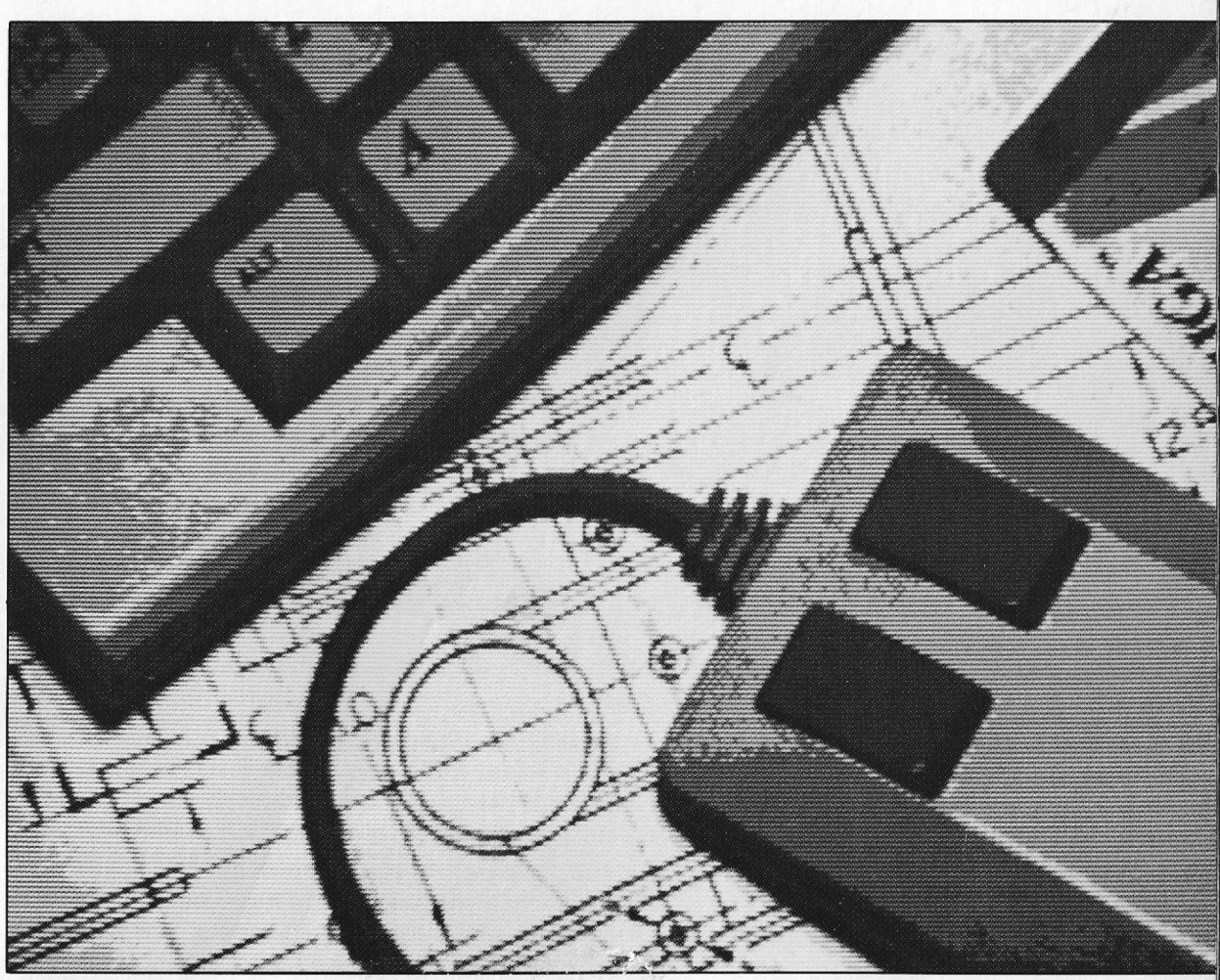
AMIGA™



Cambridge Lisp 68000 Manual

The Cambridge LISP 68000 System for the Amiga

* School of Mathematics, University of Bath, England
** The Computer Laboratory, University of Cambridge, England

This manual refers to Release 1, August 1985.

Printed in the U.S.A.

CBM Product Number 327269-01 rev 1.0 8.27.85

Cambridge LISP 68000

Table of Contents

Chapter 1: Introduction to Cambridge LISP 68000

1.1 General

This manual describes the main features of the Cambridge LISP 68000 system, and comments on the facilities provided.

Lisp exists in many dialects of which there are four major "families". The best known are InterLisp, Common Lisp, Maclisp, and Standard Lisp. To a large extent this system falls somewhere between the Common and Standard Lisp families. Standard Lisp dates back to 1966 when A C Hearn (University of Utah and Rand Corporation), wrote out a minimal list of functions. This standard basic set ensured that implementations could be made portable. By 1978 (see SIGPLAN Notices Marti J.B. *et al.* vol. 14, no. 10, p. 48-68) Standard Lisp had been upgraded but still remained a kernel-level Lisp. Subsequently it was extended and further work carried out on portability to produce an implementation known as PSL (Portable Standard Lisp). Both Cambridge LISP and PSL agree largely with the 1978 report and even share some code.

Cambridge LISP was originally developed for the IBM mainframe range of computers to provide support for a research project in computer algebra. The 68000 implementation forms a reliable base for functional style programming on this microprocessor. Cambridge LISP 68000 is intended for program development, and so it makes a policy of checking for exceptional cases (for example, car or cdr of atoms) and provides clear and concise diagnostics. A compiler is included in the package which can make dramatic improvements in execution speed. Function tracing is provided in both interpreted and compiled code. The arithmetic package puts consistency above efficiency: integers can grow to be any size; the normal arithmetic primitives know about rational numbers; and there is a well-defined interface between exact and floating point number representations. The Cambridge LISP system provides a number of character handling facilities, can select and use several input/output streams, and has a built-in LISP prettyprinter.

Cambridge LISP is a large system that provides a great deal of support for the LISP programmer; it does not attempt to use the minimal amount of store and it requires at least 512K bytes.

Note that Cambridge LISP does not support functional closures or environments (that is, the FUNARG device of classical LISP).

1.2 Definition of Conventions and Terms

Conventions:

In order to distinguish the names of LISP functions from the rest of the text, all functions appear in bold typeface - **like this**. However, function names found in examples appear exactly as they are used and in the same typeface as the rest of the example. Keywords appear in upper case, although you can type them in either upper or lower case. In some examples the name 'foo' appears, this is a nonsense name that can be used for any name. You can use this standard non-name yourself when trying things out.

The following standard abbreviations are used to define syntax:

arg	argument.
fg	flag - replaced by flags such as **t**, **f** or **nil**.
fn	function.
fp	floating point number.
lev	level - replaced by a number indicating expected level of response.
n	number - usually an integer.
var	variable.
< >	used to surround text describing the type of thing to be inserted rather than an actual name. That is, (print <function name>) where <function name> should be replaced by the actual name of the function to be printed.
[ ]	used to surround anything that is optional.

Terms:

alist	an association list or list with each member being a dotted pair. That is, ((a . b) (c . d)...).
arguments	things a function has to work with. (See section 4.2 for a full list of argument types).
atom	the smallest data object that can be manipulated in LISP. It can be a character, any type of number, a string, identifier or compiled function. Vectors are also called atoms.
bound variable	an atom that appears in a function's argument list.

circular list	a list that contains a pointer back to itself.
dotted pair	the fundamental non-atomic data object in LISP. A dotted pair has two components, called **car** and **cdr**. The dotted pair whose **car** is 'a' and whose **cdr** is 'b' is written (a . b).
dump	that directory to which the function **preserve** writes the LISP "world".
filenames	a name up to 255 characters the format of which is system dependent.
floating point	used for representing real numbers (for example, .3333333 or 2.0).
function	a named procedure that may be defined and called. It takes arguments as input and gives a value as output.
garbage collection	returning old used cells to free storage.
identifier	a named atom, which can have a value and various properties.
image	that directory from which the LISP "world" was loaded.
integer	fixed point number.
lambda	marker atom used in anonymous functions. **lambda** identifies a piece of LISP structure as representing a function.
lambdaq	behaves like **lambda**, but identifies functions whose arguments are not to be evaluated. **lambdaq** takes the place of the **fexpr/fsubr** activity found in other systems.
list	groups of atoms surrounded by matching brackets. You can also group lists together to form further lists. Thus (a), (a b c), and ((a) (a b c)) are all lists. Lists are, in fact, dotted pairs (a) being short-hand for (a . nil), and (a b c) for (a . (b . (c . nil))).
macro	a function whose output is automatically re-evaluated by the system.
number	any type of number, integer, floating point, or rational.
quotes (')	the quote symbol ' before an s-expression prevents its evaluation. This is equivalent to **quote**. A q at the end of certain (but not all) functions implies a quote symbol, for example, **setq** for **set**'.
rational	number expressed in the form 1/3, or 6/7.
real	represented by floating point number.
s-expression	atoms and lists collectively form s-expressions (or symbolic expressions).
strings	characters and spaces enclosed in " " and treated as single atoms.

unbound **variable**
or free **variable** — an identifier that is used in a function, but does not appear in the function's argument list.

vectors — ways of grouping s-expressions that are superficially similar to lists. Lists can have variable numbers of elements and new ones can be added or deleted. A vector always has a fixed number of elements, which can be extracted or replaced. Finding, for instance, the 10th element of a vector is much more efficient than the 10th element of a list. You normally use vectors to improve the efficiency of parts of programs.

1.3 Loading LISP

LISP, unlike other languages such as FORTRAN and Pascal, is essentially an interactive language. Whatever LISP is given it attempts to evaluate and obey. This means that, for most purposes, you enter and use LISP directly taking input from, and returning a value to, the terminal. For more complex use, you can enter a program from a file, and return the results to another file, while you are still in LISP.

To run LISP in a simple fashion, type

```
lisp
```

This command loads the LISP system after displaying a short message. After you have loaded LISP, another message appears giving an indication of the amount of store in use. Then an initial core image is loaded from the directory *image*, or if that is not found, from sys:l/lisp/image. Once loaded, LISP accepts instructions and evaluates them.

To finish using LISP, type

```
(stop)
```

or

```
fin
```

after which a short message appears, the normal prompt returns, and control goes back to the operating system command level.

LISP always attempts to take as much space as possible on loading. This means that there is little or no space at all left for anything else - such as opening files or running another program. To allow for this, LISP takes two keywords on loading called LEAVE and STORE. LEAVE specifies how much store should be left after loading; STORE specifies how much store LISP may run in.

```
lisp LEAVE 100K
```

means leave 100K for everything else to run in once LISP is loaded.

```
lisp STORE 400K
```

means run LISP in 400K. Note that the 'K' is not essential. It is acceptable to use the following form:

```
lisp STORE 400000
```

The keywords FROM and TO govern input and output. Initially, FROM and TO are set to the terminal (that is, the default input and output). TO directs output to a file; FROM takes input from a file.

```
lisp FROM <name>
```

```
lisp TO <name>
```

You use the keywords DUMP and IMAGE to name the directory where LISP is to write the present LISP "world," and the one containing the new LISP "world" that LISP is to load. That is to say, DUMP

directs the core image and fast load modules generated in the run to a directory by using **module** and **preserve**. IMAGE takes the name of a directory containing the initial core image and fast load modules. The defaults are the image directory and image. To dump and load your lisp world, you type

```
lisp DUMP <name>

lisp IMAGE <name>
```

Note: Cambridge LISP only accepts names up to 255 characters in length. If LISP receives filenames longer than 255 characters, it truncates them.

A full list of the keywords that the command 'lisp' accepts, including all the options, appears in Appendix B.

1.4 Example Session

This section provides a listing of a very simple Cambridge LISP session. It shows how LISP attempts to evaluate everything it receives. The session also demonstrates the basic concepts of list processing as described in this chapter.

```
 > lisp

Cambridge LISP 68000 entered in about 380 Kbytes
Store image was made at 18:15:35 on 23-Apr-85
Lisp version - Ver X     image size = 79856 bytes

Started at 15:10:41 on 10-Jul-85 after 27.00
                                  - 55.2% store used
Input: 56
Value: 56

Input: "A string"
Value: "A string"

Input: oranges

error T43: Unset variable   oranges
style 5 backtrace follows:

end of backtrace

Input: (set 'fruit '(oranges lemons apples))
Value: (oranges lemons apples)

Input: fruit
Value: (oranges lemons apples)

Input: (car fruit)
Value: oranges

Input: (cdr fruit)
Value: (lemons apples)

Input: (plus 1 3)
Value: 4

Input: (set 'a 15)
Value: 15

Input: (plus a 5)
Value: 20
```

```
Input: (plus a fruit)

error T1: Bad argument for plus (oranges lemons apples)
style 5 backtrace follows:
plus (being interpreted)

end of backtrace

Input: (set 'flavors '(orange apple
                       "passion fruit" 123))
Value: (orange apple "passion fruit" 123)

Input: flavors
Value: (orange apple "passion fruit" 123)

Input: (stop)
```

The LISP session starts with the simple command:

```
>lisp
```

which invokes a short message from the system followed by the prompt:

```
Input:
```

After the atom 56 was entered, LISP returned the corresponding value:

```
Value: 56
```

The string "A string" also returns the same value as itself. However, the third example returns an error because the input 'oranges' was treated as an unset variable. The error then gave rise to a code number, message, and backtrace.

The fourth input shows how you can set a variable to contain a value - in this case a list. Simply typing the name of this variable causes LISP to display its value. Since the value is a list, (car fruit) is the first element and (cdr fruit) is all the rest.

The next example demonstrates a very simple evaluation of the arithmetic function **plus** on the atoms 1 and 3. The following input shows how you can set a variable with a numerical value and the next shows how you can use this value in a subsequent calculation. The next input shows how the mixing of symbolic and numerical variables is nonsensical when you use an arithmetic function.

The following two inputs show how lists can contain words, strings, and numbers.

The final input gets you out of the session:

```
Input: (stop)
```

This is the correct method of signing off; lazy users often finish with the end-of-file marker **fin**.

Chapter 2: Programming in Cambridge LISP 68000

2.1 Character Set

In Cambrige LISP, as in most languages, certain characters have special properties associated with them. However, you can change them with the **setsyntax** function, although this is not recommended for anyone who is not an expert to try. The character types and their properties are as follows:

break-character	causes the character to terminate identifiers.
digit	initially applies to 0. . .9
escape	causes the character to force any character following it to be treated as a letter. This is useful for including special characters in names, for example, O!'Donnell (without the escape marker !, ' would be treated as **quote**).
ignore	causes a character to be ignored.
letter	initially applies to a. . .z. A. . .Z
macro	causes a function to be associated with a character. This is called (with no arguments) whenever the character is encountered in the input. The result returned by this function becomes an element of the list being read. This is the same as read-macro.
may-start-number	causes a character to be accepted before a number without being evaluated or returning an error.
splice	as macro except tha the function returns a list of items to be included in the list being read. This is the same as splice-read-macro.
upper-case	initially applies to A. . .Z

Characters that initially have special input properties are as follows:

break-character	**eol, ff, blank, tab, eof** (.){}#$%&=-`~^\[]+:,*:()?/
digit	0123456789
escape	!
ignore	**eol, ff, blank, tab**
letter	abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ
may-start-number	+-.
upper-case	ABCDEFGHIJKLMNOPQRSTUVWXYZ

To force strange characters to be treated as letters, you prefix them with the escape character !. For instance, as in the following example, you must type a ! before a minus sign.

```
(set 'a '(!-1 0 1))
```

The following characters also need to be prefixed in this way unless you first make them into a letter:

```
_ \ * % < > [ ] + - # ; : ? / | & $ ~ ^ ,
```

The following have special meaning when not prefixed:

```
' " ( ) {} .
```

All input should be in lower case (that is, a . . . z). However, you can make Cambridge LISP ignore case (which is useful if you want to convert a program from another Lisp). To do this, you set the flag *lower to true by typing

```
(set '!*lower t)
```

Note the use of the escape prefix before the character *. LISP accepts both upper and lower case letters interchangeably after you set *lower.

The break character % instructs LISP to ignore anything that follows. This is useful because it means that you can use % to initiate a comment. Here is an example of % introducing a comment.

```
     %setting var 'fruit'
(set 'fruit '(oranges "passion fruit"))
```

For backquote macros, you use the characters ` (backquote), , (comma), and @ (at).

To type in vectors, you use the character [.

2.2 Language Elements

Every LISP datum is either a dotted pair (so that you can apply **car** and **cdr**), or an atom (so that **car** and **cdr** give errors if you try to apply them). Various special structures built up from dotted pairs have special uses.

association lists (alists)

```
((a1 . v1) (a2 . v2).....)
```

lists

```
((a . (b . (c . nil)))
```

property lists (plists)

```
(flag1 (a1 . v1) (a2 . v2) flag2....)
```

Every identifier has a property list that you access with the function **plist**. For example,

```
(plist 'lpar)
```

might return

```
((pp!-char . 40) constant global1)
```

where the number it returns is an internally used global number. Note that (2.1) is a list of one element, the floating point number 2.1, whereas (2 . 1) is a dotted pair, whose **car** is 2 and whose **cdr** is 1.

Atoms can be

complied programs	tested by the predicate **codep**
identifiers	tested by the predicate **idp**
numbers	tested by the predicate **numberp**. (You can discriminate among different types of numbers with **fixp**, **floatp**, **rationalp**, and **smallp**)
strings	tested by the predicate **stringp**
vectors	tested by the predicate **vectorp**

Uncompiled functions take the form, for example,

```
(lambda (x) (plus x 1))
```

2.3 Evaluating S-Expressions

The example session at the end of Chapter 1, Introduction to Cambridge LISP 68000, demonstrates how Cambridge LISP evaluates simple atoms and lists. It covers the following: returning a value for an atom or string, setting variables, and carrying out the arithmetic function **plus**. This section attempts to take you a bit further by describing the particular features of Cambridge LISP concerning the definition and evaluation of functions.

Defining Functions

You can define a function in Cambridge LISP by using (de ...) or (df ...). The syntax for **de** is as follows:

```
(de <function name> <argument list> <body>)
```

For example,

```
(de mycons (a b) (cons a b))
```

where 'mycons' is the function name to be defined, a b the arguments, and (cons a b) the body or action of the function. Note that **de** is a special form that does not require you to put quote marks in front of its arguments.

As well as functions that have their arguments spread out, you can define functions that expect an evaluated list. These are sometimes known as **lexprs**. In Cambridge LISP they are defined, for example, by

```
(de myfunc arglist
  (mapcar arglist (function print)))
```

Note that **de** automatically invokes the compiler if you set the global flag ***comp** to non-**nil**. (See Section 3.2, The LISP Compiler, for a description of the compiler.) **df** is just like **de**, except that the function defined does not have its arguments evaluated. For example, you could define **de** by typing the following:

```
(df de (name args body)
    (set name (list 'lambda args body))
    (cond
        (!*comp (compile (list name))))
    name)
```

Remember that you must prefix ***comp** with the escape ! (! introduces any unusual character, such as *). As with **de**, Cambridge LISP allows you to define functions that take a list of unevaluated arguments with **df**.

```
(df mylffun arglist
  <body>)
```

de expects its arguments to be evaluated, this is known as being of type EVAL; **df** takes unevaluated arguments, this is known as type NOEVAL. Functions can also be of type SPREAD or NOSPREAD. A SPREAD function can be defined as follows:

```
(de foo (x...))
```

if evaluated, and

```
(df foo (x...))
```

if unevaluated. A NOSPREAD version of foo can be defined as follows:

```
(de foo x...)
```

if evaluated, and

```
(df foo x...)
```

if unevaluated. That is to say, NOSPREAD functions receive their arguments as a single list, while SPREAD functions receive their arguments in a one to one relationship with their argument list.

	EVAL	NOEVAL
SPREAD	(de foo (x...))	(df foo (x...))
NOSPREAD	(de foo x...)	(df foo x...)

Table 2-A: Function Types

These forms of **de** and **df** are similar to certain constructions found in other Lisps which use the words lexpr, fexpr, and expr. Following the same format as Table 2-A, Table 2-B indicates the constructs that apply to the four 'foo' functions described above.

expr	"fexpr"
lexpr	fexpr

Table 2-B: LISP Constructs

As well as functions there are macros. In macros, LISP evaluates the body to give a form that it can then evaluate. The arguments to a macro can be either SPREAD or NOSPREAD, but in both cases the name of the macro is not available (that is, arglist would be (a b c) in the call (maccons a b c)). The following macro definition:

```
(dm maccons arglist
  (list 'cons (car arglist) (maccons (cdr arglist))))
```

defines a function that is similar to **list**.

As well as **dm**, Cambridge LISP has another method of defining a macro - **dmm**. **dmm** is identical to the macro definition function found in MACLISP. Cambridge LISP has this function to help you transfer your programs from other Lisps. In MACLISP, you define a macro using this syntax:

(DEFUN <function name> MACRO (<single parameter>) <body>)

In Cambridge LISP, you omit the DEFUN and MACRO and use **dmm** instead; otherwise, the macro definition is similar to the MACLISP form.

Whenever you redefine an existing function with **de**, **df**, **dm**, or **dmm**, LISP displays the following warning message:

```
*** <function name> redefined
```

LISP is very flexible in that it allows you to redefine existing names. An existing name might be one you previously defined, or one known to the system. It is obviously dangerous to redefine those names that LISP uses internally and so a list of these 'reserved words' appears in Appendix F.

Although some other LISP systems use constructions built around the word **fexpr** to achieve the effect of **df**, **fexpr** is not part of Cambridge LISP. A bound variable list that is, in fact, a single (non-**nil**) atom is treated specially by both **lambda** and **lambdaq**. The variable gets bound to the complete list of arguments given to the function, and there is no check or constraint on how many arguments are given. This makes it reasonably easy to define functions, like **list** and **plus**, that can cope with any number of arguments.

Evaluation of Functions

When you give LISP something to evaluate, it has to decide what to do with it. The Cambridge system works in the following way:

a) if the 'function part' is a list of the form (lambda ...) or (lambdaq ...), or if it is the entry-point of a piece of compiled program, LISP processes it directly

b) otherwise, LISP replaces the function part with its value and goes back to step (a).

