command to system-dependent interface*

(CALLBELOW **"RESOLVEE" sysdep-area1 sysdep-area2**)

This command resolves a CMS file name, and if such a file exists, returns various pieces of information about it.

sysdep-area1 contains a filename, filetype and filemode, separated by blanks. Any of the field may be an *, and the filemode may be a single letter. A CMS STATE command is issued and the first file (if any) which matches the given name is used.

The value of the command is the return code from STATE. If it is zero, i.e. if a file was found, **sysdep-area2** is filled in with the following data, separated by blanks.

```
Filename                      8 bytes + 1 blank
Filetype                      8 bytes + 1 blank
Filemode                      2 bytes + 1 blank
Record form (F | V)           1 byte  + 1 blank
Logical record length        11 bytes + 1 blank
Number of records            11 bytes + 1 blank
Access (RO | RW)              2 bytes + 1 blank
Date (year)                   2 bytes + 1 blank
Date (month)                  2 bytes + 1 blank
Date (day)                    2 bytes + 1 blank
Time (hour)                   2 bytes + 1 blank
Time (minute)                 2 bytes + 1 blank
Time (second)                 2 bytes

Total                        67 bytes
```

In the following example the odd use of PRINT is simply to keep the line length within the page boundaries.

```
(SETQ RESOLVED %69'''')
Value = %69''''
(SETQ FILE "COMP2 LISP *")
Value = "COMP2 LISP *")
(CALLBELOW "RESOLVEE" FILE RESOLVED)
Value = 0
(PROGN (PRINT RESOLVED) ())
"COMP2    LISP     G2 V 00000000081 00000002537 RO 84 03 28 11 00 57"
Value = NIL
```

A negative value signals that **sysdep-area2** is too small. A value of 28 indicates that no such file could be found. For other return codes see the VM/SP CMS Command and Macro Reference manual.

GFIPLIST —— *command to system-dependent interface*

(CALLBELOW **"GFIPLIST" sysdep-area1** ["INPUT" | "OUTPUT"] ["F" , "V"] **sint sysdep-area2**)

This command creates a control block to be used for file I/O in **sysdep-area2**.

sysdep-area1 contains the CMS file name, as in RESOLVEE. **sint** specifies the record length, but is ignored for all but new F format output files. Similarly, the third argument, ["F" | "V"], is ignored for files which already exist.

A negative value signals the need for a larger **sysdep-area2** argument. Other non-zero values are:

1	Second argument neither "INPUT" nor "OUTPUT"
2	**sysdep-area1** could not be resolved for output.
3	**sint** not a number.
n	Return code from STATE, neither 0 nor 28.

FILEIN —— *command to system-dependent interface*

(CALLBELOW **"FILEIN" sysdep-area1 sysdep-area2 sint**)

This command reads one record from the file specified by **sysdep-area1**, a control block created by an earlier call to GFIPLIST. The data is placed in **sysdep-area2**. If the record length exceeds the capacity of **sysdep-area2** any excess data is lost. **sint** specifies the record number to be read.

The value is the return code from the CMS read command, see VM/SP CMS Command

FILEOUT —— *command to system-dependent interface*

(CALLBELOW **"FILEOUT" sysdep-area1 sysdep-area2 sint**)

This command writes one record into a file specified by **sysdep-area2**, a control block created by an earlier call to GFIPLIST. **sint** specifies the record number to be written, and **sysdep-area1** contains the data.

For F format files the data is truncated or padded (if possible) as needed. Padding is only done if **sysdep-area1** has a capacity equal to or larger than the record length, otherwise what ever data follows **sysdep-area1** in memory is written.

The value is the return code form the CMS write command, see the FSWRITE macro for details.

WRBLK —— *command to system-dependent interface*

(CALLBELOW **"WRBLK" sysdep-area1 sint1 sint2 sint3 sysdep-area2**)

This command performs a "block write" in to an F format file. **sysdep-area2** must be a control block for an F format output file, created by GFIPLIST. The data in **sysdep-area1** is written as **sint2** records of **sint1** bytes each, starting with record **sint3**.

This command is used chiefly by FILESEG and SAVELISP. The value is the return code from the CMS write command.

SHUT —— *command to system-dependent interface*

(CALLBELOW **"SHUT" sysdep-area**)

This command performs a CMS FINIS operation on the file specified by **sysdep-area**, a control block created by GFIPLIST.

If **sysdep-area** is recognized as an invalid control block the value is -1, otherwise it is 0.

ERASE —— *command to system-dependent interface*

(CALLBELOW **"ERASE" sysdep-area**)

This command erases the file specified by **sysdep-area**, a control block created by GFIPLIST.

RENAME —— *command to system-dependent interface*

(CALLBELOW **"RENAME" sysdep-area1 sysdep-area2**)

This command renames an existing file. **sysdep-area1** and **sysdep-area2** must each contain a CMS filename, filetype and filemode, as in the argument for RESOLVEE. They must be fully resolved names. The file named by **sysdep-area1** is renamed to the name in **sysdep-area2**

The value is the return code from the CMS RENAME command.

REPLACEF —— *command to system-dependent interface*

example (CALLBELOW **"REPLACEF" sysdep-area1 sysdep-area2**)

This command replaces one file by another. **sysdep-area1** and **sysdep-area2** each must name (as in RENAME) a file, but both must be existing files, both on the same read/write disk. The file named in **sysdep-area1** is erase and the file named in **sysdep-area2** is renamed with the first file's name.

Values returned are:

```
0       Operation succeeded
n > 0   RENAME failed with return code n.
-1      An argument is ill formed
-2      A file exists with temporary name
        (1st-name LISPUT1)
-3      Files not on same disk
-4      File named in sysdep-area2 doesn't exist
-5      File named in sysdep-area1 doesn't exist
-6      Files on read only disk
```

37.4 Memory Management

SEGLOC —— *command to system-dependent interface*

(CALLBELOW **"SEGLOC" sysdep-area**)

Returns the address of the discontiguous shared segment named in **sysdep-area**, if it exists, otherwise returns 0 (zero). If **sysdep-area** is less than eight bytes in length (including any required trailing blanks) the value returned is -8.

SHARED —— *command to system-dependent interface*

(CALLBELOW **"SHARED" sint**)

Tests the memory location specified by **sint** to determine whether it is a discontiguous shared segment linked in shared mode, or if it is private memory. Value is 0 if area is nonshared, 1 otherwise.

UNSHRSEG —— *command to system-dependent interface*

(CALLBELOW "UNSHRSEG" **sysdep-area**)

This command forces the discontiguous shared segment named in **sysdep-area** into non-shared mode. **sysdep-area** must be at least eight bytes long, with trailing blanks. If there is no such shared segment the value is 1, otherwise it is 0. If **sysdep-area** is less than eight bytes the value is -8.

RELPAGES —— *command to system-dependent interface*

(CALLBELOW "RELPAGES" **sint1** **sint2**)

This command issues a CMS Diagnose x'10', to release one or more pages of memory. **sint1** specifies the address of the first page, and **sint2** specifies the number of bytes of memory to be released. Both must be multiples of 4096.

37.5 Miscellaneous

TEMPUS —— *command to system-dependent interface*

(CALLBELOW "TEMPUS" **sysdep-area**)

This command executes a CMS Diagnose x'0C' and returns the result in **sysdep-area**. If **sysdep-area** is too small an appropriate negative value is returned.

The 32 bytes returned represent the time and date and the current virtual and total processor times. The format is:

```
8 bytes   MM/DD/YY
8 bytes   HH:MM:SS
8 bytes   virtual processor time (microseconds)
8 bytes   total processor time (microseconds)
```

LISPRET —— *command to system-dependent interface*

(CALLBELOW "LISPRET" **sint** {**sysdep-area** | 0})

This command returns control from LISP/VM to the program which called it. **sint** is passed in register 15, as a return code. If the second argument is not 0, the address of the contents of **sysdep-area** is passed in register 1, where it will be found by a caller which invoked LISP via a BALR.

OBEY —— *command to system-dependent interface*

(CALLBELOW "OBEY" **sysdep-area**)

This command attempts to pass its argument, **sysdep-area**, to the CMS command interpreter. It assumes the existence of a command or EXEC named OBEY which can be invoked via a SVC 202.

LISPXSUB —— *command to system-dependent interface*

(CALLBELOW "LISPXSUB" **sysdep-area**)

This command attempts to pass its argument, **sysdep-area**, to the private XEDIT sub-command interface, used by Lispedit. Note that if **sysdep-area** starts with the substring "CMS ", the remainder of the argument is interpreted as a CMS command.

This is the preferred interface with the CMS command interpreter.

SVC202 —— *command to system-dependent interface*

(CALLBELOW "SVC202" **sysdep-area**)

This command simply issues an SVC 202 addressed to **sysdep-area**. The value returned is the return code from the SVC.

SYSID —— *command to system-dependent interface*

(CALLBELOW "SYSID")

This command returns a value of 1 if the system is a VM/SP system. When a TSO system exists it will return a value of 2.

CONVSAL —— *command to system-dependent interface*

(CALLBELOW "CONVSAL")

This command returns a value of 0 if the system is running on the batch machine, otherwise it returns a value of 1.

USEREXT —— *command to system-dependent interface*

(CALLBELOW "USEREXT" **cvec** [{**sint** | **sysdep-area**}*])

This command provides a linkage with an, optional, user written module, the user extension. The calling sequence for the user extension is the same as for the system interface module, with **cvec** taking the place of the command string. An empty user extension exists, USEREXT ASSEMBLE, which contains extensive comments describing the linkage conventions.

A.0 MACLISP Compatibility Package

In an attempt to ease the task of porting MACLISP code to LISP/VM we have created a compatibility package. This implements most of the commonly used MACLISP functions which are not defined in LISP/VM. All of the function forms of MACLISP, EXPR, FEXPR, LEXPR, MACRO are supported. Variables are defaultly lexical, unless explicitly declared via a SPECIAL declaration. MACLISP style arrays are supported as are hunks, although CAR and CDR are not supported on hunks. In an effort to support CAR and CDR of NIL being NIL, we took the expedient of defining CAR and CDR of any non pair to return itself instead of generating an error. This allows chains of CARs and CDRs to open code and yields large performance advantages in exchange for some loss of error checking. In the same spirit of loss of error checking, the user should be warned that in LISP/VM, unbound identifiers evaluate to themselves, and extra arguments passed to functions are ignored. This should not affect the porting of debugged MACLISP programs. Functions with the same name but different semantics are handled by providing a compile time macro package which performs the appropriate renamings. Thus we only provide compatibility for compiled functions not interpreted ones. The complete MACLISP macro environment is not provided, so the user is encouraged to macro-expand code before porting, or to port the appropriate macro definitions as well.

One major area of incompatibility is that an identifier cannot be used both as a variable and a function at the same time. The other major difficulty involves input/output. MACLISP expects to run in a full duplex ASCII world, while LISP/VM under CMS is half duplex ebcdic. No real attempt to compensate for this has been made in the compatiblity package and the user will probably need to modify the source code. RETURN in LISP/VM exits from the nearest enclosing LAMBDA or PROG, thus users may need to scan their code for RETURNs from LAMBDAs nested within PROG's and replace each with a CATCH and THROW. The STATUS and SSTATUS functions are not provided, and also the various switches which can be used to control the behavior of MACLISP are not available. Finally, no attempt has been made to implement MACLISP style error handling, although CATCH and THROW are supported.

All MACLISP source files should be given filetype MACLISP to use the appropriate syntax table, and should begin with:

```
(ORADDTEMPDEFS 'MACLMACS)
```

to install the appropriate compilation renamings.

The MACLISP runtime environment is installed by:

```
(LOADVOL 'MACLISP)
(LOADVOL 'ARRAY)
```

For the format package:

```
(EXF (FORMAT COMMON) =)
```

For boolean operations on small integers:

```
(LOADVOL 'BOOLE)
```

Note that small integers are 27 bit integers not 36.

B.0 Generalized Iteration

A generalized iterative construct is available. It is not part of the basic system, but must be loaded from the LISPLIB ITERATOR. This set of macros was initially patterned on the INTERLISP iterator. Not all of the INTERLISP iterator facilities have been replicated, while a number of extensions have been added. Some of the peculiarities in the semantics can be traced to the historical development of this facility.

The basic form is a list of key words, many of which may act as operators, separated by (zero or more, depending on the operator) operands. For example, the expression:

```
(FOR X FROM 1 TO 10 BEGIN (PRINT X))
```

will print the numbers from one to ten.

The order of the operators is only partially constrained, (see "Semantics of Complex Iterators" on page 322), thus the given example could have been written as:

```
(BEGIN (PRINT X) FOR X FROM 1 TO 10)
```

or

```
(FOR X TO 10 FROM 1 BEGIN (PRINT X))
```

or in several other ways.

Each iterative expression is macro-expanded into a single PROG. No other contours are generated (exclusive of those created by user written function expressions), thus a RETURN expression will terminate the iteration, returning the value of its operand as the value of the iteration expression. The PROG will bind a number of variables. Some of these, referred to below as "iterators," have special semantics. Other variables may be specified by the user, while still others are automatically generated. The variable X in the above example is an iterator, and a prime iterator at that. The fact that it is an iterator makes it subject to loop control, i.e., its value is initialized by the FROM, is stepped in the loop, and is tested by the TO. The fact that it is a prime iterator makes it the default argument of the BEGIN operand, (e.g. PRINT, as above), if that should be determined to be a operator. Thus the following is also equivalent to the above expression:

```
(FOR X FROM 1 TO 10 BEGIN PRINT)
```

(See "Defaults" on page 319.)

We can classify the operators into a number of groups.

DECLARATIONS —— specify variables and their uses;

ITERATION-CONTROLS —— initialize, step, and perform end-of-loop tests;

PROCESSORS —— define what is to be done within the loop and what the value is to be;

MISCELLANEOUS —— is a catch-all for various remaining operators;

MODIFIERS —— which effect the meaning of preceding operators.

B.1 Declarations

FOR —— *key word and macro*

FOR {**id** | (**id** := **exp**)}
FOR OLD {**id** | (**id** := **exp**)}

Establishes **id** as a prime iterator, and optionally initializes it to the value of **exp**. If **id** is declared to be OLD, it is not bound in the PROG, and any initialization is performed outside the PROG. The first iterator declaration in an iteration expression should be a FOR, and all but the first FOR should follow ANDs.

```
(FOR X FROM 1 TO 3 BEGIN (PRINT X))
1
2
3
Value = NIL
(FOR OLD X FROM 1 TO 3 BEGIN (PRINT X))
1
2
3
Value = NIL
X
Value = 4
```

AS —— *key word and macro*

AS {**id** | (**id** := **exp**)}
AS OLD {**id** | (**id** := **exp**)}

Establishes **id** as a secondary iterator, and optionally initializes it to the value of **exp**. If **id** is declared to be OLD, it is not bound in the PROG, and any initialization is performed outside the PROG.

```
(FOR X FROM 1 TO 4
  AS Y FROM 6 TO 3
  BEGIN (PRINT (CONS X Y)))
(1 . 6)
(2 . 5)
(3 . 4)
(4 . 3)
Value = NIL
```

(The distinction between prime and secondary iterators is important in expression resolution, see page 320.)

BIND —— *key word and macro*

BIND {**id** | (**id** := **exp**)} ...

(where "**id**" may be an **id** or one of the two forms, (FLUID **id**) or (LEX **id**))

Adds the **id**(s) to those bound by the generated PROG, initialized to NIL, or to "**exp**", if given. **exp** will be evaluated outside the bindings of the PROG.

```
(FOR X FROM 1 TO 4 BIND (Y := 10) BEGIN (PRINT (CONS X Y)))
(1 . 10)
(2 . 10)
(3 . 10)
(4 . 10)
Value = NIL
```

B.2 Iteration-controls

IS —— *key word and macro*

IS **exp**

Causes **exp** to be evaluated at the start of each evaluation of the loop and its value to be assigned to the most recently declared iterator.

```
(FOR X FROM 1 TO 4
  AS Y IS (TIMES 2 X)
  BEGIN (PRINT (CONS X Y)))
(1 . 2)
(2 . 4)
(3 . 6)
(4 . 8)
Value = NIL
```

BECOMING —— *key word and macro*

BECOMING **exp**

Causes **exp** to be evaluated at the end of each evaluation of the loop and its value to be assigned to the most recently declared iterator.

```
(FOR X FROM 1 TO 4
   AS (Y := 1) BECOMING (TIMES 2 Y)
   BEGIN (PRINT (CONS X Y)))
(1 . 1)
(2 . 2)
(3 . 4)
(4 . 8)
Value = NIL
```

BY —— *key word and macro*

BY **exp**

(without either FROM/TO or IN/ON forms)

Causes **exp** to be evaluated at the end of each evaluation of the loop and its value to be added to the most recently declared iterator.

```
(FOR X FROM 1 TO 4
   AS (Y := 1) BY (TIMES 2 Y)
   BEGIN (PRINT (CONS X Y)))
(1 . 1)
(2 . 3)
(3 . 9)
(4 . 27)
Value = NIL
```

FROM —— *key word and macro*

FROM **exp**

Causes the most recently declared iterator to be initialized to the value of **exp**, evaluated outside the generated PROG. The value of **exp** should be numeric.

```
(FOR X FROM 1 TO 2 BEGIN (PRINT X))
1
2
Value = NIL
```

TO —— *key word and macro*

TO **exp**

The value of **exp**, computed once, outside the generated PROG, is used as an end-of-loop test. The test applies to the most recently declared iterator. The nature of the test depends on the BY operator.

```
(FOR X FROM 4 TO 6
   AS Y FROM 10 TO 0
   BEGIN (PRINT (CONS X Y)))
(4 . 10)
(5 . 9)
(6 . 8)
Value = NIL
```

BY —— *key word and macro*

BY exp

(used with FROM and/or TO)

The value of **exp**, computed at the end of each iteration, is added to the most recently declared iterator. The sign of the value determines the end-of-loop test. If the value is positive the loop will terminate when the sum exceeds the TO value, if zero the loop terminates immediately, if negative, when the sum is less than the TO value. In all cases, the loop is evaluated at least once. If BY is omitted, it defaults to 1.

```
(FOR X FROM 8 BY (QUOTIENT X -2) BEGIN (PRINT X))
8
4
2
1
Value = NIL
   (Note, (QUOTIENT 1 -2) = 0)
```

UPTO —— *key word and macro*

UPTO exp

The value of **exp**, computed once, outside the generated PROG, is used as an end-of-loop test. The iteration terminates as soon as the iterator exceeds the value of the UPTO expression. The test is performed at the top of the loop, allowing zero iterations.

```
(FOR X FROM 4 UPTO 6 BEGIN (PRINT X))
4
5
6
Value = NIL
(FOR X FROM 4 UPTO 3 BEGIN (PRINT X))
Value = NIL
```

DOWNTO —— *key word and macro*

DOWNTO exp

The value of **exp**, computed once, outside the generated PROG, is used as an end-of-loop test. The iteration terminates as soon as the iterator is less than the value of the DOWNTO expression. The test is performed at the top of the loop, allowing zero iterations.

```
(FOR X FROM 4 DOWNTO 2 BEGIN (PRINT X))
4
3
2
Value = NIL
(FOR X FROM 4 DOWNTO 5
   BEGIN (PRINT X))
Value = NIL
```

IN —— *key word and macro*

IN {exp | (id := exp)}

Causes **exp**, evaluated outside the generated PROG, to be bound to the specified **id**, if present, or to a generated variable. The value is tested for "PAIRness" at the top of the loop, with the iteration terminating if it is not a PAIR. On each iteration the most recently declared iterator will be assigned successive CARs of this value, while the variable is re-set to its CDR, unless a stepping expression has been specified with BY, IS or BECOMING.

```
(FOR X IN '(1 2 3 4) BEGIN (PRINT X))
1
2
3
4
Value = NIL
(FOR X IN (Y := '(1 2)) BEGIN (PRINT X) (PRINT Y))
1
(1 2)
2
(2)
Value = NIL
```

IN —— key word and macro

IN **exp** called **id**

(N.B. CALLED is a key word.)

Equivalent to:

IN (**id** := **exp**)

```
(FOR X IN '(1 2) CALLED Y BEGIN (PRINT X) (PRINT Y))
1
(1 2)
2
(2)
Value = NIL
```

IN —— key word and macro

IN OLD {**id** | (**id** := **exp**)}

(N.B. OLD is a key word.)

This differs from the previous operator in that the variable is used to hold the successive values of the list being traversed. If no initial value is specified, the value at the time of entry is used. In either case the variable is not bound by the generated PROG, but is referenced as a free variable.

```
(SETQ X '(1 2 3))
Value = (1 2 3)
(FOR Y IN OLD X BEGIN (PRINT Y))
1
2
3
Value = NIL
X
Value = NIL
```

IN —— key word and macro

IN VECTOR {**exp** | (**id** := **exp**)}

(N.B. VECTOR is a key word.)