LISP repeatedly evaluates functions until they make sense. All LISP built-in functions are just variables that have been given initial values that are the entry-points of the corresponding pieces of code. It is an error to try to apply an expression that does not turn into a recognizable function when evaluated enough times - LISP can detect this in most simple cases, but can be put in a loop sometimes.

Note: An evaluation loop as described above has just the same status as a loop of the form XX (go XX): it is a user error and NOT a failure in the LISP system.

Some other LISP systems only evaluate function parts once, or perhaps twice before demanding that a recognizable form has been reached. For example, in Cambridge LISP, although it traps an attempt to use the function 'a' after

```
(setq a 'a)
```

as being a simple circular definition,

```
(setq a '(a a))
```

sets up an infinite loop when you use the function 'a'

Cambridge LISP uses the symbol **lambdaq** when a function must avoid evaluating its arguments too soon. The use of **lambdaq** is just like that of **lambda**, except that it inhibits the evaluation of its

arguments. Therefore, you can redefine the (built-in) function **quote** without disturbing the system by typing

```
(setq quote '(lambdaq (a) a))
```

To get things explicitly evaluated, you can call **eval** within the body of a function introduced in this way.

Note that the previous use of **setq** to introduce a special form is not recommended, in that the LISP compiler requires all special forms to be flagged as such, so really an extra

```
(flag '(quote) 'fexpr)
```

would be needed. If you define functions using **putd**, **de**, and **df**, LISP automatically takes care of this technical point.

The Cambridge LISP interpreter implements the 'implied progn' feature of many LISP systems, where you can omit the function **progn** (used to group LISP statements) in nearly all circumstances. If you give a function too few arguments, then LISP supplies additional arguments of **nil**. If you give too many arguments, then LISP ignores excess arguments after it has evaluated them (that is, in the case of EVAL functions). For example, using **progn**, you could type

```
Input: (de test()
(progn
(print 'This)
(print 'is)
(print 'a)
(print 'test)
)
)
Value: test

Input: (test)
This
is
a
test
Value: test
```

and without **progn**

```
Input: (de test()
(print 'This)
(print 'is)
(print 'a)
(print 'test)
)

*** test redefined
Value: test

Input: (test)
This
is
```

```
a
test
Value: test
```

This can then be compiled and run:

```
Input: (compile '(test)
)

** 64 bytes    60 msecs     compiling test
Value: (test)

Input: (test)
This
is
a
test
Value: test
```

2.4 Arithmetic, Logical, and Elementary Functions

In Cambridge LISP integers can grow to be any size (up to store limits). Rational numbers can have arbitrary numerators and denominators. Floating point numbers are single-precision in IEEE format. (rational fp) calculates that rational number corresponding to the interpretation of the fp number as a certain integer multiplied by a power of 2.

There are faster versions of the arithmetic routines, ones that only work on smallish numbers, which all start with the letter i, for example, **iplus**. These allow numbers to be in the range -2**24 to 2**24 (yes, 25 bits including the sign). (Note that **smallp** checks if a number is "smallish"). In compiled code most of these are turned into open code which goes much faster than normal LISP arithmetic but which does not check for valid arguments or overflow. These are normally only used in time-critical compiled code.

Arithmetic functions

Function	Description
difference	subtract two args
divide	value is cons of quotient and remainder
expt	raise to a power (incl. floating point)
minus	negate one thing
plus	add any number of args
quotient	)truncate if given integer args
remainder	)
rational	exact rational quotient or convert to rational form
sqrt	floating point function
times	multiply any number of args

Predicates on numbers

Predicate	Description
fixp	true if arg is an integer
floatp	true if arg is f.p.
greaterp	test >
lessp	test <
eqn	test = (numerical equality, for example, 1=1.0)
equal	test = (exact equality only, but 1=1.0)

Table 2-C: Arithmetic Functions, Numeric Predicates, and Logical Operations

Logical operations

Operation	Description
leftshift U V	U is shifted left by V spaces
logand U1 U2 ...	logical and of all its arguments
logand2 U V	as **logand** but taking only two arguments
logor U1 U2 ...	logical or
logor2	as **logor** see **logand2**
logxor U1 U2 ...	logical exclusive or
logxor2	as **logxor** see **logand2**
setdiff U V	set difference (U - V)
union U V	set union (U + V)
xn U V	set intersection (U . V)

(continuation of Table 2-C)

Cambridge LISP assumes logical numbers to be 24-bit; if they are not, it takes the bottom 24-bits of internal representation. This feature can provoke some strange results if you use very big numbers. For instance, **logand**, **logor**, and **logxor** do 24-bit boolean operations on small integers. If you give these functions a big number, they first reduce the number modulo 2**24. **leftshift** shifts a 24 bit integer left (or right if second arg is negative) a specified number of places. **leftshift** keeps within 24 bits, though, and it is not a general multiply routine for powers of 2. As described above, there are faster versions that only work on small integers: **ileftshift**, **ilogor**, **ilogxor**, etc.

The following elementary functions are all floating point:

sin	cos	exp	log
log10	tan	cot	atan

Table 2-D: Elementary Functions

Examples:

```
Input: (rational 1 3)
Value: (1/3)
```

and

```
Input: (float 3)
Value: 3.0

Input: (float (rational 1 3))
Value: 0.3333333
```

2.5 Input and Output

Input and output in LISP is based on the idea of selectable streams. You can select these streams when you load LISP by using the keywords FROM and TO. (For further details on the use of these keywords, see Section 1.3 Loading LISP.) At the start of a run, the stream you specify with the keyword FROM is open and selected for input (the default input is the keyboard of your terminal) and TO is open and selected for output (the default output is the screen of your terminal). Thus, you type input in directly to the terminal and, similarly, all values appear on the terminal screen.

One common occasion when you should change the default streams is when you run a LISP program. Usually you run LISP interactively: you type the instructions at the terminal and the values return directly to your screen. But what if the program is large, or if the code exists in a file on disk? In fact, you can still run LISP interactively. All that you need to do is to use the function **rdf**. This function takes one or two arguments: the first argument to **rdf** indicates the file you want to read from; the second argument indicates the file you want to write to. If you only give one argument name, then **rdf** assumes the TO to be the terminal (that is, the default). Note that, as you refer to the files from within LISP, the rules governing the prefixing of strange characters are in force. Also, LISP only accepts filenames of up to 255 characters in length (excluding any escape characters); ones longer than this are truncated.

Here is the syntax you use for **rdf**:

```
(rdf '<filename> ['<filename>])
```

For example,

```
(rdf 'my!-test)
```

attempts to locate a file with the name 'my!-test' in the current directory. If you can open the file, you can read in and execute the code stored there. In the example, as no second filename is quoted, the results return to the terminal. After the code in the file is executed, control returns to the state it was before you ran the program (that is, LISP is still running and interactive use can continue).

You should end each file of code with the end-of-file marker **fin**. It is usually a good idea to add a number of extra closing brackets as well, for example,

```
fin))))))))))))))))))))))
```

If you forget **fin**, the system stops at the end of the file, but it also gives a warning message to that effect.

To change the stream selection temporarily and return to using interactive LISP, you use **rdf** from inside LISP. **rdf**, however, expects the contents of the file to be in LISP code. Entering data requires the use of the more sopisticated form of stream selection described below.

Apart from using the read-file **rdf**, you can also select streams with the read-stream **rds** and write-stream **wrs**. However, before you can select another stream using these commands, you must first open the stream for input or output.

To open a stream for input, you use the following syntax:

```
(open '<filename> 'input)
```

To open a stream for output, you use the following syntax:

```
(open '<filename> 'output)
```

Note that a stream must be open before you can select it. The second argument to **open** specifies whether it is read to, or written from.

After you have specifically selected and used a stream, you should close it. To close a stream, you use the following syntax:

```
(close '<filename>)
```

Once you have opened a stream, you can select it. To select a stream to be written to, you use the following syntax:

```
(wrs '<filename>)
```

To select a stream to be read from, you use the following syntax:

```
(rds '<filename>)
```

To un-select the output stream and return to the terminal, type

```
(wrs nil)
```

To un-select the input stream and return to the terminal, type

```
(rds nil)
```

Although it is easiest to access files from within the current directory, you can also open a file within another directory using the following format:

```
(open '(<dirname> '<filename>) 'input)
```

Each directory name and filename can be up to a maximum of 255 characters. (Of course, if you give each name 255 characters, you can build a full name of directories, subdirectories, and files that is as long as you like.)

rds and **wrs** alter all input and output (with the possible exception of diagnostics which should always appear on the terminal). In particular, both **rds** and **wrs** fail to keep the data to be read in, and the LISP statements for execution, apart from each other.

The stream that you select for output need not necessarily be either a file or directory name; a device name is equally acceptable in certain circumstances. For instance, if you wanted to send output to the printer, you could type

```
(open 'prt!: 'output)

(wrs 'prt!:)

(close 'prt!:)
```

If you change the TO stream from the default (the terminal) to the printer, it is essential that you leave the printer enough room in which to work. To do this, you use the keyword LEAVE when you load LISP.

```
lisp LEAVE 100K
```

As soon as you use **rds**, LISP stops listening to the terminal and reads from the named file. It continues to read from the file until it meets either another **rds** or an end-of-file. On finding an end-of-file (either explicitly with **fin** or implicitly by exhausting the data or code in the file), control passes back to the primary state - in most cases the system control level and not interactive LISP. One common use of **rds** and **wrs** is inside programs, for example, to read in data.

Output

There are four printing styles available in LISP. The names of the printing functions that make up these four styles are specific to Cambridge LISP; other Lisps tend to refer to the different sorts of print by a numerical suffix. The first of these styles produces output that you can read back into LISP at a later stage, as described above. This style prints strings with " marks around them and prefixes any unusual characters in atoms with escape marks (that is, with !). To specify this print style, you can use the following:

Function	Description
prin	prints an atom or list (and does nothing else)
print	as prin but then ends the print line
printm	as print but it attempts to leave a left margin when an expression takes many lines to print.

The next set of routines print in a way that is useful when the program is trying to produce non-LISP-like output: they just print the characters of atoms and strings without any extra markers.

Function	Description
princ	basic print routine
princs	as princ but starts new line if not a beginning
printc	as princ followed by call to terpri
printcm	as printc but leaves a left margin of specified width.

Sometimes you may find it useful to produce circular structures. The previous print styles go into an infinite loop if you ask them to print such a list. The following family, however, is safe

Function	Description
prinl	basic print for looped structure
printl	as prinl terminated by a terpri
princl	as prinl with no escapes
printcl	as princl with terpri.

You use the last style of printing for displaying LISP programs, as they provide a consistent style of indentation.

Function	Description
superprint	print with indentation
superprintm	as **superprint** but with a left margin
prettyprint	similar to **superprint** (see below).

For explicit control of page layout, you can use the following functions:

Function	Description
terpri	end a record (with a newline)
eject	as terpri but prints 'end-of-page', if available
linelength	set the logical width of paper that print can use
spaces	synonym for xtab
ttab	tab to specified column
xtab	tab by a given number of columns.

Input

The read routines are as follows:

Function	Description
readch	get next character
read	get next s-expression or atom.
readq	read quiet - like **read** but with no echo
readchq	readch quiet - like **readch** but with no echo.

Characters obtained by calls to **readch** can be classified by:

Function	Description
digit	is it 0,1,2,3,4,5,6,7,8 or 9?
liter	is it A,B,... ,Z,a,b,...,z?
breakp	detects blank, comma, brackets, etc. The exact definition depends on the results of **setsyntax**.

Characters can be packed in a buffer and then assembled into complete atoms. Buffer-handling routines are as follows:

Function	Description
clearbuff	clear the buffer (must be called first)
pack	puts characters in the buffer
mkatom	convert buffer contents to an atom
numob	convert buffer contents to a number
mkstring	converts buffer into a string.

numob is not happy if the buffer contents are anything other than a sensible string of digits, possibly with a leading sign. You must not call **pack** before you have used **clearbuff** to set things to a standard initial state. **read**, either called explicitly by you or called by the system to read in some more LISP code, destroys the buffer contents. You should, therefore, only call **pack**, etc., from within a nest of routines that both starts and finishes the atom assembly process.

explode takes any argument and returns a list of characters. The characters are similar to those **prin** would produce if you gave it the same list or atom as an argument. In particular, **explode** converts numbers into characters.

Two special print functions can be confused - **superprint** and **prettyprint**. Both **superprint** and **prettyprint** print with indentation. The difference lies in where they start. **prettyprint** is exactly the same as **superprint**, except that it starts at the position on the page that the last print command left the cursor. **prettyprint** is roughly defined as follows:

```
(de prettyprint (x) (superprintm x (posn)))
```

whereas **superprint** is defined as:

```
(de superprint (x) (superprintm x 0))
```

For example,

```
(progn (princ "ABCDEFG")
       (prettyprint '(("AB") ("CD") ("EF"))))
```

produces

```
ABCDEFG (("AB") ("CD") ("EF"))
```

but

```
(progn (princ "ABCDEFG")
       (superprint '(("AB") ("CD") ("EF"))))
```

produces

```
ABCDEFG
(("AB") ("CD") ("EF"))
```

2.6 Error Handling

LISP attempts to return full detailed error messages whenever anything unexpected happens. The normal error message consists of a line containing an error number and a textual message corresponding to it. After the error message, LISP returns a full backtrace showing the functions called prior to the error and any arguments passed. The backtrace can be extremely verbose, especially in cases of runaway recursion; however, you can cancel the backtrace output by typing CTRL-C. You can also alter the action of the LISP error handler by using the function **backgag**. The function **backgag** takes a number in the range 0 to 9 and alters the amount of information produced after an error. Specifying 0 or nil gives no notification of errors at all; 1 just gives the error message, but no backtrace; other values up to the default of 5 give increasing amounts of information. Values greater than 5 give more information than is usually necessary and can even lead to greater confusion.

A more sophisticated error handling mechanism is possible if you use the function **errorset**. This function allows a LISP program to keep control over things even when it is failing, bug-ridden, or otherwise unhappy. It also allows you to handle errors in different ways in different parts of a LISP program. For example,

```
(errorset x fg)
```

evaluates x and returns (cons <value of x> nil) normally. If an error occurs during the evaluation of x, control returns to **errorset** and the value returned is atomic. In fact the value is a numeric code identifying the sort of error involved. To provide recovery from syntax errors and the like, you can wrap **errorset** round read routines. You can use **errorset** round print routines so that calculation can keep going even if some intermediate results cannot be printed in full and so you can use it to give the effect of backtracking. The value specified as fg is a number in the range 0 to 5. This number controls the amount of information produced about the error in the same way as the argument passed to **backgag** above. Notice, in particular, that a call specifying fg as nil suppresses the error message entirely. **errorset** cannot trap fatal conditions like 'bus error'.

Here is an example of the use of **errorset**:

```
(errorset '(car 'a) nil)

Value: 24
```

If you look in the error table, you can find the meaning of the value that **errorset** returns. For instance, if you look up 24 (that is, the value in the example), you find that the error results from an attempt to get the **car** of an atom.

2.7 Catch and Throw

The functions **catch** and **throw** provide a general control structure. You only use the two functions in conjunction; a call to **throw** causes a non-local transfer of control to an earlier call of **catch**. Multiple uses of the functions are possible as **throw** jumps to a **catch** that you have specified with the same tag value. For example,

```
(de test() (plus 2 (throw x 99)))

Value: test
```

defines a function 'test'. Although this contains a call to 'plus,' the **throw** causes a non-local jump out of the evaluation of 'plus' and directly into a preceding call to **catch** with the same tag 'x'.

catch takes an expression as the second argument which is evaluated normally. If **catch** encounters a **throw**, it leaves off the evaluation of the argument as **throw** causes a change of control. Thus

```
(catch x (plus 2 3))

Value: 5
```

simply evaluates the given expression. However, a call of the form

```
(catch x (plus 7 (test)))

Value: 99
```

returns the value specified in the **throw** and ignores the partially evaluated calls to 'plus'.

2.8 AVL

The term AVL comes from the initial letters of the two Russian scientists G.M. Adel'son-Vel'skii and E.M. Landis who designed an elegant algorithm for a balanced tree structure now known as an AVL tree. This structure is most useful as an efficient method for internal storage.

Balanced trees are those where the number of nodes on each branch are either equal, or +1 or -1 of each other and hence have the minimum number of levels. The following figure provides a diagramatic representation of a tree.

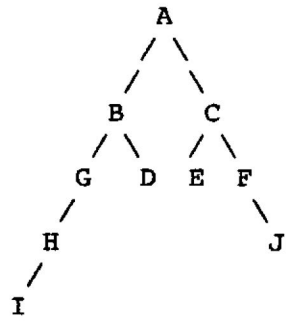


Figure 2-A: Tree Structure

However, the tree in Figure 2-A balances at C, but not at B; therefore, Figure 2-A is not an AVL tree.

An AVL tree represents sorted data as interlinked nodes. Each new entry that is inserted creates a new level. If many new items are added the tree could become unbalanced by increasing the length of the search path. However, the AVL tree includes certain factors that enable re-balancing when new elements are added. To print the tree in sorted order the Amiga follows the AVL algorithm. The AVL tree module therefore provides an efficient balanced tree lookup and deletion which is used in the printing of circular lists. (See Section 4.3 for **printl, prinl, printcl** as well as functions starting with **avl-**).

Examples:

```
Input: (setq fred nil)
Value: nil

Input: (setq fred (avl!-add 67 fred))
Value: ((67) nil)

Input: (setq fred (avl!-add 65 fred 'lessp))
Value: ((67) ((65) nil))
```

2.9 Object List

You can check all the available functions by looking at the object list. LISP keeps the names of all identifiers - and hence all functions - that it knows about in its *object list*. This list is externally represented as a lexagraphically ordered list. The function **oblist**, which does not need any arguments, returns a list of all items held in this structure. It is thus a sort of index to the collection of available functions. As the object list is usually very long, it is a good idea to send it to the printer instead of the terminal. You can then study the contents of the list at leisure. To obtain a printed copy of the object list, type

```
lisp leave 100k

(open 'prt!: 'output)

(wrs 'prt!:)

(print oblist)

(close 'prt!:)
```

Note: In Cambridge LISP the object list has an internal structure that is made up of a collection of balanced trees. This structure is efficient, but it means that you cannot use certain operations on it (see the function **oblist** in Section 4.3 in Chapter 4, Functions and Variables).

LISP does not usually allow you to apply the same name to more than one thing, because it looks up the name in the oblist and comments if you have re-used a name. You can, however, remove a name from the oblist so that no one else can find the corresponding function. The name then becomes free and can be defined. To remove a name from the **oblist** so that you can redefine it, you use **remob**. Nevertheless, any pointers that existed before you used **remob** remain. So that, if you **remob** fred, the function 'foo', which still contains a call to 'fred', continues to carry out the original action of 'fred' even if you have redefined the name 'fred'.

To create a unique identifier that is not interned on the **oblist**, and hence not **eq** to anything, you use the function **gensym**. These are probably of most use to experts!

Chapter 3: System components

3.1 The Top Level of Control

Supervisor and Interpreter

The LISP supervisor controls the process of reading and executing LISP programs. This is part of the top level of control.

The normal way of talking to Cambridge LISP is known as *eval mode*, in contrast to the *evalquote mode* described in the LISP 1.5 manual. However, you can make Cambridge LISP simulate the older evalquote scheme. So, if you have existing programs, you can make use of them directly by redefining the initial supervisor. For more details, see the function **preserve** in Chapter 4, Functions and Variables. In eval mode, you give LISP a sequence of expressions to evaluate and obey. This sequence ends with an occurrence of the word **fin**. Note that LISP evaluates expressions in exactly the way it would process them if they appeared within a LISP function, and so, in particular, literal data needs quote marks. Thus

```
(cons 'a 'b)
```

evaluates to a list containing the atoms a and b - indicated in this manual by the syntax (a . b). However,

```
(cons a b)
```

conses together whatever the values of the (global) variables **a** and **b** are. For example, with quotes

```
(cons 'a 'b)

Value: (a . b)
```

and without

```
(setq a '(oranges))

Value: (oranges)

(setq b '(pears))

Value: (pears)

(cons a b)

Value: ((oranges) pears)
```

The LISP supervisor displays various amounts of information as it executes a program. The options available for controlling this output are as follows:

(prarg lev) — sets echo level for input to level = lev: lev = 0 no printing, lev = 1 no printing, but prompt given; lev = 2 prettyprint.

(prval lev)	sets format for printing result of evaluation: lev = 0 no message; lev = 1 print (default); lev = 2 uses princ; lev = 3 superprint.
(prmsg lev)	lev = 0 no messages; lev = 1 'INPUT:' and 'Value:'.
(verbos lev)	lev = 0 quiet; lev = 1 message at each garbage collection.
(outradix n)	sets the radix for printing of numbers to be n. The only legal values of n are 2, 8, 10, and 16.

When you load Cambridge LISP, it gives the prompts

```
Input:
```

and

```
Value:
```

To change the value of these prompts, you can alter the value of prompt* and value*. For example:

```
(setq prompt!* "?")

(setq value!* "=")
```

To remove all prompts, type

```
(prmsg 0)
```

and to return them, type

```
(prmsg 1)
```

Although Cambridge LISP initially requires all input to be in lower case, you can make LISP equate all cases by setting the flag *lower to true. To do this, type

```
(setq !*lower t)
```

Cambridge LISP 68000 is a value cell system. This means that for each identifier in any LISP program, LISP uses a single word to hold the current value of that variable. When variables are bound or unbound, LISP updates the contents on the value cell as appropriate. This scheme has the advantage of being efficient and lends itself to graceful error recovery. It does, however, mean that it is hard, if not impossible, to provide a function closure mechanism such as in LISP systems based on association lists. So far as a casual user is concerned, the visible manifestations of a value cell LISP implementation are as follows:

a) A name can only be associated with one value at once. If you try to use a local variable called car, you cannot get at the built-in function **car** while within the scope of your local variable.

b) Function definition and **cset** are just variants of **set**. If you assign to a name not bound as a local variable somewhere, LISP stores your assigned value globally.

c) Functions used as functional arguments should not reference free variables. This rule can be relaxed if you understand the technicalities and possible consequences of not keeping to it.

Saving a Core Image

If you develop a large program, you may find it useful to dump the state of LISP to a disk file and tnen pick it up again in a later run. To dump the state of LISP to disk, you use the function called **preserve**. For example, type

```
(preserve)
```

This function writes a core-image to the file with the name LISPROOT in the DUMP (or IMAGE) directory. LISP stops after it has executed **preserve**, even if you have embedded the call to **preserve** deep in some other functions. On restart, LISP loses all information about what was happening when you called **preserve**. **preserve** writes to members LISPROOT and LISPERRS, together with any other members defined by the FASL facility. If IMAGE and DUMP are the same, then **preserve** renames the previous LISPROOT as BACKROOT before dumping the core. You can only load core images at the very start of a LISP run. On entry to LISP, the parameter IMAGE specifies the directory containing the core image to be loaded. For example,

```
LISP IMAGE=SYS:L/LISP/IMAGE DUMP=SAVEDIR
 . . .
  (preserve)
```

dumps a core image to the directory SAVEDIR, and

```
LISP IMAGE=SAVEDIR
 . . .
```

reloads it and continues with the computation. If **preserve** writes out a file successfully, LISP stops with a return code of at least 200 to signal this fact.

```
(preserve '<initialsupervisor>)
```

is a form that you can use to package systems in a secure way. If you preserved your LISP image using the above example, when you came reload the image file, the function initialsupervisor would be called instead of the standard LISP supervisor. Once you exit initialsupervisor, LISP stops. (Note that initialsupervisor is a user's function, that is to say, it does not exist as **initialsupervisor**).

3.2 The LISP Compiler

Serious users of LISP need to invoke the compiler to convert their programs from s-expressions into hard executable machine code. This process normally saves store and results in functions that run much faster. The compilation process, however, takes some liberties with LISP semantics, and so you cannot use it indiscriminantly. To use the compiler at all, you must use an image directory containing the module LISPCOMP for the LISP system. If f1, f2 ... are functions that you have just defined, the following call:

```
(compile '(f1 f2 ...))
```

replaces the definitions of the functions with compiled code that is essentially equivalent.

Consider the simple function 'test'.

```
Input: (de test ()
            (print 'This)
            (print 'is)
            (print 'a)
            (print 'test))
```

You can compile 'test' and run it as follows:

```
Input: (compile '(test))

** 64 bytes    60 msecs     compiling test
Value: (test)

Input: (test)
This
is
a
test
Value: test
```

You can compile functions defined using **de**, **df**, **dm**, or **dmm** automatically by setting the flag ***comp** to non-**nil**. For example,

```
Input: (setq !*comp t)
Value: t

Input: (de test()
(print 'Hello)
(print 'World))

*** test redefined
** 42 bytes    60 msecs     compiling test
Value: test
```

The compiler has two major semantic differences from the interpreter that make compilation more effective.

(a) In compiled code, LISP treats variables bound either by **lambda** or by **prog** as local to the function binding them unless you have specially declared them to be fluid. Thus, if you use free variables and the LISP binding rules for them, you must warn the compiler.

(b) The compiler freezes some assumptions and knowledge about the routines that the function being compiled calls. These assumptions can sometimes lead to inconsistencies if the user attempts to redefine functions later. The compiler does not treat calls to user-defined special forms (that is, **lambdaq** things) properly unless the special form has been defined before any attempt is made to compile a call to it. (For further details, see the function **declare** in Section 4.3, Functions and Variables.)

These restrictions arise because the compiler generates open code or special calling sequences for many standard LISP functions. LISP treats certain function names specially and causes the compiler to generate fixed sequences of orders irrespective of what definition you think should be associated with the name. These functions include **car**, **cdr**, **cons**, **putv**, **getv**, the integer arithmetic routines **iplus**, **iminus**, etc., and a few more. (A complete list appears in Table 3-A.) If you redefine one of these functions, it alters the behaviour of the interpreted code but does not affect the compiled code in any way. In all other cases, if you redefine a function, all references to that name see the new definition.

To tell the compiler to treat a variable 'properly' with respect to lambda bindings, before calling the compiler, you should type

```
(fluid '(var1 var2 ...))
```

to declare the variables. You can remove the declaration after compilation with a corresponding call to **unfluid**. Note that **fluid** takes the place of declarations known as SPECIAL and COMMON in some other LISP systems. When LISP has compiled all the user functions, you can recover a substantial amount of space by telling LISP to throw away the compiler. To tell LISP to throw away the compiler, type

```
(excise 'lispcomp)
```

Normally, the compiler calculates the arguments to a function call in an order that suits it. If you set *ord to true, LISP evaluates a function's arguments in a strict left to right order. This is helpful if you want to make use of the side effects of the left to right action. However, this is not usually good practice.

allocatequote	**and**	**apply**
casego	**catch**	**cdifference**
cminus	**conc**	**cond**
cons	**cplus**	**ctimes**
cxxxxr family	**eq**	**equal**
errorset	**flagp**	**function**
gensym	**get**	**getd**
getv	**go**	**iadd1**
idifference	**igreaterp**	**ileftshift**
ilessp	**ilogand**	**ilogand2**
ilogor	**ilogor2**	**ilogxor**
ilogxor2	**imax**	**imin**
iminus	**iplus**	**iplus2**
irightshift	**isub1**	**itimes**
itimes2	**izerop**	**lambda**
lambdaq	**list**	**logand**
logor	**logxor**	**map**
mapcan	**mapcon**	**max**
ncons	**or**	**planthalfword**
plus	**prog**	**prog2**
progn	**putv**	**return**
rplaca	**rplacd**	**rplacw**
setq	**times**	**xcons**

and all macros

Table 3-A: Functions Fixed by The Compiler

Load-on-call Facility

The following functions provide for a load-on-call facility in LISP. Note that LISP loads all the code from members of the directory called IMAGE (unless the call designates another directory for that function) and writes any generated code to the same directory, unless you specify a DUMP keyword when you call LISP. You should avoid the filenames LISPROOT, BACKROOT, and ERRORMSG as the basic system uses these names. Also LISPCOMP, LISPREAD, LISPXRD, LISPAVLT, LISPRDM, LISPSPRI, and LISPEDIT are part of the base system.

`(module membername)`

The compiler writes Fast Load (FASL) data for functions it compiles to the file membername in directory DUMP; it writes these files as binary data. If you specify membername as **nil**, or not at all, LISP switches off FASL output and the compiler produces code in store. If you replace a module, then it is essential that DUMP refers to a different directory from the one you specified as IMAGE. Note that you must call **preserve** at the end of the LISP run if you have used this function.

`(excise membername)`

If membername is the name of a FASL file that you have loaded, (excise membername) replaces the definitions of all functions in that file with references to their disk versions. This frees the space that the compiled code bodies consumed, but this space only becomes available at the next garbage collection. If you call the functions again, the disk file reloads automatically.

`(preserve)`

To write a file LISPROOT to IMAGE (or DUMP, as described above) in a format suitable for LISP to reload it at the start of a subsequent run, you type (preserve). If the core image is being

written to IMAGE, then the previous image is renamed as BACKROOT in case of disaster. **preserve** also closes the output to a module thereby removing the necessity for a (module nil). After executing **preserve**, LISP stops. Note that core images are very large and you should check that there are sufficient free blocks available on the disk before using this function. You should take particular care if your system is floppy-based.

If you give the appropriate value to **verbos**, it monitors the loading of modules. From this it is possible to see how much store thrashing there is.

3.3 The LISP Editor

The Structure Editor is an integral part of Cambridge LISP. You can call the editor whenever you want to edit an interpreted function, property list, or expression. The editor allows you to wander through the list expression at will, searching for substructures and making alterations. The result is always a valid LISP expression, because it is impossible to have the wrong number of brackets in a structure editor! Editing with the LISP editor is completely safe as there is an **undo** facility and when you leave the editor there is an opportunity to repudiate all the changes you made. In addition, to assist you when you edit the same function repeatedly, the editor remembers the last expression edited and can even be made to re-enter at the same place in the expression it was immediately before the exit. There is also a facility that enables you to add new functions to the editor; however, the commands provided should be sufficient for your needs. Only advanced users should attempt to add new functions!

You can enter the editor with one of the functions **editf**, **editv**, **editp**, or **edite**, which allow you to edit functions, values, property lists, and general LISP expressions. The first three functions are special forms, so you need not quote their arguments. In fact the editor remembers the last call to the edit entry function, so calling these without an argument re-edits that last object.

All the time the editor is working it focusses attention on one node of the tree that forms the expression, called the *current position*. As you move the position with the various commands, the editor keeps a list, called the *edit chain*, of the previous current positions. The editor uses this list whenever you want to undo anything.

The simplest of edit commands is **p**. **p** prints the current expression to a limited depth (default value 3). **p** displays parts of the expression deeper than this as a **&** sign. If you want the whole sub-expression, then you can use **pp** to call the prettyprinter to display it. This can cause problems with circular structures: **?** uses **printl** for completely safe printing. The prompt character in the editor is **#**.

For example, if your call to the editor was

```
(edite '(a b c (d (e (f)))))
edit:
#p                              Print the expression
(a b c (d &))

#?
(a b c (d (e (f))))

#
```

to move into the expression all you need to give is the number of the node in the list.

```
#3 p
c
#
```

There are a large number of commands available within the editor. These editing commands can be grouped into three types: global activities, moving in the structure, and changing the structure. They

are listed here in summary form.

Global Commands

e X Evaluate X from within the editor.

lp (X) Repeat the editing command(s) X until an error.

ok Leave the editor after a satisfactory edit.

p Print from the current position to a maximum depth given by the variable **editplev**. This is initially set to 3.

pp Prettyprint from the current position.

save Leave the editor while remembering where in the structure one is, so a subsequent re-entry comes back to the same point.

stop Leave the editor without instantiating any changes.

undo Undo the last editor command, including undo. (Note that !*undo undoes everything whatever the number of changes - including undoing undos. It can therefore go on forever).

? Print the current position using the careful list printing fuction **printl**.

?? Print the undolist that is preserved on a save exit.

Moving Within the Structure

bk Move back to the previous expression at the same level.

f X Find the first occurrence of X in print order and set that as the current position.

n Change the current position to be the *n* th element of the top level list from the current position, where *n* is an integer. If *n* is negative, then the counting is from the right end. A value of 0 returns to the next highest level.

nx Move to the next expression at the same level.

up Back up one level.

0 Go back to the next highest level.

Go back to the start of the structure.

Changing the Structure

All these commands work from the current position.

(*n* **)** For positive values of *n* delete the nth object in the current list

bi *n m* Insert a left bracket before the *n* th element and a right bracket after the *m* th element, so moving them both in.

bo *n* Remove the brackets from the *n* th element, so moving it both out.

delete X

Delete the first occurrence of X.

delete (X **thru** Y)

A multiple form of delete.

insert X **after** Y

Insert the structure X after the first occurrence of the structure Y. X may be a series of items, as in

```
insert a b c after d
```

li *n* Put a left bracket before the *n* th element of the current list and a right bracket at the end.

lo *n* Remove a left bracket from before the *n* th element. This may lose the rest of an expression

move X **to before** Y

move X **to after** Y

Move parts of structures around.

replace X **with** Y

Replace the first occurrence of X with Y.

sw X Y Swop the first X for the first Y.

ri *n m* Move the right bracket of the *n* th element in to after its *m* th element.

ro *n* Move the right bracket of the *n* th element to the end of the current expression.

3.4 Prettyprinter

To invoke the prettyprinter, you use the function **prettyprint**. **prettyprint** takes any LISP expression and prints it in an indented form. The prettyprinter uses certain other functions starting with **pp-**, but these functions are not for general use.

The prettyprinter makes code intelligible. A program written in the default unindented style can be fairly meaningless to everyone; code that is indented properly can be clear to everyone. The prettyprinter is hence a useful aid to understanding obscure code. Since it is easy to mis-bracket a LISP function when typing it in, you should look at the pretty-printed form to check that it was what you intended.

You can use the prettyprinter almost everywhere. One major use is as a circular list printer (for further details, see **prinl**, **printl**, and **princl** in Chapter 4, Functions and Variables).

3.5 Debugging

There are two occasions when debugging is necessary. The first is when a program has bugs of the kind that stop it from working. The second is when a program has bugs of the kind that prevent its efficient working.

Checking code for bugs is easiest when the code is interpreted. The general rule is: if it works when interpreted, it should work when compiled. Normally, when a you compile a function, the interpreted definition is lost. However, if you set the flag ***savedef** to true, then LISP saves the previous definition on the property list under the tag **savedef**. You can then recover and edit the interpreted version. For example:

```
Input: (setq !*savedef t)
Value: t

Input: (de foo (x) (print x))
Value: foo

Input: (compile '(foo))
** 16 bytes  40 msecs    compiling foo
Value: (foo)

Input: (get 'foo 'savedef)
Value: (lambda (x) (print x))
```

When evaluation of a piece of LISP code fails due to errors, LISP generates a backtrace to show you all the function calls. You can overcome this, as described earlier, by using **errorset** to stay in control and give a graded amount of information. Alternatively, you can set **backgag** to various levels of backtrace information. The levels of **backgag** are 0 (totally silent recovery), 1 (the message only), 2 (message and list of functions), and then in increasing verbosity up to 9 which prints out the whole stack. The higher levels should only be of any interest to system programmers. Notice that you can always exit a backtrace with CTRL-C. (For further details, see the function **setbr** in Chapter 4, Functions and Variables.)

The function **trace** is a useful debugging tool. You apply **trace** to a function that you want to log. **trace** sends information on each time the function is called and the arguments that are associated with it **trace** has the advantage that it works on both interpreted and compiled code, although it is always easier to work with the code in interpreted mode. To cancel **trace**, you use the function **untrace**. (For further details, see the function **gcdaemon** in Chapter 4, Functions and Variables.)

embed provides more detailed information than **trace**. It takes the form:

```
(embed '<oldname> <new function definition>)
```

where "new function definition" has a call to the old name inside. The old definition is then stored and you can recover it with **unembed**. For example, if the value of a certain variable was of interest before and after the use the function 'foo', then it could be embedded like this:

```
(embed 'foo
          '(lambda (x)
             ...
                (setq x (foo ...))
                ...
                x))
```

(The periods (that is, ...) refer here to the details of the 'body'.)

Normally, LISP produces compiled code that is 'safe', that is to say, all the **car**s and **cdr**s have been checked to see if their operation is legal. This involves a cost (about 6% of the size of the compiled code), and so an option exists to inhibit it.

```
(carcheck n)
```

directs the compiler to check at the level n, where n=0 means no checking at all, n=1 requests checking that is safe, but unable to provide very clear diagnostics when something does go wrong (the overheads for this cost 4 bytes per access), and n=2 checks all is safe and gives full information. Level 0 is clearly dangerous. At level 0 programmer errors can lead to system corruption.

You can also gauge and tune the efficiency of a program with the functions **profile** and **mapstore**. A call to **profile** generates code that collects certain statistics. At level 1, **profile** causes counts to be collected at the entry point to each routine. At level 2, **profile** causes counts to be collected near each conditional branch and each label in the compiled code. You can then access the statistics with the function **mapstore**. By analysing the numbers of calls, etc., you can see what part of your program is inefficient. (For further details, see both **count** and **speak** in Chapter 4, Functions and Variables.)

Chapter 4: Functions and Variables

4.1 Introduction

This chapter lists all the functions and variables in Cambridge Lisp with a description of the format of each in a header. Each formal parameter has a name suffixed with its allowed type. Lower case tokens are names of classes and upper case tokens are parameter names referred to in the definition. The type of the value returned by the function (if any) is suffixed to the parameter list. If it is not commonly used the parameter type may be a specific set enclosed in brackets {...}. For example:

(**putd** FNAME:id, TYPE:ftype, BODY:{lambda function-pointer}):id

where FNAME is name of the function being defined, TYPE is the type of the function being defined and BODY is a lambda expression or a function-pointer. **putd** returns the name of the function being defined.

Functions which accept formal parameter lists of arbitrary length have the type class and parameter enclosed in square brackets indicating that zero or more occurrences of that argument are permitted. For example:

and([U:any]):extra-boolean

and is a function that accepts zero or more arguments which may be of any type. The type of the function is specified after the header. An EVAL type function receives its arguments ready evaluated and a NOEVAL function receives unevaluated arguments. Normally this is dealt with by the system, but the distinction is important when you use **apply** as you must ensure that the arguments are already in the correct form. SPREAD type functions have their arguments passed in one-to-one correspondence with their formal parameters. NOSPREAD functions receive their arguments as a single list.

Finally, the header line shows the module of the system where a function resides. Base means the function is part of the initial load; Core means it is always loaded, but is defined in LISP. Other names are modules that are found in the IMAGE directory.

The first part of this section describes argument types; the second part the available functions and variables. Part three is a list of unavailable names that are reserved because of their internal use. The final part is a list of error messages with their corresponding numbers.

4.2 Argument Types

Note that functions do not necessarily give an error with an argument of a type other than that specified but the results should not be relied on.

alist — A list with each member being a dotted pair. That is, ((a.b)(c.d) ...).

atom — Any type of number, string, id, vector, or compiled function.

boolean — The set of global variables **t** and **nil**, or their values **t** and **nil**.

constant — All atoms except ids.

extra-boolean — Any value in the system, all except **nil** having the interpretation **t**.

filename — This is an AmigaDOS filename of up to 255 characters in length.

ftype — The class of definable function types. The ids **expr**, **fexpr**, **subr**, **fsubr** **macro** and **mmacro**.

function — Anything that can be used as a function, for example, lambda expression, pointer to binary code, id which is defined as a function.

id — Equivalent to the normal LISP atom, with a property list and value so it can be bound or assigned to. Numbers, vectors etc. are treated specially as they do not need this full mechanism and so cannot be used where an id is specified.

logical — Integers that can be represented in one word, that is, less than 2**24. You can pass arguments of this type to any function that expects a number or integer, but the type of the result may not necessarily be a logical value.

sinteger — Signed integers that can be represented in one word, that is, modulus less than 2**24. You can pass arguments of this type to any function that expects a number or integer, but the type of the result may not necessarily be a logical value. The compiler uses them extensively, and there is a class of functions to handle them.

integer — Signed integers that can be of any size.

n-mod-p — Integers reduced mod p where p is set by **setmod**.

number — All items of type integer, floating, rational.

string — A string of characters enclosed in " ".

4.3 Functions and Variables

Use the character . in the input notation for lists, and if a and b are any structures, (a . b) represents a dotted-pair with a as its **car** and b as its **cdr**. To use the atom '.' see the entries under **!** and **period**.

(
You use brackets in LISP input to form lists. To use the atom (see the entries under **!** and **lpar**.

{
The curly brackets are used as super parentheses. A } closes sufficient brackets to reach either a { or top level, whichever comes first.

!
! is the default escape character, which causes the following character to be treated as an ordinary letter. This means that characters with special properties, such as (or ., can be used as part of an identifier. See **setsyntax** for how to change the escape character and for a list of characters that have special properties initially.

cr
Value is carriage return.

eof
readch returns a special marker when it reaches the end of a stream. The initial value of **eof** is this value, and so a program might read:

```
(setq A (readch))
(cond ((eq A $eof$) (go ENDSTREAM)))
```

eol
readch returns a newline character object when it reads to the end of a line. The initial value of **eol** is this atom. (prin eol) therefore has the effect of **terpri**.

ff
Value is a form feed character object.

***echo**
The variable ***echo** controls echoing of input within **read**. If the value of ***echo*** is

nil	no echo
pretty	formatted printing as reading progresses
t	character by character echo

The calls to **read** made by the LISP supervisor (read/eval/print loop) have ***echo** controled by (prarg n). Initial value is 0. See **prarg**.

***comp**
Type: Variable
When the variable ***comp** is true, function definitions are compiled. See **de**, **df** and **dm**. Default is **nil**.

***lower**
Type:Variable
When true all uppercase letters are translated to their lowercase equivalent.

***nolinke**
When true, the compiler does not generate fast link-and-return instructions. This makes tracing and debugging easier, but there is a cost in time and store.

***ord**
Type: Variable
Normally the compiler calculates the arguments to a function call in an order chosen by the compiler. If ***ord** is true, then the compiler uses a strict left to right order.

***pgen**
Type: Variable
When true the compiler generates an assembly listing. The form of the listing is not compatible with standard assembler, and is only for checking purposes.

***plap**
Type: Variable
The compiler generates an intermediate macro form that is machine independent. When ***plap** is true this form is printed.

***pwrds**
Type: Variable
Controls the printing of a size and time message from the compiler.

***savedef**
Type: Variable
Normally when a function is compiled the interpreted definition is lost. If ***savedef** is true then the previous definition is saved on the property list under the tag **savedef**.

***symmetric**
Type: Variable
If this is true the form of printing produced by **superprint** and **prettyprint** is suitable for entry as input. Initial value is **t**.

%
Comments are (by default) introduced just by **%**, and they last until the end of a line. Since they are implemented by a **splice** readmacro the character that introduces comments can be changed (see **setsyntax**). The function that swallows comments is called **comment-read-macro-function**, but of course any user defined function that reads characters to the end of line works just as well.

(**abs** U:number):number
Type: EVAL, SPREAD, Base
Returns the absolute value of its argument.

(**acons** U:any V:any):list
Type: EVAL, SPREAD, Base
Like **cons** (c.f.) but produces a different internal tag. This can be checked with the functions **aconsp** or **constype**.

(**aconsp** U:any):boolean
Type: EVAL, SPREAD, Base
Check if U is a list created by **acons**.

(**add1** U:number):number
Type: EVAL, SPREAD, Base
Returns the number U incremented by 1. Equivalent to, but faster than, a call to (plus U 1). See also **sub1**.

(**and** [U:any]):extra-boolean
Type: NOEVAL, NOSPREAD, Base
and evaluates each U until a value of **nil** is found or the end of the list is encountered. If a non-**nil** value is the last value it is returned, otherwise **nil** is returned. For bit-wise comparisons see **logand** and **logor**.

(**append** U:list V:list):list
Type: EVAL, SPREAD, Core
Returns a constructed list in which the last element of U is followed by the first element of V. The list U is copied, V is not.