Causes a variable, either generated or specified, to be initialized by the value of **exp**, evaluated outside the generated PROG. In addition generated variables are initialized to MAXINDEX of that value, and to 0. These are the vector, limit and index variable for the most recently declared iterator. The index is tested, and if it is larger than the limit the iteration terminates. On each iteration the corresponding iterator is assigned the current element of the vector, while the index is stepped by one, unless a stepping expression has been specified.

```
(SETQ A '<1 2 3 4>)
Value = <1 2 3 4>
(FOR X IN VECTOR A BEGIN (PRINT X))
1
2
3
4
Value = NIL
```

IN —— *key word and macro*

IN VECTOR {**exp** | (**id1** := **exp**)} INDEX **id2**

(N.B. VECTOR and INDEX are key words.)

Differs from the previous case in that the INDEX variable is specified rather than generated, and thus is accessible to the programmer.

```
(SETQ A '<100 200 300 400>)
Value = <100 200 300 400>
(FOR X IN VECTOR A INDEX Y BEGIN (PRINT (LIST Y X)))
(0 100)
(1 200)
(2 300)
(3 400)
Value = NIL
```

ON —— *key word and macro*

ON {**exp** | (**id** := **exp**)}
ON OLD {**id** | (**id** := **exp**)}
ON **exp** CALLED **id**

(N.B. OLD and CALLED are key words.)

The ON operator differs from IN in assigning the entire remaining object to the iterator, rather than the CAR, and in not accepting a modifier of VECTOR.

```
(FOR X ON '(1 2 3) BEGIN (PRINT X))
(1 2 3)
(2 3)
(3)
Value = NIL
```

INSIDE —— *key word and macro*

INSIDE {**exp** | (**id** := **exp**)}
INSIDE OLD {**id** | (**id** := **exp**)}
INSIDE **exp** CALLED **id**

(N.B. OLD and CALLED are key words.)

Differs from the IN operator in that the iteration is evaluated for the final, non-PAIR value of the object, as well as for the CARs.

```
(FOR X INSIDE '(1 2 3 . 4) BEGIN (PRINT X))
1
2
3
4
Value = NIL
```

INLIST —— *key word and macro*

INLIST {**exp** | (**id** := **exp**)}
INLIST OLD {**id** | (**id** := **exp**)}
INLIST **exp** CALLED **id**

(N.B. OLD and CALLED are key words.)

Differs from the ON operator in that the iteration is evaluated for the final, non-PAIR value of the object, as well as for the CDRs.

```
(FOR X INLIST '(1 2 3 . 4) BEGIN (PRINT X))
(1 2 3 . 4)
(2 3 . 4)
(3 . 4)
4
Value = NIL
```

BY —— *key word and macro*

BY {**operator.** | **exp**}

(used with IN, ON, INSIDE or INLIST)

If the **operator** form is used, the generated variable holding the list (or the original variable, if OLD was specified) is updated to the value of the operator applied to that variable. If an **exp** is given the "holding" variable is updated to the value of the **exp**. Use the CALLED operator to name the "holding" variable for use in **exp**.

The test for **operator** is described below.

```
(FOR X IN '(1 2 3 4 5 6) CALLED Y BY (CDDR Y) BEGIN (PRINT X))
1
3
5
Value = NIL
```

(Note that if the list had had an odd number of elements an error would have occurred.)

BY —— *key word and macro*

BY {**operator** | **exp**}

(used with IN VECTOR)

Differs from the previous case, in that the value computed uses the current index and is used to set the next value of the index into the vector.

```
(FOR X IN VECTOR '<1 2 3 4> INDEX Y BY (PLUS Y 2) BEGIN (PRINT X))
1
3
Value = NIL
```

IS —— *key word and macro*

IS **exp**

(Used with IN, ON, et cetera)

Causes **exp** to be evaluated at the start of each evaluation of the loop and its value to be assigned to the most recently declared iterator. When the iteration is over a list the value becomes the new list, when over a vector, the new index.

```
(FOR X IN '(1 2 3 4 5 6) CALLED Y IS (CDDR Y)
  BEGIN (PRINT X))
3
5
Value = NIL
(FOR X IN VECTOR '<1 2 3 4 5> INDEX Y IS (PLUS Y 2)
  BEGIN (PRINT X))
3
5
Value = NIL
```

BECOMING —— *key word and macro*

BECOMING **exp**

(Used with IN, ON, et cetera)

Causes **exp** to be evaluated at the end of each evaluation of the loop and its value to be assigned to the most recently declared iterator. Differs from BY in this context only in that no substitution of "holding" variable occurs.

```
(FOR X ON '(((A) B) C) CALLED Y BECOMING (CAR Y)
  BEGIN (PRINT X))
(((A) B) C)
((A) B)
(A)
Value = NIL
```

CYCLING —— *key word and macro*

CYCLING

When present, causes the most recently declared iterator to be re-initialized when its normal terminating condition occurs.

```
(FOR X FROM 1 TO 5
  AS Y CYCLING FROM 1 TO 3
  BEGIN (PRINT (CONS X Y)))
(1 . 1)
(2 . 2)
(3 . 3)
(4 . 1)
(5 . 2)
Value = NIL
(FOR X FROM 1 TO 5
  AS Y CYCLING IN '(A B)
  BEGIN (PRINT (CONS X Y)))
(1 . A)
(2 . B)
(3 . A)
(4 . B)
(5 . A)
Value = NIL
```

WHILE —— *key word and macro*

WHILE **exp**

The value of **exp** is computed at the start of each iteration, and the expression terminates the first time it returns NIL.

```
(FOR X IN '(1 2 B 3 4) WHILE (NUMBERP X) BEGIN (PRINT X))
1
2
Value = NIL
```

UNTIL —— key word and macro

UNTIL **exp**

The value of **exp** is computed at the start of each iteration, and the expression terminates the first time it does not return NIL.

```
(FOR X IN '(A B 2 C D) UNTIL (NUMBERP X) BEGIN (PRINT X))
A
B
Value = NIL
```

REPEATWHILE —— key word and macro

REPEATWHILE **exp**

The value of **exp** is computed at the end of each iteration, and the expression terminates the first time it returns NIL.

```
(FOR X IN '(1 2 B 3 4) REPEATWHILE (NUMBERP X)
  BEGIN (PRINT X))
1
2
B
Value = NIL
```

REPEATUNTIL —— key word and macro

REPEATUNTIL **exp**

The value of **exp** is computed at the end of each iteration, and the expression terminates the first time it does not return NIL.

```
(FOR X IN '(A B 2 C D) REPEATUNTIL (NUMBERP X)
  BEGIN (PRINT X))
A
B
2
Value = NIL
```

WHEN —— key word and macro

WHEN **exp**

The value of **exp** is computed before the body of the loop and the body is skipped when ever that value is NIL.

```
(FOR X IN '(A 1 2 B C 3) WHEN (NUMBERP X)
  BEGIN (PRINT X))
1
2
3
Value = NIL
```

UNLESS —— key word and macro

UNLESS **exp**

The value of **exp** is computed before the body of the loop and the body is skipped when ever that value is not NIL.

```
(FOR X IN '(A 1 2 B C 3) UNLESS (NUMBERP X)
  BEGIN (PRINT X))
A
B
C
Value = NIL
```

B.3 Processors

These operators all are followed by one or more **exps**, which constitute the body of the loop.

BEGIN —— key word and macro

BEGIN **exp** ...

The value of the iteration expression is NIL (unless another value is forced, see the VALUE operation, below).

```
(FOR X IN '(A B C)
   AS Y FROM 5 TO 7
   BEGIN (PRINT X) (PRINT Y))
A
5
B
6
C
7
Value = NIL
```

COLLECT —— key word and macro

COLLECT **exp** ...

The value of the iteration expression is a list of the successive values of the last **exp**.

```
(FOR X FROM 1 TO 10 BY 2 COLLECT (PRINT X) (ADD1 X))
1
3
5
7
9
Value = (2 4 6 8 10)
```

JOIN —— key word and macro

JOIN **exp** ...

The value of the iteration expression is a list formed by NCONCing the successive values of the last **exp**.

```
(FOR X IN '(A B C) JOIN (PRINT X) (LIST X X))
A
B
C
Value = (A A B B C C)
```

JOINPAIR —— key word and macro

JOINPAIR **exp** ...

Similar to JOIN, but the assumption is that the values will be PAIRs, so RPLACD is used to build the resulting value. Saves a function call (NCONC) in the inner loop.

```
(SETQ A '(A B C))
Value = (A B C)
(SETQ B (FOR X IN A JOINPAIR (PRINT X) (LIST X))
A
B
C
Value = (A B C)
(EQ A B)
Value = NIL
```

SUM —— *key word and macro*

SUM **exp** ...

The value of the iteration expression is the sum of the values of the last **exp**.

```
(FOR X FROM 3 TO 8 SUM (TIMES X X))
Value = 199
```

COUNT —— *key word and macro*

COUNT **exp** ...

The value is the number of iterations.

```
(FOR X FROM 1 TO 20 BY (PLUS X X) COUNT (PRINT X))
1
3
9
Value = 5
```

ANY —— *key word and macro*

ANY **exp** ...

The iteration continues until either a normal termination occurs or until the last **exp** has a non-NIL value. In the first case the value of the iteration expression is NIL, in the second it is non-NIL, specifically 1 (one).

```
(FOR X IN '(T H I S I S A T E S T) ANY (EQ X 'A))
Value = 1
(FOR X IN '(T H I S I S A T E S T) ANY (EQ X 'Z))
Value = NIL
(FOR X IN '(T H I S NIL A NIL B) ANY (NULL X))
Value = 1
```

NOTALWAYS —— *key word and macro*

NOTALWAYS **exp** ...

The iteration continues until either a normal termination occurs or until the last **exp** has a NIL value. In the first case the value of the iteration expression is NIL, in the second it is non-NIL, specifically 1 (one).

```
(FOR X IN '(A A A X A A) NOTALWAYS (EQ X 'A))
Value = 1
(FOR X IN '(A A A A A A) NOTALWAYS (EQ X 'A))
Value = NIL
```

ALWAYS —— *key word and macro*

ALWAYS **exp** ...

The iteration continues until either a normal termination occurs or until the last **exp** has a NIL value. In the first case the value of the iteration expression is NIL, in the second it is non-NIL, specifically 1 (one).

```
(FOR X IN '(1 2 3 4 5) ALWAYS (NUMBERP X))
Value = 1
(FOR X IN '(1 2 3 X 5) ALWAYS (NUMBERP X))
Value = NIL
(FOR X IN '(NIL NIL NIL NIL NIL) ALWAYS (NULL X))
Value = 1
```

NEVER —— *key word and macro*

NEVER **exp** ...

The iteration continues until either a normal termination occurs or until the last **exp** has a non-NIL value. In the first case the value of the iteration expression is non-NIL, specifically 1 (one), in the second it is NIL.

```
(FOR X IN '(A B C D E) NEVER (NUMBERP X))
Value = 1
(FOR X IN '(V W X 4 Z) NEVER (NUMBERP X))
Value = NIL
(FOR X IN '(NIL NIL NIL NIL NIL) NEVER X)
Value = 1
```

ALWAYSTRY —— *key word and macro*

ALWAYSTRY **exp** ...

The iteration continues until a normal termination occurs. If the last **exp** always returns a value of NIL, that is the value of the iteration expression. Otherwise the value is the value of the first prime iterator at the last instance in which the last **exp** has a non-NIL value.

```
(FOR X IN '(1 A 2 B) ALWAYSTRY (PRINT X) (NUMBERP X))
1
A
2
B
Value = 2
(FOR X IN '(A B C) ALWAYSTRY (PRINT X) (NUMBERP X))
A
B
C
Value = NIL
(FOR X IN '(A NIL B NIL) ALWAYSTRY (PRINT X) (NULL X))
A
NIL
B
NIL
Value = NIL
```

UNIQUELY —— *key word and macro*

UNIQUELY **exp** ...

The iteration continues until either a normal termination occurs or until the last **exp** has a returned a non-NIL value for two iterations. In the first case the value of the iteration is the value of the first prime iterator at the time of the single success, or NIL, if there was none. In the second it is NIL.

```
(FOR X IN '(A 1 B) UNIQUELY (PRINT X) (NUMBERP X))
A
1
B
Value = 1
(FOR X IN '(1 A 2 B) UNIQUELY (PRINT X) (NUMBERP X))
1
A
2
Value = NIL
(FOR X IN '(A NIL B) UNIQUELY (PRINT X) (NULL X))
A
NIL
B
Value = NIL
```

MAXIMIZE —— *key word and macro*

MAXIMIZE **exp** ...

The values of the last **exp** are examined, and those which are numeric are compared, determining the maximum value. The value of the iteration expression is the value of the first prime iterator which corresponds to that maximum.

```
(FOR X FROM 10 TO 6
  AS Y IN VECTOR '<2 1 5 4 3>
  MAXIMIZE (PLUS X Y))
Value = 13
```

MINIMIZE —— *key word and macro*

MINIMIZE **exp** ...

The values of the last **exp** are examined, and those which are numeric are compared, determining the minimum value. The value of the iteration expression is the value of the first prime iterator which corresponds to that minimum.

```
(FOR X FROM 10 TO 6
  AS Y IN VECTOR '<2 4 5 1 3>
  MINIMIZE (PLUS X Y))
Value = 9
```

B.4 Miscellaneous

AND —— *key word*

AND

Closes the definition of the current prime iterator, and establishes an new one.

```
(FOR X FROM 1 TO 4
  AND FOR Y FROM 6 TO 3
  BEGIN (PRINT (CONS X Y)))
(1 . 6)
(2 . 5)
(3 . 4)
(4 . 3)
Value = NIL
```

FIRST —— *key word and macro*

FIRST **exp** ...

The **exp**(s) are evaluated once, before the iteration is started. All variables declared by the BIND operator will have been initialized, but iterators will not have been.

```
(FOR X FROM 1 TO 3
   BIND (A := 2) (B := 3) C
   FIRST
      (SETQ C (TIMES A B))
      (PRINT "Starting")
   COLLECT (LIST X A B C))
"Starting"
Value = ( (1 2 3 6)
          (2 2 3 6)
          (3 2 3 6) )
```

FINALLY —— *key word and macro*

FINALLY **exp** ...

The **exp**(s) are evaluated after the termination of the iteration, but before returning the value. At this time all FROM iterators will have been stepped (to the values which are not used in the loop), while all IN/ON iterators will have their last value.

```
(FOR X IN '(1 2 3 4 5)
   AS Y FROM 1 TO 3
   COLLECT (LIST X Y)
   FINALLY (PRINT (LIST X Y)))
(3 4)
Value = ((1 1) (2 2) (3 3))
```

EACHTIME —— *key word and macro*

EACHTIME **exp** ...

The **exp**(s) are placed in the body of the loop, preceding the WHEN and UNLESS tests. Thus they are evaluated on every iteration. Their values are discarded.

```
(FOR X IN '(1 A 2 B) EACHTIME (PRINT X)
  WHEN (NUMBERP X) COLLECT X)
1
A
2
B
Value = (1 2)
```

REPLACE —— *key word and macro*

REPLACE [WITH] **exp**
REPLACE **id** WITH **exp**

The REPLACE operator must follow the declaration of an IN/ON variable, or must specify an IN/ON variable as **id**.

The list (or vector) which corresponds to the variable will be updated (via RPLACA or SETELT) to the successive values of **exp**. If no process operator (COLLECT, etc.) is specified the value of the iteration is NIL.

There can be as many REPLACE operators as there are IN/ON variables.

```
(SETQ A '(1 2 3))
Value = (1 2 3)
(FOR X IN A REPLACE (ADD1 X))
Value = ()
A
Value = (2 3 4)
(SETQ B '<10 20 30>)
Value = <10 20 30>
(FOR X IN VECTOR B REPLACE X WITH (ADD1 X))
Value = ()
B
Value = <11 21 31>
(FOR X IN VECTOR B
  AND FOR Y IN A
  REPLACE X WITH Y
  REPLACE Y WITH X)
Value = ()
A
Value = (11 21 31)
B
Value = <2 3 4>
```

VALUE —— key word and macro

VALUE **exp**

The value of **exp**, computed immediately before returning from the iterative expression, becomes the value of the expression, overriding any normal value.

```
(SETQ A 2)
Value = 2
(FOR (X := 1.0) BIND Y Z
   BEGIN
      (SETQ Z (* X X))
      (SETQ Y (/ (- Z A) (* 2.0 X)))
      (SETQ X (- X Y))
   REPEATUNTIL (= A Z)
   VALUE X)
Value = 1.414213562373
```

YIELDING —— key word and macro

YIELDING **exp**

(with predicate-like and selection operators)

On each iteration which results in (or could result in) a "true" value, or which would select a value of the first prime iterator, **exp** is evaluated and its value saved. Upon completion of the iteration expression the last computed value of **exp** becomes the value of the iteration expression.

```
(FOR X FROM 10 TO 6
   AS Y IN VECTOR '<2 1 5 4 3>
   AS I FROM 0
   MAXIMIZE (PLUS X Y)
   YIELDING I)
Value = 2
(FOR X IN '(A A Z A A) NOTALWAYS (EQ X 'A) YIELDING X)
Value = Z
```

YIELDING —— key word and macro

YIELDING VECTOR

(with "list valued" operators. N.B. VECTOR is a key word.)

Causes the value, normally a list, to be returned as a vector.

```
(FOR X FROM 1 TO 5 COLLECT X YIELDING VECTOR)
Value = <1 2 3 4 5>
(FOR X IN '(A B C D)
   AS Y IN VECTOR '<M N O>
   JOIN (LIST X Y) YIELDING VECTOR)
Value = <A M B N C O>
```

B.5 Modifiers

OLD —— *key word*

operator OLD

Indicates that the variable controlled by the preceding operator (FOR, AS, etc.), or nominally suppling its value (IN, ON, etc.) is to be used explicitly in the generated code, rather than being rebound. See various examples, e.g. page 303 and page 307.

CALLED —— *key word*

operator **exp** CALLED **id**

Allows the programmer to specify the variable to be used to hold the list being processed by an IN-type operator. See page 307 for example.

VECTOR —— *key word*

IN VECTOR **exp**

Indicates that the iteration is to be over a vector rather than a list. See page 307.

VECTOR —— *key word*

YIELDING VECTOR

Indicates that a normally list-valued operator is to return a vector. See page 317.

INDEX —— *key word*

IN VECTOR **exp** INDEX **id**

Allows the programmer to specify the variable to be used to hold the index for an iteration over a vector. See page 308.

B.6 Fine control

The user can exercise control

RETURN

As stated above, the entire iterative expression is transformed into a single PROG, exclusive of any explicit function expressions included. Thus a RETURN expression will cause immediate termination of the iteration, with the results of any VALUE, YIELDING, and/or FINALLY code ignored.

```
(FOR X IN '(1 2 A 3 4)
   EACHTIME
     (COND
       ( (NULL (NUMBERP X))
         (RETURN "Help!") ))
   SUM X)
Value = "Help!"
```

EXIT

In addition, the body to the loop, that is the EACHTIME code, the WHEN/UNLESS test, and the processing code, is enclosed in a SEQ. Thus an EXIT expression in any of those places will "drop you out the bottom".

```
(FOR X IN '(5 2 4 1 3)
   AS Y IN '(1 2 5 3 4)
   COLLECT
      (COND
        ( (GREATERP X 3) X )
        ( (EQ (PLUS X Y) 4)
          (EXIT ()) )
        ( 'T Y )))
Value = (5 4 4)
```

GO

Finally, all the code specified by FIRST, EACHTIME and FINALLY, as well as all but the last expression in the processing code (and even that, in BEGIN) are made into the PROG or SEQ body. Thus identifiers become labels which can be "GOne" to, with infinite possibility for errors.

```
(FOR X IN '(1 A 2 B)
   BEGIN
      (COND ((NUMBERP X) (GO N)))
      (PRINTEXP "Not number ")
      (PRINO X)
      (GO E)
  N   (PRINTEXP "Number ")
      (PRINO (PLUS 10 X))
  E   (PRINT ',))
Number 10,
Not number A,
Number 12,
Not number B,
Value = NIL
```

B.7 Defaults

The iterator construct treads a fine line between conciseness and over-cleverness. (Which side of the line it is on is a matter of heated debate.) This conciseness is achieved by the provision of a wide range of default actions. Several of these have been used in the preceding examples, without comment.

missing FROM/TO/BY

The presence of either FROM or TO associated with an iterator implies a default value for the other and for BY, if not provided. A defaulted FROM becomes 1 (one). A defaulted TO causes the termination test to be omitted. A defaulted BY becomes either 1 (one) or -1 (minus one), depending on whether the TO value is greater or less than the FROM value.

```
(FOR X TO 5 BY 2 BEGIN (PRINT X))
1
3
5
Value = NIL
(FOR X TO -2 BEGIN (PRINT X))
1
0
-1
-2
Value = NIL
```

default operator (BEGIN)

Many operators require a single following expression or variable. If additional expressions are found after the required one, they are treated as if they were preceded by a BEGIN.

```
(FOR X IN '(1 2 3) (PRINT X))
1
2
3
Value = NIL
(FOR X IN '(1 A 2) WHEN (NUMBERP X) (PRINT X))
1
2
Value = NIL
```

expression resolution

Many operators are related to specific iterators. In these cases a heuristic test for "function-ness" is done, and if the expression satisfies the test the "expression" is treated as an operator to be applied to the corresponding iterator. An expression is considered a function if it is:

1. a macro or function expression;
2. an built-in function;
3. an fbpi;
4. of the form (FUNCTION expression);
5. an identifier whose value passes the "function" test.

Note that during compilation various identifiers which you might wish to be treated as functions may not have the requisite values. When in doubt, use the FUNCTION form.

```
(FOR X IN '(A 1 B 2) WHEN NUMBERP (PRINT X))
1
2
Value = NIL
```

For processor operators, which take one or more expressions, the default "function-ness" test is applied to the last expression. If it passes it is applied to all the prime iterators. This is the crucial distinction between prime and secondary iterators.

```
(FOR X IN '(1 2 3) COLLECT ADD1)
Value = (2 3 4)
(FOR X IN '(A B C) AND Y IN '(U V W) COLLECT LIST)
Value = ((A U) (B V) (C W))
(FOR X IN '(1 3 5)
   AS Y IN '(10 9 8)
   AND Z IN '(100 200 300)
   COLLECT
      (LAMBDA (A B) (PLUS A B Y)))
Value = (111 212 313)
```

Finally, for those operators which require at least one expression, if no expression is given, their associated iterator(s) is used.

```
(FOR X IN '(1 2 3) COLLECT YIELDING VECTOR)
Value = <1 2 3>
```

default prime iterators

Every iteration expression has at least one prime iterator. If none is explicitly declared (with a FOR) then one is generated. Any operators which refer to a "last declared iterator" may be used, before any other iterators are declared.

```
(FROM 1 TO 5 AS Y IN '(A B C D E F G) COLLECT Y)
Value = (A B C D E)
```

multiple defaults

Of course, these various default all work together.

```
(IN '(1 2 3) AND IN '(5 6 7) COLLECT LIST)
Value = ((1 5) (2 6) (3 7))
(TO 10 AND IN '(1 C 5) WHEN NUMBERP JOIN LIST)
Value = (1 1 3 5)
```

B.8 Peculiarities

The historical development of the iterator has left various traces. There are several aspects of the semantics which are only explicable as remnants of the INTERLISP iterator, which was the ultimate pattern for this facility. While some of the original operators have been dropped, and a number of new operators have been added, an attempt was made to maintain as much compatibility as possible. In some cases this has resulted in somewhat unmnemonic operators, and unwieldy and unsafe semantics.

operators

Among the processing operators, there are several, e.g. NEVER, ANY, UNIQUELY, which perform tests during the iteration, and either select a value from the iterator variables, or simply return a boolean value indicating success or failure. For any given option there is only one type (selector or predicate-like), and there is no clear pattern to the choice.

In general, the YIELDING operator will convert either into the other, thus:

```
(FOR X IN A ANY (EQ X 'Z))
```

is predicate-like, returning either NIL or 1. To covert it to a selector one may write:

```
(FOR X IN A ANY (EQ X 'Z) YIELDING X).
```

Similarly,

```
(FOR X IN A UNIQUELY (NULL X))
```

is a selector, in this case always returning NIL. To convert it to a predicate, one may write:

```
(FOR X IN A UNIQUELY (NULL X) YIELDING 1).
```

semantics of TO

The TO operator has semantics which can be both expensive and misleading.

The termination test is defined in terms of the sign of of the BY expression. If BY is specified, and is a constant a single test can be generated, as is the case when BY is defaulted but FROM and TO are both constants. In all other cases a double test is needed, first to determine the sign of the BY value, then to see if the TO value has been passed, either upward or downward.

This has two effects. The generated code contains tests which, in many cases will never be used, and the test must follow the loop, thus precluding the possibility of zero iterations. This is so because the value of BY may depend on variables which are changed inside the loop.

Having some FROM/TO loops test at the beginning (which would be possible in the constant cases) while others tested at the end seemed potentially confusing. In order to allow simpler cases to be complied efficiently, as well as allowing zero iterations, the UPTO and DOWNTO operators have been introduced.

semantics of BY

When an explicit BY is provided for an IN/ON iterator, there is no automatic check for its validity. A simple example of this is mentioned in the preceding description, where it is noted that a (... BY (CDDR X) ...) may cause an error if the list being iterated over has an odd number of elements.

In general, the BY expression will be evaluated whenever there is a pair left, i.e. at least one element. Since the default BY is CDR, it is always correct. For an explicit BY it is the programmer's responsibility to assure correctness, either by control of the arguments or by tests in the BY expression itself.

semantic exposures

The semantics of the generated form are dangerously exposed. Three stereotyped labels, and one stereotyped variable, are included in each expanded iteration expression. The "top-of-the-loop" is labeled $$LP, the "end-of-the-body" (before final tests) is labeled $$ITERATE, and the FINALLY code (if any), and the return are labeled $$OUT. One could use GOs to explicitly control the evaluation, but it is not recommended. The use of EXIT provides the ability to "(GO $$ITERATE)" from inside the loop, while RETURN will do most (albeit not all) that a "(GO $$OUT)" does. The variable $$VAL is used to hold the the putative final value. It can be explicitly set, with SETQ, however the operator VALUE has been provided to allow a user specified value to be given.

B.9 Semantics of Complex Iterators

order of evaluation

There are two points in the loop where iterators are reset. All IN/ON iterators and IS iterators are set at the top of the loop. All FROM/TO iterators and BECOMING iterators are set at the end. In addition, all FROM iterators and any variable with an explicit initial-value will have its first value during the FIRST code and at initial entry into the loop.

All new values in each place are computed before any assignments are done. Thus all "new value" expressions (such as BY, IS and BECOMING expressions) are evaluated in the context of the last iteration. In addition the pre-body and post-body termination tests differ. Termination because of exhaustion of IN/ON iterators prevents any of the pre-body assignments being done. The post-body tests (from TO iterators) are performed after the assignments are complete.

Because of this, during the evaluation of the FINALLY code IN/ON and IS iterators will have the values they had during the last iteration while TO and BECOMING iterators will have their *next* value. (The UPTO/DOWNTO test are pre-body, but the assignments are post-body so they will have their *next* value during the FINALLY code.)

Constraints on order of operators

The code generator which translates the iterative expressions into LISP/VM will accept a wide variety of expressions as equivalent. This does not show in the examples, where a highly stereotyped style has been used.

There do exist some constraints on the form of iterative expressions.

The prime constraint is simply that the first object in the iterative expression must be one of the operators, not one of the key words.

The next constraint is that (with one exception) the iterative control operators must follow the declaration of the iterator they refer to. Thus, two FROMs, without an intervening AS or AND {FOR} are illegal, as they would both be governing the same iterator. A FROM, e.g., which is preceded by no iterator declarations is legal, however. It is taken as governing the default prime iterator. If a FOR is encountered before an AS or AND, it usurps the, until then unnamed, default iterator, and the preceding control expressions govern it. Similarly an AND sets up another unnamed prime iterator which can be captured by a FOR after control operators have effected it.

TABLE OF SYSTEM FUNCTIONS, VARIABLES AND COMMANDS

See Figure 1 on page 4 for a description of the operand types.

+ —— *function and compile-time macro*

(+ **num** ...) = **num**
Return the sum of all the arguments.

& —— *macro*

(& [**item1** [**item2** ...]])
Back trace printer, unQUOTEd arguments. See page 229.

$CHECKMODE —— *CMS interface function*

($CHECKMODE **item**)
Returns () * or a CMS filemode letter.

$CLEAR —— *CMS interface function*

($CLEAR)
Clear the screen if the console is a 32xx.

$ERASE —— *CMS interface function*

($ERASE **filearg**)
Erase a file.

$EST —— *CMS interface function*

($EST {**bORcvec** | **id**})
Tests the status of the disk with the given filemode.

$FCOPY —— *CMS interface function*

($FCOPY **filespec1** **filespec2** [APPEND])
Copy the contents of **filespec1** creating, or appending to **filespec2**. See page 275.

$FILEDATE —— *CMS interface function*

($FILEDATE **filearg**)
Return a string of the form "mm/dd/yy hh:mm" or ().

$FILEP —— *CMS interface function*

($FILEP **filearg**)
Verify arguments for reasonableness and return a list (fn ft fm). See page 273.

$FILEREC —— *CMS interface function*

($FILEREC **filespec** **sint**)
Returns a string containing the specified record or (). Will read from files or console.

$FILESIZE —— *CMS interface function*

($FILESIZE **filearg**)
Returns () or the file size in bytes (this is a pessimistic upper bound for v-format files).

$FINDFILE —— *CMS interface function*

($FINDFILE **filespec** **list**) = (fn ft fm) or ()
Fill-in a filespec from the list of filetypes until a file is found. See page 274.

$FINDLINK —— *CMS interface function*
($FINDLINK [**bORcvec** | **id**])
Return a modeletter if the argument denotes a linked disk.

$INFILEP —— *CMS interface function*
($INFILEP **filearg**) = (fn ft fm) or ()
Locate an input file. See page 273.

$LISTFILE —— *CMS interface function*
($LISTFILE **filearg**)
Returns a list of all files that match the arguments. See page 274.

$MODELETTER —— *CMS interface function*
($MODELETTER {**bORcvec** | &id})
Return only a legal CMS modeletter. Translate NIL and * to A.

$OUTFILEP —— *CMS interface function*
($OUTFILEP **filearg**)
Value is (fn ft fm) if it can be an output file. See page 273.

$OUTOLDP —— *CMS interface function*
($OUTOLDP **filearg**)
Value is (fn ft fm) if file exists and it can be an output file. See page 273.

$READFLAG —— *CMS interface function*
($READFLAG)
Value is non-() if there are lines in the program (or console) stacks.

$REPLACE —— *CMS interface function*
($REPLACE **filespec1** **filespec2**) The data in **filespec2** replaces the data in **filespec1**. **filespec2** is erased, and the data in **filespec1** is lost.

$SCREENSIZE —— *CMS interface function*
($SCREENSIZE)
If the current console is a 32xx, the value is (rows cols) or (rows cols REMOTE).

$SHOWLINE —— *CMS interface function*
($SHOWLINE **cvec** **sint**)
Display the string on the specified line of a 32xx.

$T-DELTA —— *CMS interface function*
($T-DELTA **num**)
Value is elapsed total cpu time.

$T-TIME —— *CMS interface function*
($T-TIME)
Total cpu time in milliseconds since last logon.

$TIMESTAMP —— *CMS interface function*
($TIMESTAMP)
Value is current date and time as a string "mm/dd/yy hh:mm".

$V-DELTA —— *CMS interface function*
($V-DELTA **num**)
Value is virtual cpu time in milliseconds since **num**.

$V-TIME —— CMS interface function

($V-TIME)

Value is virtual cpu time in milliseconds since last logon.

* —— *function and compile-time macro*

(* **num** ...) = **num**

Return the product of all the arguments.

** —— *function*

(** **num1** &number2) = **num**

Raise **num1** to the **num2** power.

- —— *function and compile-time macro*

(- **num1 num2** ...) = **num**

Return the difference between **num1** and **num2**.

/ —— *function*

(/ **num1 num2** ...) = **num**

Return the quotient of **num1** divided by **num2**.

? —— *function*

(? **item1** [**item2** ...])

The & operator with evaluated arguments. See page 236.

? —— *command to system-dependent interface*

(CALLBELOW "?" {**cvec** | **sint**})

Query to SYSDEP, to verify commands.

? —— *key word*

Request to the break loop to repeat the error message.

= —— *function*

(= **item1 item2**) = **boolean**

Value is TRUE when **item1** is EQUAL to **item2**.

ABSTAB —— function

(ABSTAB **sint** &stream)

Forced tab to a specific position.

ABSVAL —— function

(ABSVAL **num**)

Absolute value. See page 104.

ADDOPTIONS —— function

(ADDOPTIONS [**id item**] ...)

Adds pairs to OPTIONLIST See page 70.

ADDRESSOF —— function and compile-time macro

(ADDRESSOF **item**)

Returns the address part of pointer as a small integer.

ADDTEMPDEFS —— function

(ADDTEMPDEFS **filearg**)

Augment the compiler's OR state with the definitions in the specified file.

ADDTOLIST —— *function*

(ADDTOLIST **list item**)
Adds an item to a list if not already there. See page 118.

ADD1 —— *function*

(ADD1 **num**)
Increments its argument by one. See page 105.

ALINE —— *function -- FOR THE EXPERT LISP/VM USER*

(ALINE **sint1 sint2**)
Where **sint2** is a power of 2.
Rounds up to a power of two. See page 107.

ALL-NON-DESC —— *function*

(ALL-NON-DESC **item list**)
Value is a list of all the non-descendible components in the argument. If **list** is a pair, update it to build value.

ALLFUNCTIONS —— *function*

(ALLFUNCTIONS {**id** | **bpi**})
Returns list of functions called by a bpi.

AND —— *macro*

(AND [**exp** ...])
Evaluates expressions until one returns NIL. See page 86.

AND-P —— *macro*

(AND-P [**pred** ...])
Produce a predicate that test for the AND of **pred** ...

ANDBIT —— *function*

(ANDBIT **bvec** ...)
Logical AND of bit vectors. See page 131.

APPEND —— *function*

(APPEND **item1 item2** ...)
Another name for CONC.

APPEND2 —— *function*

(APPEND2 **item1 item2**)
Concatenate exactly two lists.

APPLY —— *built-in function*

(APPLY **app-ob list** [**sd**])
Apply a function to a list of arguments, possibly in a specified environment. See page 57.

ASMTIME —— *function*

(ASMTIME)
Returns time and date of system assembly.

ASSEMBLE —— *function*

(ASSEMBLE **list1** [**list2**])
Where **list1** is either a name/LAP program list or a list of name/LAP program lists.

The LAP assembler See page 63.

ASSOC —— function
(ASSOC **item list**)
Search a list of name-value pairs, uses EQUAL. See page 118.

ASSQ —— function
(ASSQ **item list**)
Search a list of name-value pairs, uses EQ. See page 118.

ATOM —— built-in function
(ATOM **item**)
Tests for not-pairness. See page 73.

AUGMENTGLOBAL —— function
(AUGMENTGLOBAL **list sd**)
Where list is of the form ((name . value) ...).
Add bindings to non-lambda environment of state descriptor.

AUGMENTSTACK —— function
(AUGMENTSTACK **list sd**)
Where list is of the form ((name . value) ...).
Add FLUID bindings to stack environment of state descriptor.

BATCHERROR —— function
(BATCHERROR **item1 item2**)
Action when break is entered in batch mode.

BFARGCOUNT —— function
(BFARGCOUNT **item**)
Returns the minimum number of arguments expected by a built-in function.

BFP —— built-in function
(BFP **item**)
Tests for built-in function type. See page 74.

BGREATERP —— function
(BGREATERP **bvec1 bvecn**)
Compares two bit vectors. See page 136.

BINTP —— built-in function
(BINTP **item**)
Tests for long integer type. See page 76.

BITSUBSTRING2NUM —— function
(BITSUBSTRING2NUM **bvec sint1 sint2**)
Numeric value of up to 24 bits of bitstring.

BOOLEANP —— function
(BOOLEANP)
Synonym for TRUEP, used for message clarity.

BOUNDEDBYP —— function
(BOUNDEDBYP **bpi**)
Searches environment chain for bpi in frame. See page 236.

BOUNDP —— *function*
(BOUNDP **id**)
Tests for the existence of a binding.

BPILEFT —— *function*
(BPILEFT)
Returns bytes of bpi space remaining.

BPILIST —— *definition option*
(BPILIST . **boolean**)
Request printing of an assembly listing on the listing stream. See page 67.

BPINAME —— *function*
(BPINAME **bpi**)
Returns the "name" id of a bpi.

BPIP —— *function*
(BPIP **item**)
Tests for bpi data types. See page 74.

BREAK —— *function*
(BREAK)
Forces an error break.

BUILDLIST —— *macro*
(BUILDLIST **id1 id2 exp**)
Constructs a list from left to right, using **id1** and **id2** as temporary variables. The resulting list is the value of **id1**.

BVECP —— *built-in function*
(BVECP **item**)
Tests for bit vector type. See page 74.

BVEC2CVEC —— *function*
(BVEC2CVEC **bvec**)
Changes type and contents count of bit vector. See page 130.

BYTES —— *function*
(BYTES **item**)
Value is a list, (bytes ids sds), that describes the storage occupied by the argument.

*C*R* —— *function and compile-time macro*
(C*R **cvec item**)
Where string has the form "{A | D}...".
Does nested CAR and CDRs.

CAAAAR —— *function and compile-time macro*
(CAAAAR **item**)
(CAR (CAR (CAR (CAR x)))). See page 109.

CAAADR —— *function and compile-time macro*
(CAAADR **item**)
(CAR (CAR (CAR (CDR x)))). See page 109.

CAAAR —— *function and compile-time macro*
(CAAAR **item**)
(CAR (CAR (CAR x))). See page 109.

CAADAR —— *function and compile-time macro*
(CAADAR **item**)
(CAR (CAR (CDR (CAR x)))). See page 109.

CAADDR —— *function and compile-time macro*
(CAADDR **item**)
(CAR (CAR (CDR (CDR x)))). See page 109.

CAADR —— *function and compile-time macro*
(CAADR **item**)
(CAR (CAR (CDR x))). See page 109.

CAAR —— *function and compile-time macro*
(CAAR **item**)
(CAR (CAR x)). See page 109.

CADAAR —— *function and compile-time macro*
(CADAAR **item**)
(CAR (CDR (CAR (CAR x)))). See page 109.

CADADR —— *function and compile-time macro*
(CADADR **item**)
(CAR (CDR (CAR (CDR x)))). See page 109.

CADAR —— *function and compile-time macro*
(CADAR **item**)
(CAR (CDR (CAR x))). See page 109.

CADDAR —— *function and compile-time macro*
(CADDAR **item**)
(CAR (CDR (CDR (CAR x)))). See page 109.

CADDDR —— *function and compile-time macro*
(CADDDR **item**)
(CAR (CDR (CDR (CDR x)))). See page 109 and page 116.

CADDR —— *function and compile-time macro*
(CADDR **item**)
(CAR (CDR (CDR x))). See page 109 and page 116.

CADR —— *function and compile-time macro*
(CADR **item**)
(CAR (CDR x)). See page 109 and page 116.

CALL —— *built-in function*
(CALL [**item** ...] **app-ob**)
Apply last argument to list of otner arguments. See page 58.

CALLBELOW —— *function and compile-time macro*
(CALLBELOW **cvec** [**sysdep-area** ...])
Invoke system dependent code See page 291.

CALLEDBYP —— *function*

(CALLEDBYP **bpi**)

Searches control chain for calling bpi. See page 236.

CAPACITY —— *function*

(CAPACITY **bORcvec**) = **sint**

Maximum possible size of **bORcvec**. See page 132.

CAR —— *built-in function*

(CAR **item**)

First element of a pair. See page 109 and page 116.

CASEGO —— *macro*

(CASEGO **exp list** ...)

Where list has the form (**item id**).

N way branch, uses EQ. See page 85.

CATCH —— *macro*

(CATCH **id1 exp** [**id2** [**item** ...]])

Establishes stopping point for THROWs to **id1**. See page 87.

CATCHALL —— *macro*

(CATCHALL **exp** [**id** [**item** ...]])

Establishs stopping point for all THROWs. See page 87.

CBOUNDP —— *function*

(CBOUNDP **id**)

Test for binding on the control chain.

CDAAAR —— *function and compile-time macro*

(CDAAAR **item**)

(CDR (CAR (CAR (CAR x)))). See page 109.

CDAADR —— *function and compile-time macro*

(CDAADR **item**)

(CDR (CAR (CAR (CDR x)))). See page 109.

CDAAR —— *function and compile-time macro*

(CDAAR **item**)

(CDR (CAR (CAR x))). See page 109.

CDADAR —— *function and compile-time macro*

(CDADAR **item**)

(CDR (CAR (CDR (CAR x)))). See page 109.

CDADDR —— *function and compile-time macro*

(CDADDR **item**)

(CDR (CAR (CDR (CDR x)))). See page 109.

CDADR —— *function and compile-time macro*

(CDADR **item**)

(CDR (CAR (CDR x))). See page 109.