(**apply** FN:{id function} ARGS:any-list):any
Type: EVAL, SPREAD, Base
FN must be a function in the form of a code pointer or lambda expression, or else an id which has been defined as a function. ARGS must be a list of arguments in a form ready to be bound to the formal parameters of FN (that is, if FN expects evaluated arguments then they must be already evaluated). The result of evaluating FN with the values given in ARGS bound to its formal parameters is returned. If ARGS contains more items than FN has formal parameters, then the excess items are ignored, and if ARGS has fewer items the excess formal parameters are set to **nil**.

(**arccos** U:number):number
Type: EVAL, SPREAD, Base
arccos calculates the arc cosine of the argument which must be numeric. The answer is a floating point number and is expressed in radians.

(**arcsin** U:number):number
Type: EVAL, SPREAD, Base
As **arccos** except the function is arc sine.

(**ascii** CODE:integer):character
Type: EVAL, SPREAD, Base
Returns the character corresponding to the given internal code, for example, (**ascii** 48) returns the character !0.

(**assoc** U:any V:alist):{dotted-pair nil}
Type: EVAL, SPREAD, Base
If U occurs as the **car** portion of an element of the alist V, the dotted-pair in which U occurred is returned, else **nil** is returned.

(**atan** U:number):number
Type: EVAL, SPREAD, Base
As **arccos** except the function is arc tangent.

(**atom** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is an atom, including any type of constant (see **constantp**). If **atom** is true, then **car** or **cdr** would be illegal.

(**atsoc** U:any V:alist):{dotted pair nil}
Type: EVAL, SPREAD, Core
atsoc is exactly like **assoc** except that it uses **eq** to test if a tag has been found instead of **equal**. See also **sassoc**.

(**avl-add** KEY:any TREE:avltree ORDER:function):avltree
Type:EVAL, SPREAD, LISPAVLT
The KEY is added to the AVL balanced tree, using ORDER as the ordering predicate. ORDER must be an EVAL/SPREAD function of two arguments. This form of adding to the tree uses an **equal** test. To print the tree in order see **values-in-tree**.

(**avl-delete** KEY:any TREE:avltree ORDER:function):avltree
Type: EVAL, SPREAD, LISPAVLT
The KEY is deleted from the AVL tree.

(**avl-lookup** KEY:any TREE:avltree ORDER:function):boolean
Type: EVAL, SPREAD, LISPAVLT
The KEY is looked up on the tree using an **equal** test.

(**avlq-add** KEY:any TREE:avltree ORDER:function):avltree
Type: EVAL, SPREAD, LISPAVLT
The KEY is added to the AVL balanced tree, using ORDER as the ordering predicate. ORDER must be an EVAL/SPREAD function of two arguments. This form of adding to the tree uses an **eq** test. To print the tree in order see **values-in-tree**.

(**avlq-delete** KEY:any TREE:avltree ORDER:function):avltree
Type: EVAL, SPREAD, LISPAVLT
The KEY is deleted from the AVL tree.

(**avlq-lookup** KEY:any TREE:avltree ORDER:function):boolean
Type: EVAL, SPREAD, LISPAVLT
The KEY is looked up on the tree using an **eq** test.

(**backgag** U:integer):integer
Type: EVAL, SPREAD, Core
You can control the amount of information printed after an error by the use of **backgag**. This function takes one argument, which sets the level of printing desired. Arguments should be integers in the range 0 to 5: 0 means no notification of errors at all; 1 gives just the header message ERROR: ; 2 in addition notes the names of functions being obeyed. Codes 3, 4 and 5 give progressively fuller printing of variables found on the stack. **nil** is treated as meaning 0, and any argument out of range is treated as 5, the default level of printing. The value of this function is the previous value.

bcons
See **acons**.

bconsp
See **aconsp**.

(**bcplbacktrace**)
Type: Base
Prints map of stack in format useful to system programmers.

(**bcplread**):any
Type: Base
A simple read function, used for the bootstrap process. Its use is not recommended.

blank
The atom **blank** has an initial value that is the character blank or space. To test if ch is a space, you can either go: (eq ch blank) or (eq ch (quote !)). See entry under ! for further explanation of the above.

blank-internal-code
Variable containing the ASCII value of **blank**.

(**boundp** U:id):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is the name of a variable that has either been bound by **prog** or as an argument of a function, or has been given a value by **set**.

break-character
Flag used by the **setsyntax** function.

(**breakp** U:any):boolean
Type: EVAL, SPREAD, Base
breakp tests its argument to see if it is a character such as a dot, bracket or blank that would terminate an atom. See also **digit, liter**.

caaaar
Type: EVAL, SPREAD, Core
Any name of the form **c**XXXX**r** where the Xs represent the characters a or d is treated as a combination of the basic functions **car** and **cdr**. Thus (**caddr** U) is equivalent to (**car** (**cdr** (**cdr** U))). The Amiga implementation allows up to four letters between the **c** and the **r**, so **caaaar** to **cddddr** are provided for.

caaadr
see **caaaar**.

caaar
see **caaaar**.

caadar
see **caaaar**.

caaddr
see **caaaar**.

caadr
see **caaaar**.

caar
see **caaaar**.

cadaar
see **caaaar**.

cadadr
see **caaaar**.

cadar
see **caaaar**.

caddar
see **caaaar**.

cadddr
see **caaaar**.

caddr
see **caaaar**.

cadr
see **caaaar**.

call-library (LIBPTR:integer OFFSET:number, Registers. vector):integer
LIBPTR is a pointer to a library returned from a call to openlibrary by **exec**. OFFSET is an offset in the library and Registers is a vector containing values to be placed in A0 - A3, D0 - D3 respectively. The result is the value of D0.

(**car** U:dotted-pair):any
Type: EVAL, SPREAD, Base
car(**cons** a b) ==> a. The left part of U is returned. The type mismatch error occurs if U is not a dotted-pair.

(**car-nil-legal** U:boolean):boolean
Type: EVAL, SPREAD, Base
After a call of (car-nil-legal t) the forms (car nil) and (cdr nil) (which would both normally result in errors) evaluate to **nil**. This facility is provided to ease the problems of transferring certain Interlisp code to the Cambridge system. See also the options given on entry to Lisp and MacLisp.

(**carcheck** N:integer):integer
Type: EVAL, SPREAD, LISPCOMP
Normally compiled code is produced that is safe, in that it checks to see that all **car** and **cdr** operations are legal. This involves a cost (about 6% of the size of compiled code), and so an option for inhibiting it is provided. (**carcheck** n) directs the compiler to check at level n, where n=0 means no checking (and SO PROGRAMMER ERRORS CAN LEAD TO SYSTEM CORRUPTION); n=1 requests checking that is safe, but unable to provide very clear diagnostics when something does go wrong (overhead = 4 bytes per access); and n=2 is both safe and informative. The previous level of checking is returned.

(**casego** U:any [V:(any . label)])
Type: NOEVAL, NOSPREAD, Base
A case is a list of the form (value label) where both value and label are atoms. U is evaluated (once) and its value compared in turn against the value in each case. If a match is found control is transferred to the label, as if a (**go** label) had been obeyed. If none of the values match, the entire **casego** construction is taken to have the value **nil**, and no transfer of control occurs. The value and label of each case are not evaluated and so must be literal values. **casego** must occur in a context where **go** would be legal.

(**catch** TAG:id EXP:any FAIL:any)
Type: NOEVAL, SPREAD, Base
catch provides a method of non-local transfer of control. EXP is evaluated; if while doing this **throw** is used with the tag TAG then **catch** exits with the value of the **throw** (or the value of FAIL if given).

ccons
see **acons**.

cconsp
see **aconsp**.

cdaaar
see **caaaar**.

cdaadr
see **caaaar**.

cdaar
see **caaaar**.

cdadar
see **caaaar**.

cdaddr
see **caaaar**.

cdadr
see **caaaar**.

cdar
see **caaaar**.

cddaar
see **caaaar**.

cddadr
see **caaaar**.

cddar
see **caaaar**.

cdddar
see **caaaar**.

cddddr
see **caaaar**.

cdddr
see **caaaar**.

cddr
see **caaaar**.

(**cdifference** U:n-mod-p V:n-mod-p):n-mod-p
Type: EVAL, SPREAD, Base
The result is U-V, with all numbers reduced mod p in the range [0,p-1]. The number p is set by **setmod**.
See also **mdifference**.

(**cdr** U:dotted-pair):any
Type: EVAL, SPREAD, Base
cdr(**cons** a b) = = > b. The right part of U is returned. The type mismatch error occurs if U is not a dotted-pair.

(**changetype** L:list T:integer):list
Type: EVAL, SPREAD, Base
changetype causes the pair L to be of type determined by T. T = 0 gives a normal **cons** type; T = 1 gives an **acons**; . . . , T = 8 gives an **hcons**. Values of T outside the range [0,8] give an error.

(**character** N:integer):integer
Type: MACRO, LISPSPRI
character gives the character corresponding to the integer N by reading the **character-atom-table**.

character-atom-table
A table of characters used by the reader and **character**. It translates from internal code to ASCII, and allows synonyms.

(**clearbuff**):nil
Type: Base
Routine with no arguments that clears an internal buffer BOFFO that is used for constructing atoms in. Must be called before an attempt is made to use **pack** etc. See **pack, numob, mkatom**.

(**close** FILE:any):any
Type: EVAL, SPREAD, Base
Closes the file with file-handle FILE, releasing store used for buffers and control blocks. FILE can refer to a member of a directory (see **open**). **nil** is returned. An error occurs if the file cannot be closed.

(**cminus** U:n-mod-p):n-mod-p
Type: EVAL, SPREAD, Base
Returns -U mod p. See **cdifference**.

(**cmod** U:integer):n-mod-p
Type: EVAL, SPREAD, Base
Reduces the integer U mod p in the range [0,p-1]. The integer p is usually but not always a prime number. It is set by **setmod**.

(**codep** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a pointer to compiled code.

comma
Initial value of **comma** is the atom ,. See description of **blank** and **$**. **comma** is also used in **read** following ` (backquote) where **comma** introduces a non quoted object.

comma-internal-code
Contains the internal ASCII code for comma.

(**compile** U:id-list):id-list
Type: EVAL, SPREAD, LISPCOMP
compile takes a list of names of functions and compiles them. See also **carcheck, *pgen, *plap, profile, *pwrds**. Example: (compile '(FUN1 FUN2 FUN3))

(**compress** U:id-list):{atom}-{vector}
Type: EVAL, SPREAD, LISPREAD
U is a list of single character identifiers which is built into a Standard LISP entity and returned. Numbers, strings, and identifiers with the escape character prefixing special characters are recognized. Identifiers are interned on the OBLIST. If an entity cannot be parsed out of U or characters are left over after parsing an error occurs.

(**comprop** U:id-list PROPNAME:id):id-list
Type: EVAL, SPREAD, LISPCOMP
For each id in U its property with name PROPNAME is compiled. These properties must be lambda expressions.

(**conc** [U:anylist]):any-list
Type: NOEVAL, NOSPREAD, Base
The lists passed to **conc** are concatenated by modifying the structures, so not using up store. **nconc** is similar to **conc** but only allows for two arguments. See also **append**.

(**cond** [U:cond-form]):any
Type: NOEVAL, NOSPREAD, Base
A cond-form is a list of the form (predicate expression ... expression). The predicate of each U is evaluated until a non-**nil** value is encountered. The sequence of expressions following this predicate are evaluated and the value of the last one becomes the value of **cond**. If all the predicates evaluate to **nil** then the value of **cond** is **nil** and if no expressions follow a predicate, the value returned if this predicate succeeds is the value of this predicate.

(**cons** U:any V:any):dotted-pair
Type: EVAL, SPREAD, Base
Returns a dotted-pair which is not **eq** to anything preexisting and has U as its **car** part and V as its **cdr** part.

(**consp** U:any):boolean
Type: EVAL, SPREAD, Base
See **aconsp**.

constant
If an identifier is flagged with **constant** the compiler presumes that its value never changes, and so picks up that value at compile time. Initially **nil**, **t**, **blank**, **rpar**, etc., are constants.

(**constantp** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a constant (a number, string, function-pointer, or vector).

(**constype** U:list):integer
Type: EVAL, SPREAD, Base
constypB, ... , hcons.

(**copy** U:any):any
Type: EVAL, SPREAD, Core
copy takes a list and returns one that has the same gross structure, but which does not share store with the original. The function fails if the list to be copied has been made cyclic through the use of **rplaca/d**. **copy** does not duplicate atoms or vectors: the copying operation is confined to lists as made up using **cons**.

(**cos** U:number):number
Type: EVAL, SPREAD, Base
As **arccos** except the function is cosine.

(**cot** U:number):number
Type: EVAL, SPREAD, Base
As **arccos** except the function is cotangent.

(**count** N:integer):integer
Type: EVAL, SPREAD, Base
After (count n) has been obeyed, LISP allows about n **cons** operations (or equivalent store-consuming actions) and then gives an error saying 'CONS COUNTER OVERFLOW'. Note that the counting is VERY approximate, and that generally n should have a value of several thousand. **count** returns as its value the previous value of the **cons** counter. If a count of zero is established, the conscounter trap is disabled. See **speak**. **count** is intended for use with **errorset** when trying out code that may run berserk.

(**cplus** U:n-mod-p V:n-mod-p):n-mod-p
Type: EVAL, SPREAD, Base
Returns U + V mod p. See **cdifference**.

(**cquotient** U:n-mod-p V:n-mod-p):n-mod-p
Type: EVAL, SPREAD, Base
Returns the quotient of U and V mod p. See **cdifference**.

(**crecip** U:n-mod-p):n-mod-p
Type: EVAL, SPREAD, Base
Returns the reciprocal of U mod p. See **cdifference**

(**ctimes** U:n-mod-p V:n-mod-p):n-mod-p
Type: EVAL, SPREAD, Base
Returns U*V mod p. See **cdifference**.

dash
The value of the atom **dash** is initially the character -, a dash or minus sign.

(**date**):string
Type: EVAL, SPREAD, Base
This returns a string giving the current date.

dcons
see **acons**.

dconsp
see **aconsp**.

(**de** NAME:id PARAMS:{id id-list} FN:any):id
Type: NOEVAL, NOSPREAD, Base
The function FN with formal parameter(s) specified by PARAMS is added to the set of defined functions with the name NAME. Any previous definitions of the function are lost. The function is left unchanged unless the ***comp** variable is **t** in which case the expression FN is compiled. The name of the defined function is returned.

(**declare** U:id-list):nil
Type: EVAL, SPREAD, LISPCOMP
You can use this to warn the compiler that the functions in the id-list will eventually be special forms (**df/lambdaq**). You only need **declare** when you reference such functions before you define them, and such forward references are not recommended anyway. **declare** works by giving the listed functions dummy definitions, and so overwrites any existing definitions they have.

(**define** U:dlist):id-list
Type: EVAL, SPREAD, Base
A "dlist" is a list in which each element is a two-element list: (ID:id DEF:any). Each ID in U has the definition DEF placed in its value as a lambda/lambdaq expression. **define** is really only useful in an *evalquote* system. It is easier to use **de**. The value of **define** is a list of the first elements of each two element list.

(**deflist** U:dlist IND:id):id-list
Type: EVAL, SPREAD, Base
A "dlist" is a list in which each element is a two element list: (ID:id PROP:any). Each ID in U has the indicator IND with property PROP placed on its property list by the PUT function. The value of **deflist** is a list of the first elements of each two element list. Like **put, deflist** may not be used to define functions.

(**defprop** U:dlist IND:id):list
Type: EVAL, SPREAD, Core
As **deflist** but tries to compile the properties before placing them on the property list. No error occurs if a property is not compilable.

(**deleq** U:any V:list):list
Type: EVAL, SPREAD, Core
As **delete** but uses **eq** rather than **equal** as the test.

(**delete** U:any V:list):list
Type: EVAL, SPREAD, Core
Returns V with the first top level occurrence of U removed from it.

(**denq** N:number):integer
Type: EVAL, SPREAD, Base
If N is a LISP number, its denominator is returned. If N is not rational its denominator is the integer 1. See **numq**.

(**df** NAME:id PARAM:{id id-list} FN:any):id
Type: NOEVAL, NOSPREAD, Base
The function FN with formal parameter(s) specified by PARAM is added to the set of defined functions with the name NAME. Any previous definitions of the function are lost. The function created is of type **fexpr** unless the ***comp** variable is **t** in which case the expression FN is compiled and an **fsubr** is created. The name of the defined function is returned.

(**difference** U:number V:number):number
Type: EVAL, SPREAD, Base
Returns U - V.

(**digit** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a digit, otherwise **nil**. Note that a digit is a character and not a number. (that is, U = !2 returns **t** but U = 2 returns **nil**).

(**divide** U:number V:number):dotted-pair
Type: EVAL, SPREAD, Base
The dotted-pair (quotient . remainder) is returned. The quotient part is computed the same as by **quotient** and the remainder the same as by **remainder**. An error occurs if division by zero is attempted.

(**dm** MNAME:id PARAM:{id id-list} FN:any):id
Type: NOEVAL, NOSPREAD, Base
The macro FN with formal parameter(s) specified by PARAM is added to the set of defined functions with the name MNAME. The result of the macro should be an expression to be evaluated. Any previous definitions of the function are overwritten. The function created is of type macro and the name of the macro is returned. If ***comp** is true then the macro is compiled.

(**dmm**) MNAME:id PARAM:{id id-list} FN:any):id
Type: NOEVAL, NOSPREAD, Base
Format as above. Although **dmm** is essentially similar to **dm**, it follows the style of MACLISP macros and has been included for purposes of compatibility with MACLISP.

dollar
The initial value of **dollar** is the character $.

dollar-internal-code
Contains the internal ASCII code for dollar.

(**dumpcore**)
Type: Base
This is a system-programmers function used to print the internal state of LISP memory. It can be expected to print several thousand lines of information that are neither useful nor comprehensible to the normal user.

(**dumpfile** NAME:id):nil
Type: EVAL, SPREAD, Base
Normally the directory identified as DUMP (or IMAGE) is used for preserving the state of LISP. This can be changed by use of **dumpfile**.

econs
see **acons**

econsp
see **aconsp**

(**editf** FN:id):any
Type: NOEVAL, SPREAD, LISPEDIT
editf enters the structure editor to edit the function defined with the name ID. For more details of the editor see Section 3.3. If the editor is entered with no argument then the last edit is resumed, either from the start if the edit was left via **stop** or **ok**, or at the same place if the last edit was left by **save**.

(**editp** U:id):any
Type: NOEVAL, SPREAD, LISPEDIT
editp enters the structure editor to edit the property list of the identifier U. For more details of the editor see Section 3.3 and **editf**.

(editv U:id):any
Type: NOEVAL, SPREAD, LISPEDIT
editv enters the structure editor to edit the value of the identifier U. For more details of the editor see Section 3.3 and **editf**.

(eject):nil
Type: Base
Causes a skip to the top of the next output page if the destination supports carriage controls.

(embed NAME:id NEWDEF:function)
Type: EVAL, SPREAD, Core
embed is used to provide more detailed tracing than is available with **trace**. The definition of the function NAME is replaced by NEWDEF, where this new definition may contain calls to the current value. The old definition is stored and can be recovered by **unembed**. For example if the global variable GLOB was of interest before and after the function CHANGE one could use the following to get the information:

```
(embed 'CHANGE (lambda (X)
    (print (list "Value of GLOB on entry:" GLOB))
    (CHANGE X)
    (print (list "changed by CHANGE to:" GLOB))))
```

emsg*
Is set by errors to the error message produced, so is useful after **errorset** catches the error.

(endmodule)
see **module**.

eol-internal-code
Variable containing the ASCII value of **eol**.

(eq U:any V:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U points to the same object as V (it tests for equal pointers). **eq** is not a reliable comparison between numeric arguments in general, but is correct for integers with absolute value <2**24. For portability this should not be relied on.

(eqcar U:any V:any):boolean
Type: EVAL, SPREAD, Core
This is equivalent to (eq (car U) V) except that if U is atomic it returns the answer **nil**.

(eqn U:any V:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U and V are **eq** or if U and V are numbers and have the same numeric value after any necessary conversions have been performed (see Section 2.4). Floating point numbers are **eqn** only if they have, bit for bit, the same internal representation. There is no allowance made for rounding error, and so **eqn** on floating point arguments should be used only when this precise comparison is wanted.

eqsign
The initial value of **eqsign** is =.

(**equal** U:any V:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U and V are the same. Dotted-pairs are compared recursively to the bottom levels of their trees. Vectors must have identical dimensions and equal values in all positions. Strings must have identical characters. Function pointers must have **eq** values. Numbers are not **equal** if their types differ, for example, (equal 1 1.0) = nil.

(**error** NUMBER:integer MESSAGE:any)
Type: EVAL, SPREAD, Base
NUMBER and MESSAGE are passed back to a surrounding **errorset** (the Standard LISP reader has an **errorset**). MESSAGE is placed in the global variable **emsg*** and NUMBER becomes the value of the **errorset**. **error** can be called with a single argument which becomes the message, the error number defaulting to zero. Fluid variables and local bindings are unbound to return to the environment of the **errorset**. Global variables are not affected by the process.

error-count
Count of the errors detected by the **supervisor**.

(**errorset** U:any FLAG:integer):any
Type: EVAL, SPREAD, Base
A LISP program can call **eval** to get a fragment of code explicitly evaluated. If this is done, however, errors in the code cause a complete backtrace of all the functions being executed. **errorset** is a variant on **eval** that overcomes this problem, and allows the user to obey code while keeping control if the code turns out to be faulty. The result returned is the same as (list (eval U)) if the evaluation works, but if there is an error the result returned is atomic and is a number identifying the type of error that occurred. (see **error**). The value of FLAG determines how much information about the error is reported to the user. A value of zero results in no notification about the error and values 1 to 5 result in progressively more information about the functions being obeyed and the variables on the stack being printed.

(**eval** U:any):any
Type: EVAL, SPREAD, Base
U is evaluated as a piece of LISP code with respect to the current collection of variable bindings. **eval** is the function used by the LISP interpreter to evaluate LISP code.

(**evenp** U:number):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is an even integer.

(**evlis** U:any-list):any-list
Type: EVAL, SPREAD, Base
evlis returns a list of the evaluation of each element of U.

(**exec** REGISTERS:vector):number
Type: EVAL, SPREAD, Base
exec is used to call the exec library. Number is the offset in the library and registers is a vector containing values to be put in registers D0 - D3, A0 - A3. If a string is passed in the vector it is converted to a machine pointer to an appropriate C string.

(**excise** NAME:{id id-list})
Type: EVAL, SPREAD, Base
LISP has a load-on-call mechanism, and the function **excise** can be used to unload previously loaded modules. If NAME is **nil** then everything is unloaded, otherwise the named module(s) are unloaded. As a particular case, **excise** can be used to recover the store taken up by the LISP compiler, in that (excise 'lispcomp) purges the relevant functions. Note that after **excise** has been called, further reference to the excised module results in it being reloaded.

(**exp** U:number):number
Type: EVAL, SPREAD, Base
As **arccos** except the function is the exponential function

(**expand** L:list FN:function):list
FN is a defined function of two arguments to be used in the expansion of a macro. **expand** returns a list in the form:
(FN L[0] (FN L[1] ... (FN L[n-1] L[n]) ...))
where "n" is the number of elements in L, L[i] is the ith element of L.

(**explode** U:any):id-list
Type: EVAL, SPREAD, Base
Returned is a list of single-character identifiers representing the characters that print as the value of U. The primitive data types have these formats:

integer	Leading zeros are suppressed and a minus sign prefixes the digits if the integer is negative.
floating	The value appears in the format [-]0.nn...nnE[-]mm if the magnitude of the number is too large or small to display in [-]nnnn.nnnn format. The crossover point is determined by the implementation.
id	The characters of the print name of the identifier are produced with special characters prefixed with the escape character.
string	The characters of the string are produced surrounded by double quotes ". . .".
function-pointer	The value of the function-pointer is created as a list of characters conforming to the conventions of the system site.
vector	The elements of the vector are produced surrounded by brackets %<...>%.

The type mismatch error occurs if U is not a number, identifier, string, or function-pointer.

(**explodec** U:any):id-list
Type: EVAL, SPREAD, Base
As **explode** but no escape characters are produced in ids.

(**explodecn** U:any):integer-list
Type: EVAL, SPREAD, Base
As **explodec** but the list is of the internal codes of the characters rather than the characters.

(**explodeN** U:any):integer-list
Type: EVAL, SPREAD, Base
As **explode** but the list is of the internal codes of the characters rather than the characters.

expr
See **fexpr**.

(**expt** U:number V:number):number
Type: EVAL, SPREAD, Base
Returns U raised to the V power, where V cannot be an integer of unlimited precision. (that is, V can be a floating point number or small integer). A floating point U to an integer power V does not have V changed to a floating number before exponentiation.

f
The initial value of **f** is **nil**, and so **f** can be used as a name for 'false'. Unlike some other systems, in Cambridge LISP **f** can be used as a bound variable, and the local binding overrides the global value.

fcons
see **acons**.

fconsp
see **aconsp**.

fexpr
fexpr is not a real part of Cambridge LISP. See **getd**.

fin
The atom fin is used to mark the end of a LISP program, and it is recommended that most files end with the sequence:
fin))))))))))))))))))))))))))))))))
If **fin** is omitted, the system stops on end-of-file, but prints a warning message to this effect.

(**fix** U:number):integer
Type: EVAL, SPREAD, Base
Returns an integer which corresponds to the truncated value of U. The result of conversion must retain all significant portions of U. If U is an integer it is returned unchanged.

(**fixp** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is an integer (a fixed number).

(**flag** U:id-list V:id):nil
Type: EVAL, SPREAD, Base
U is a list of ids to be flagged with V. The effect of flag is that **flagp** has the value **t** for those ids that were flagged. Both V and all the elements of U must be identifiers or the type mismatch error occurs.

(**flagp** U:any V:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U has been previously flagged with V, else **nil**. Returns **nil** if either U or V is not an id.

(**float** U:number):floating
Type: EVAL, SPREAD, Base
The floating point number corresponding to the value of the argument U is returned. Some of the least significant digits of an integer may be lost due to the implementaion of floating point numbers. **float** of a floating point number returns the number unchanged. If U is too large to represent in floating point an error occurs.

(**floatp** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a floating point number and **nil** otherwise.

(**fluid** IDLIST:id-list):nil
Type: EVAL, SPREAD, Core
The ids in IDLIST are declared as fluid type variables (ids not previously declared are initialized to **nil**). Variables in IDLIST already declared fluid are ignored. Changing a variable's type from global to fluid is not permissible and results in an error.

(**fluidp** U:any):boolean
Type: EVAL, SPREAD, Core
If U has been declared fluid (by declaration only) **t** is returned, otherwise **nil** is returned.

(**fntype** U:function):{(ftype . nargs) atom}
Type: EVAL, SPREAD, Base
If U is either a piece of binary code, or a lambda expression, the result returned is a dotted-pair (type . nargs) specifying the type and number of arguments that U requires. The possible types are **expr**, **fexpr**, **subr**, and **fsubr**, and if nargs is given as negative, it means that the function accepts an indefinite number of arguments. If you give fntype a non-functional or malformed argument, it returns some atomic value.

(**for** VAR:id LOW:integer STEP:integer HIGH:integer ACTION:nil BODY:any):nil
Type: MACRO, Core
The BODY is evaluated for VAR having the values LOW, LOW + STEP, . . . , until the value of VAR is greater than HIGH. Note that the ACTION must be **nil**. This form of the **for** macro is for compatibility with certain packages.

(**foreach** VAR:id INON:{**in on**} L:list T:{**do collect conc**} BODY:any):any
Type: MACRO, Core
for provides a method of calling the map functions. The combinations of INON and T are as follows:

on	do	gives the **map** function
on	collect	gives the **maplist** function
on	conc	gives the **mapcon** function
in	do	gives the **mapc** function
in	collect	gives the **mapcar** function
in	conc	gives the **mapcan** function

fsubr
fsubr is not a real part of Cambridge LISP.

(**function** fn:function):function
Type: NOEVAL, NOSPREAD, Base
The function fn is to be passed to another function. If fn is to have side effects its free variables must be fluid or global. **function** is like **quote** (indeed, the interpreter does not distinguish them) but if (function (lambda ...)) occurs in code that is being compiled, the lambda expression is compiled, and indeed may be expanded into in-line code, instead of a separate **subr**.

g
The names chosen to represent generated symbols created by **gensym** are initially of the form **gn** where n is a small integer. The prefix **g** can be changed through the use of **symnam**.

(**gcd** U:integer V:integer):integer
Type: EVAL, SPREAD, Base
The positive integer that is the greatest common divisor of U and V is returned.

(**gcdaemon** N:integer):any
Type: EVAL, SPREAD, user
If you define a function **gcdaemon**, it is called at the end of each garbage collection if it is compiled and not traced. **gcdaemon** is handed one argument, which is the serial number of the garbage collection just completed. The **gcdaemon** facility is inhibited within execution of itself, so garbage collections triggered by work done within **gcdaemon** do not normally lead to re-entry to the function. Error exits from the **gcdaemon** function lead to the function's definition being removed, and so after the deliberate creation of such an error state **gcdaemon** should be redefined.

gcons
see **acons**

gconsp
see **aconsp**

(**gctime**):integer
Type: EVAL, SPREAD, Base
The value returned by **gctime** is the amount of time (in milliseconds) consumed by 'overheads' since the start of the current LISP run. For these purposes, 'overheads' include initial loading of LISP, the dynamic loading of modules and garbage collection.

(**gensym**):id
Type: EVAL, SPREAD, Base
Creates an identifier which is not interned on the OBLIST and consequently not **eq** to anything else.

(**gensym1** U:id):id
Type: EVAL, SPREAD, Base
This generates a symbol in the same way as **gensym** does, but forces the printname of the generated atom to start with the given identifier.

(**geq** U:number V:number):boolean
Type: MACRO, Core
Returns **t** if U >= V.

(**get** U:any IND:any):any
Type: EVAL, SPREAD, Base
Returns the property associated with indicator IND from the property list of U. Returns **nil** if U or IND are not ids. **get** cannot be used to access functions (use **getd** instead).

(**getd** FNAME:any):{nil dotted-pair}
Type: EVAL, SPREAD, Base
If FNAME is not the name of a defined function, **nil** is returned. If FNAME is the name of an x**subr**, then the dotted-pair (xsubr . function-pointer) is returned where the function-pointer is that associated with FNAME. If FNAME is the name of an **xexpr** then the dotted-pair (xexpr . lambda) is returned, the lambda expression being the definition of the function. If FNAME is the name of a macro then the dotted pair (macro . lambda) is returned with lambda being the body of the macro.

(**getv** V:vector INDEX:integer):any
Type: EVAL, SPREAD, Base
Returns the value stored at position INDEX of the vector V. The type mismatch error may occur and an error occurs if the INDEX does not lie within 0...(**upbv** V) inclusive.

(**global** IDLIST:id-list):nil
Type: EVAL, SPREAD, Core
The ids of IDLIST are declared global type variables. If an id has not been declared previously it is initialized to **nil**. Variables already declared global are ignored. Changing a variable's type from global to fluid is not permissible and results in an error.

(**globalp** U:any):boolean
Type: EVAL, SPREAD, Core
If U has been declared global or is the name of a defined function, **t** is returned, else **nil** is returned.

(**go** LABEL:id)
Type: NOEVAL, NOSPREAD, Base
go alters the normal flow of control within a **prog** function. The next statement of a **prog** function to be evaluated is immediately preceded by LABEL. A **go** may only appear in the following situations:

1) At the top level of a **prog** referencing a label which also appears at the top level of the same **prog**.
2a) As the consequent of a cond item of a cond appearing on the top level of a **prog**.
2b) As the consequent of a **cond** item which appears as the consequent of a **cond** item to any level.
3a) As the last statement of a **progn** which appears at the top level of a **prog** or in a **progn** appearing in the consequent of a **cond** to any level subject to the restrictions of 2a,b.
3b) As the last statement of a **progn** within a **progn** or as the consequent of a **cond** to any level subject to the restrictions of 2a,b and 3a.

An error occurs if LABEL does not appear at the top level of the **prog** in which the **go** appears or if the **go** has been placed in a position not defined by the rules. See also **casego**. **go** cannot be used for non-local transfers of control. For such facilities see **throw**.

(**greaterp** U:number V:number):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is strictly greater than V, otherwise returns **nil**.

(**gts** U:id):any
Type: EVAL, SPREAD, Base
This returns the current value of the identifier U, or **nil** if it is unset. **gts** is intended particularly for evaluating non-local variables, where it is a cheap but restricted equivalent to (eval U), except perhaps for its treatment of unset values.

hcons
see **acons**.

hconsp
see **aconsp**.

(iadd1 U:integer**)**:integer
Type: EVAL, SPREAD, Base
Similar to **add1**. All the routines with names iXXXX where XXXX is the name of an arithmetic operation, are index mode operations. They must only be called with arguments that are integers less than 2**24, and must be called in such a way that the result satisfies the same constraints. Failure to adhere to these constraints (for example, overflow conditions, bignum inputs,...) may not be detected and may lead to inconsistent behaviour. The routines do not necessarily check their arguments' types or ranges, but do at least never return a value that does not print as a small number. The LISP compiler can turn these routines into reasonably efficient in-line code, which should be much faster than use of the more general arithmetic routines. It must be stressed that only small numbers are valid and this constraint is not checked by the system.

(idifference U:integer V:integer**)**:integer
Type: EVAL, SPREAD, Base
Similar to **difference**. See **iadd1**.

(idp U:any**)**:boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is an id (that is, an atom that is not a constant).

(igreaterp U:integer V:integer**)**:integer
Type: EVAL, SPREAD, Base
Similar to **greaterp**. See **iadd1**.

(ileftshift U:integer V:integer**)**:integer
Type: EVAL, SPREAD, Base
Similar to **leftshift**, but does not accept a negative second argument. **irightshift** is provided for right shifts. See **iadd1**.

(ilessp U:integer V:integer**)**:integer
Type: EVAL, SPREAD, Base
Similar to **lessp**. See **iadd1**.

(ilogand [U:integer]**)**:integer
Type: NOEVAL, NOSPREAD, Base
Similar to **logand**. See **iadd1**.

(ilogand2 U:integer V:integer**)**:integer
Type: EVAL, SPREAD, Base
Similar to **logand2**. See **iadd1**.

(ilogor [U:integer]**)**:integer
Type: NOEVAL, NOSPREAD, Base
Similar to **logor**. See **iadd1**.

(ilogor2 U:integer V:integer**)**:integer
Type: EVAL, SPREAD, Base
Similar to **logor2**. See **iadd1**.

(ilogxor [U:integer]**)**:integer
Type: NOEVAL, NOSPREAD, Base
Similar to **logxor**. See **iadd1**.

(**ilogxor2** U:integer V:integer):integer
Type: EVAL, SPREAD, Base
Similar to **logxor2**. See **iadd1**.

(**imax** [U:integer]):integer
Type: NOEVAL, NOSPREAD, Base
Similar to **max**. See **iadd1**.

(**imax2** U:integer V:integer):integer
Type: EVAL, SPREAD, Base
Similar to **max2**. See **iadd1**.

(**imin** [U:integer]):integer
Type: NOEVAL, NOSPREAD, Base
Similar to **min**. See **iadd1**.

(**iminus** U:integer):integer
Type: EVAL, SPREAD, Base
Similar to **minus**. See **iadd1**.

(**iminusp** U:integer):boolean
Type: EVAL, SPREAD, Base
Similar to **minusp**. See **iadd1**.

(**imin2** U:integer V:integer):integer
Type: EVAL, SPREAD, Base
Similar to **min2**. See **iadd1**.

input
The atom input is a marker word used in calls to **open**.

(**internalcode** U:id):integer
Type: EVAL, SPREAD, Base
If U is a single character id its ASCII code is returned. See **ascii**.

(**iplus** [U:integer]):integer
Type: NOEVAL, NOSPREAD, Base
Similar to **plus**.

(**iplus2** U:integer V:integer):integer
Type: EVAL,SPREAD, Base
Similar to **plus2**. See **iadd1**.

(**iquotient** U:integer V:integer):integer
Type: EVAL,SPREAD, Base
Similar to **quotient**. See **iadd1**.

(**iremainder** U:integer V:integer):integer
Type: EVAL,SPREAD, Base
Similar to **remainder**. See **iadd1**.

(**irightshift** U:integer V:integer):integer
Type: EVAL, SPREAD, Base
Since **ileftshift** can not accept a negative second argument, this routine is provided. It shifts a (small) number right. See **ileftshift, iplus**.

(**isqrt** U:integer):integer
Type: EVAL, SPREAD, Base
Returns the square-root of U if it is exact, otherwise **nil**.

(**isub1** U:integer):integer
Type: EVAL, SPREAD, Base
Similar to **sub1**. See **iadd1**.

(**itimes** [U:integer]):integer
Type: EVAL, SPREAD, Base
Similar to **times**. See **iadd1**.

(**itimes2** U:integer V:integer):integer
Type: EVAL, SPREAD, Base
Similar to **times2**. See **iadd1**.

(**izerop** U:integer):integer
Type: EVAL, SPREAD, Base
Similar to **zerop**, but does not recognize floating point zero. See **iadd1**. In fact, (izerop xx) = (eq xx 0).

label
The label facility for producing recursive functions is not supported in Cambridge LISP.

lambda
lambda is a marker atom that identifies a piece of LISP structure as representing a function. The correct syntax for its use is

(lambda variables expr1 expr2 ... exprn)

where variables is a list of formal arguments that the function needs, and the expressions are the body of the function. See **lambdaq**.

lambdaq
lambdaq introduces a function that receives its arguments unevaluated. Except for supressing argument evaluation **lambdaq** behaves exactly like **lambda**. The **lambdaq** facility in Cambridge LISP takes the place of **fexpr/fsubr** activity in some other systems.

(**last** U:list):any
Type: EVAL, SPREAD, Base
Returns the last element of the list U; for instance if U is the list (A B C D E), then E is returned. **last** should not be given an atomic argument.

lcurly-internal-code
Variable containing the ASCII value of {.

(**leftshift** U:logical V:integer):logical
Type: EVAL, SPREAD, Base
The 24-bit value U is shifted left by V places, keeping only the bottom 24 bits of the result. If the second argument is negative, a right shift is indicated. See also **logand, logor** and **logxor**.

(**leftmost-key** T:avl-tree):any
Type: EVAL, SPREAD, AVLTREE
Returns the leftmost key and value in an AVL tree. It should be used in conjunction with the other AVL tree functions.

(**length** X:any):integer
Type: EVAL, SPREAD, Core
The top level length of the list X is returned.

(**lessp** U:number V:number):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is strictly less than V, otherwise returns **nil**.

lexpr
See **fexpr**.

(**linelength** LEN:{integer nil}):integer
Type: EVAL, SPREAD, Base
If LEN is an integer the maximum line length to be printed before the print functions initiate an automatic **terpri** is set to the value LEN. The initial linelength is 72 characters. If LEN is **nil** the current linelength is returned but is not reset. Values of the line length less than 24 should not be used.

(**list** [U:any]):list
Type: NOEVAL, NOSPREAD, Base
A list of the evaluation of each element of U is returned.

(**liter** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a character of the alphabet, **nil** otherwise.

(**log** U:number):number
Type: EVAL, SPREAD, Base
As **arccos** except the function is the natural logarithm.

(**logand** [U:logical]):logical
Type: NOEVAL, NOSPREAD, Base
The result returned is the logical **and** of all the arguments, treated as 24 bit quantities. See **logor**, **logxor** and **leftshift** for further operations on 24 bit logical words.

(**logand2** U:logical V:logical):logical
Type: EVAL, SPREAD, Base
logand2 behaves exactly like **logand** expect that it expects exactly two arguments. Compiled references to **logand** get converted into sequences of calls to **logand2**. See also **plus2**, **times2** etc.

(**logor** [U:logical]):logical
Type: NOEVAL, NOSPREAD, Base
logor forms the logical (that is, bitwise) **or** of a sequence of 24 bit values, and is otherwise similar to **logand**.

(**logor2** U:logical V:logical):logical
Type: EVAL, SPREAD, Base
logor, but expecting exactly two arguments. See **logand2**.

(**logp** U:number):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a integer in the range 0 to 2**24-1, that is, if the binary representation of U is at most 24 bits long, so U can be used directly in **logand**, **logor**, etc.

(**logxor** [U:logical]):logical
Type: NOEVAL, NOSPREAD, Base
As **logand** and **logor**, but forms the bitwise exclusive **or** (non-equivalence) of its arguments.

(**logxor2** U:logical V:logical):logical
Type: EVAL, SPREAD, Base
logxor but expecting exactly two arguments.

(**log10** U:number):number
Type: EVAL, SPREAD, Base
As **arccos** except the function is logarithm to the base 10.

lpar
The initial value of lpar is the atom '(', a left parenthesis.

lpar-internal-code
Variable containing the ASCII value of **(**.

(**lposn**):integer
Type: Base
lposn always returns zero. It is intended to record the line on the page.

lsubr
See **fexpr**.

macro
Macros are introduced by calls to **dm**, and subsequently calls of the form (name args) get trapped, and the entire function application passed to the macro definition for processing. See **dm** for details.

macro:
Atom used internally to mark macros.

(**make-constant** NAME:id VALUE:any):id
Type: EVAL, SPREAD, LISPRDMC
The identifier NAME is made into a constant with value VALUE. NAME cannot now be bound as an argument to a function.

(**map** x:list FN:function):any
Type: EVAL, SPREAD, Core
Applies FN to successive **cdr** segments of X. (that is, X, (cdr X), (cddr X)...). **nil** is returned.

(**mapc** X:list FN:function):any
Type: EVAL, SPREAD, Core
FN is applied to successive **car** segments of list X. (that is, (car X), (cadr X), (caddr X)...). **nil** is returned.

(**mapcan** X:list FN:function):any
Type: EVAL, SPREAD, Core
A concatenated list of FN applied to successive **car** segments of X is returned. Note that FN must return a value that is a list for **mapcan** to work.

(**mapcar** X:list FN:function):any
Type: EVAL, SPREAD, Core
A constructed list of FN applied to successive **car** segments of list X is returned.

(**mapcon** X:list FN:function):any
Type: EVAL, SPREAD, Core
A concatenated list of FN applied to successive cdr segments of X is returned. (that is, X, (cdr X), (cddr X)...). Note that FN must return a value that is a list.

(**maplist** X:list FN:function):any
Type: EVAL, SPREAD, Core
A constructed list of FN applied to successive **cdr** segments of X is returned.

(**mapstore** U:boolean)
Type: EVAL, SPREAD, Base
mapstore displays details of all compiled functions present in the LISP heap. If any of these were compiled with the statistics option (see **profile**) the counts are displayed and reset to zero. (mapstore t) displays a similar map of the core LISP system for use by system programmers and the like.

(**max** [U:number]):number
Type: NOEVAL, NOSPREAD, Base
Returns the largest of the values in U. If two or more values are the same the first is returned.

(**max2** U:number V:number):number
Type: EVAL, SPREAD, Base
Returns the larger of U and V. If U and V are the same value U is returned (U and V might be of different types). This function is used in the expansion of **max**.

(**mdifference** U:n-mod-p V:n-mod-p):n-mod-p
Type: EVAL, SPREAD, Base
Returns U - V mod p in the range [-p/2,p/2] where p is set by **setmod**. See also **cdifference** etc.

(**member** A:any B:list):extra-boolean
Type: EVAL, SPREAD, Core
Returns **nil** if A is not a member of list B, otherwise returns the remainder of B whose first element is A. The function **equal** is used to compare list elements.

(**memq** A:any B:list):extra-boolean
Type: EVAL, SPREAD, Core
Same as **member** but an **eq** check is used for comparison.

(**min** [U:number]):number
Type: NOEVAL, NOSPREAD, Base
Returns the smallest of the values in U. If two or more values are the same the first of these is returned.

(**minus** U:number):number
Type: EVAL, SPREAD, Base
Returns -U.

minus-internal-code
Variable containing the ASCII value of the minus character.

(**minusp** U:number):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a negative number, otherwise **nil**.