CDAR —— *function and compile-time macro*

(CDAR **item**)

(CDR (CAR x)). See page 109.

CDDAAR —— *function and compile-time macro*
(CDDAAR **item**)
(CDR (CDR (CAR (CAR x)))). See page 109.

CDDADR —— *function and compile-time macro*
(CDDADR **item**)
(CDR (CDR (CAR (CDR x)))). See page 109.

CDDAR —— *function and compile-time macro*
(CDDAR **item**)
(CDR (CDR (CAR x))). See page 109.

CDDDAR —— *function and compile-time macro*
(CDDDAR **item**)
(CDR (CDR (CDR (CAR x)))). See page 109.

CDDDDR —— *function and compile-time macro*
(CDDDDR **item**)
(CDR (CDR (CDR (CDR x)))). See page 109 and page 116.

CDDDR —— *function and compile-time macro*
(CDDDR **item**)
(CDR (CDR (CDR x))). See page 109 and page 116.

CDDR —— *function and compile-time macro*
(CDDR **item**)
(CDR (CDR x)). See page 109 and page 116.

CDR —— *built-in function*
(CDR **item**)
Second element of pair. See page 109 and page 116.

CEVAL-ID —— *function*
(CEVAL-ID **id**)
Control chain ID evaluator, drops lexicals. See page 48.

CEVAL-LEX-ID —— *function*
(CEVAL-LEX-ID **id**)
Control chain ID evaluator, sees caller's local variables. See page 48.

CGREATERP —— *function*
(CGREATERP **cvec1 cvec2**)
Compare (collating sequence) two character vectors. See page 136.

CHARP —— *function and compile-time macro*
(CHARP **item**)
Tests for one character identifier. See page 75.

CHAR2NUM —— *function*
(CHAR2NUM **id**)
Where **id** has a one character print name.
Returns the number, 0 to 255, for character id.

CLEAR-GC-HISTORY —— function

(CLEAR-GC-HISTORY)
Sets to zero the space utilization averages maintained by the garbage collector.

CLOSEDFN —— special form

(CLOSEDFN **item**)
Where **item** is a function expression or an macro expression. In interpreter = QUOTE, compiles to QUOTEd bpi. See page 57.

CLOSURE —— built-in function

(CLOSURE **exp sd**)
Creates a FUNARG from an expression and a state descriptor. See page 49.

COMCELLP —— function

(COMCELLP **item**)
Test for existence of a communication cell.

COMMON-READTABLE —— function

(COMMON-READTABLE [IN | OUT | INOUT])
Return the readtable that defines Common Lisp syntax. See page 259.

COMPILE —— function

(COMPILE **list1** [**list2**])
Where **list1** is either a name/expression list or a list of name/expression lists. The LISP/VM compiler. See page 63.

COMPILEDEF —— macro

(COMPILEDEF **id**)
Compiles and replaces the value of **id**.

CONC —— function

(CONC **item** ...)
Concatenate lists. See page 113.

CONCAT —— function

(CONCAT **item1** ...)
Concatenate lists or vectors of the same kind.

COND —— special form

(COND [**clause** ...])
Where **clause** is either a character vector (a comment) or of the form (**exp** ...). Nested IF-THEN-ELSEs. See page 85.

CONDERR —— function

(CONDERR **sint item1 item2 app-ob**)
Where **item1** is either a character vector of a list of character vectors. System error signal.

CONS —— built-in function

(CONS **item1 item2**)
Creates a new pair from its arguments. See page 109 and page 113.

CONSFN —— function

(CONSFN **item1 item2** ... **itemn**)
(CONS **item1** (CONS **item2** ... (CONS ... **itemn**) ...)).

CONSOLEPRINT —— *function and compile-time macro*
(CONSOLEPRINT **cvec**)
Output directly to console without using a stream. See page 254.

CONSOLEREAD —— *function*
(CONSOLEREAD) = **cvec**
Input directly from the console (or CMS stack) without using a stream.

CONSTANT —— *macro*
(CONSTANT **exp**)
Where item will be evaluated at compile-time.
QUOTEs the value of its argument. See page 79.

CONTAINSQ —— *function*
(CONTAINSQ **item1 item2**)
Searches **item2** (any pair/vector structure) for EQ **item1**.

CONVERSATIONAL —— *function*
(CONVERSATIONAL)
Tests for batch/non-batch operation.

CONVSAL —— *command to system-dependent interface*
(CALLBELOW "**CONVSAL**")
Tests for batch versus conversational mode See page 298.

COPY —— *function*
(COPY **item**)
Creates a structurally exact copy. See page 141.

COPY-IOTABLES —— *function*
(COPY-IOTABLES) = syntax-name
Replace the values of CURREADTABLE and OUTREADTABLE with copies. See page 263.

COPY-READSTYLE —— *function*
(COPY-READSTYLE [**readtable**])
Copy only the style flags in a readtable. See page 260.

COPY-READTABLE —— *function*
(COPY-READTABLE [**readtable**])
Make a complete copy of a readtable. See page 260.

COS —— *function*
(COS **num**)
Cosine, argument in radians. See page 108.

COUNT —— *macro*
(COUNT **exp id** ...)
Evaluate **exp** and count how many times **id** ... are called.

COUNT-CALLS —— *macro*
(COUNT-CALLS **exp**)
Evaluate **exp** and count the total number of contours entered.

COUNT-EXPR —— *function*

(COUNT-EXPR expr) = number
Count the number of characters in the external form of expr. See page 255.

COUNT-PAIRS —— macro
(COUNT-PAIRS **exp**)
Evaluate **exp** and count calls by caller and callee.

CREATE-SBC —— function
(CREATE-SBC **id**)
Creates a shallow binding cell for an id.

CSET-ID —— function
(CSET-ID **id item**)
Control chain assignment, drops lexicals. See page 80.

CSET-LEX-ID —— function
(CSET-LEX-ID **id item**)
Control chain assignment, sees caller's local variables. See page 80.

CURINSTREAM —— variable
Default input stream See page 277.

CURLINE —— function
(CURLINE **stream**)
Current record number in a FILE stream.

CUROUTSTREAM —— variable
Default output stream. See page 277.

CURREADTABLE —— variable
Value is the default readtable used by all input functions. See page 262.

CURRENT-SYNTAX —— variable
Name of syntax installed most recently by READ-INIT. See page 263.

CURRENTTIME —— function
(CURRENTTIME)
Returns the current time and date as string.

CURRINDEX —— function
(CURRINDEX **stream**)
Position in the line buffer of a stream.

CURS —— function
(CURS)
Value is the current focus in Lispedit.

CVECP —— built-in function
(CVECP **item**)
Test for character vector type. See page 74.

CVEC2BVEC —— function
(CVEC2BVEC **cvec**)
Copies with change of type. See page 130.

CVEC2ID —— *function*
(CVEC2ID **cvec**)
Creates an un-interned identifier. See page 97.

CYCLES —— *function*
(CYCLES **item**)
Returns vector of sub-cycles in an object. See page 144.

CYCLESP —— *function*
(CYCLESP **item**)
Tests for existence of cycles in an object.

DEBUGMODE —— *function*
(DEBUGMODE [**boolean**])
Sets abend/hard-stop switch for fatal errors.

DEFAULT-INDEXED-FILE —— *variable*
The default indexed file used by the SAVE command.

DEFINATE —— *function -- FOR THE EXPERT LISP/VM USER*
(DEFINATE **list1 id1 id2 list2**)
Where **list1** is a name/expression list, **list2** is an option list, **id1** is the input specification and **id2** is the action request.
The generic exp transformer for definition. See page 64.

DEFINE —— *function*
(DEFINE **list1** [**list2**])
Where **list1** is either a name/expression list or a list of name/expression lists.
Sets exps as values, making LAMs applicable. See page 64.

DEFINE-CHAR-CLASS —— *function*
(DEFINE-CHAR-CLASS charspec class [**readtable**])
Modify the class to which a character belongs during input parse. See page 261.

DEFINE-DISPATCH-CHAR —— *function*
(DEFINE-DISPATCH-CHAR charspec [def] [class] [**readtable**])
Define a character to be a dispatch character during input parse. See page 262.

DEFINE-DISPATCH-MACRO —— *function*
(DEFINE-DISPATCH-MACRO dchar char fn [**readtable**])
Define a character to be a dispatch macro during input parse. See page 262.

DEFINE-PURE —— *function*
(DEFINE-PURE **list1** [**list2**])
Where **list1** is either a name/expression list or a list of name/expression lists.
Expands all macros in the expressions and replaces F*CODE by their interpretive components.

DEFINE-READ-MACRO —— *function*
(DEFINE-READ-MACRO charspec fnspec [class] [**readtable**])
Define a character to trigger a read macro during input parse. See page 261.

DEFINE-SYNTAX —— *function*
(DEFINE-SYNTAX rolename charspec [**readtable**]) = charnum or NIL

Define a character to perform a pre-define input or output function. See page 261.

DEFIOSTREAM —— *function*
(DEFIOSTREAM **list sint1 sint2** [&smin3.])
Creates an input/output stream. See page 278.

DEFLAM —— *macro*
(DEFLAM **id bv-list** [**item** ...])
Constructs a LAM expression from **bv-list** and **items** (unevaluated), and calls DEFINE.

DEFLIST —— *function*
(DEFLIST **list item**)
Where list is of the form ((**id item**) ...).
Adds to the property lists of identifiers. See page 99.

DEFMAC —— *macro*
(DEFMAC **id bv-list** [**item** ...])
Constructs a macro expression from **bv-list** and **items** (unevaluated), and calls DEFINE.

DEFUN —— *macro*
(DEFUN **id bv-list** [**item** ...])
Constructs a function expression from **bv-list** and **items** (unevaluated), and calls DEFINE.

DEPLOAD —— *function*
(DEPLOAD {**id1** | **list1**} {**id2** | **list2**})
Where **list1** is a list of one or more LISPLIB names.
Loads missing functions from a bpi's call tree See page 286.

DIFFERENCE —— *function and compile-time macro*
(DIFFERENCE **num1 num2** ...)
Numeric difference of two arguments. See page 103.

DIGITP —— *function and compile-time macro*
(DIGITP **item**)
Test identifier as |0 - |9. See page 75.

DIG2FIX —— *function and compile-time macro*
(DIG2FIX **id**)
Convert identifiers |0 - |9 to small integers.

DISABLE —— *function and compile-time macro*
(DISABLE)
Suppresses recognition of interrupts, cf., ENABLE.

DISABLE-EXCEPTIONS —— *function*
(DISABLE-EXCEPTIONS **id**)
Disables the signaling of floating point underflow or significance exceptions.

DISABLED —— *macro*
(DISABLED [**exp** ...])

Causes its body, [**exp** ...], to be evaluated in a state in which interruptions are deferred.

DISPATCHER —— *function*
(DISPATCHER **item sint**)
System function, gets control on interrupt.

DIVIDE —— *function*
(DIVIDE **num1 num2**)
Quotient and remainder for two arguments. See page 106.

DO —— *macro*
(DO **list1 list2** [**exp** ...])
Where **list1** declares variable, with initial and subsequent values, and **list2** defines the termination test, epilogue and value.
MACLISP based iterator.

DOWNCASE —— *function*
(DOWNCASE {**cvec** | **id** | **list**})
Shifts character vector, id, or list thereof to lower case. See page 135 and page 97.

EBCDIC —— *function*
(EBCDIC **sint**)
Returns the character id for **sint** 0 to 255.

ECONST —— *macro*
(ECONST **sint**)
Expands to quoted value of (EBCDIC **sint**).

EDIT —— *macro*
(EDIT **item**)
Invoke Lispedit to edit the value of **item**.

EDIT-OFF —— *function*
(EDIT-OFF)
To discard the data associated with Lispedit. This data is re-created the next time EDIT is invoked.

ELT —— *function and compile-time macro*
(ELT {**vec** | **bORcvec** | **list**} **sint**)
Gets an element of a list or vector, by index. See page 125, page 132 and page 116.

ELTABLEP —— *function*
(ELTABLEP **item**)
Tests **item** for validity as a first argument to ELT or SETELT.

EMBED —— *function*
(EMBED **id exp**)
Redefines function, may use old definition See page 239.

EMBEDDED —— *function*
(EMBEDDED)
Returns list of EMBEDed functions See page 240.

ENABLE —— *function and compile-time macro*
(ENABLE)
Allows recognition of interrupts, cf DISABLE.

ENABLE-EXCEPTIONS —— *function*
(ENABLE-EXCEPTIONS **id**)
Enables the signaling of floating point underflow or significance exceptions.

ENVIRONEVAL —— *function*
(ENVIRONEVAL **sd1 sd2**)
Combine control and environment from two state descriptors.

EOFP —— *function and compile-time macro*
(EOFP **stream**)
Test a stream for end-of-file condition.

EOL-IS-EOF —— *function -- FOR THE EXPERT LISP/VM USER*
A stream advancing function that forces end-of-file at the end of the current line buffer.

EOLP —— *function and compile-time macro*
(EOLP **stream**)
Test a stream for end-of-line condition.

EQ —— *built-in function*
(EQ **item1 item2**)
Tests for identity. See page 77.

EQQ —— *macro*
(EQQ **item1 item2**)
Where **item1** is a pair/vector structure.
Simple pattern matcher See page 144.

EQSUBSTLIST —— *function*
(EQSUBSTLIST **list1 list2 item**)
Replaces old items by new. Uses EQ, no update. See page 142.

EQSUBSTVEC —— *function*
(EQSUBSTVEC **vec1 vec2 item**)
Replaces old items by new. Uses EQ, no update See page 142.

EQUAL —— *function*
(EQUAL **item1 item2**)
Tests access equivalence. See page 77.

ERASE —— *command to system-dependent interface*
(CALLBELOW **"ERASE" cvec**)
Erase a file See page 296.

ERRCATCH —— *macro*
(ERRCATCH **exp [id [item ...]]**)
Like ERRSET, but true value, optional flag. See page 90.

ERROR —— *function and compile-time macro*
(ERROR **item**)

In-line entry to break, UNWIND forced.

ERROR-PRINT —— *function*
(ERROR-PRINT **item stream**)
Print function for break loop, normally PRETTYPRINT.

ERRORINSTREAM —— *variable*
Default input stream in break loop See page 277.

ERRORN —— *function*
(ERRORN **item** ...)
Like ERROR, with multi-part message.

ERROROUTSTREAM —— *variable*
Default output stream in break loop. See page 277.

ERRORR —— *function and compile-time macro*
(ERRORR **item**)
Inline entry to break, FIN allowed.

ERRSET —— *macro*
(ERRSET **exp [item** ...])
Intercepts UNWINDs from inside evaluations. See page 89.

ESC-BODY-TABLE —— *variable*
Value is table used by Reader and Printer. See page 264.

ESC-FIRST-TABLE —— *variable*
Value is table used by Reader and Printer. See page 264.

EVAL —— *built-in function*
(EVAL **exp** [**sd**])
Evaluate expression in current or given environment. See page 46.

EVAL-GLOBAL-ID —— *function*
(EVAL-GLOBAL-ID **id**)
Non-lambda environment ID evaluator. See page 47.

EVAL-GVALUE —— *function*
(EVAL-GVALUE **id**)
Returns the global value cell contents associated with **id**. See page 48.

EVAL-ID —— *function*
(EVAL-ID **id**)
Returns fluid value of id in current stack. See page 47.

EVAL-LEX-ID —— *function*
(EVAL-LEX-ID **id**)
Returns value of id, sees caller's lexicals. See page 47.

EVALANDFILEACTQ —— *macro*
(EVALANDFILEACTQ [**id**] **exp**)
Like FILEACTQ with EXF-time evaluation.

EVALFUN —— *built-in function*

(EVALFUN **exp [sd]**)
Evaluate expression, dropping lexicals See page 46.

EVALQ —— special form
(EVALQ **id**)
Returns the value of its unevaluated argument, **id**.

EVENP —— function
(EVENP **item**)
Tests for even integers.

EXF —— macro
(EXF **filespec1 filespec2 [id item]** ...)
Evaluate the expressions in a file.

EXF-FILETYPES —— variable
A list of identifiers that specify default filetypes for files specified only by name to various system functions and commands.

EXIT —— special form
(EXIT [**exp**])
Leave a SEQ with value. See page 84.

EXITLISP —— CMS interface function
(EXITLISP [**sint [sysdep-area]**])
Exit to host system.

EXP —— function
(EXP **num**)
Exponentiation function, floats its argument. See page 108.

EXPT —— function
(EXPT **num1 num2**)
Exponentiation function, FIX**FIX gives FIX. See page 104.

EXTERNAL-EVENTS-CHANNELS —— variable
Actions for various interrupts generated outside of LISP/VM.

EXTSTATE —— function
(EXTSTATE **list1 list2 sd**)
Where **list1** is (**id** ...) and **list2** (**item** ...).
Augments the environment of an state descriptor, without capturing control.

F —— variable
Allows F to be false, NIL.

FASTSTREAMP —— function
(FASTSTREAMP **item**)
Pragmatic test for fast I/O stream. See page 77.

FBPIP —— built-in function
(FBPIP **item**)
Test for fbpi type. See page 74.

FETCH —— function

(FETCH **rstrm**)
Read and reconstitute an item from a LISPLIB.

FETCHPSMINT —— function
(FETCHPSMINT **cvec sint1 sint2**)
Like SUBSTRING but return a positive small integer.

FILE —— definition option
(FILE . **rstrm**)
Specifies the output LISPLIB for definition operators. See page 68.

FILE-SYNTAX —— variable
Value defines the mapping from CMS filetypes to syntax names. See page 263.

FILEACTQ —— macro
(FILEACTQ **id item**)
Adds expression to LISPLIB, evaluated at load.

FILEIN —— command to system-dependent interface
(CALLBELOW **"FILEIN" sysdep-area1 sysdep-area2 sint**)
Read a record from a file. See page 295.

FILELISP —— CMS interface function
(FILELISP **filearg**)
Save the workspace on disk and exit to CMS.

FILEOUT —— command to system-dependent interface
(CALLBELOW **"FILEOUT" sysdep-area1 sysdep-area2 sint**)
Write a record to a file See page 295.

FILEQ —— macro
(FILEQ **id item**)
Add an arbitrary item to a LISPLIB.

FILESEG —— function
(FILESEG **filearg**)
Save the shared part of the system on disk.

FILL-VEC —— function
(FILL-VEC &vec. **item**)
Replaces each element of &vec. with **item**.

FIN —— function
(FIN [**item**])
This form is recognized by the break loop, it is not evaluated. Used to continue from the break.

FINITEP —— function
(FINITEP **list**)
Returns NIL if **item** is an infinite (i.e. circular) list.

FIX —— function
(FIX **num**)
Integer part of a floating point number. See page 101.

FLOAT —— *function*
(FLOAT **num**)
Converts integers to floating point numbers. See page 101.

FORCE-GC —— *function*
(FORCE-GC **id**)
Calls the garbage collector, forcing a specified data area (NILsec, STACK, HEAP or UBPIspace) to be extended.

FORCE-GLOBAL —— *function*
(FORCE-GLOBAL **id [sd]**)
Forces binding in nearest non-lambda environment.

FREEZE-SHARED-SEGMENT —— *function*
(FREEZE-SHARED-SEGMENT)
Makes the shared segment area out of bounds.

FUNARG —— *special form*
(FUNARG **exp sd**)
Creates a funarg object from its arguments. See page 49.

FUNARGP —— *built-in function*
(FUNARGP **item**)
Test for FUNARG type. See page 76.

FUNCALL —— *built-in function*
(FUNCALL **app-ob item1 item2** ...)
Invoke **app-ob** after evaluating it first in both compiled and interpreted programs. See page 57.

FUNCTION —— *built-in function*
(FUNCTION **exp**)
Makes a funarg out of an unevaluated argument. See page 49.

*F*CODE* —— *special form*
(F*CODE **exp list** [**lap-statement** ...])
Where the first argument is a declaration and the remaining arguments are LAP instructions.
Allows LAP code to be fed values.

GC-SLOP —— *function*
(GC-SLOP [**sint**])
Returns or sets the tolerance allowed in the garbage collector in testing for any individual minimum free space.

GC-SPAN —— *function*
(GC-SPAN [**sint**])
Returns or sets the rate at which past history is to be discounted by the garbage collector in computing estimates of space usage.

GCCOUNT —— *function*
(GCCOUNT)
Number of times RECLAIM has been called.

GCEXIT —— *function*

(GCEXIT)
Either NIL or a user supplied function which will be invoked at the end of each garbage collection.

GCMSG —— *function*
(GCMSG **boolean**)
Turns on and off RECLAIMs statistic message.

GCNPLIST —— *command to system-dependent interface*
(CALLBELOW "**GCNPLIST**" **cvec1** {"**INPUT**" | "**OUTPUT**"} **sysdep-area2**)
Create a console I/O PLIST See page 293.

GE —— *function*
(GE **item1 item2**) = **boolean**
Value is true if **item1** is greater than or equal to **item2**. Both arguments must be numbers, bit vectors, character vectors or identifiers. See page 146.

GENSYM —— *function and compile-time macro*
(GENSYM)
Creates a gensym. See page 97.

GENSYMP —— *built-in function*
(GENSYMP **item**)
Tests for gensym type. See page 75.

GET —— *function*
(GET {**id** | **list**} **item**)
Searches a property or name-value list. See page 98 and page 118.

GET-CHAR-CLASS —— *function*
(GET-CHAR-CLASS charspec [**readtable**]) = class
Extract the character class to which a character belongs in a readtable. See page 261.

GET-READTABLE —— *function*
(GET-READTABLE [name [mode]]) = **readtable** or NIL
Fetch the readtable that defines a named syntax. See page 264.

GET-READTABLE-FLAGS —— *function*
(GET-READTABLE-FLAGS [**readtable**]) = flaglist
Extract the style flags from a readtable. See page 261.

GET-SYNTAX-CHAR —— *function*
(GET-SYNTAX-CHAR role-name [**readtable**]) = charnum or NIL
Extract from a readtable the character associated with one of the pre-defined input or output roles. See page 261.

GETZEROVEC —— *function*
(GETZEROVEC **sint**)
Creates an **sint** element integer vector with all elements set to zero. See page 123.

GFIPLIST —— *command to system-dependent interface*
(CALLBELOW "**GFIPLIST**" **cvec1** {"**INPUT**" | "**OUTPUT**"} **cvec2 sint sysdep-area3**)

Create a file I/O PLIST See page 295.

GGREATERP —— *function* See page 146.
(GGREATERP **item1 item2**)
Generic comparison, arbitrary order of types.

GLOEXTSTATE —— *function*
(GLOEXTSTATE **list1 list2 sd**)
Augments the non-lambda environment of an state descriptor, without capturing control.

GO —— *special form*
(GO **label**)
Transfer control to a label in a SEQ. See page 84.

GREATERP —— *function*
(GREATERP **num1 num2**)
Compare two numbers. See page 102.

GROWTH-BIAS —— *function*
(GROWTH-BIAS [**list**])
Returns or sets the weighting factors for area growth used by the garbage collector.

GT —— *function*
(GT **item1 item2**) **= boolean**
Value is true if **item1** is greater than **item2**. Both arguments must be numbers, bit vectors, character vectors or identifiers. See page 146.

GVALUE —— *special form*
Returns the contents of the global value cell of an unevaluated identifier. See page 48.

HASHCVEC —— *function*
(HASHCVEC **cvec**)
Computes a small integer hash code for any character vector. See page 140.

HASHEQ —— *function*
(HASHEQ **item**)
Computes a small integer hash code for any argument. See page 139.

HASHID —— *function*
(HASHID **id**)
Computes a small integer hash code for any identifier. See page 140.

HASHTABLE-CLASS —— *function*
(HASHTABLE-CLASS **hashtable**)
Returns the key class of **hashtable**. See page 137.

HASHTABLEP —— *built-in function*
(HASHTABLEP **item**)
Tests for hashtable type. See page 76.

HASHUEQUAL —— *function*
(HASHUEQUAL **item**)

Computes a small integer hash code for any object. See page 140.

HCHAIN —— *function*

(HCHAIN **hashtable1 [hashtable2]**)
Chains **hashtable2** to **hashtable1** as an ancestor, or returns the chain field of **hashtable1**. See page 138.

HCLEAR —— *function*

(HCLEAR **hashtable**)
Removes all key/value pairs from **hashtable**. See page 139.

HCOUNT —— *function*

(HCOUNT **hashtable**)
Returns the number of key/value pairs in **hashtable**. See page 138.

HEAPLEFT —— *function*

(HEAPLEFT)
Returns bytes of HEAP space remaining.

HEXEXP —— *function*

(HEXEXP **item**)
Convert a pointer to a hexadecimal string.

HEXNUM —— *function*

(HEXNUM **int**)
Convert an integer to a hexadecimal string.

HEXSTRINGPART —— *function*

(HEXSTRINGPART **cvec sint1 sint2**)
Substring to hexadecimal, 1 2 3 or 4 chars.

HEXSTRING2NUM —— *function*

(HEXSTRING2NUM &cvec.)
Constructs the small integer which corresponds to the hexadecimal characters in &cvec..

HGET —— *function*

(HGET **hashtable item1 [item2]**)
Returns value of key **item1** in **hashtable**, with optional default value. See page 137.

HGETPROP —— *function*

(HGETPROP **hashtable item1 item2**)
Searchs **hashtable** for an **item2** property on the **item1** property list. See page 137.

HGFACTS —— *function*

(HGFACTS **hashtable [num1 num2]**)
Returns a list of the current count/size ratios for growth and shrinkage. See page 138.

HKEYS —— *function*

(HKEYS **hashtable**)
Returns a list of all the keys in **hashtable**. See page 137.

HPUT —— *function*

(HPUT **hashtable item1 item2**)
Adds or changes an item in a hashtable. See page 138.

*HPUT** —— *function*
(HPUT* **a-list hashtable**)
Adds all the name-value pairs from **a-list** to **hashtable**. See page 138.

HPUTPROP —— *function*
(HPUTPROP **hashtable item1 item2 item3**)
Adds **item3** to **hashtable** as the **item2** property of **item1**. See page 139.

HREM —— *function*
(HREM **hashtable item**)
Removes the key/value pair associated with **item** from **hashtable**. See page 139.

HREMPROP —— *function*
(HREMPROP **hashtable item1 item2**)
Removes the **item2** property from **item1** in **hashtable**. See page 139.

IDENTITY —— *function and compile-time macro*
(IDENTITY **item**)
(LAMBDA (X) X). See page 95.

IDENTP —— *built-in function*
(IDENTP **item**)
Tests for identifier type. See page 74.

IFCAR —— *function and compile-time macro*
(IFCAR **item**)
If argument is a pair, CAR, otherwise NIL. See page 110.

IFCDR —— *function and compile-time macro*
(IFCDR **item**)
If argument is a pair, CDR, otherwise NIL. See page 110.

INITIALOPEN —— *function*
(INITIALOPEN)
Does a fixup of the non-lambda bound streams.

INITSUPV —— *variable*
NIL or expression to start user's supervisor.

INPUT-SYNTAX —— *function*
(INPUT-SYNTAX **id**) = **readtable** or NIL
Fetch the **readtable** associated with input from a file with a filetype of **id**. See page 264.

INTERN —— *function*
(INTERN **cvec**)
Character vector to identifier, copies print name. See page 97.

INTERSECTION —— *function*
(INTERSECTION **list1 list2**)
Set intersection of two lists, no duplicates. See page 115.

INTERSECTIONQ —— *function*

(INTERSECTIONQ **list1 list2**)
Set intersection, uses EQ, no duplicates. See page 115.

INTP —— *built-in function*

(INTP **item**)
Test for integerness. See page 76.

INT2RNUM —— *function*

(INT2RNUM **num**)
Converts integers to floating point numbers. See page 101.

INUMP —— *function*

(INUMP **item**)
Test for integers in the range -2^{31} to $2^{31}-1$, suitable for use as elements of an integer vector.

IS-CONSOLE —— *function*

(IS-CONSOLE **stream**)
Tests a stream for (DEVICE . CONSOLE) property. See page 77.

IVECP —— *built-in function*

(IVECP **item**)
Tests for integer vector type. See page 74.

LAM —— *macro*

(LAM **bv-list** [**exp** ...])
Extension of the LAMBDA special form. See page 60.

LAMBDA —— *special form*

(LAMBDA **bv-list** [**exp** ...])
Special form for applicable expressions. See page 53.

LAPLIST —— *definition option*

(LAPLIST . **boolean**)
Requests a listing of the assembler code. See page 68.

LAP370 —— *function*

(LAP370 **list1** [**list2**])
An archaic synonym for ASSEMBLE.

LAST —— *function*

(LAST **list**)
Final item of a list (ignoring final CDR). See page 116.

LAST-EXP —— *function*

(LAST-EXP)
Returns the last expression SUPV evaluated.

LAST-VALUE —— *function*

(LAST-VALUE)
Returns the last value computed by SUPV.

LASTPAIR —— *function*

(LASTPAIR **list**)

Final pair of a list. See page 117.

LE —— *function*
(LE **item1 item2**) = **boolean**
Value is true if **item1** is less than or equal to **item2**. Both arguments must be numbers, bit vectors, character vectors or identifiers. See page 146.

LEFTSHIFT —— *function*
(LEFTSHIFT **num sint**)
Multiply by a power of two. See page 107.

LENGTH —— *function*
(LENGTH **item**)
Number of items in a list. See page 121.

LESSP —— *function*
(LESSP **num1 num2**)
GREATERP, backwards. See page 102.

LISPRET —— *command to system-dependent interface*
(CALLBELOW **"LISPRET" sint sysdep-area**)
Return control to invoking environment. See page 297.

LISPXSUB —— *command to system-dependent interface*
(CALLBELOW **"LISPXSUB" cvec**)
Pass a command string to XEDIT subcommand interface. See page 297.

LISP370-READTABLE —— *function*
(LISP370-READTABLE [IN | OUT | INOUT]) = **readtable**
Make a readtable that defines LISP/370 syntax. See page 259.

LIST —— *function and compile-time macro*
(LIST **item** ...)
Makes a list of its arguments. See page 113.

LIST-PAIRS —— *macro*
(LIST-PAIRS **exp**)
Like COUNT-PAIRS but prints a file CALL PAIRS A.

LISTING —— *definition option*
(LISTING . **stream**)
Provides a stream for program listings. See page 68.

LISTOFFLUIDS —— *function*
(LISTOFFLUIDS **bpi**)
Returns variables bound FLUID by a bpi.

LISTOFFREES —— *function*
(LISTOFFREES **bpi**)
Returns variables used free by a bpi.

LISTOFFUNCTIONS —— *function*
(LISTOFFUNCTIONS **bpi**)
Returns names of functions called by a bpi.

LISTOFLEXICALS —— *function*

(LISTOFLEXICALS **bpi**)
Returns variables bound LEXically by a bpi.

LISTOFQUOTES —— *function*

(LISTOFQUOTES **bpi**)
Returns all items QUOTEd by a bpi.

LISTOFSAME —— *function*

(LISTOFSAME **list**)
Tests for identical types on list members.

LIST2IVEC —— *function*

(LIST2IVEC **list**)
Where all elements of list are fixed point numbers.
Converts a list to a vector of integers. See page 124.

LIST2RVEC —— *function*

(LIST2RVEC **list**)
Where all elements of list are numbers.
Converts a list to a vector of floating nums. See page 123.

LIST2VEC —— *function*

(LIST2VEC **list**)
Converts a list to a vector of exps. See page 123.

LN —— *function*

(LN **num**)
Natural logarithm. See page 108.

LOADCOND —— *function*

(LOADCOND **filearg**)
Loads currently undefined items from a LISPLIB. See page 285.

LOADVOL —— *function*

(LOADVOL **filearg**)
Loads all items from a LISPLIB. See page 285.

LOG —— *function*

(LOG **num**)
Logarithm, base 10. See page 108.

LOG2 —— *function*

(LOG2 **num**)
Logarithm, base 2. See page 108.

LOTSOF —— *function*

(LOTSOF **item** ...)
Returns "infinite list" of arguments.

LT —— *function*

(LT **item1 item2**) = **boolean**
Value is true if **item1** is less than **item2**. Both arguments must be numbers, bit vectors, character vectors or identifiers. See page 146.

MAADDTEMPDEFS —— *function*

(MAADDTEMPDEFS **filearg**)
Adds items from a LISPLIB to MACRO-APP-SD. See page 70.

MACRO-APP-SD —— *definition option*

(MACRO-APP-SD . **sd**)
Specifies macro application environment for definition. See page 68.

MACRO-INVALIDARGS —— *function*

(MACRO-INVALIDARGS **id list cvec**)
Used by macros to report invalid argument errors.

MACRO-MISSINGARGS —— *function*

(MACRO-MISSINGARGS **id list** &smint)
Used by macros to report missing or excess argument errors.

MAKE-BVEC —— *function*

(MAKE-BVEC **sint**)
Creates a new bit vector. See page 129.

MAKE-CVEC —— *function*

(MAKE-CVEC **sint**)
Create an empty character vector with given capacity. See page 129.

MAKE-FULL-BVEC —— *function*

(MAKE-FULL-BVEC **sint boolean**)
Creates a bit-vector containing **sint** instances of **boolean**.

MAKE-FULL-CVEC —— *function*

(MAKE-FULL-CVEC **sint** [**id** | **cvec** | **sint** .])
Create a character vector, filled with a given character. See page 129.

MAKE-FULL-IVEC —— *function*

(MAKE-FULL-IVEC **sint int**)
Creates a integer vector containing **sint** instances of **int**.

MAKE-FULL-RVEC —— *function*

(MAKE-FULL-RVEC **sint rnum**)
Creates a real vector containing **sint** instances of **rnum**.

MAKE-FULL-VEC —— *function*

(MAKE-FULL-VEC **sint item**)
Creates a vector containing **sint** instances of **item**.

MAKE-HASHTABLE —— *function*

(MAKE-HASHTABLE **id1** [**id2**])
Creates a new, empty, hashtable. See page 137.

MAKE-INSTREAM —— *function*

(MAKE-INSTREAM **filespec** [recnum]) = stream or ()
Value is a stream positioned at the given record if the file exists. Can also create console and internal streams. See page 277.

MAKE-IVEC —— *function*

(MAKE-IVEC **sint**)

Create a vector of integers (32 bits). See page 123.

MAKE-OUTSTREAM —— *function*
(MAKE-OUTSTREAM **filespec** [width [recnum]])
Value is an output stream positioned at the specified record if possible. The width argument is used only for a new file. Can also make console and internal streams. See page 278.

MAKE-RNUM —— *function*
(MAKE-RNUM)
Create a new floating point number cell.

MAKE-RVEC —— *function*
(MAKE-RVEC **sint**)
Create a vector of floating point numbers. See page 123.

MAKE-VEC —— *function*
(MAKE-VEC **sint**)
Create a vector of exps. See page 123.

MAKESTRING —— *function and compile-time macro*
(MAKESTRING **cvec**)
Returns**cvec** arg, compiles with data in-line. See page 131.

MAKETRTTABLE —— *function*
(MAKETRTTABLE {**cvec** | **list**} **item**)
Builds translate and test tables. See page 131.

MAP —— *macro*
(MAP **app-ob list** ...)
Apply function to successive CDRs. See page 91.

MAPC —— *macro*
(MAPC **app-ob list** ...)
Apply function to successive items. See page 91.

MAPCAN —— *macro*
(MAPCAN **app-ob list** ...)
Apply function to successive items, NCONC values. See page 92.

MAPCAR —— *macro*
(MAPCAR **app-ob list** ...)
Apply function to successive items, LIST values. See page 91.

MAPCON —— *macro*
(MAPCON **app-ob list** ...)
Apply function to successive CDRs, NCONC values. See page 92.

MAPE —— *macro*
(MAPE **app-ob** {**vec** | **list**} ...)
Applies operator to vector elements, returns size. See page 94.

MAPELT —— *macro*
(MAPELT **app-ob** {**vec** | **list**} ...)
Applies operator to vector elements, returns vec. See page 94.

MAPHASH —— *function*
(MAPHASH **app-ob hashtable**)
Applies **app-ob** (a two argument function) to each key and value in **hashtable**. See page 139.

MAPLIST —— *macro*
(MAPLIST **app-ob list** ...)
Apply function to successive CDRs, LIST values. See page 91.

MAPOBLIST —— *function*
(MAPOBLIST **app-ob**)
Applies function to elements of obarray. See page 95.

MASKNUM —— *function*
(MASKNUM **int sint**)
Rightmost n bits of a number.

MATEMPDEFINE —— *function*
(MATEMPDEFINE **list1** [**list2**])
Where **list1** is either a name/expression list or a list of name/expression lists. Adds definitions to MACRO-APP-SD. See page 70.

MATEMPSETQ —— *macro*
(MATEMPSETQ **id item** [**list**])
Adds values to MACRO-APP-SD.

MAX —— *function and compile-time macro*
(MAX **num** ...)
Largest of a set of numbers. See page 104.

MAXIMUM-SPACE —— *function*
(MAXIMUM-SPACE [**list**])
Returns or sets the maximum allowed sizes for the individual data areas. These limits are checked by the garbage collector.

MAXINDEX —— *function*
(MAXINDEX **vec**)
Largest valid index, = (SUB1 (SIZE x)). See page 124.

MBPIP —— *built-in function*
(MBPIP **item**)
Tests for mbpi type. See page 74.

MDEF —— *built-in function*
(MDEF **macro-app-ob item** [**sd**])
Perform one macro expansion.

MEMB —— *function*
(MEMB **item list**)
Membership of item in list, uses EQ. See page 117.

MEMBER —— *function*
(MEMBER **item list**)
Membership of item in list, uses EQUAL. See page 117.

MEMQ —— *function*

(MEMQ **item list**)

Membership of item in list, uses EQ. See page 117.

MESSAGE —— *definition option*

(MESSAGE . **stream**)

Provides a stream for information and warning messages. See page 69.

MIN —— *function and compile-time macro*

(MIN **num** ...)

Smallest of a set of numbers. See page 104.

MINIMUM-FREE —— *function*

(MINIMUM-FREE [**list**])

Returns or sets the desired minimum free area for the various data spaces. The garbage collector uses these values in allocating storage.

MINUS —— *function*

(MINUS **num**)

Changes the sign of a number. See page 105.

MINUSP —— *function*

(MINUSP **item**)

Tests for number less than zero. See page 102.

MLAMBDA —— *special form*

(MLAMBDA **bv-list** [**exp** ...])

Special form for macro applicable expressions. See page 59.

MONITOR —— *function*

(MONITOR **id** [**list1** [**list2**]])

Sets a function to print on call and return. See page 237.

MONITOR-ON-OFF —— *variable*

Global switch to allow monitoring PRINT et.al.

MOVEVEC —— *function*

(MOVEVEC **vec1 vec2**)

Copies contents of one vector into another. See page 127.

MSUBST —— *function*

(MSUBST **item1 item2 item3**)

Equivalent to SUBST, but copies only changed structure, uses EQUAL. See page 142.

MSUBSTQ —— *function*

(MSUBSTQ **item1 item2 item3**)

Equivalent to SUBSTQ, but copies only changed structure, uses EQ. See page 142.

MTON —— *function*

(MTON **sint1 sint2**)

Creates list of integers from n to m. See page 116.

NAMED-SYNTAX —— *variable*

Value defines the mapping from syntax names to readtables. See page 262.

NAMEDERRSET —— *macro*
(NAMEDERRSET **id exp [item ...]**)
ERRSET with tag usable by THROW. See page 90.

NAMELISP —— *function*
(NAMELISP **filearg**)
Query or modify the default file parameters used by SAVELISP or FILELISP.

NCONC —— *function*
(NCONC **list** ...)
APPENDs by updating, no copying. See page 119.

NCONC2 —— *function*
(NCONC2 **list list2**)
NCONC with exactly two arguments.

NCONS —— *function*
(NCONS **pair item1 item2**)
Update CAR+CDR of 1st arg by 2nd & 3rd args. See page 111.

NCONSTKD —— *command to system-dependent interface*
(CALLBELOW **"NCONSTKD"**)
Returns no. of physical lines in console stack. See page 292.

NE —— *function*
(NE **item1 item2**) **= boolean**
Value is NIL when **item1** is EQUAL to **item2**.

NEQ —— *function and compile-time macro*
(NEQ **item1 item2**) **= boolean**
Value is NIL when **item1** is EQ to **item2**.

NEWQUEUE —— *function*
(NEWQUEUE **sint1 sint2**)
Changes maximum queue length for an interrupt.

NEXT —— *function and compile-time macro*
(NEXT **[stream]**)
Advance an input stream one item. See page 250.

NILFN —— *function and compile-time macro*
(NILFN)
(LAMBDA () ()). See page 96.

NILLEFT —— *function*
(NILLEFT)
Returns bytes of NIL space remaining.

NILSD —— *function*
(NILSD)
Returns the state descriptor for the root of the stack.

NMAPCAR —— *macro*

(NMAPCAR **app-ob list** ...)
Apply function to successive CARs, RPLACA 1st arg.. See page 93.

NMAPELT —— macro
(NMAPELT **app-ob** {**vec** | **list**} ...)
Applies operator to vector elements, SETELTs 1st v. See page 94.

NOLINK —— definition option
(NOLINK . **boolean**)
Controls the creation of a bpi following assembly. See page 69.

NOMERGE —— definition option
(NOMERGE . **boolean**)
Suppresses the merging of binding contours by the compiler. See page 69.