(**min2** U:number V:number):number
Type: EVAL, SPREAD, Base
Returns the smaller of its arguments. If U and V are the same value, U is returned (U and V might be of different types). This function is used in the expansion of **min**.

(**mkatom**):id
Type: Base
If a sequence of characters have been assembled in the buffer BOFFO (by calls to **clearbuff** and **pack**), **mkatom** can be used to form the LISP identifier made out of the characters. **mkatom** consults LISP's internal hash tables and name lists (see **oblist**) and makes certain that identifiers are defined uniquely by their printnames.

(**mkquote** X:any):list
Type: EVAL, SPREAD, Core
This is a function of one argument and its definition is equivalent to (lambda (X) (list 'quote X)).

(**mkstring**):string
Type: Base
Creates a LISP string from the characters in BOFFO.

(**mkvect** UPLIM:integer):vector
Type: EVAL, SPREAD, Base
Defines and allocates space for a vector with UPLIM+1 elements accessed as 0...UPLIM. Each element is initialized to **nil**. An error occurs if UPLIM is < 0 or there is not enough space for a vector of this size.

mmacro:
A marker for mmacros.

(**mminus** U:n-mod-p):n-mod-p
Type: EVAL, SPREAD, Base
Returns -U mod p. See **mdifference**.

(**mmod** U:integer):n-mod-p
Type: EVAL, SPREAD, Base
Reduces U to range [-p/2,p/2]

(**module** U:id):nil
Type: EVAL, SPREAD, Base
This directs the compiler to put the code that it generates onto a module for later use. The code is written directly to the DUMP directory, if present; otherwise, it is written to the IMAGE directory, as well as being kept in store in case it is needed in some bootstrapping process. The module selected by **module** replaces any previous one with the same name, and empty modules eventually get purged from the system. You should avoid the name **nil** and any starting with the four characters 'LISP.' (**endmodule**) or another call to **module** terminates a module.

(**mplus** U:n-mod-p V:n-mod-p):n-mod-p
Type: EVAL, SPREAD, Base
Returns U + V mod p. See **mdifference**.

(**mquotient** U:n-mod-p V:n-mod-p):n-mod-p
Type: EVAL, SPREAD, Base
Returns the quotient of U and V mod p. See **mdifference**.

(**mrecip** U:n-mod-p):n-mod-p
Type: EVAL, SPREAD, Base
Returns the reciprocal of U mod p. See **mdifference**.

(**mtimes** U:n-mod-p V:n-mod-p):n-mod-p
Type: EVAL, SPREAD, Base
Returns U*V mod p. See **mdifference**.

(**nconc** U:list V:list):list
Type: EVAL, SPREAD, Core
Concatenates V to U without copying U. The last **cdr** of U is modified to point to V.

(**ncons** U:any):dotted-pair
Type: EVAL, SPREAD, Base
Is the same as (cons U nil). c.f. **cons, xcons**.

(**neq** U:any V:any):boolean
Type: EVAL, SPREAD, Core
neq is equivalent to (not (equal U V)).

nil
nil is an identifier that LISP uses in a variety of special ways. It is therefore not possible to use it either as a function name or a variable name. The first special use is that all lists normally terminate with a reference to the atom **nil**, and so (A B C) is 'really' (A B C . nil). The effect of this on the normal programmer is that the test **null**, as used to see if the end of a list has been reached, can be seen to be equivalent to (eq xx nil). The second special use of **nil** is as the standard denotation for 'false'. All LISP predicates return **nil** for false (most return **t** for true). **nil** is used so often in LISP programs that it has been defined to stand for itself, and so it is possible to write (cons a nil) rather than (cons a (quote nil)).

(**not** U:any):boolean
Type: EVAL, SPREAD, Base
If U is **nil**, return **t** else return **nil** (same as **null** function).

(**null** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is **nil**.

(**numberp** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a number (integer, rational or floating).

(**numob**):number
Type: Base
If the characters placed in the buffer BOFFO (by **pack** etc) represent a number, **numob** forms that number as a proper LISP object. If **mkatom** (q.v.) had been called instead of **numob** it would have created an identifier that had the same print format as a number but which had the properties of a variable rather than of a number.

(**numq** U:number):integer
Type: EVAL, SPREAD, Base
Returns the numerator of the number U. If U is not a rational the value returned is the argument. Giving **numq** a non numeric argument is an error.

(**oblist**):id-list
Type: Base
Returns a lexigraphically ordered list of all the atoms that can be reached by **read**. However it should be noted that the 'object list' in Cambridge LISP is a collection of balanced trees, so **rplac** operations on the value of **oblist** have no effect on the object list.

(**onep** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is the number 1 or 1.0. There is no error if the item is not numeric. The effect is like (eqn U 1).

(**open** FILE:any HOW:id):any
Type: EVAL, SPREAD, Base
Open the file with the name FILE for output if HOW is **eq** to **output**, or input if HOW is **eq** to **input**. Note that a file may be at most 255 characters (excluding any escape characters). After calls to **pdsinput** and **pdsoutput**, FILE is treated as the name of a member of the directory specified by these calls. Alternatively, FILE can be a list consisting of the name of the directory and the membername required. If a third argument, of the same form as FILE is given to **open**, the effect is the same as **close** performed on the third argument, followed by **open** performed on the first. If the file is opened successfully a file-handle is returned. This handle should be used to refer to the file when using other I/O routines. An error occurs if HOW is something other than **input** or **output** or the file cannot be opened.

(**or** [U:any]):extra-boolean
Type: NOEVAL, NOSPREAD, Base
U is any number of expressions which are evaluated in order of their appearance. When one is found to be non-**nil** it is returned as the value of **or**. If all are **nil**, **nil** is returned.

(**orderp** U:any V:any):boolean
Type: EVAL, SPREAD, Base
This is a predicate defining a self-consistent order relation between lists. If U and V are identifiers this relation reduces to alphabetic order. For more complex structures the details of the ordering should not be relied on.

output
A flag value, used as the second argument to **open**.

(**outradix** RADIX:integer):previous radix
Type: EVAL, SPREAD, Base
Sets the radix that will be used when printing subsequent integer values. Legal arguments are 2, 8, 10 and 16 (decimal). Note that only small numbers are printed under control of this option - numbers bigger than 2**24 are always printed in decimal.

(**pack** U:any V:extra-boolean):nil
Type: EVAL, SPREAD, Base
The function **pack** converts its argument into a string of characters and places these in the atom assembly buffer BOFFO. Subsequently the characters placed in BOFFO can be turned into a real LISP identifier or number through calls to **mkatom** or **numob**. BOFFO must be initialized by a call of **clearbuff** before **pack** is used. Note that the standard LISP read and print routines clear BOFFO. (that is, if the result of **pack** is printed the buffer is then empty causing a subsequent **mkatom** to fail.) If **pack** is given a negative second argument, it forces all the characters it packs into lower case. If it is given a positive argument, it forces all the characters it packs into upper case.

(**packbyte** U:any):nil
Type: EVAL, SPREAD, Base
As **pack** but acting on internal codes.

(**pair** U:list V:list):alist
Type: EVAL, SPREAD, Core
U and V are lists which must have an identical number of elements. If not, an error occurs. Returned is a list where each element is a dotted-pair, the **car** of the pair being from U, and the **cdr** the corresponding element from V.

(**pairp** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a dotted-pair.

(**pdsinput** FILE:any):nil
Type: EVAL, SPREAD, Base
Causes calls to **open**, **rds** and **close** to refer to members of the directory called FILE. **pdsinput** calls can not be nested to go down multiple levels of the directory. It is cancelled by (pdsinput nil).

(**pdsoutput** FILE:any):nil
Type: EVAL, SPREAD, Base
As **pdsinput** but for **open**, **wrs** and **close**.

period
The initial value of the atom **period** is the atom ".".

period-internal-code
Variable containing the ASCII value of the period character.

(**plist** U:id):plist
Type: EVAL, SPREAD, Base
The function **plist** called with one argument that is an identifier returns the property list of that atom. The format of such a property list is a list of flags (see **flag**, **flagp**) and dotted pairs of (name . property). (See also **put**, **get**.) Note that in Cambridge LISP the properties **expr** and **fexpr** (see **getd**) found in LISP 1.5 do not appear on the property list.

(**plus** [U:number]):number
Type: NOEVAL, NOSPREAD, Base
Forms the sum of all its arguments.

(**plus2** U:number V:number):number
Type: EVAL, SPREAD, Base
Returns the sum of U and V. This function is used in the expansion of **plus**.

(**posn**):integer
Type: Base
Returns the number of characters in the output buffer (that is, position in output line). When the buffer is empty, 0 is returned.

(**prarg** U:level):level
Type: EVAL, SPREAD, Core
where the levels are 0, 1 and 2. After (prarg 0) the LISP supervisor does not echo what it reads and interprets. (prarg 1) the supervisor echoes its arguments using print, (prarg 2) causes it to superprint them. See **prval** and **prmsg**. The initial state is no echo. The function returns the previous value.

(**preserve** U:function)
Type: EVAL, SPREAD, Base
A LISP core-image can be created by executing (preserve) where the name DUMP refers to a directory. The state of LISP is written out onto the file, in a format suitable for loading as an initial image. After the system has been dumped, LISP stops. Core-images are reloaded by providing them to LISP under the directory IMAGE when the system starts up. If a function was specified by U when **preserve** was done this function is called when LISP starts up again. If none was specified then the supervisor, that was being used when **preserve** was called, is used. The image is written to the member LISPROOT.

(**prettyprint** U:any):any
Type: EVAL, SPREAD, LISPSPRI
Print the LISP expression U in an indented style.

(**prin** U:any):any
Type: EVAL, SPREAD, Base
The value of U is printed with any special characters preceded by the escape character. The value of U is returned.

(**princ** U:any):any
Type: EVAL, SPREAD, Base
The value of U is printed with no escape characters. The value of U is returned.

(**princl** U:any):any
Type: EVAL, SPREAD, LISPSPRI
As **prinl** but with no escape characters.

(**princs** U:any):any
Type: EVAL, SPREAD, LISPCOMP
The same as **princ** except that it prints a newline before U if the line is not at the beginning.

(**prinhex** U:number V:integer):number
Type: EVAL, SPREAD, Base
The number U is printed in hexadecimal in a field width V.

(**prinl** U:any):any
Type: EVAL, SPREAD, LISPSPRI
Like **prin** but treats circular lists correctly.

(**print** U:any):any
Type: EVAL, SPREAD, Base
The value of U is printed, with escape characters, followed by a new line. **print** fails if given cyclic structures, and there is no guarantee that the output it produces will be acceptable to the **read** function - in particular **gensyms** and binary code print legibly but not in a way where they can be re-input. There are two variables called ***printdepth** and ***printlength** which control how deep or long a list is printed. Initial values are 80 for depth and 2000 for length.

(**printc** U:any):any
Type: EVAL, SPREAD, Base
As for **print** but with no escape characters.

printcl
As **princl** plus terpri.

(**printcm** U:any N:integer):any
Type: EVAL, SPREAD, Base
As for **printc** but leaving a left margin of size N on line overflow.

(**printl** U:any):any
Type: EVAL, SPREAD, LISPSPRI
Prints circular lists without looping for ever. Reference points are labeled in the output with %Ln: and referred to by %Ln.

(**printm** U:any N:integer):any
Type: EVAL, SPREAD, Base
As for **print** but leaving a left margin of size N on line overflow.

printprompt* and **printvalue***
Prints the prompt string and value string of the user interface. The default is **printc**. This means that prompt can be on the same line as string.

(**prmsg** U:boolean):boolean
Type: EVAL, SPREAD, Core
If U is **nil** then the printing of messages by the LISP supervisor is inhibited. The function returns the previous value.

(**profile** U:integer)
Type: EVAL, SPREAD, LISPCOMP
After a call to **profile**, the LISP compiler generates code that includes statistic gathering orders to order U. The default (U=0) collects no statistics. With U=1 counts are collected at the entrypoint to each routine. With U=2 counts are collected near each conditional branch and each label in the compiled code. The statistics so gathered can be accessed by **readcount** or (more commonly) by **mapstore**.

(**prog** VARS:id-list [PROGRAM:{id any}]):any
Type: NOEVAL, NOSPREAD, Base
VARS is a list of ids which are considered fluid when the **prog** is interpreted and local when compiled. The **prog**'s variables are allocated space when the **prog** form is invoked and are deallocated when the **prog** is left. **prog** variables are initialized to **nil**. The PROGRAM is a set of expressions to be evaluated in order of their appearance in the **prog** function. Identifiers appearing in the top level of the PROGRAM are labels which can be referenced by **go**. The value returned by the **prog** function is determined by a **return** function or **nil** if the **prog** "falls through".

(**progn** [U:any]):any
Type: NOEVAL, NOSPREAD, Base
U is a set of expressions which are executed sequentially. The value returned is the value of the last expression.

(**prog2** U:any V:any):any
Type: EVAL, SPREAD, Base
prog2 is just like **progn** except that it may only be used to combine two expressions. It is provided for compatibility with other LISP systems.

prompt*
Type: Variable
The prompt string used by the supervisor. It can be changed by **setq**. Its initial value is "Input: ". It is printed by **printprompt***

(**prval** U:any):integer
Type: EVAL, SPREAD, Core
prval can be used to control the way in which the LISP supervisor prints the values of expressions it processes. After (prval 0) it prints nothing. (prval 1) which is the default, uses **print** to display the result of a computation, while after (prval 2) the function **printc** is used. If U is 3, values are **superprinte**d. The function returns the previous value.

pts
pts is a synonym for **set**.

(**put** U:id IND:id PROP:any):any
Type: EVAL, SPREAD, Base
The indicator IND with the property PROP is placed on the property list of the id U. If the action of **put** occurs, the value of PROP is returned. If either of U and IND are not ids the type mismatch error occurs and no property is placed. **put** *cannot* be used to define functions (use **putd** instead).

(**putd** FNAME:id TYPE:ftype BODY:function):id
Type: EVAL, SPREAD, Base
Creates a function with name FNAME and definition BODY of type TYPE. If **putd** succeeds the name of the defined function is returned. The effect of **putd** is that **getd** returns a dotted-pair with the function's type and definition. Likewise the **globalp** predicate returns **t** when queried with the function name. If function FNAME already exists, then this warning message appears
*** FNAME redefined
The function defined by **putd** is compiled before definition (changing **exprs** to **subrs** and **fexprs** to **fsubrs**) if the ***comp** global variable is non-**nil**.

(**putv** V:vector INDEX:integer VALUE:any):any
Type: EVAL, SPREAD, Base
Stores VALUE into the vector V at position INDEX. VALUE is returned. The type mismatch error may occur and if INDEX does not lie in 0...**upbv**(V) an error occurs.

(**quote** U:any):any
Type: NOEVAL, NOSPREAD, Base
Returns U unevaluated.

quote-internal-code
Variable containing the ASCII code for a single quote sign.

quotemark
Initial value is the character .

(**quotep** U:any):boolean
Type: EVAL, SPREAD, LISPSPRI
Returns **t** if U is of the form (quote xxxxx).

(**quotient** U:number V:number):number
Type: EVAL, SPREAD, Base
The quotient of U divided by V is returned. If U and V are integers the result is an integer and (remainder U V) is the corresponding remainder. An error occurs if you attempt to divide by zero.

(**random**):integer
Type: Base
Returns a random integer in the range 0 to 2**24-1. The pseudo-random sequence used is initialized to a number based on the time of day LISP was run. The seed can be fixed by an option at load time.

(**rational** U:number V:number):rational number
Type: EVAL, SPREAD, Base
Divides U by V, leaving the result as an exact fraction. If either U or V is floating point, it is converted to rational number form before the division is attempted.

(**rationalp** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a rational number, and **nil** otherwise.

rcurly-internal-code
The internal ASCII code for }.

(**rdf** INFILE:id OUTFILE:id):nil
Type: EVAL, SPREAD, Core
Reads and executes the LISP code in the given file, with the output going to the other file. If OUTFILE is null, or is ommitted (the normal use) then the terminal is used.

(**rds** FILE:any):any
Type: EVAL, SPREAD, Base
Input from the currently selected input file is suspended and further input comes from the file with name FILE. If FILE is **nil** the standard input device is selected. When end of file is reached on a non-standard input device, the standard input device is reselected. When end of file occurs on the standard input device the Standard LISP reader terminates. **rds** returns the name of the previously selected input file.

(**read**):any
Type: LISPREAD
Returns the next expression from the file currently selected for input. Valid input forms are: dot-notation, list-notation, numbers, function-pointers, strings, and identifiers with escape characters. Identifiers are interned on the OBLIST. **read** returns **eof** when the end of the currently selected input file is reached.

(**read-token**):atom
Type: LISPREAD
Reads one symbol and sets the variable **token-type** to one of **number**, **symbol** or **break-character**.

(read-tokenq):atom
Type: LISPREAD
As **read-token** but does not cause echo.

(readch):id
Type: LISPREAD
Returns the next character from the file currently selected for input. If all the characters in an input record have been read, the id **eol** is returned. If the file selected for input has all been read the id **eof** is returned. Note that the normal LISP escape character conventions and macro expansion do not operate in character by character reading.

(readchq):id
Type: LISPREAD
As **readch** but does not cause echo.

(readcount U:function V:any):integer
Type: EVAL, SPREAD, Base
Reads a call-count accumulated in a function compiled with the **profile** option set. If V is non-**nil** then the count is reset to 0.

(readq):any
Type: LISPREAD
As **read** but does not cause echo.

(recip U:number):number
Type: EVAL, SPREAD, Base
recip finds the reciprocal of a number by calling (quotient 1 U). In this sense the reciprocal of any integer other than + or -1 is zero.

(reclaim):nil
Type: Base
A user call to the function **reclaim** forces LISP to garbage-collect. A side effect of garbage collection may (see **verbos**) be the printing of some store-use statistics.

(remainder U:number V:number):number
Type: EVAL, SPREAD, Base
If both U and V are integers the result is the integer remainder of U divided by V. If either parameter is floating point, the result is the difference between U and V*(U/V) all in floating point. The sign of the remainder is always the same as the sign of V. An error occurs if V is zero.

(remd FNAME:id):{nil dotted-pair}
Type: EVAL, SPREAD, Base
Removes the function named FNAME from the set of defined functions. Returns the (ftype . function) dotted-pair or **nil** as does **getd**. The global/function attribute of FNAME is removed and the name may be used subsequently as a variable.

(remflag U:id-list V:id):nil
Type: EVAL, SPREAD, Base
Removes the flag V from the property list of each member of the list U. Both V and all the elements of U must be ids or the type mismatch error occurs. **nil** is returned.

(**remob** U:id):id
Type: EVAL, SPREAD, Base
If U is present on the OBLIST it is removed. This does not affect U having properties, flags, functions and the like. U is returned.

(**remprop** U:any IND:any):any
Type: EVAL, SPREAD, Base
Removes the property with indicator IND from the property list of U. Returns the removed property or **nil** if there was no such indicator.

(**repeat** BODY:any PRED:any):nil
Type: MACRO, Core
The BODY is evaluated repeatedly until PRED returns a non-**nil** value. The BODY is evaluated at least once.

(**return** U:any)
Type: EVAL, SPREAD, Base
Within a **prog**, **return** terminates the evaluation of a **prog** and returns U as the value of the **prog**. The restrictions on the placement of **return** are exactly those of **go**. Improper placement of **return** results in an error.

(**reverse** U:list):list
Type: EVAL, SPREAD, Core
Returns a copy of the top level of U in reverse order.

(**reversewoc** U:list):list
Type: EVAL, SPREAD, Base
reversewoc reverses a list without creating a copy, and so is destructive. It can be used to great effect in building lists which naturally are calculated in a left to right fashion where it can replace repeated uses of **append**. c.f. **reverse**.

rpar
The initial value of the atom **rpar** is **)**. c.f. **lpar**

rpar-internal-code
Variable containing the ASCII value for **)**.

(**rplaca** U:dotted-pair V:any):dotted-pair
Type: EVAL, SPREAD, Base
The **car** portion of the dotted-pair U is replaced by V. If dotted-pair U is (a . b) then (V . b) is returned. The type mismatch error occurs if U is not a dotted-pair.

(**rplacd** U:dotted-pair V:any):dotted-pair
Type: EVAL, SPREAD, Base
The **cdr** portion of the dotted-pair U is replaced by V. If dotted-pair U is (a . b) then (a . V) is returned. The type mismatch error occurs if U is not a dotted-pair.

(**rplacw** U:dotted-pair V:dotted-pair):dotted-pair
Type: EVAL, SPREAD, Core
Equivalent to (rplaca (rplacd U (cdr V)) (car V))

(**sassoc** U:any V:alist FN:function):any
Type: EVAL, SPREAD, Core
Returns the same result as (assoc U V) if U is present in V, otherwise the evaluation of function FN is returned.

(**select** U:any [V:pair] W:any):any
Type: MACRO, Core
V is an association list. U is compared for **equal**ity with the successive **car**s of V, and when found **select** returns the evaluation of the **cdr**. If there is no match then W is evaluated. **select** expands into a **cond** function.

(**set** EXP:id VALUE:any):any
Type: EVAL, SPREAD, Base
EXP must be an identifier or a type mismatch error occurs. The effect of **set** is replacement of the item bound to the identifier by VALUE. If the identifier is not a local variable or has not been declared GLOBAL it is automatically declared fluid. EXP must not evaluate to **t** or **nil** otherwise an error occurs because **t** or **nil** cannot be changed.

(**setbtr** LEV:integer):integer
Type: EVAL, SPREAD, Base
setbtr sets the level of backtrace information.

0 Totally silent error recovery.
1 Give the message only.
2 Message and list of function on the stack.
3 Like 2 with extended list of functions.
4 Like 3 but includes all fluids bound.
5 Message, functions, fluids and arguments to compiled functions.
6 As 5 with expressions prettyprinted.
7 As 6 with the loop printer **printl** used.
8 As 5 with compiler temporaries as well.
9 BCPL stack printed before style 8.

Values higher than 8 are only useful for system developers. The value returned is the previous level value. Note that **setbtr** works at the lowest level, and as such calls to other functions, especially **errorset** nullifies the effect of this function. In general, you are recommended to use the **backgag** function which changes the backtrace characteristics at a higher, and hence more predictable, level.