NONINTERRUPTIBLE —— definition option
(NONINTERRUPTIBLE . **boolean**)
Controls the compilation of interrupt polling instructions. See page 69.

NONSTOREDP —— function and compile-time macro
(NONSTOREDP **item**)
Tests for nonstored (i.e. not heap resident). See page 73.

NOSTOP —— definition option
(NOSTOP . **boolean**)
Suppresses error break during macro expansion. See page 69.

NOT —— built-in function
(NOT **item**)
True in argument is NIL, NIL otherwise. See page 73.

NOT-P —— macro
(NOT-P **pred**)
Creates a predicate that test for the negation of **pred**.

NOTEFILE —— function
(NOTEFILE **stream**)
Returns position information for file stream.

NREMOVE —— function
(NREMOVE **list item** [**sint**])
Delete one (or **sint**) occurrence(s) of **item** in **list**. Uses EQUAL and updates input list.

NREMOVEQ —— function
(NREMOVEQ **list item** [**sint**])
Delete one (or **sint**) occurrence(s) of **item** in **list**. Uses EQ and updates input list.

NREVERSE —— function
(NREVERSE **list**)
Reverses a list in place by updating. See page 120.

NSTACKED —— command to system-dependent interface
(CALLBELOW **"NSTACKED"**)
Returns number of lines in program stack. See page 292.

NSUBST —— *function*

(NSUBST **item1 item2 item3)**
Update a structure with item substitutions, uses EQUAL. See page 145.

NSUBSTQ —— *function*

(NSUBSTQ **item1 item2 item3)**
Update a structure with item substitutions, uses EQ. See page 145.

NULL —— *built-in function*

(NULL **item)**
Tests for NIL value. See page 73.

NULLOUTSTREAM —— *variable*

A no-op stream, a data sink. See page 277.

NUMBEROFARGS —— *function*

(NUMBEROFARGS **bpi)**
Returns number of argument for a bpi.

NUMP —— *built-in function*

(NUMP **item)**
Tests for number types. See page 75.

NUM2TIME —— *function*

(NUM2TIME **int)**
Unpacks small int. to time/date (see TIME2NUM).

OBARRAY —— *function*

(OBARRAY)
Returns copy of the obarray. See page 99.

OBEY —— *CMS interface function*

(OBEY **cvec)**
Passes a command to the host system.

OBEY —— *command to system-dependent interface*

(CALLBELOW **"OBEY" cvec)**
Pass a command string to a user supplied OBEY module or EXEC.. See page 297.

ODDP —— *function*

(ODDP **item)**
Odd test for integers. See page 102.

ONE-OF —— *macro*

((ONE-OF **id1** ...) **idn)**
Test for arg EQ to member of QUOTEd set.

OP-RECOGNITION-SD —— *definition option*

(OP-RECOGNITION-SD . **sd)**
Specifies operator recognition environment for definition. See page 69.

OPTIMIZE —— *definition option*

(OPTIMIZE . **sint)**
Controls the amount of optimization by the compiler. See page 69.

OPTIONLIST —— *variable*

A-list which controls DEFINATE's actions. See page 67.

OR —— *macro*

(OR **exp** ...)

Evaluates expressions until one returns non-NIL. See page 86

OR-P —— *macro*

(OR-P **pred** ...)

Creates a predicate that test for the OR of **pred** ...

ORADDTEMPDEFS —— *function*

(ORADDTEMPDEFS **filearg**)

Adds items from a LISPLIB to OP-RECOGNITON-SD. See page 70.

ORBIT —— *function*

(ORBIT **bvec** ...)

Logical or of bit vectors. See page 131.

ORTEMPDEFINE —— *function*

(ORTEMPDEFINE **list1** [**list2**])

Where item is either a name/expression list or a list of name/expression lists. Adds definitions to OP-RECOGNITION-SD. See page 71.

ORTEMPSETQ —— *macro*

(ORTEMPSETQ **id item** [**list**])

Adds values to OP-RECOGNITION-SD.

OUTPUT-SYNTAX —— *function*

(OUTPUT-SYNTAX **id**) = **readtable** or NIL

Fetch the readtable associated with output to a file with a filetype of **id**. See page 264.

OUTREADTABLE —— *variable*

Value is the default readtable used by all output functions. See page 262.

PACKHEXSTRING —— *function*

(PACKHEXSTRING **cvec**)

Creates the character vector whose hexadecimal form is represented by **cvec**.

PAIRP —— *built-in function*

(PAIRP **item**)

Tests for pair type. See page 73.

PANICMSG —— *command to system-dependent interface*

(CALLBELOW "PANICMSG" **cvec**)

Writes to console unconditionally. See page 292.

PARAMETERS —— *CMS interface function*

(PARAMETERS)

Returns the parameter with which LISP/VM was invoked from CMS.

PLACEP —— *function and compile-time macro*

(PLACEP **item**)

Tests for read place holder type. See page 75.

PLUS —— *function and compile-time macro*
(PLUS **num** ...)
Sums a set of numbers. See page 103.

PLUSP —— *function*
(PLUSP **item**)
Tests for positive numbers. See page 101.

PNAME —— *function*
(PNAME **id**)
Returns a copy of the print name of an id. See page 98 and page 130.

POINTFILE —— *function*
(POINTFILE **pair stream**)
Repositions a file stream.

POLLUP —— *function and compile-time macro*
(POLLUP)
Test for upward branch interrupt.

POP —— *macro*
(POP **id1** **[id2]**)
Pops an item from a list. Value is the new CDR, which becomes value to **id1**.
Optionally assigns CAR to **id2**.

POPP —— *macro*
(POPP **id1** **[id2]**)
Pop an item from a list, setting **id1** to new list. Value is the old list. Optionally assigns CAR to **id2**.

POST —— *function*
(POST **sint item**)
Signal an interrupt.

POST-SELECT —— *function*
(POST-SELECT **sint1 item sint2**)
Signals an interrupt, enables selected POLL.

PPRINT-TABLE —— *function*
(PPRINT-TABLE {IN | OUT | INOUT}) = **readtable**
Copy the style flags in a readtable and set them for prettyprinting. See page 260.

PRETTYPRINT —— *function and compile-time macro*
(PRETTYPRINT **item [stream]**)
Formated print, with TERPRI. See page 253.

PRETTYPRIN0 —— *function and compile-time macro*
(PRETTYPRIN0 **item [stream]**)
Formated print, no TERPRI. See page 253.

PRINM —— *function and compile-time macro*
(PRINM **list [stream]**)
Print list elements, may default stream. See page 253.

PRINT —— *function and compile-time macro*
(PRINT **item [stream]**)
Print item and TERPRI, may default stream. See page 253.

PRINT-CHAR —— *function and compile-time macro*
(PRINT-CHAR **id [stream]**)
Put character on a stream, may default stream. **id** must have a one-character print name. See page 255.

PRINTEXP —— *function and compile-time macro*
(PRINTEXP **cvec [stream]**)
Print contents of character vector, may default stream. See page 254.

PRINTVAL —— *function*
(PRINTVAL **cvec stream**)
Print routine for SUPV.

PRINTWARN —— *function*
(PRINTWARN **list**)
Does PRINM on CUROUTSTREAM, forcing new line.

PRIN0 —— *function and compile-time macro*
(PRIN0 **item [stream]**)
Print item, may default stream. See page 253.

PRIN0R —— *function*
(PRIN0R **item sint [stream]**)
Print item right justified in a field of width **sint**.

PRIN1 —— *function and compile-time macro*
(PRIN1 **item [stream]**)
Print atom, may default stream. See page 254.

PRIN1B —— *function and compile-time macro*
(PRIN1B **item [stream]**)
Print atom + blank, may default stream. See page 254.

PRIN2CVEC —— *function*
(PRIN2CVEC **item**)
Convert an exp to a character vector, as if printed. See page 130.

PROFILE-FILE —— *variable*
The value must be a list of the form (filenamrbl.filetype) to specify a file that is EXFed every time the workspace is loaded.

PROG —— *macro*
(PROG **list [{exp | id} ...]**)
Bind, initialize variables; establish labels. See page 82.

PROGN —— *special form*
(PROGN **[exp ...]**)
Evaluate expressions in seq., value is last. See page 81.

PROGRAM-EVENTS —— *variable*
A-list of actions on errors, list.

PROG1 —— *function and compile-time macro*
(PROG1 **exp** ...)
Evaluate expressions in seq., value is first. See page 83.

PROG2 —— *function and compile-time macro*
(PROG2 **exp exp** ...)
Evaluate expressions in seq., value is second. See page 83.

PROPLIST —— *function*
(PROPLIST **id**)
Returns id's property list, partial copying. See page 98.

PRY —— *function*
(PRY **item** ...)
Forces entry to machine debugger.

PUSH —— *macro*
(PUSH **item id**)
CONS the value of item onto a list, value of id. Sets id to new list.

PUT —— *function*
(PUT {**id** | **list**} **item1 item2**)
Add or change a name-value property. See page 98 and page 119.

PUT-EXPR —— *function*
(PUT-EXPR **item sint cvec**)
Print **item** into **cvec** up to a maximum length of **sint** If the capacity is exceeded, return NIL. See page 255.

PUTBACK —— *function*
(PUTBACK **item stream**)
Replaces item onto head of input stream. See page 249.

QASSQ —— *macro*
(QASSQ **item list**)
ASSQ, functional version of in-line macro. See page 118.

QCAAAAR —— *macro*
(QCAAAAR **item**)
(QCAR (QCAR (QCAR (QCAR x)))). See page 110.

QCAAADR —— *macro*
(QCAAADR **item**)
(QCAR (QCAR (QCAR (QCDR x)))). See page 110.

QCAAAR —— *macro*
(QCAAAR **item**)
(QCAR (QCAR (QCAR x))). See page 110.

QCAADAR —— *macro*
(QCAADAR **item**)
(QCAR (QCAR (QCDR (QCAR x)))). See page 110.

QCAADDR —— *macro*
(QCAADDR **item**)

(QCAR (QCAR (QCDR (QCDR x)))). See page 110.

QCAADR —— *macro*
(QCAADR **item**)
(QCAR (QCAR (QCDR x))). See page 110.

QCAAR —— *macro*
(QCAAR **item**)
(QCAR (QCAR x)). See page 110.

QCADAAR —— *macro*
(QCADAAR **item**)
(QCAR (QCDR (QCAR (QCAR x)))). See page 110.

QCADADR —— *macro*
(QCADADR **item**)
(QCAR (QCDR (QCAR (QCDR x)))). See page 110.

QCADAR —— *macro*
(QCADAR **item**)
(QCAR (QCDR (QCAR x))). See page 110.

QCADDAR —— *macro*
(QCADDAR **item**)
(QCAR (QCDR (QCDR (QCAR x)))). See page 110.

QCADDDR —— *macro*
(QCADDDR **item**)
(QCAR (QCDR (QCDR (QCDR x)))). See page 110 and page 116.

QCADDR —— *macro*
(QCADDR **item**)
(QCAR (QCDR (QCDR x))). See page 110 and page 116.

QCADR —— *macro*
(QCADR **item**)
(QCAR (QCDR x)). See page 110 and page 116.

QCAR —— *macro*
(QCAR **item**)
Like CAR, but no argument type check made. See page 110 and page 116.

QCDAAAR —— *macro*
(QCDAAAR **item**)
(QCDR (QCAR (QCAR (QCAR x)))). See page 110.

QCDAADR —— *macro*
(QCDAADR **item**)
(QCDR (QCAR (QCAR (QCDR x)))). See page 110.

QCDAAR —— *macro*
(QCDAAR **item**)
(QCDR (QCAR (QCAR x))). See page 110.

QCDADAR —— *macro*

(**QCDADAR item**)
(QCDR (QCAR (QCDR (QCAR x)))). See page 110.

QCDADDR —— *macro*
(**QCDADDR item**)
(QCDR (QCAR (QCDR (QCDR x)))). See page 110.

QCDADR —— *macro*
(**QCDADR item**)
(QCDR (QCAR (QCDR x))). See page 110.

QCDAR —— *macro*
(**QCDAR item**)
(QCDR (QCAR x)). See page 110.

QCDDAAR —— *macro*
(**QCDDAAR item**)
(QCDR (QCDR (QCAR (QCAR x)))). See page 110.

QCDDADR —— *macro*
(**QCDDADR item**)
(QCDR (QCDR (QCAR (QCDR x)))). See page 110.

QCDDAR —— *macro*
(**QCDDAR item**)
(QCDR (QCDR (QCAR x))). See page 110.

QCDDDAR —— *macro*
(**QCDDDAR item**)
(QCDR (QCDR (QCDR (QCAR x)))). See page 110.

QCDDDDR —— *macro*
(**QCDDDDR item**)
(QCDR (QCDR (QCDR (QCDR x)))). See page 110 and page 116.

QCDDDR —— *macro*
(**QCDDDR item**)
(QCDR (QCDR (QCDR x))). See page 110 and page 116.

QCDDR —— *macro*
(**QCDDR item**)
(QCDR (QCDR x)). See page 110 and page 116.

QCDR —— *macro*
(**QCDR item**)
Like CDR, but no argument type check made. See page 110 and page 116.

QCEQUAL —— *function and compile-time macro*
(**QCEQUAL cvec1 cvec2**)
Compare contents of two cvecs without type checking.

QCSIZE —— *function and compile-time macro*
(**QCSIZE cvec**)
Like SIZE for character vectors, no check. See page 132.

QEAPPEND —— *macro*
((QEAPPEND {**id** | **cvec** | **sint**} ...) **cvec2**)
SUFFIXes QUOTEd chars to **cvec2**, no check.

QECHAR —— *macro*
((QECHAR **sint**) **cvec**)
Extracts char from **cvec**, no check.

QEFILL —— *macro*
((QEFILL {**id** | **cvec** | **sint**}) **cvec2**)
Pads **cvec2** with QUOTEd character, no check.

QENUM —— *macro*
(QENUM **cvec sint**) = charnum
Extracts char code from **cvec**, no check.

QEQQ —— *macro*
(QEQQ **item1 item2**)
Where **item1** is a pair/vector structure.
Like ECQ, but no type checking. See page 144.

QESET —— *macro*
(QESET **cvec sint** charnum)
Update one character in a cvec.

QESTORE —— *macro*
((QESTORE **sint** {**id** | **cvec** | **sint**} ...) **cvec2**)
Replaces chars in **cvec2** by QUOTEd chars.

QESUFFN —— *macro*
(QESUFFN **cvec sint**)
Suffix a character to a character vector.

QETEST1 —— *macro*
((QETEST1 **sint** {**id** | **cvec** | **sint**} ...) **cvec2**)
Test for char in **cvec2**, no check.

QGET —— *macro*
(QGET {**id** | **list**} **item**)
GET, functional version of in-line macro. See page 118.

QINSERT —— *function and compile-time macro* -- *FOR THE EXPERT LISP/VM USER*
(QINSERT **sint1 sint2 cvec int**)
Insert bytes from fixed number into character vector.

QINSERTFP —— *function and compile-time macro* -- *FOR THE EXPERT LISP/VM USER*
(QINSERTFP **sint cvec rnum**)
Insert floating point number into character vector.

QINSERTSTG —— *function and compile-time macro* -- *FOR THE EXPERT LISP/VM USER*
(QINSERTSTG **sint cvec1 cvec2**)
Insert cvec into cvec.

QLAST —— *macro*
(QLAST **list**)

In-line LAST, no cycle check.

QLENGTH —— *macro*
(QLENGTH **list**)
In-line LENGTH, no check for cycle. See page 122.

QLENGTHCODE —— *function and compile-time macro*
(QLENGTHCODE **vec**)
In-line find number of bytes in vector no type check. See page 132.

QMEMQ —— *macro*
(QMEMQ **item list**)
MEMQ, functional version of in-line macro. See page 117.

QNCONC —— *macro*
(QNCONC **list item**)
In-line NCONC, no cycle check.

QNCONS —— *macro*
(QNCONS **pair item1 item2**)
Update both CAR and CDR, no type checks. See page 112.

QRPLACA —— *macro*
(QRPLACA **pair item**)
Like RPLACA, no type check. See page 111 and page 119.

QRPLACD —— *macro*
(QRPLACD **pair item**)
Like RPLACD, no type check. See page 111 and page 119.

QRPLPAIR —— *macro*
(QRPLPAIR **pair1 pair2**)
Update one pair from the second, no type checks. See page 111.

QRPLQ —— *macro*
(QRPLQ **item1 item2**)
Where **item2** is a pair/vector structure.
Like RCQ, but no type checking. See page 146.

QSABSVAL —— *function and compile-time macro*
(QSABSVAL **sint**)
Like ABSVAL of small integer, no type check. See page 105.

QSADD1 —— *function and compile-time macro*
(QSADD1 **sint**)
Like ADD1 of small integer, no type check. See page 105.

QSAND —— *function and compile-time macro*
(QSAND **sint1 sint2**)
Bitwise AND of small integers, no check. See page 107.

QSBITS —— *macro*
((QSBITS **sint1 sint2**) **sint3**)
Extracts bits from small integer, no check.

QSCAPACITY —— *macro*

(QSCAPACITY **cvec**)
In-line CAPACITY for character vectors, no type check on **cvec**.

QSDEC1 —— *macro*

(QSDEC1 **sint**)
Like QSSUB1, but won't cross zero. See page 105.

QSDIFFERENCE —— *function and compile-time macro*

(QSDIFFERENCE **sint1 sint2**)
Like DIFFERENCE of small integers, no check. See page 105.

QSETBITS —— *macro*

((QSETBITS **sint1 sint2**) **sint3 sint4**)
Sets bits in small integer, no check.

QSETQ —— *macro*

(QSETQ **item1 item2**)
Where **item1** is a pair/vector structure.
Like SETQP, but no type checking, nor boolean value. See page 144.

QSETSIZE —— *macro*

(QSETSIZE **cvec sint**)
Update the current length of a cvec, no type checks. See page 134.

QSETVELT —— *macro*

(QSETVELT **vec sint item**)
Like SETELT of vector, no check. See page 126.

QSGREATERP —— *macro*

(QSGREATERP **sint1 sint2**)
Like GREATERP of small integers, no check. See page 102.

QSINC1 —— *macro*

(QSINC1 **sint**)
Like QSADD1, but won't cross zero. See page 105.

QSLEFTSHIFT —— *function and compile-time macro*

(QSLEFTSHIFT **sint1 sint2**)
Like LEFTSHIFT of small integers, no check. See page 107.

QSLESSP —— *macro*

(QSLESSP **sint1 sint2**)
Like LESSP of small integers, no check. See page 103.

QSMAX —— *function and compile-time macro*

(QSMAX **sint1 sint2**)
Like MAX of small integers, no check. See page 104.

QSMIN —— *function and compile-time macro*

(QSMIN **sint1 sint2**)
Like MIN of small integers, no check. See page 104.

QSMINUS —— *function and compile-time macro*

(QSMINUS **sint**)

Like MINUS of small integers, no check. See page 105.

QSMINUSP —— *macro*
(QSMINUSP **sint**)
Like MINUSP of small integers, no check. See page 102.

QSNOT —— *function and compile-time macro*
(QSNOT **sint**)
Bitwise NOT of small integer, no check. See page 107.

QSODDP —— *macro*
(QSODDP **sint**)
Like ODDP of small interger, no check. See page 102.

QSOR —— *function and compile-time macro*
(QSOR **sint1 sint2**)
Bitwise OR of small integers, no check. See page 107.

QSPLUS —— *function and compile-time macro*
(QSPLUS **sint1 sint2**)
Like PLUS of small integers, no check. See page 103.

QSPLUSP —— *macro*
(QSPLUSP **sint**)
Like PLUSP of small integers, no check. See page 101.

QSQUOTIENT —— *function and compile-time macro*
(QSQUOTIENT **sint1 sint2**)
Like QUOTIENT of small integers, no check. See page 106.

QSREMAINDER —— *function and compile-time macro*
(QSREMAINDER **sint1 sint2**)
Like REMAINDER of small integers, no check. See page 106.

QSSUB1 —— *function and compile-time macro*
(QSSUB1 **sint**)
Like SUB1 of small integers, no check. See page 105.

QSTIMES —— *function and compile-time macro*
(QSTIMES **sint1 sint2**)
Like TIMES of small integers, no check. See page 106.

QSXOR —— *function and compile-time macro*
(QSXOR **sint1 sint2**)
Bitwise XOR of small integers, no check. See page 108.

QSZEROP —— *macro*
(QSZEROP **sint**)
Like ZEROP of small integers, no check. See page 102.

QUIET —— *definition option*
(QUIET . **boolean**)
Suppresses messages to the console. See page 69.

QUOTE —— *special form*

(QUOTE **item**)
Returns argument un-evaluated. See page 79.

QUOTIENT —— function
(QUOTIENT **num1 num2** ...)
(CAR (DIVIDE x y)), quotient after division. See page 103.

QVASSQ —— macro
(QVASSQ **item** &vec.)
In-line VASSQ, no type check on &vec..

QVELT —— macro
(QVELT **vec sint**)
Like ELT of vector, no type check. See page 126.

QVMAXINDEX —— function and compile-time macro
(QVMAXINDEX **vec**)
MAXINDEX for vectors, no checking. See page 125.

QVSIZE —— function and compile-time macro
(QVSIZE **vec**)
SIZE for vectors, no checking. See page 125.

RANDOM —— function
(RANDOM)
Return a random 32-bit integer based on the system seed.

RAW-READTABLE —— function
(RAW-READTABLE [IN | OUT | INOUT]) = **readtable**
Make a readtable with no macro characters at all. See page 260.

RCLASS —— function
(RCLASS **cvec rstrm**)
Returns the class byte from LISPLIB item. See page 283.

RCOPYITEMS —— function
(RCOPYITEMS **filespec1 filespec2 list**)
Copy item(s) from one LISPLIB to another. See page 284.

RDEFIOSTREAM —— function
(RDEFIOSTREAM **list**)
Create a stream for LISPLIB operations. See page 283.

RDLINE —— command to system-dependent interface
(CALLBELOW "**RDLINE**" **sysdep-area1 sysdep-area2**)
Reads line from console. See page 293.

RDROPITEMS —— function
(RDROPITEMS **filespec list**)
Delete item(s) from a LISPLIB file. See page 285.

RE-ENABLE-INT —— function
(RE-ENABLE-INT **sint**)
Forces a one in the units bit of the service priority of interrupt channel **sint**, thus allowing one more try. Used to allow second level external interrupts.

READ —— *function and compile-time macro*
(READ [**stream**])
Reads any expression, may default stream. See page 249.

READ-CHAR —— *function and compile-time macro*
(READ-CHAR [**stream**])
Fetch one character from a stream. See page 249.

READ-INIT —— *function*
(READ-INIT [syntax-name]) = syntax-name
Set CURREADTABLE and OUTREADTABLE to a specified named syntax.
See page 263.

READ-LINE —— *function*
(READ-LINE **stream**)
Reads a line (record). See page 250.

READ-NEXT-LINE —— *function*
(READ-NEXT-LINE [**stream**])
Advance a stream to a new input record.

READ-ONE-LINE —— *function*
(READ-ONE-LINE [**stream**])
Like READ, but close all open lists and vectors at the end of the current line buffer.

READTABLEP —— *built-in function*
(READTABLEP x) = x or NIL
The type predicate for readtables. See page 76 and page 260.

RECLAIM —— *function*
(RECLAIM)
The garbage collector.

RELPAGES —— *command to system-dependent interface*
(CALLBELOW "**RELPAGES**" **sint1 sint2**)
Release virtual pages. See page 297.

REMAINDER —— *function and compile-time macro*
(REMAINDER **num1 num2**)
(CADR (DIVIDE x y)), remainder after division. See page 106.

REMALLPROPS —— *function*
(REMALLPROPS **id**)
Remove all items from a property list. See page 99.

REMOVE —— *function*
(REMOVE **list item** [**sint**])
Delete one (or **sint**) occurrence(s) of **item** in **list**. Uses EQUAL and copies input list.

REMOVEQ —— *function*
(REMOVEQ **list item** [**sint**])
Delete one (or **sint**) occurrence(s) of **item** in **list**. Uses EQ and copies input list.

REMPROP —— *function*

(REMPROP **id item**)

Remove a specific item from a property list. See page 99 and page undefined.

RENAME —— *command to system-dependent interface*

(CALLBELOW "**RENAME**" **sysdep-area1 sysdep-area2**)

Rename a disk file. See page 296.

REPLACEF —— *command to system-dependent interface*

(CALLBELOW "**REPLACEF**" **sysdep-area1 sysdep-area2**)

Safe rename for disk files. See page 296.

RESETQ —— *macro*

(RESETQ **id [item]**)

Assigns to a variable, returns previous value. See page 79.

RESOLVEE —— *command to system-dependent interface*

(CALLBELOW "**RESOLVEE**" **sysdep-area1 sysdep-area2**)

Returns data for resolved file. See page 294.

RESTARTSD —— *variable*

Resume point on warm start, state descriptor.

RETURN —— *special form*

(RETURN **[item]**)

Exit from a function expression contour with value. See page 85.

REVERSE —— *function*

(REVERSE **list**)

Reverses order of a list, no updating. See page 114.

RIGHTSHIFT —— *function and compile-time macro*

(RIGHTSHIFT **num sint**)

Divide by a power of two. See page 107.

RKEYIDS —— *function*

(RKEYIDS **filearg**)

Identifiers matching LISPLIB keys. See page 283.

RNUMP —— *built-in function*

(RNUMP **item**)

Test for floatingness. See page 76.

RNUM2INT —— *function*

(RNUM2INT **num**)

Integer part of a floating point number. See page 101.

RPACKFILE —— *function*

(RPACKFILE **filespec**)

Garbage collect a LISPLIB file. See page 284.

RPLACA —— *built-in function*

(RPLACA **pair item**)

Update the first element of a pair. See page 111 and page 119.

RPLACD —— *built-in function*
(RPLACD **pair item**)
Update the second element of a pair. See page 111 and page 119.

RPLACSTR —— *function and compile-time macro*
(RPLACSTR **cvec1 sint1 sint2 cvec2** [**sint3** [**sint4**]])
Where any of the **sint** arguments may also be NIL.
Update part of a character vector (if there is room). See page 135.

RPLPAIR —— *function*
(RPLPAIR **pair1 pair2**)
Replace CAR+CDR of 1st arg by contents of 2nd. See page 111.

RPLQ —— *macro*
(RPLQ **item1 item2**)
Where **item2** is a pair/vector structure.
Updates components of structure from ID values. See page 145.

RREAD —— *function*
(RREAD **cvec rstrm**)
Read an item from a LISPLIB, by key. See page 283.

RREADLIST —— *function*
(RREADLIST **rstrm list1** {**sint** | **list2**})
Read from **rstrm** those items with keys in **list1** and/or class equal to **sint** or member of **list2**.

RSETCLASS —— *function*
(RSETCLASS **cvec sint** &rstrm)
Sets the class byte in LISPLIB item. See page 284.

RSHUT —— *function*
(RSHUT **rstrm**)
Close a LISPLIB stream. See page 284.

RVECP —— *built-in function*
(RVECP **item**)
Test for real vector type. See page 74.

RWRITE —— *function*
(RWRITE **cvec item rstrm**)
Write an item into a LISPLIB, by key. See page 284.

SASSOC —— *function*
(SASSOC **item list app-ob**)
ASSOC with function supplied for failure. See page 118.

SAVELISP —— *CMS interface function*
(SAVELISP **filearg**)
Save the workspace on disk and stay in LISP/VM.

SAVELISP-FILE —— *variable*
The default file parameters to be used by SAVELISP or FILELISP.

SBCP —— *function*

(SBCP **id**)
Test for existence of a shallow binding cell.

SBOUNDP —— function
(SBOUNDP **id**)
Tests for binding in stack part of environment.

SCANAND —— macro
(SCANAND **app-ob list** ...)
Apply function to successive list items until NIL. See page 93.

SCANOR —— macro
(SCANOR **app-ob list** ...)
Apply function to successive list items until nonNIL. See page 93.

SECD1 —— function
May not be invoked directly.
A dummy function marking an interpreter stack frame which inherits local variables.

SECD2 —— function
May not be invoked directly.
A dummy function marking an interpreter stack frame which does not inherit local variables.

SEESWHAT —— macro
(SEESWHAT **id** ...)
Return useful information about ids.

SEGLOC —— command to system-dependent interface
(CALLBELOW **"SEGLOC" sysdep-area**)
Returns address of a named shared segment. See page 296.

SEGMENTNAME —— function
(SEGMENTNAME)
Returns the file name of this segment image.

SEGTIME —— function
(SEGTIME)
Returns time and date of segment image.

SELECT —— macro
(SELECT **exp1 list** ... **exp2**)
Where each **list** is of the form (**exp exp** ...).
A case construct. See page 86.

SEQ —— special form
(SEQ **exp** ...)
Evaluate expressions sequentially, allow GOs. See page 81.

SET —— built-in function
(SET **id item**)
Assign value of one argument to value of other. See page 80.

SET-ECHO-PRINT —— function

(SET-ECHO-PRINT **boolean**)
Turn echo printing on/off in SUPV.

SET-GLOBAL-ID —— *function*
(SET-GLOBAL-ID **id item**)
Non-lambda environment assignment. See page 80.

SET-GVALUE —— *function*
(SET-GVALUE **id item**)
Stores **item** into the global value cell of **id**. See page 48.

SET-ID —— *function*
(SET-ID **id item**)
Sets fluid value of id in current stack. See page 80.

SET-LEX-ID —— *function*
(SET-LEX-ID **id item**).
Sets value of id, sees caller's lexicals. See page 80.

SET-MCASE —— *function*
(SET-MCASE **stream**)
Forces a console stream to mixed-case reading.

SET-READTABLE-FLAGS —— *function*
(SET-READTABLE-FLAGS flagname ... [**readtable**])
(SET-READTABLE-FLAGS flaglist [**readtable**])
Set one or more style flags in a readtable. See page 260.

SET-QUAL —— *function*
(SET-QUAL **stream id**)
Forces an input edit mode in a console stream.

SET-S —— *macro*
(SET-S (**id access-path instance**) **item**)
Update a field in a defined structure.

SET-STREAM-A-LIST —— *function*
(SET-STREAM-A-LIST **stream list**)
Update operator on fast streams. See page 281.

SET-STREAM-BUFFER —— *function*
(SET-STREAM-BUFFER **stream item**)
Update operator on fast streams. See page 281.

SET-STREAM-P-LIST —— *function*
(SET-STREAM-P-LIST **stream item**)
Update operator on fast streams. See page 281.

SET-UCASE —— *function*
(SET-UCASE **stream**)
Forces a console stream to upper-case reading.

SET-VALUE-PRINT —— *function*
(SET-VALUE-PRINT **boolean**)
Turn value printing on/off in SUPV.

SETANDFILEQ —— *macro*
(SETANDFILEQ **id item**)
Assign and add to LISPLIB.

SETSIZE —— *function and compile-time macro*
(SETSIZE **bORcvec sint**)
Change the components count of a character vector. See page 134.

SETDIFFERENCE —— *function*
(SETDIFFERENCE **list1 list2**)
Members of one list which are not in other. See page 116.

SETDIFFERENCEQ —— *function*
(SETDIFFERENCEQ **list1 list2**)
Same as SETDIFFERENCE, but uses EQ. See page 116.

SETELT —— *function and compile-time macro*
(SETELT {**vec** | **bORcvec** | **list**} **sint item**)
Update operator for vectors, indexed. See page 126, page 135 and page 119.

SETFUZZ —— *function*
(SETFUZZ **pair**)
Change the numeric comparison tolerance.

SETLINE —— *function*
(SETLINE **sint stream**)
Set record number in FILE stream.

SETQ —— *macro*
(SETQ **item1 item2**)
Assigns components of **item2** to corresponding (unevaluated) identifiers in **item1**. If **item1** is an identifier equivalent to the traditional SETQ. See page 79.

SETQP —— *macro*
(SETQP **item1 item2**) = boolean
Where **item1** is a pair/vector structure.
Assigns components of structure to IDs. Reports on success or failure. See page 142.

SFP —— *built-in function*
(SFP **item**)
Tests for special form. See page 75.

SHARED —— *command to system-dependent interface*
(CALLBELOW "**SHARED**")
Returns SHARED/NONSHARED status. See page 296.

SHAREDITEMS —— *function*
(SHAREDITEMS **item**)
Return vector of multireachable items in **item**. See page 145.

SHAREDP —— *function*
(SHAREDP)
Tests for SHARED/NONSHARED mode.

SHOW-CALLS —— macro

(SHOW-CALLS **exp**)
Print a file SHOW CALLS A containing a detailed call trace.

SHOW-S —— macro

(SHOW-S **id access-path instance**)
Display a structure instance with field names.

SHUT —— function

(SHUT **stream**)
Close a stream, non-op on console streams. See page 280.

SHUT —— command to system-dependent interface

(CALLBELOW "SHUT" **sysdep-area**)
Close a file. See page 296.

SIMPLEIDP —— built-in function

(SIMPLEIDP **item1**)
The predicate that test for identifier that is not a gensym, a special form, or a built in function.

SIN —— function

(SIN **num**)
Sine, argument in radians. See page 108.

SINTP —— built-in function

(SINTP **item**)
Test for small integer type. See page 75.

SIZE —— function

(SIZE **item**)
Number of elements in a list or vector. See page 124, page 131, and page 122.

SKIP —— function and compile-time macro

(SKIP **sint [stream]**)
Does n TERPRIs, may default stream. See page 254.

SORT —— function

(SORT **list**)
Quicksort on lists, uses SORTGREATERP. See page 122.

SORTBY —— function

(SORTBY **app-ob list**)
Sorts list of items with given access function. See page 122.

SORTGREATERP —— function

(SORTGREATERP **item1 item2**)
Initialized to GGREATERP, for SORT.

SOURCELIST —— definition option

(SOURCELIST . **boolean**)
Requests printing of the source expression on the listing stream. See page 70.

STACKLEFT —— function

(STACKLEFT)

Returns bytes of STACK space remaining.

STACKLIFO —— *variable*
Stream which feeds the console stack. See page 277.

STANDARD-READTABLE —— *function*
(STANDARD-READTABLE [IN | OUT | INOUT]) = **readtable**
Make a readtable that defines the normal syntax of LISP/VM. See page 259.

STARTTIME —— *function*
(STARTTIME)
Returns time/date of initial entry to LISP/VM.

STATE —— *built-in function*
(STATE [**a-list**])
Capture environment and control. See page 49.

STATEP —— *built-in function*
(STATEP **item**)
Test for state descriptor type. See page 76.

STORAGE —— *function*
(STORAGE **item**)
Number of bytes occupied by a single object. See page 132.

STRDEF —— *macro*
(STRDEF **structure-def**)
Equivalent to function expression, for structured definitions.

STREAM-A-LIST —— *function*
(STREAM-A-LIST **stream**)
Returns a-list component of a fast stream. See page 280.

STREAM-BUFFER —— *function*
(STREAM-BUFFER **stream**)
Returns buffer component of a fast stream. See page 280.

STREAM-BUFFER-DESCRIPTOR —— *function*
(STREAM-BUFFER-DESCRIPTOR **stream**)
Extracts the buffer descriptor, <buffer start current end>, from **stream**.

STREAM-DESCRIPTOR —— *function*
(STREAM-DESCRIPTOR **stream**)
Returns descriptor component of a fast stream. See page 281.

STREAM-P-LIST —— *function*
(STREAM-P-LIST **stream**)
Returns the system control block of a fast stream. See page 281.

STREAMP —— *function*
(STREAMP **item**)
Heuristic is-this-a-stream? test, = PAIRP. See page 76.

STRING2ID-N —— *function*
(STRING2ID-N **cvec sint**)

Converts the nth token in a string to an id. See page 132.

STRING2PINT-N —— *function*
(STRING2PINT-N **cvec sint**)
Converts the nth token in a string to a number. See page 132.

STRPOS —— *function*
(STRPOS **cvec1 cvec2 sint item**)
Searches string for sub-string. See page 133.

STRPOSL —— *function*
(STRPOSL **table cvec sint item**)
Searches string for specified character(s). See page 133.

STRTRT —— *function*
(STRTRT **table cvec pair**)
Locates and identifies characters in string. See page 133.

SUBLOAD —— *function*
(SUBLOAD **filespec {id | list}**)
Load a subset of a LISPLIB. See page 285.

SUBST —— *function*
(SUBST **item1 item2 item3**)
Create a structure with item substitutions, uses EQUAL. See page 141

SUBSTQ —— *function*
(SUBSTQ **item1 item2 item3**)
Create a structure with item substitutions, uses EQ. See page 141

SUBSTRING —— *function*
(SUBSTRING **cvec sint1 sint2**)
Create a new string from part of another. See page 132.

SUB1 —— *function*
(SUB1 **num**)
Subtracts one from its argument. See page 105.

SUFFIX —— *function and compile-time macro*
(SUFFIX **id cvec**)
Adds a character to a character vector, lengthening it. See page 134.

SUPERMAN —— *function*
(SUPERMAN)
The next-to-root function, binds error streams.

SUPV —— *function*
(SUPV **stream1 stream2**)
System READ/EVAL/PRINT loop.

SUPV-PRINT —— *function*
Print function used by SUPV. Initially bound to the value of PRETTYPRINT.

SUPV-READ —— *function*
(SUPV-READ **stream**)

The function called by the top-level supervisor, SUPV, to obtain input.

SUPV-SYNTAX —— *variable*
Value is the name of the syntax used by normal read loops. See page 263.

SVC202 —— *command to system-dependent interface*
(CALLBELOW "**SVC202**" **sysdep-area**)
Issues SVC 202 for arbitrary property list. See page 298.

SYM-BREAK-TABLE —— *variable*
Value is table used by Reader and Printer. See page 264.

SYSID —— *function*
(SYSID)
Returns the cpu id for the host system.

SYSID —— *command to system-dependent interface*
(CALLBELOW "**SYSID**")
Returns 1, meaning CMS. See page 298.

SYSKEY —— *function*
(SYSKEY)
Returns machine/time stamp for this work space.

TAB —— *function and compile-time macro*
(TAB **sint** [**stream**])
Sets output position, may default stream. See page 255.

TAILP —— *function*
(TAILP **item list**)
Test 1st arg for EQ to (C[D...]R 2nd arg). See page 117.

TEMPDEFINE —— *function*
(TEMPDEFINE **item** [**list**])
Where **item** is either a name/expression list or a list of name/expression lists.
Adds augmented STATEs to OPTIONLIST.

TEMPSETQ —— *macro*
(TEMPSETQ **id item** [**list**])
Performs both an ORTEMPSETQ and a MATEMPSETQ.

TEMPUS —— *command to system-dependent interface*
(CALLBELOW "**TEMPUS**" **sysdep-area**)
Returns time, date and CPU usage. See page 297.

TEMPUS-FUGIT —— *function*
(TEMPUS-FUGIT)
Returns elapsed virtual time.

TEREAD —— *function and compile-time macro*
(TEREAD [**stream**])
Discards remainder of line, may default stream. See page 249.

TERPAGE —— *function*
(TERPAGE **sint** [**stream**])

Advance the stream to a multiple of page length.

TERPRI —— *function and compile-time macro*
(TERPRI **[stream]**)
Forces output of line, may default stream. See page 254.

TEST-S —— *macro*
(TEST-S **id access-path instance**)
Test if a defined field is present in an instance of the structure.

THROW —— *function and compile-time macro*
(THROW {**id** | **sint**} **item**)
Return control to a CATCH, evaluating EXITs. See page 87.

THROW-PROTECT —— *macro*
(THROW-PROTECT **exp1 exp2**)
Evaluates an expression, despite THROW/UNWIND. See page 88.

TIMES —— *function and compile-time macro*
(TIMES **num** ...)
Computes product of a set of numbers. See page 103.

TIME2NUM —— *function*
(TIME2NUM **cvec**)
Where **cvec** is a time/date, as returned from CURRENTTIME, e.g., encode time-date string as small integer.

TINLL —— *command to system-dependent interface*
(CALLBELOW **"TINLL"**)
Returns console input line length. See page 292.

TOULL —— *command to system-dependent interface*
(CALLBELOW **"TOULL"**)
Return console output line length. See page 292.

TPLINE —— *command to system-dependent interface*
(CALLBELOW **"TPLINE" sysdep-area1 sysdep-area2**)
Display line on console. See page 293.

TRACE —— *macro*
(TRACE **exp id** ...)
Evaluate **exp** while MONITORing id .... See page 237.

TRANSLIST —— *definition option*
(TRANSLIST . **boolean**)
Requests printing of the macro expanded code. See page 70.

TRIMSTRING —— *function*
(TRIMSTRING **cvec**)
Forces the capacity of a string to minimum. See page 134.

TRUEFN —— *function and compile-time macro*
(TRUEFN)
(LAMBDA () 'T). See page 96.

TT —— *macro*

(TT **exp**)
Evaluate **exp** and return the total and virtual cpu time used.

TYPEOF —— *function and compile-time macro*

(TYPEOF **item**)
Returns the type-byte of the pointer **item** as a small integer.

UASSOC —— *function*

(UASSOC **item list**)
Search a list of name-value pairs, uses UEQUAL. See page 118.

UEQUAL —— *function*

(UEQUAL **item1 item2**)
Tests for update equivalence. See page 78.