(**setdiff** U:list V:list):list
Type: EVAL, SPREAD, Core
Returns the set difference between two lists; that is those members of U that are not members of V using a **member** test.

(**setmod** U:integer):integer
Type: EVAL, SPREAD, Base
(setmod p) sets the modulus for the **cplus** and **mplus** families of modular arithmetic functions. (setmod 0) returns the current modulus without resetting it.

(**setplist** U:id PLIST:alist):alist
Type: EVAL, SPREAD, Base
Replaces the property list of U with PLIST. The function returns the previous property list owned by U.

(**setq** VARIABLE:id VALUE:any):any
Type: NOEVAL, NOSPREAD, Base
If VARIABLE is not local or GLOBAL it is by default declared fluid. The value of the current binding of VARIABLE is replaced by the value of VALUE. VARIABLE must not be **t** or **nil** or an error occurs. **setq** is the normal assignment operator in LISP.

(**setreturncode** U:integer):integer
Type: EVAL, SPREAD, Base
Sets the eventual return codes from LISP. (setreturncode nil) reads the return code currently set up.

(**setsyntax** U:chars V:property W:value):chars
Type: EVAL, SPREAD, LISPRDMC
U specifies a character or characters that are to be given special properties with respect to input. U can be a single-character id, a list of such ids or a string, which is treated as a list of characters. V specifies the property and if W is non-**nil** this property is set up for each character. If W is **nil** then the property is cancelled for the given characters. V can take the following values:

escape	causes the character to force any character following it to be treated as a letter.
break-character	causes the character to terminate identifiers.
digit	initially applies to 0. . .9, not wise for user to change!
upper-case	initially applies to a. . .z, not wise for user to change!
macro	value should be a function, which is called (with no arguments) whenever the character is encountered in the input. The result returned by this function becomes an element of the list being read. Same as **read-macro**.
may-start-number	character accepted in front of a number. Initially applies + and -.
splice	as for macro except a list of items to include in the list being read is expected to be returned from the function. A special case is when the function returns **nil**, when the macro character plus anything read by the function are ignored by the main reader. (This is how comments are implemented). Same as **splice-read-macro**

The characters that have special input properties initially are:

ignore	**eol, ff, blank, tab**
break-character	**eol, ff, blank, tab, eof** (.){}#$%&=-`~^|\[]+;,*:()?/
escape	!
digit	0123456789
may-start-number	+-.
upper-case	ABCDEFGHIJKLMNOPQRSTUVWXYZ
letter	abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ

The value returned is that of U.

(**sin** U:number):number
Type: EVAL, SPREAD, Base
As **arccos** except the function is sine.

(**smallp** U:number):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is an integer that can be stored as a single word.

(**sort** U:any-list PREDICATE:id):any-list
Type: EVAL, SPREAD, Core
The list U is sorted with respect to the given predicate, for example (sort '(7 5 9 1 5) 'greaterp) returns (9 7 5 5 1).

spaces
Synonym for **xtab**.

(**speak**):integer
Type: Base
(speak) returns the current value of the **cons** counter. Note that a value of 0 means that the counter is disabled, and that the number inspected by speak is only updated when a garbage collection occurs.

(**sqrt** U:number):number
Type: EVAL, SPREAD, Base
Returns the square root of U. If U is an integer that is a perfect square the result is an integer, otherwise floating point.

(**stop** U:integer)
Type: EVAL, SPREAD, Base
Exits directly from LISP. (stop n) gives a return code of at least n. See **setreturncode** for a more general way of controlling return codes.

(**stringconcat** U:string V:string):string
Type: EVAL, SPREAD, Base
Returns the concatenation of the strings U and V.

(**stringp** U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a string.

(**sublis** X:alist Y:any):any
Type: EVAL, SPREAD, Core
The value returned is the result of substituting the **cdr** of each element of the alist X for every occurrence of the **car** part of that element in Y.

subr
subr is a property name used by some other LISP systems in association with the implementation of compiled code and the basic built-in functions. Since Cambridge LISP uses a rather different scheme for coping with function definitions, imported programs that rely heavily on the manipulation of **subr** properties may need substantial revision.

(**subst** U:any V:any W:any):any
Type: EVAL, SPREAD, Core
The value returned is the result of substituting U for all occurrences of V in W.

(**sub1** U:number):number
Type: EVAL, SPREAD, Base
If U is a number then U-1 is returned. If U is not a number then an error is given. The effect is the same as (difference U 1) but is faster.

(**superprinm** U:any N:integer)
Type: EVAL, SPREAD, LISPSPRI
Same as **superprint** but leaves a left margin of width N.

(**superprint** U:any):any
Type: EVAL, SPREAD, LISPSPRI
Prints U in an indented format (if it does not all fit on one line) which is intended to make the structure of the list more readily visible. The detailed print style is tuned for the display of LISP programs, and so some words (for example, prog, lambda, quote) are treated specially by superprint, forcing it to split lines in standardized places. The value returned is the argument.

(**superprintm** U:any N:integer)
Type: EVAL, SPREAD, LISPSPRI
Same as **superprint** but leaves a left margin of width N, and terminates with a number of newlines.

(**supervisor**)
Type: Core
supervisor is a user entry into the standard LISP loop that reads, echoes and evaluates LISP code. It can be useful for processing data-files that contain executable LISP statements. Reading the atom **fin** at the top level terminates the call to **supervisor**. The **supervisor** catches any **throw** to the tag supervisor and also it catches the tag **nil**, which is a universal catcher. Thus a throw not caught by the user program remains in LISP.

(**symnam** U:id):id
Type: EVAL, SPREAD, Base
symnam takes one argument, which must be an identifier. Until you call **symnam** again, this identifier provides the initial part of the print representation of atoms created by **gensym**. Note that for internal reasons, there is a store-use penalty in using more than a few dozen different **symnams**. See also **gensym1**. The value returned is the previous value.

t
The atom **t** is the standard LISP representation of 'true', and most built-in LISP predicates return either **t** for true or **nil** for false. You should not use **t** as a name for a local or bound variable.

tab
Variable having the value of the character tab.

(**tan** U:number):number
Type: EVAL, SPREAD, Base
As **arccos** except the function is tangent.

(**tempus-fugit**)
Type: Base
A call to **tempus-fugit** results in LISP displaying a line of text containing timing and core-use figures.

(**terpri**):nil
Type: Base
The current print line is terminated (that is, a new line is started).

(**throw** TAG:id VAL:any)
Type: MACRO, Core
Used in conjuction with **catch** this provides a form of non-local control. Control continues from the last **catch** in the execution path that has the tag TAG, and the **catch** returns the value VAL.

(**time**):integer
Type: Base
Returns the elapsed time (in milliseconds) used so far this run, excluding overheads (see **gctime**).

(timeofday):string
Type: Base
Returns a string giving time of day.

(times [U:number]**)**:number
Type: NOEVAL, NOSPREAD, Base
Returns the product of all its arguments.

(times2 U:number V:number**)**:number
Type: EVAL, SPREAD, Base
Returns the product of U and V. This function is used in the expansion of **times**.

token-type
This variable is set by the tokenizer of the LISP reader to indicate the type of the token read. Possible values are: **break-character**, **result-of-read-macro**, **number**, **symbol** or **escape**. The meanings of these are obvious when taken with **setsyntax**.

(trace U:{id id-list}**)**:{id id-list}
Type: EVAL, SPREAD, Base
Sets up tracing for any function whose name appears in the list U or the function U.

(tracecount N:integer**)**:nil
Type: EVAL, SPREAD, Base
Causes system to suppress next N items of trace output. This is very useful for delayed errors by switching on **trace** just before a problem occurs.

(tracesetq U:id-list**)**:id-list
Type: EVAL, SPREAD, Base
After a call to **tracesetq** *interpreted* use of **set** or **setq** to update the variables named in U leads to a message being printed. Unlike **trace**, **tracesetq** just has one list of traced variables, and to stop tracing a call (tracesetq nil) should be used.

(ttab U:integer**)**:nil
Type: EVAL, SPREAD, Base
Enough spaces are printed to move the next character position in the line to U.

(tyi):integer
Type: Base
Reads one character and returns its internal code.

(tyipeek):integer
Type: Base
Returns the internal code for the next character in the input without reading it. That is, a subsequent call to **tyi** returns this character.

(tyiq):integer
Type: Base
As **tyi** but does not echo what it reads.

(tyo U:integer**)**
Type: EVAL, SPREAD, Base
Prints the character with internal code U.

(unembed U:id)
Type: EVAL, SPREAD, Core
Undoes effect of **embed**.

(unfluid IDLIST:id-list):nil
Type: EVAL, SPREAD, Core
The variables in IDLIST that have been declared as fluid variables are no longer considered as fluid variables. Others are ignored. This affects only compiled functions as free variables in interpreted functions are automatically considered fluid.

(unglobal U:id):nil
Type: EVAL, SPREAD, Core
Undoes effect of **global**.

(union U:list V:list);list
Type: EVAL, SPREAD, Core
Returns a list of all the items from U and V. If some item is in both U and V it only occurs once in the union list. See **append** for list merging/concatenation that does not remove duplicate entries, and **xn** for list intersection.

(unset U:id):any
Type: EVAL, SPREAD, Base
Is equivalent to (set U INDEFINITE! VALUE) but returns the old value of the variable.

(untrace U:id-list)
Type: EVAL, SPREAD, Base
Undoes the effect of **trace** for the variables named in U.

(upbv U:any):integer
Type: EVAL, SPREAD, Base
Returns the upper limit of U if U is a vector, or 0 if it is not.

value*
Type: Variable
The **supervisor** prints the values of evaluation following the printing (with **princ**) of this variable. It is initiallized to the string "Value: ". It is printed by **printvalue***

(values-in-tree T:avl-tree):list
Type: EVAL, SPREAD, LISPAVLT
values-in-tree returns the list of value in the AVL tree in order. See also the functions **avl-add** and **avlq-add**.

(vectorp U:any):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a vector.

(verbos N:integer):integer
Type: EVAL, SPREAD, Base
Sets garbage message level to N. N=0 for no messages; N=1 for garbage collection; N>1 for commentary on FASL activity. The initial setting is N=0. The value returned is the previous value.

(**while** PRED:any BODY:any):nil
Type: MACRO, Core
while evaluates the BODY repeatedly while the expression PRED returns a non-**nil** result. The BODY may be evaluated zero times.

(**wrs** FILE:any):any
Type: EVAL, SPREAD, Base
Output to the currently active output file is suspended and further output is directed to the file with file-handle FILE. The file named must have been opened for output. If FILE is **nil** the standard output device is selected. **wrs** returns the file-handle of the previously selected output file.

(**xcons** U:any V:any):dotted-pair
Type: EVAL, SPREAD, Base
Is the same as (cons V U). c.f. **ncons** and **cons**

(**xn** U:list V:list):list
Type: EVAL, SPREAD, Core
Returns the set intersection of the lists U and V, that is, those members of U that are also in V. See also **union**.

(**xtab** U:integer):nil
Type:EVAL, SPREAD, Base
Prints U spaces on the current line.

(**younger** U:list V:list):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U and V are both lists and U was created by a **cons** that took place after the **cons** that made V. This provides a cheap, consistent but rather arbitrary order relation on list structures.

(**zerop** U:number):boolean
Type: EVAL, SPREAD, Base
Returns **t** if U is a number and it is either the integer 0 or the floating point number 0.0. c.f. **onep**.

Appendix A: Summary of Standard LISP

As Cambridge LISP 68000 is similar to Standard LISP, this section simply covers the differences between the two. Although Cambridge LISP 68000 contains functions not found in Standard LISP, it does include most of the Standard LISP functions as a subset. Those mentioned here either do not exist in Cambridge LISP 68000, or do, but in a somewhat different form.

Identifiers

COMPRESS (U:id-list):{atom}-{vector} Type: EVAL, SPREAD
COMPRESS is identical to the **compress** found in Cambridge LISP, except that the list of single character identifiers that make up U are not interned on the oblist.

EXPLODE (U:{atom}-{vector}):id-list Type: EVAL, Spread
This is identical to **explode** except in that the primitive data types do not include vectors.

INTERN (U:{id,string}):id Type EVAL, SPREAD
INTERN searches the oblist for an identifier with the same print name as U and returns the identifier on the oblist if a match is found. Any properties and global values associated with the uninterned U are lost. If U does not match any entry, a new one is created and returned. If U has more than the maximum number of characters permitted by the implementation (the minimum number is 24) an error occurs:

***** Too many characters to INTERN

Function definition

DE (FNAME:id, PARAMS:id-list, FN:any):id Type: NOEVAL, NOSPREAD

Almost identical to **de**. The function created by **DE** is of type EXPR unless the !*COMP variable is T in which case the EXPR is compiled.

DF (FNAME:id, PARAMS:id-list, FN:any):id Type: NOEVAL, NOSPREAD
Almost identical to **df**. The function created by **DF** is of type FEXPR unless the !*COMP variable is T in which case the FEXPR is compiled.

GETD (FNAME:any):{NIL, dotted-pair} Type: EVAL, SPREAD
Almost identical to **getd**. NIL is returned if FNAME is not the name of a defined function, if it is then the dotted-pair

(TYPE:ftype . DEF:{function-pointer, lambda})

is returned.

PUTD (FNAME:id, TYPE:ftype, BODY:function):id Type: EVAL, SPREAD
Almost identical to **putd**. However, if the function FNAME has already been declared as a GLOBAL or FLUID variable, the error message:

***** FNAME is a non-local variable

occurs and the function is not defined.

Variables and Bindings

A *variable* is a place holder for a Standard LISP entity which is said to be *bound* to the variable. The *scope* of a variable is the range over which the variable has a defined value. There are three different binding mechanisms in Standard LISP:

Local binding
Global binding
Fluid binding

These are similar to those in Cambridge LISP.

Error handling

ERRORSET (U:any, MSGP:boolean, TR:boolean):any Type: EVAL, SPREAD
Although **ERROR** is the same as **error**, **ERRORSET** is slightly different from **errorset**. If an error occurs during the evaluation of U, the value of NUMBER from the ERROR call is returned as the value of ERRORSET. In addition, if the value of MSGP is non-NIL, the MESSAGE from the ERROR call is displayed upon both the standard output device and the currently selected output device unless the standard output device is not open. The message appears prefixed with five asterisks. The MESSAGE list is displayed without top level parentheses. The MESSAGE from the ERROR call is available in the global variable EMSG!*. Error numbers generated by Standard LISP functions are implementation dependent.

If no error occurs during the evaluation of U, the value of (LIST (EVAL U)) is returned.

If an error has been signaled and the value of TR is non-NIL, a traceback sequence is initiated on the selected output device. The traceback displays information such as unbindings of FLUID variables, argument lists, and so on in an implementation dependent format.

Boolean functions and conditionals

Although **AND**, **NOT** and **OR** are the same as in Cambridge LISP 68000, **COND** is slightly different.

COND ([u:cond-form]):any Type: NOEVAL, NOSPREAD
The antecedents of all U's are evaluated in order of their appearance until a non-NIL value is encountered. The consequent of the selected U is evaluated and becomes the value of COND. If no antecedent is non-NIL the value of COND is NIL. An error is detected if a U is improperly formed.

Arithmetic Functions

ABS, ADD1, DIFFERENCE, DIVIDE, EXPT, FIX, FLOAT, GREATERP, LESSP, MAX, MAX2, MIN, MIN2, MINUS, PLUS, PLUS2, TIMES and **TIMES2** are identical to those found in Cambridge LISP. The following are, however, slightly different.

QUOTIENT (U:number, V:number):number Type:EVAL, SPREAD
The quotient of U divided by V is returned. Division of two positive or two negative integers is conventional. When both U and V are integers and exactly one of them is negative the value returned

is the negative truncation of the absolute value of U divided by the absolute value of V. An error occurs if division by zero is attempted.

REMAINDER (U:number, V:number):number Type: EVAL, SPREAD
If both U and V are integers the result is the integer remainder of U divided by V. If either parameter is floating point, the result is the difference between U and V*(U/V) all in floating point. If either number is negative the remainder is negative. If both are positive or both are negative the remainder is positive. An error occurs if V is zero.

SUB1 (U:number):number Type: EVAL, SPREAD
Returns the value of U less 1. If U is a FLOAT type number, the value returned is U less 1.0.

The Interpreter

APPLY (FN:{id.function}, ARGS:any-list):any Type: EVAL, SPREAD
APPLY returns the value of FN with actual parameters ARGS. The actual parameters in ARGS are already in the form required for binding to the formal parameters of FN.

EVAL, EVLIS, EXPAND, FUNCTION, and **QUOTE** are identical to those in Cambridge LISP.

Input and Output

This is different from Cambridge LISP. The user communicates with Standard LISP through "standard devices". The default devices are selected in accordance with the conventions of the implementation site. Other input and output devices or files may be selected for reading and writing using the functions described below.

CLOSE (FILEHANDLE:any):any Type: EVAL, SPREAD
Closes the file with the internal name FILEHANDLE writing any necessary end of file marks. The value of FILEHANDLE is that returned by the corresponding OPEN. The value returned is the value of FILEHANDLE. An error occurs if the file can not be closed.

EJECT ():NIL
Causes a skip to the top of the next output page. Automatic EJECTs are executed by the print functions when the length set by the PAGELENGTH function is exceeded.

LINELENGTH (LEN:{integer, NIL}):integer Type: EVAL, SPREAD
If LEN is an integer the maximum line length to be printed before the print functions intitiate an automatic TERPRI is set to the value of LEN. No initial Standard LISP line length is assumed. The previous line length is returned except when LEN is NIL. This special case returns the current line length and does not cause it to be reset. An error occurs if the requested line length is too large for the currently selected output file or LEN is negative or zero.

LPOSN ():integer
Returns the number of lines printed on the current page. At the top of the page, 0 is returned. **POSN** is the same as **posn**

OPEN (FILE:any, HOW:id):any Type: EVAL, SPREAD
Open the file with the system dependent name FILE for output if HOW is EQ to OUTPUT, or input if HOW is EQ to INPUT. If the file is opened successfully, a value which is internally associated with the file is returned. This value must be saved for use by RDS and WRS. An error occurs if HOW is something other than INPUT or OUTPUT or the file can not be opened.

PAGELENGTH (LEN:{integer, NIL}):integer Type: EVAL, SPREAD
Sets the vertical length (in lines) of an output page. Automatic page EJECTs are executed by the print functions when this length is reached. The initial vertical length is implementation specific. The previous page length is returned. If LEN is 0, no automatic page ejects occur.

PRINC (U:id):id Type: EVAL, SPREAD
U must be a single character id such as produced by EXPLODE or read by READCH or the value of !$EOL!$. The effect is the character U displayed upon the currently selected output device. The value of !$EOL!$ causes termination of the current line like a call to TERPRI.

PRINT (U:any):any Type: EVAL, SPREAD
Displays U in READ readable format and terminates the print line. The value of U is returned.

PRIN1 (u:any):any Type EVAL, SPREAD
U is displayed in a READ readable form. The format of display is the result of EXPLODE expansion; special characters are prefixed with the escape character !, and strings are enclosed in "...". Lists are displayed in list-notation and vectors in vector-notation.

PRIN2 (U:any):any Type: EVAL, SPREAD
U is displayed upon the currently selected print device but output is not READ readable. The value of U is returned. Items are displayed as described in the EXPLODE function with the exceptions that the escape character does not prefix special characters and strings are not enclosed in "...". Lists are displayed in list-notation and vectors in vector-notation. The value of U is returned.

RDS (FILEHANDLE:any):any Type: EVAL, SPREAD
Input from the currently selected input file is suspended and further input comes from the file named. FILEHANDLE is a system dependent internal name which is a value returned by OPEN. If FILEHANDLE is NIL the standard input device is selected. When end of file is reached on a non-standard input device, the standard input device is reselected. When end of file occurs on the standard input device the Standard LISP reader terminates. RDS returns the internal name of the previously selected input file. **WRS** is almost identical to **wrs**.

READ, **READCH**, and **TERPRI** are similar to their Cambridge LISP counterparts.

LISP reader

An EVAL read loop was chosen to drive a Standard LISP system in order to provide a continuity in functional syntax. Choices of messages and the amount of extra information displayed are decisions left to the implementor.

QUIT ()
Causes termination of the LISP reader and control to be transferred back to the operating system.

System global variables

These variables provide global control of the LISP system, or implement values which are constant throughout execution. The following are identical to their Cambridge counterparts: **!*COMP**, **EMSG!***, **!$EOF!$**, **!$EOL!$**, **NIL** and **T**. **!*GC** and **!*RAISE** are different.

!*GC

This has the initial value of NIL. It controls the printing of garbage collector messages. If NIL no indication of garbage collection may occur. In non-NIL various system dependent messages may be displayed.

!*RAISE

This has the initial value NIL. If it is non-NIL all characters input though Standard LISP input/output functions are raised to upper case. If !*RAISE is NIL, characters are input as they stand.

Appendix B: Customization of LISP

As you make more complex use of LISP, you can use the various core preservation functions to produce customized versions. For example, if a customized system is available in the directory 'custom', then

```
lisp image=custom
```

runs it.

The process of customization needs the form

```
lisp image=old_image  dump=new_image
```

to preserve modules in the directory *new_image* using the core image in *old_image* as a base.

When it starts, LISP attempts to acquire as much space as it can, leaving a little behind for other operating system activities. You can elleviate this anti-social behavior by using certain options (see below for a list of valid options). Messages produced at the start and finish of each run give you some indication of how much store is used and how much was available. You do not need to tell LISP how to allocate the store it is given - it has its own flexible scheme so that for example neither stack nor freestore can run out while there is some of the other left. To some extent this flexible storage allocation scheme applies to space for lists, atoms, numbers, floating point numbers, compiled code, vectors and internal tables. Note also that the LISP compiler does not take up much space until it is used, and you can remove it (using **excise**) when it is finished with.

The options available are:

FROM — If present expects a AmigaDOS file name to use as the standard input to LISP. The default is the terminal.

TO — If present expects a AmigaDOS file name to use as the standard output to LISP. The default is the terminal.

IMAGE — If present expects the name of a directory in which to find the initial core image and fast load modules. The default is IMAGE.