UGEQUAL —— *function*

(UGEQUAL **item1 item2**)
Test for update equivalence and unify gensyms.

UMEMBER —— *function*

(UMEMBER **item list**)
Searches list, using UEQUAL. See page 117.

UNEMBED —— *function*

(UNEMBED **id**)
Undoes EMBED. See page 239.

UNFREEZE-SHARED-SEGMENT —— *function*

(UNFREEZE-SHARED-SEGMENT)
Reestablishes access to the shared segment.

UNION —— *function*

(UNION **list1 list2**)
Set union of two lists. See page 115.

UNIONQ —— *function*

(UNIONQ **list1 list2**)
Set union of two lists, uses EQ. See page 115.

UNSHRSEG —— *command to system-dependent interface*

(CALLBELOW **"UNSHRSEG" sysdep-area**)
Force named shared segment to non-shared mode. See page 297.

UNWIND —— *function*

(UNWIND [**sint** [**item**]])
THROWs to the nth ERRSET. See page 88.

UPCASE —— *function*

(UPCASE {**cvec** | **id** | **list**})
Returns character vector, id, or list in upper case. See page 136 and page 97.

USEREXT —— *command to system-dependent interface*

(CALLBELOW **"USEREXT" cvec** [{**sint** | **sysdep-area**}*])
Escape to locally implemented SYSDEP commands. See page 298.

VAPPLY —— *function*

(VAPPLY pred0 **app-ob** [pred1. **item1**] ...)

Verify arguments, apply **app-ob** and then verify value. See page 58.

VASSQ —— *function*

(VASSQ **item** &vec.)

Searches &vec. for a pair whose CAR is EQ to **item**. Returns the pair, if found, NIL otherwise.

VECTOR —— *function and compile-time macro*

(VECTOR [**item** ...])

Makes a vector of its arguments. See page 123.

VECP —— *built-in function*

(VECP **item**)

Test for vector capable of holding arbitrary objects . See page 73.

VECTOROFSAME —— *function*

(VECTOROFSAME **vec**)

Tests for identical types on vector members.

VEC2LIST —— *function*

(VEC2LIST **vec**)

Makes a list of the contents of a vector. See page 115, page 126 and page 133.

VERSION —— *function*

(VERSION)

Returns character vector with version id, creation date.

VGREATERP —— *function*

(VGREATERP **vec1 vec2**)

Subroutine of GGREATERP, orders vectors.

VMEMQ —— *function*

(VMEMQ **item vec**)

Returns index of EQ element of a vector. See page 126.

VSCANAND —— *macro*

(VSCANAND **app-ob** {**vec** | **list**} ...)

Like SCANAND, for vectors. See page 94.

VSCANOR —— *macro*

(VSCANOR **app-ob** {**vec** | **list**} ...)

Like SCANOR, for vectors. See page 94.

WHOCALLED —— *function*

(WHOCALLED **sint**)

"Name" of **sint**th bpi up the control chain.

WHOSEES —— *macro*

(WHOSEES **id** ...)

Return a list of functions that call or use the **id**s free or quoted.

WRAP —— *function*

(WRAP **list item**)

Where item2 is either the "x" or a list of "x"s.
"Wraps" list items e.g. (a b) -> ((x a) (x b)). See page 95.

WRBLK —— *command to system-dependent interface*
(CALLBELOW **"WRBLK" sysdep-area1 sint1 sint2 sint3 sysdep-area2)**
Write blocked records to a file. See page 295.

WRITE-LINE —— *function*
(WRITE-LINE **cvec stream)**
Writes a line on an output stream. See page 254.

WSTIME —— *function*
(WSTIME)
Returns time and date of work space creation.

XORBIT —— *function*
(XORBIT **bvec** ...)
Exclusive OR of bit vectors. See page 131.

XORCAR —— *function and compile-time macro*
(XORCAR **item)**
If arg is pair, CAR, otherwise arg. See page 110.

XORCDR —— *function and compile-time macro*
(XORCDR **item)**
If arg is pair, CDR, otherwise arg. See page 110.

ZEROP —— *function*
(ZEROP **item)**
Tests for zero value. See page 101.

32XXPLST —— *command to system-dependent interface*
(CALLBELOW **"32XXPLST" sint sysdep-area)**
Returns information about a virtual device. See page 293.

32XXWRIT —— *command to system-dependent interface*
(CALLBELOW **"32XXWRIT" sint1 sysdep-area sint2 sint3)**
Performs the CP operation DIAG 58 output to console. See page 294.

GLOSSARY

&

1. External characters used in an elided expression.
2. Keyword used to denote focus in certain commands.

( ) External characters used as expression and pair delimiters.

< > External characters used as vector delimiters.

" External character used as string delimiter and last expression repetition.

|

1. External character used as an escape character.
2. Metasymbol representing alternatives.

% External character used to delimit special syntactic constructs. The dispatching character.

, Keyword used to separate fields in a structured definition.

; Keyword used to separate fields in a structured definition.

. External character used as the dot of a dotted pair.

...

1. External characters used in an elided expression
2. Metasymbols used to denote zero or more instances of the preceding object.

: External character used as delimiter separator in bit strings.

' External character used to specify an unevaluated expression.

= Keyword used to represent a default value and to denote sharing in bound variable lists of function and macro expressions.

$ External character used to identify a line as a command or expression.

? External character used in a pattern and to recall previous commands.

* Keyword used for indefinite repetition or undefined.

+ - External characters used for numbers.

{} Metasymbols used for grouping.

a-list: An association list. A list of pairs used to define a mapping from a collection of keys to a collection of values. A property list is an a-list connected with an identifier. Indexed files are collections of property lists.

access function: A function which returns a component of a composite object.

actual parameter: An argument.

applicable object: Function, state descriptor, or funarg.

application: A stage of evaluation during which previously evaluated arguments are passed to a previously evaluated object (e.g., a function) which then performs its operation.

argument: An object which is associated with a corresponding bound variable during evaluation of an expression.

assemble action: To transform LAP code into object code.

assembly: The third phase of compilation. Object code is generated during this phase.

association list: A-list.

associative cell: A key-value pair of an a-list.

atom: Any object which is not a pair; this includes NIL, identifiers, numbers, bpis, funargs, state descriptors, vectors, hashtables, readtables, and %.EOF.

back-trace: Print out of the stack.

bind: To associate a value with a variable. There are two types of binding: lambda binding and non-lambda binding.

binding:

1. An association of a value with a variable.
2. The value associated with a variable.

body: The part of the function or macro definition which specifies what operations are to take place when the function is invoked.

bound variable: A variable named explicitly in the bound-variable list of the definition of a function or macro. When the function or macro is invoked, the variable is associated with a corresponding argument. A bound variable is lexically scoped within the definition unless specifically dynamically scoped.

bpi: An object which results from the compilation of a lambda expression. A bpi includes an fbpi and an mbpi.

break loop: A READ-EVAL-PRINT loop activated by an error signal.

bucket: The collection of keys which have been hashed to the same hashcode.

built-in function: A function-like operator, or the expression containing this identifier as the expression operator, which is recognized directly by LISP/VM.

catch: To receive control at a catch point.

catch point: A distinguished point in the control part of the environment to which control can return via a THROW or UNWIND.

circular list: List which includes itself or part of itself as a list element or tail.

closure: The combination of an expression and an environment.

command: Input to Editor which allows a more general form of expressions.

command mode: Status of the editor when awaiting a command.

command token: The leading part of a command which identifies the purpose of the command.

compilation: The process of conversion of an expression into an equivalent form called a bpi which evaluates more efficiently. Because the compiler freezes information during compilation, subsequent evaluation can, in some subtle cases, produce different results from direct evaluation of the original form.

compile action: To transform a reduced expression into LAP code.

composite object: An object containing accessible parts.

conformal: Of the same shape.

constant: A data object that evaluates to itself; a number, state descriptor, vector, hashtable, readtable, %.EOF. or NIL.

continuous evaluation: Process of evaluating an expression without any explicit control by the programmer.

control: Status of evaluation with respect to nesting of expressions.

control chain: The chain of invocations that led to the current control.

contour: A boundary created by the evaluation of one of a set of expressions which define limits for transfer of control. Contours may contain bound variables and labels. These expressions include EVAL, APPLY, MDEF, function and macro expressions and the expressions which generate function and macro expressions (PROG, SELECT, DO, CATCH, CATCHALL, THROW-PROTECT, ERRSET, NAMEDERRSET, ERRCATCH, MAPC, MAP, MAPCAR, MAPLIST, MAPCAN, MAPCON, NMAPCAR, SCANOR, SCANAND, MAPE, MAPELT, NMAPELT, VSCANOR, VSCANAND, DISABLED, LAM, iteration operators, and any operators created by the user which generate function and macro expressions).

conversion: The transformation of a form of one type into a form of another type.

current length: The number of characters in the existing vector object.

data object: The typed internal form of an expression.

data type: One of a set of classes. Each data object in the system is a member of one such class. These include: identifier, number, bpi, funarg, state descriptor, pair, list, stream, vector, hashtable, readtable, and the miscellaneous types.

define action: To transform any expression by expanding an outermost LAM, if present.

definition: The association of a possibly transformed expression with a variable.

delimiter: A character which is part of the external form and which is used to to mark the beginning or end of some syntactically meaningful unit.

directory: Indexed file.

dispatching character: %. It reads an optional digit string and then one more character and uses that character to select a function to run as a macro-character function.

display information: Name information retained in a bpi after compilation.

disposition property: One of the properties in an indexed file which controls what happens to an indexed file object when it is loaded.

disposition function: A function that is an acceptable value for a disposition property.

dotted pair: A pair.

edit environment: Data and status information associated with the editing of a single object.

edit session: All edit environments which occur between the commands LISPEDIT and EXIT.

element: A component of a pair, list, or vector.

elided expression: An external form in which some details are replaced by ellipsis marks (& and ...).

environment: The combination of the current local environment, previous local environments, and non-lambda environment.

environment of compilation: The operator recognition environment and the macro-application environment.

environment of evaluation: The current environment.

error: Any condition which causes an error signal to be generated.

error signal: The invocation of CONDERR. If an error is signaled during evaluation, an error break loop may take effect under programmer control.

escape character: A character which is part of the external form and denotes that the character which follows should be interpreted as itself and included as part of the internal form.

evaluation: The process of computing a value corresponding to an expression. If the expression is a constant, evaluation terminates immediately with the return of that value. Otherwise, the expression is a list where the CAR determines the process of further evaluation. A macro is macro-applied and the evaluation process repeated. In the case of a function, the arguments are evaluated, the function is applied and the evaluation process terminates.

evaluation pointer: The current control within an expression being interactively evaluated. When displayed, the evaluation pointer is highlighted.

expression:

1. An arbitrary data object.
2. A data object which can be evaluated, usually part of a function.

external characters: Characters which are part of the external form but not the internal form. These include: () <> , ' " | % and &.

external form: A representation of a data object which is produced by by LISP/VM to be read by the programmer, or written by the program to be read by LISP/VM. Some external forms offer a guarantee of read/print equivalence.

fast stream: A stream that has a buffer to speed up input/output.

fbpi: A compiled function expression.

field definition: A component of a structured definition.

fluid binding: An association of a bound variable, which has been specified as FLUID, with its value. An identifier may be resolved to a fluid binding which occurs past a contour in an environment.

focus: The current component of the top expression. It appears as an implicit argument to many edit commands. When displayed, the focus is highlighted.

formal parameter: Bound variable.

frame: See stack frame.

free variable: An occurance of a variable name whose binding does not occur within the immediately enclosing function definition.

funarg: A data object that represents a closure. The type of expression in the funarg characterizes the funarg (for example, function funarg and number funarg).

function: An applicable object that does not contain environment information. It includes the built-in functions, function expressions, and fbpis.

function expression: A list which contains LAMBDA as its CAR at the time when the interpreter recognizes the operator.

fuzz: The approximation implicit in floating point operations.

garbage collector: Automatic process which identifies the memory cells whose contents are no longer useful for the computation in progress and then makes them available for some other use.

gensym: Identifier which is generated by the system and is unique within a given input operation (e.g., an invocation of the function READ). In contrast to a stored identifier, a gensym cannot have a property list or a non-lambda/fluid binding.

hashtable: A collection of buckets containing key-value pairs.

head: The first element of a list.

heap: The default location where stored objects are kept. The heap is periodically garbage collected.

Heval: Iterative debugging interpreter.

identifier: A data object which consists of a name, a value, and possibly an a-list, called a property list.

identifier resolution: The process of finding a binding for a particular instance of an identifier.

immediate property: A property of an indexed file which, in contrast to the pushdown property, keeps no backup of values.

indexed file: A CMS file which contains a collection of named property lists, called members.

interactive evaluation: Process of evaluating an expression with the capability of pausing during evaluation so that the programmer can determine how the evaluation is progressing.

interactive system: A programming system which allows immediate response to commands.

internal form: The form in which a data object is maintained within the system.

interned identifier: A stored identifier which has the relation between its internal and external form maintained in the obarray.

interpretation: In general, the process of evaluation but most frequently used to indicate evaluation of a form which has not been previously compiled.

invocable object: An object which receives arguments and yields a value. It includes applicable objects and macro-applicable objects.

invocation: A process of application for applicable objects or macro-application for macros.

iteration: Repeated evaluation of a sequence of expressions.

jaunt expression: An expression whose operator is a state descriptor. It causes resumption of a previously saved environment.

key: One of the components of an associative cell.

key-value pair: An associative cell.

key word: An identifier whose significance lies in its name, not its value. or as a marker in system-related objects.

label: An identifier which identifies a point to which control can transfer through a GO.

label-scope: A boundary created by the evaluation of one of a set of expressions which define limits for the definition of labels. These expressions include SEQ and PROG.

lambda binding: A binding which takes place when a lambda expression is applied.

lambda expression: function expression, macro expression, or bpi.

large integer: A number which has type BNUM and a range which includes all values not in small integers.

lexical binding: The default form of lambda binding (as opposed to fluid binding). Identifier resolution rules do not search for lexical bindings beyond the resolution boundary.

lexically present function expression: An expression which is recognized as a function expression after exactly one evaluation of its operator and which is itself not the result of any evaluation. Macro applications are not counted in this process.

lexically present macro expression: An expression which is recognized as a macro expression after exactly one evaluation of its operator and which is itself not the result of any evaluation. Macro applications are not counted in this process.

library: A key-addressed CMS file containing arbitrary objects.

list: Sequences of expressions which is an alternative view of a pair. A list has a more convenient notation than a pair.

local variable: A bound variable in a local environment.

local environment: The set of all lambda bindings within a function or macro.

macro: A macro expression, special form or mbpi.

macro-applicable object: An invocable object (a macro or a macro-funarg) which is used by the programmer to extend the syntax.

macro application: Process of passing the argument to a macro and of returning the value of the macro. Macro application takes place during evaluation when the result of macro application is normally evaluated again. It also takes place during the first stage of compilation.

macro application environment: An environment in which the compiler performs all macro applications. It is used to add temporary definitions or redefine existing definitions of macros and functions invoked during the macro applications which are triggered by the actual compilations. This serves to isolate the compiler from the compiled expression. This environment is normally empty.

macro closure: The combination of a macro expression and an environment.

macro expansion: Effect of macro application.

macro expression: A list which contains MLAMBDA as its CAR at the time when the interpreter recognizes the operator.

mapping operator: An operator that applies another operator to successive components of a data object.

mbpi: A compiled macro expression.

member: A single property list in an indexed file.

message: Output from the system.

metasymbol: Symbol which is not part of LISP/VM but which is used to describe LISP/VM syntax.

miscellaneous types: NIL and %.EOF.

NIL: A constant usually used to denote the empty list or the boolean value 'false'. NIL may also be written as ().

non-checking: Not performing type checking. This is done as an optimization.

non-lambda binding: The value associated with a variable which is not a bound variable.

non-lambda environment: The part of an environment which contains non-lambda bindings. There can be several non-lambda environments.

non-NIL: A term used to denote the boolean value 'true'. Any object which is not NIL.

non-stored identifier: An identifier which is not stored in the obarray, a gensym.

null list: Empty list or NIL.

number: Floating point, small and large integers.

obarray: The master index used to relate internal to external forms for stored, interned identifiers.

object: The internal form of all data.

object array: Obarray.

object code: Code which is executed directly on the IBM/370.

operand: An element of the CDR of a list in the context of evaluation.

operator: The CAR of a list in the context of evaluation. If the operator does not evaluate to a function or macro, an error is signalled.

operator recognition environment: An environment used by the compiler which adds and replaces bindings from the non-lambda environment. It is used to selectively redefine operators during compilation, for reasons of efficiency and to achieve macros local to a file. Macros are recognized here and applied in the macro application environment.

OR-environment: Operator recognition environment.

order of evaluation: The sequence in which parts of a composite expression are evaluated.

pair: A composite object having two components, a CAR and a CDR. The pair type includes the alternate views of pairs, lists and streams.

predicate: A function usually used for testing whose result is either NIL or non-NIL. Wherever possible, the non-NIL value returned is one that could be of use.

prettyprinting: Formatted printing.

previous local environment: A local environment which was created during an invocation of an applicative object which either invoked the currently evaluating object or an object whose evaluation eventually caused the invocation of the currently evaluating object.

Printer: The functions that map the internal form of data objects to their external forms.

program: an expression or collection of expressions which perform a specific task.

property: A property-name and value pair of a property list.

property list: An a-list that is available with each identifier, or as a member of an indexed file. The property list is composed of properties.

property name: One component of a property.

pushdown property: A property of indexed files which allows values to be backed up until explicitly discarded.

quote: To not evaluate.

Reader: The functions that map the external form of data objects to their internal forms.

real number: A number stored using System/370 double precision floating point format, yielding 53 to 56 bits of precision for the mantissa and a range of approximately 10^{74}.

realize action: To transform a defined expression which may have a macro invocation as the value of the operator, to one in which all such outer macro invocations have been macro-applied and the resultant realized expression is a function or macro expression.

recursive:

1. Self-referential.
2. See recursive.

reduce action: To transform a realized expression, by expanding all macros, into a reduced expression.

resolution: Identifier resolution.

resolution boundary: A boundary which appears within an environment to limit identifier resolution.

s-expression: A symbolic expression. Expression.

shallow binding cells: An optimization used to decrease the time taken to find the value of a variable.

shared substructure: The parts of an object which occupy the same heap storage.

small integer: A number which has type SNUM and a range -2^{26} to $2^{26}-1$ (-67,108,864 to 67,108,863).

special form: A macro-like operator, or the expression containing this operator as the expression operator, which is recognized directly.

stack: A sequence of stack frames in temporal order of creation.

stack frame: An object which contains the local environment and the current control.

state: The environment, control chain, and current control of an invocation.

state descriptor: A data object which captures an environment.

stored identifier: An identifier that can have a property list and any kind of binding.

stream: A data object which includes system information as well as input data. It always has a direction, either input or output. A stream is an alternate view of a pair.

string: A character or bit vector.

strong hashtable: A hashtable containing key-value pairs, each of which must be explicitly deleted.

structural equality: Equality of objects with regard to shape and value.

structure: See structured definition.

structured definition: A collection of access functions for data objects.

system command: A key word used by the system dependent interface.

tail: The result of applying one or more CDRs to a list.

throw: To transfer control to a catch point.

top expression: The entire data object under consideration by an edit session.

transformed expression: An expression on which the define, realize, reduce, compile, or assemble actions have been performed.

trapped function: A function which is modified to allow interactive evaluation.

traversal: Shifting the focus within the top expression.

truncated expression: An external form that does not show a complete representation of an internal object. The external form has been cut short at an arbitrary point.

type: Data type.

type checking: Verification that the type of an argument is consistent with that expected by the operator.

uninterned identifier: A stored identifier which is not in the obarray.

unspecified value: The consistency of the value returned by evaluation of an expression is not guaranteed.

value: Result of evaluating an expression. In the case of an identifier, the value is the value of the binding.

updating: Replacing a component of an object.

variable: An identifier in the context of evaluation.

vector: A one-dimensional collection of expressions with efficient indexed access and a compact storage representation.

vector capacity: Maximum size of a vector.

weak hashtable: A hashtable containing key-value pairs, each of which may be deleted when its key is no longer referenced from outside the hashtable.

workspace: Status of the system at a point in time. The workspace can be saved. The workspace captures the edit session.

Index

SH20-6477-0

LISP/VM User's Guide Printed in U.S.A. SH20-6477-0

IBM®

LISP/VM 5798-DQZ
User's Guide
SH20-6477

READER'S COMMENT FORM

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

____

____

____

____

What is your occupation? ______________________________

Number of latest Newsletter associated with this publication: ____________________

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

SH20-6477-0

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 68Y
P.O. Box 2750
Irving, Texas 75061

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

Cut or Fold Along Line

LISP/VM User's Guide Printed in U.S.A. SH20-6477-0