DUMP — If present expects the name of a directory (max 255 chars) in which to put the core image and fast load modules generated in the run, by use of **module** and **preserve**. The default is the image directory.

LEAVE — Expects a number (in units of 1024) that LISP will leave for the operating system. The default is 20. For example:

```
lisp leave 100k
```

(this leaves 100 kilobytes)

STORE — Expects a number (in units of 1024) that LISP will use for this run. It is useful to give exactly reproducable runs. For example:

```
lisp store 400k
```

(this runs lisp in about 400 kilobytes)

OPT — LISP can accept a number of options, given as a string to the OPT input. Few of these are of interest to normal users.

- D<n> — Debug mode is set to n. This is of no interest to general users.
- G<n> — Causes the system to trap after garbage collection n.
- I — Set car-nil-legal mode on entry. See also **car-nil-legal**.
- M — Obey a **mapstore** at end of run.
- M<n> — Obey a **mapstore** at end of run; if n>1 then do a **bcplmapstore**.
- N<n> — Set the maximum allowable length of an integer to be 9n decimal digits. If this is exceeded an error occurs.

P<n> Set print depth to n for error messages.

R<n> Set the seed for the random number generator to n. Useful for reproducable runs.

X<n> Set auto excise mode to n. This controls how readily LISP unloads a module after a garbage collection. The default is n=0, which causes unloading if less than 40Kbytes are free; n=1 reduces this to 20K. n=2 inhibits autoexcise while n=3 causes all modules to be unloaded on a garbage collection.

Saving a core image

You may decide to customise your version of LISP in order to keep all your own features so that they can be used all the time. To do this they must be saved in a core image. First create a directory called (for instance) 'myimage' (the name can be anything up to 255 characters). You then need approximately 261Kbytes in which to hold the core image. After creating the directory, type

```
lisp dump=myimage
```

to load LISP. You may then make any changes you wish, for example, alter the prompt, or change the default linelength. The function **preserve** should then be used.

```
(preserve)
```

preserve writes a core-image to the file with the name LISPROOT in the DUMP (or IMAGE) directory. After **preserve** has been executed LISP stops, even if the call to **preserve** was embedded deep in some other functions. On restart all information about what was happening when **preserve** was called is lost. **preserve** writes to members LISPROOT and LISPERRS, together with any other members defined by the FASL facility.

To use the saved core image again type:

```
lisp image=myimage
```

To make this image your default image, rename the original image by renaming sys:l.lisp.image as sys:l.lisp.oldimage.

If anything should go wrong with **preserve**, check to see if you have a large file called BACKROOT and an empty file called LISPROOT. If you have you should rename BACKROOT as LISPROOT and you should be back where you started. If IMAGE and DUMP are the same, then the previous LISPROOT is renamed BACKROOT before the core is dumped.

It is also useful, particularly when developing a large program, to be able to dump the state of LISP to a disc file, then pick it up again in a later run. Core images can only be loaded at the very start of a LISP run. On entry to LISP, the parameter IMAGE gives the directory in which to find the coreimage that is to be loaded. For example,

```
LISP IMAGE=SYS:L/LISP/IMAGE DUMP=SAVEDIR
 . . .
  (preserve)
```

dumps a coreimage to the directory SAVEDIR, and

```
LISP IMAGE=SAVEDIR
  . . .
```

reloads it and continues with the computation. If **preserve** writes out a file successfully, LISP stops with a return code of at least 200 to signal this fact.

```
(preserve 'initialsupervisor)
```

is a form that can be used to package systems in a secure way. When the image file gets reloaded, the function initialsupervisor is called rather than the standard LISP supervisor. If initialsupervisor is exited then LISP stops. (Note that initialsupervisor is a user's function, that is to say, it does not exist as **initialsupervisor**).

Appendix C: Converting to Cambridge LISP

The purpose of this section is not to give a complete cross-reference of every feature found in Cambridge LISP to those found in other Lisps, it is rather to give an indication of where other Lisps differ. In the design of Cambridge LISP great care was taken to include some degree of compatibility with other Lisps. Many functions were included to facilitate compatiblity with Maclisp, U.C.I., Standard Lisp and Interlisp. P.S.L., although a separate language, contains similar code to Cambridge LISP and is therefore largely compatible. If the areas outlined below are considered and amended, programs written in other Lisps should run under Cambridge LISP 68000 with very little further alteration.

At first the most obvious difference between Cambridge LISP and other Lisps is that it is written in lower case. Programs from other Lisps should, therefore, be either converted into lower case, or ***lower** should be set to true. Once ***lower** has been used, all upper case letters are automatically translated into lower case.

Comments in Cambridge LISP are by default introduced by the character **%** (Maclisp for instance uses a ;). This character can always be changed to that used in another dialect - such as a ;. The function **setsyntax** is used to change the use of characters. Comment characters can hence be exchanged globally to the Cambridge default, or the default can be changed to be the same.

Cambridge LISP uses the character **!** as its standard escape to introduce strange characters. This is the same as Standard Lisp, but U.C.I. uses a **/** instead. For a description of setting an escape (as well as comments and case) see section 2.1.

Certain other characters have different meanings in other Lisp dialects. U.C.I. considers + and ^ to be letters; Cambridge LISP does not, although it can be made to do so. Maclisp uses + as a short form for the **iplus** of Cambridge LISP, or the ***plus** found in U.C.I.

The print commands used in Cambridge LISP are quite different from those used in other Lisps. The usual form found elsewhere is the command followed by a number e.g. Prin1, Prin2 etc. Here the numbers have been removed and the commands differentiated in a more logical way using a letter as a mnemonicon to remind you of the action of each command. For example:

prin	simple printing of an atom or list
print	as prin, but terminating the print line
printm	as print, except leaves a left margin
princ	basic print routine
princs	as princ, but starts a new line if necessary
printc	as princ followed by call to terpri
printcm	as printc, except leaves specified left margin
prinl	simple print routine for looped structure
printl	as prinl, except terminated by terpri

and so on.

The most fundamental difference between Cambridge LISP and some of the other major dialects is apparent in the way functions are defined. Cambridge LISP uses value cells, whereas others, such as U.C.I., use value cells and what can only be called "function cells". (See 3.1 for a description of value cells). If we take the function 'foo', this is defined in U.C.I using the command **defprop** (where the 'prop' part refers to the property list), this has the properties **fexpr** and **expr** which are not found in Cambridge LISP. So you can enter

```
(get 'foo 'expr)
```

in U.C.I. where in Cambridge LISP the form would be

```
(getd 'foo)
```

get returns the property associated with the indicator from the property list of foo, in Cambridge LISP it must be replaced by **getd** as **get** can not be used to access functions. Section 2.3 describes the function definitions **de** and **df** as well as the macro definitions **dm** and **dmm**. **dmm** is a direct translation of the Maclisp DEFUN MACRO.

LET and **DO** found in other dialects are not present in Cambridge LISP; there is no real translation.

In some Lisp dialects the action of compilation is separate. In Cambridge LISP there is internal compilation. In some functions compilation is automatic if not specified. Compilation can always be initiated from within LISP. This should be noted when you translate programs to run under Cambridge LISP.

Appendix D: Possible error messages

0		User call to error function
1 -	5	Bad argument for plus
7		Bad argument for a division function
8		Bad argument for minus
9		Malformed number in buffer detected by numob
10		Bad argument for evenp
11		Bad argument for shift function
12 -	13	Bad argument for logand
14 -	15	Bad argument for logor
16 -	17	Bad argument for logxor
18		Argument for remob not an identifier
19 -	21	Bad argument for expt
22 -	23	Bad argument for greaterp or lessp
24		Attempt to take car of an atom
25		Attempt to take cdr of an atom
26		Rplaca given atomic first argument
27		Rplacd given atomic first argument
28		Orderp can not process gensyms
29		Bad argument for gts
30 -	31	Bad syntax for quote function
32		Bad argument for unset
33 -	34	Bad syntax in cond expression
35		Bad argument for plist
36		Second argument for open neither input/output
37		Bad first argument for open
38		Mkatom failed; atom assembly buffer not set up
39		Numob failed; atom assembly buffer not set up
40		Atom assembly buffer empty
41		Pack failed - atom assembly buffer not set up
42		Bad argument for pack
43		Unset variable
44		Illegal object used as a function
45		Undefined function
46		Circular definition of function
47		Unset variable in macro-expansion
48		Funargs not implemented
49		Bad syntax in lambda expression
50		Illegal call to codel function
51		Illegal call to fcode function
52		Illegal call to lambdaq function
53		Too many arguments in lambda expression
54		Too many arguments for function
55		Illegal call to (lambda x ...) function
56		Illegal call to a macro
57 -	58	Illegal item in list of bound variables
59		Illegal item in list of prog variables
60		Return not directly in a prog
61		Attempt to divide by 0.0
62		Bad argument for fix
63		Argument for fix > 10**9
64 -	66	Bad format for define

67		Bad argument for set or setq
68		Attempt to set the value of nil
69 -	71	Bad syntax for setq
73 -	74	Bad format for deflist
75		Bad format in deflist/fexpr
76		Bad argument for put
77		Bad argument for flag
85		Bad argument for remflag
86		Bad argument for remprop
87		Bad argument for prop
89		Not enough store for vector request
90		Casego not directly in a prog
91		Go not directly in a prog
92 -	96	Bad argument for times
97		Atom too long (limit is 253 chars)
98		Bad syntax in number
99 -	100	Bad argument for xtab
101 -	102	Bad argument for ttab
108		Illegal multiple assignment
109		Bad argument for linelength
110 -	111	Bad syntax for prog
112		Bad syntax for go
113		Label not found
114		Attempt to divide by zero
115		Unable to convert fp number to rational form
129		Store jam
131		Bad argument for gctrap
132		Not enough store to load basic LISP system
133		Failure in LISP supervisor
134		Bad argument for tobig
135		Bad argument for torat
136		Numeric argument expected
138		Argument for prin1 should be atomic
139		Error detected by operating system
140 -	142	Attempt to divide by zero
143		Bad syntax for define
144		Bad syntax for deflist
145 -	147	Error in mkvect
148 -	150	Error in access to vector
151 -	153	Error in attempt to update vector element
154		Open failed - file does not exist
155		Open failed - file already in use
156		Bad argument for preserve
157		Preserve failed - file not found
158		Preserve failed - file already opened
159		Bad argument for digch
160		Non-atomic arg to chdig
161 -	162	Argument for chdig not a digit
163		Bad argument for trace
164		Bad argument for untrace
165		End of file detected by read
166		Not an atom for orderp
168		Output radix must be 2,8,10 or 16
169		Bad argument for gcd

170 Null string not allowed
173 Illegal car access in compiled code
174 Illegal cdr access in compiled code
175 Illegal rplaca in compiled code
176 Illegal rplacd in compiled code
177 Rplacw read access illegal
178 Rplacw write access illegal
179 Bad argument for count
180 Type assumptions wrong in compiled code
181 Bad use of car,cdr,rplaca/d in compiled code
182 Bad argument to modular arithmetic routine
183 Bad argument for arcsin/arccos
184 Bad argument for atan
185 Bad argument for exp
186 Bad argument for log/log10
187 Bad argument for expt (base = 0.0)
188 Bad argument for sin/cos
189 Negative argument for sqrt
190 Bad argument for tan/cot
191 Bad argument for tan/cot
192 Bad argument for abs
196 Boffo overflowed in pack
197 Bad use of 'function'
198 Bad function name for define
199 Maximum length of integer exceeded
201 Bad argument for (integer) sqrt
202 Negative argument for (integer) sqrt
203 Exponent too large in floating point number
206 Bad argument for bps
207 Length for bps not a small integer
208 Type code bad in bps
210 Overflow of quotecell region
211 Unable to open dumpfile
213 Type code bad for getbps
214 Attempt to open sysin for output
215 Attempt to open sysprint for input
216 Close failed on file
217 Lost file in close
218 Bad syntax in use of de, df or dm
219 - 220 Conflict between flag and indicator name
225 Module already loaded
226 Module not found
227 Loading module did not load required function
228 Bad syntax in call to fasl
229 Module empty or irrelevant
230 Format error in module
231 Bad argument for dumpfile
232 Dumping inhibited by return code
233 Failed to write up core image file
234 Bad argument for setreturncode
235 Bad member name for faslcopy
236 Member not found for faslcopy
237 Faslcopy unable to open output file
242 File already open as a sequential dataset

244	Attempt to open file for both read & write
246	Lost file in wrs
248	Bad argument for wrs
249	Lost file in rds
251	Bad argument for rds
252	Fast load failed; system may be inconsistent
253	Function definition not recovered from module
254	Type code bad in excise
255	Bad member in image file found by faslcopy
256	Bad argument for module
257	Module already in use
258	Unable to open module output file
265	Cons counter overflow
266	Console break signalled

Appendix E: Examples

This appendix contains a few example lisp programs.

PPFILE - this function takes a file containing lisp source and prettyprints it out to another file. This is useful for locating errors in programs caused by misplaced brackets. Note that PPFILE will lose all the comments that are not on a line by themselves.

```
USAGE:
(ppfile 'fred 'ppfred) prettyprints file fred to ppfred
(ppfile 'fred '*)      prettyprint file fred to screen
(ppfile 'fred)         same as above
(ppfile 'fred 'prt!:)   prettyprint file fred to the
                       lineprinter (if you have one !)

(de ppfile (infile outfile)
   (prog (s oldout oldin)
      (setq oldin (rds (open infile 'input)))
      (setq oldout (wrs (open outfile 'output)))
loop  (commentfind)
      (setq s (read))
      (cond
         ((eq s !$eof!$) (go end))
         (t (prettyprint s) (go loop)))
end   (rds oldin)
      (wrs oldout)
      (close outfile)
      (close infile)))
```

PPFILE2 is the same as PPFILE, but removes all comments

```
(de ppfile2 (infile outfile)
   (prog (s oldout oldin)
      (setq oldin (rds (open infile 'input)))
      (setq oldout (wrs (open outfile 'output)))
loop  (setq s (read))
      (cond
         ((eq s !$eof!$) (go end))
         (t (prettyprint s) (go loop)))
end   (rds oldin)
      (wrs oldout)
      (close outfile)
      (close infile)))
```

COMMENTFIND - this is a function used by PPFILE to write out any comments found on a line by themselves

```
(de commentfind nil
   (prog (temp)
loop  (setq temp (tyipeek))
      (cond
         ((eq temp 40) (return nil))
         ((member temp '(32 10 !$eol!$)) (tyo (tyi)))
         ((eq temp 37) (pcomment))
         (t (return nil)))
      (go loop)))
```

PCOMMENT - this is a function used by COMMENTFIND to print out a line

```
(de pcomment nil
   (prog nil
loop  (cond
         ((eq (tyipeek) 10) (tyo (tyi)) (return nil))
         (t (tyo (tyi))))
      (go loop)))
```

SAVEFUN - this function can be used to write function definitions to a file.

USAGE:

```
(savefun 'myprog 'myfile)
```

This saves the definition of myprog to myfile.

```
(savefun '(prog1 prog2 prog3) 'myfile)
```

saves the definitons of prog1, prog2 and prog3. Note that myprog must not be compiled because this will destroy the interpreted definition.

```
(de savefun (funs file)
   (prog (oldout)
      (setq oldout (wrs (open file 'output)))
      (cond
         ((atom funs) (prettyprint (mygetd funs)))
         (t (mapc funs 'ppmygetd)))
      (terpri)
      (printc 'fin)
      (wrs oldout)
      (close file)))

(de ppmygetd (!*!*thingz)
   (prettyprint (mygetd !*!*thingz)))
```

DISPLAY - displays the file "file" to the screen.

USAGE:

```
(display examples)
```

displays this file. Note that DISPLAY is a fexpr and therefore does not evaluate its argument - hence no quote is needed for its argument.

```
(df display (file)
   (prog (temp oldinput oldll)
      (setq oldll (linelength 80))
      (setq oldinput (rds (open file 'input)))
loop  (setq temp (readch))
      (cond
         ((eq temp !$eof!$) (go end))
         ((eq temp !$eol!$) (terpri) (go loop))
         (t (princ temp) (go loop)))
end   (terpri)
      (linelength oldll)
      (rds oldinput)
      (close file)))
```

MYGETD - this function is used by savefuns to get a function definition. It also copes with global definitions flagging and property lists.

```
(de mygetd (def)
   (prog (fun head)
      (setq fun (getd def))
      (cond ((plist def) (ppplist def)))
      (cond
         ((null fun)
            (cond
               ((boundp def)
                  (return
                     (list
                        'setq
                        def
                        (mkquote (eval def)))) )
               ((plist def) (return nil))
               (t (printerror (list def 'is 'not 'set))
                  (return nil)))) )
      (setq head (car fun))
      (return
         (cond
            ((eq head 'expr)
               (append (list 'de def) (cddr fun)))
            ((eq head 'fexpr)
               (append (list 'df def) (cddr fun)))
            ((eq head 'macro)
               (cond
                  ((codep (cdr fun))
                     (printerror
                        (list
                           "macro "
                           def
                           " is a codepointer"))
                     nil)
                  (t (append
                        (list 'dm def)
                        (cddr fun)))) )
            (t (printerror
                  (list
                     "The function "
                     def
                     " is compiled"))
               nil)))) )
```

PPPLIST(def) prettyprints out the lisp commands used to put the properties of def onto its property list.

```
(de ppplist (def)
   (prog (l)
      (setq l (plist def))
loop  (cond
         ((null l) (return nil))
         ((atom (car l))
            (prettyprint
               (list
                  'flag
                  (mkquote (list def))
                  (mkquote (car l))))
            (setq l (cdr l)))
         (t (prettyprint
               (list
                  'put
                  (mkquote def)
                  (mkquote (caar l))
                  (mkquote (cdar l))))
            (setq l (cdr l))))
      (go loop)))
```

PRINTERROR - this function is used by mygetd to print error messages to the screen.

```
(de printerror (l)
   (prog (oldoutput)
      (setq oldoutput (wrs nil))
      (printlist l)
      (terpri)
      (wrs oldoutput)))
```

PRINTLIST - this function is used by printerror to print a list.

```
(de printlist (l)
   (cond
      ((null l) (terpri))
      (t (princ (car l))
         (princ " ")
         (printlist (cdr l))))  )
```

Appendix F: Reserved words

Although Cambridge LISP 68000 has no reserved words as such, it does make use of certain words internally which if used elsewhere could lead to problems. All the words in this appendix are used by the system and should not be redefined.

*XXXXXXXX
The following functions, with names begining with * are part of the compiler.

*bcc	*carcheck	*catchbinder
*catchunbinder	*cmpd2d	*cmpd2g
*cmpd2h	*cmpd2q	*cmpd2s
*compare	*dealloc	*fluidbind
*fluidrstr	*fntype	*gentrace
*jumpnil	*jumpt	*lbase
*lbl	*link	*linke
*load	*loadaddr	*loadexp
*loadnil	*lquote	*mapsk1
*moved2d	*moved2g	*moved2h
*moved2s	*moveg2d	*moveh2d
*movei2d	*moveq	*moveq2d
*moves2d	*newglobal	*stnil
*store	*throw	*tstd

The following variables with names beginning with * are for internal use and should not be modified.

*block	*backgag	*carcheckflag
*cdl	*countflag	*ctrace
entries	*gonest	*itree
*macro	*maxnargs	*n
*opt	*prarg	*predicate
*prmsg	*prval	*quotes
*trace	*unnamed	*xdefn

Functions used or required by the compiler

allocatequote
allocatequote.open
apply.open
atom.comtst
atsoc.conditionalmacro
callglob
car.anyreg
car.anyregr
car.conditionalmacro
car.open
casego.compfn
catch.compfn
cdifference.open
cdr.anyreg
cdr.anyregr
cdr.conditionalmacro
cdr.open
cminus.open
commutative.conditionalmacro
conc.compilermacro
cond.compfn
cons.compfn
cons.open
cplus.open
ctimes.open
eq.comtst
eq.conditionalmacro
eqcar.condtstmacro
equal.conditionalmacro
equal.open
errorset.open
eval.open
evlis.conditionalmacro
flagp.open
function.compfn
gensym.open
get.open
getbps
getd.open
getentry
getv.open
go.compfn
gts.open
iadd1.open
idifference.open
igreaterp.comtst
ileftshift.open
ilesscdr
ilessp.comtst
ilogand.compilermacro
ilogand2.open
ilogor.compilermacro
ilogor2.open
ilogxor.compilermacro
ilogxor2.open
imax.compilermacro
imin.compilermacro
iminus.open
iplus.compilermacro
iplus2.open
irightshift.open
isub1.open
itimes.compilermacro
itimes2.compilermacro
itimes2.conditionalmacro
itimes2.open
izerop.compilermacro
list.compfn
logand.compilermacro
logor.compilermacro
logxor.compilermacro
makebps
mapcar.conditionalmacro
maplist.conditionalmacro
max.compilermacro
min.compilermacro
mkquote.compilermacro
ncons.open
pairp.compilermacro
planthalfword
planthalfword.open
plus.compilermacro
prog.compfn
putv.open
quote.conditionalmacro
quotecell
return.compfn
rplaca.open
rplacd.open
rplacw.open
setq.compfn
times.compilermacro
xcons.open
zerop.compilermacro

Flags or properties used by the compiler

anyreg	**anyregr**
cnewnam	**codelist**
compfn	**compiler**
compilermacro	**comtst**
conditionalmacro	**condtail**
condtstmacro	

Function used by the AVL tree module

mark-balanced	**mark-double-unbalanced**
mark-left-unbalanced	**mark-right-unbalanced**
rotate-left	**rotate-right**
values-in-tree1	

Print functions

prinl0	Used by the circular list printer.
prinl1	Used by the circular list printer.
prinl2	Used by the circular list printer.

Miscellaneous

The following names should be avoided also:

case-shift-constant	Constant used by the ***lower** system.
fasl	Internal function used in the process of loading modules
gctrap	The function **gctrap** is intended for system programmer use while debugging the garbage collector. It is not useful to the normal user.

Commodore Business Machines, Inc.
1200 Wilson Drive, West Chester, PA 19380

Commodore Business Machines, Limited
3370 Pharmacy Avenue, Agincourt, Ontario, M1W 2K4