

CAMBRIDGE LISP



REFERENCE
MANUAL

CAMBRIDGE LISP



PART NO 0410, 009
ISSUE NO 1
JULY 1985

© Copyright Acorn Computers Limited 1985

Neither the whole or any part of the information contained in, or the product described in, this manual may be reproduced in any material form except with the prior written approval of Acorn Computers Limited (Acorn Computers).

The product described in this manual and products for use with it, are subject to continuous developments and improvement. All information of a technical nature and particulars of the product and its use (including the information in this manual) are given by Acorn Computers in good faith.

In case of difficulty please contact your supplier. Deficiencies in software and documentation should be notified in writing, using the Acorn Scientific Fault Report Form to the following address:

Sales Department
Scientific Division
Acorn Computers Ltd
Fulbourn Road
Cherry Hinton
Cambridge
CB1 4JN

All maintenance and service on the product must be carried out by Acorn Computers' authorised agents. Acorn Computers can accept no liability whatsoever for any loss or damage caused by service or maintenance by unauthorised personnel. This manual is intended only to assist the reader in the use of the product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any error or omission in, this manual, or any incorrect use of the product.

Published by Acorn Computers Limited,
Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN.

Within this publication the term BBC is used as an abbreviation for the British Broadcasting Corporation.

NOTE: A User Registration Card is supplied with the hardware. It is in your interest to complete and return the card. Please notify Acorn Scientific at the above address if this card is missing.

ISBN 0 907876 42 0 Acorn Scientific

Contents

	Part I	1
1	Introducing Acorn Cambridge LISP	3
1.1	Installing LISP	3
1.2	A Short Introduction to Acorn Cambridge LISP	3
1.3	Running LISP	4
1.3.1	Start-up Options	5
1.4	Use of Space	6
1.5	An example session with Cambridge LISP	6
1.6	Finding out what is available	7
1.7	Compatibility with Acornsoft LISP	8
2	Preparing Programs	9
2.1	General Remarks	9
2.1.1	Special Characters	9
2.1.2	Case	9
2.1.3	Defining Functions	10
2.1.4	Macros	10
2.1.5	Error Recovery	11
2.2	The LISP Editor	11
2.2.1	Introduction	11
2.2.2	General background	12
2.2.3	Editor commands	13
2.2.4	Entering an s-expression to the editor	15
2.2.5	Limitations	16
3	Implementation Features	17
3.1	Preserve	20
3.2	Load-on-call Facility (FASL)	21
4	Input and output	23
4.1	I/O Routines	23
4.1.1	Open/Close	23
4.1.2	wrs/rds	23
4.2	Printing	23
4.3	Reading	25
	Part II	27
5	Functions and Variables	29
5.1	Characters	31
5.2	Specialised variables	32

5.3	Atoms and Values	33
5.4	Dotted Pair Functions	34
5.4.1	Tagged Cons cells	36
5.5	List Processing Functions	38
5.6	List Equality and Searching	40
5.7	Pointer Replacement Functions	43
5.8	List Manipulating Functions	44
5.9	Creation of symbols	46
5.10	Flags and Property Lists	47
5.11	Function Definitions as Values	49
5.12	Vector Operations	50
5.13	The AVL Module	51
5.14	Arithmetic Functions	52
5.15	Basic Arithmetic Operations	56
5.15.1	Modulo Arithmetic Functions	59
5.15.2	Rational Arithmetic Operations	61
5.15.3	Trigonometric Calculating Functions	62
6	Control Structures	63
6.1	Common LISP Control Structures	66
7	Loops	69
8	Logic Functions	71
8.1	Bit-level Operations	71
9	I/O and File Handling	75
9.1	Files	75
9.2	Printing	76
9.3	The Programmable Reader	80
9.4	Syntax	81
9.4.1	Character level syntax	83
9.5	Interacting with the LISP supervisor	84
9.6	Saving Work	85
10	Evaluating Functions	87
10.1	Declarations and Binding	88
10.2	Function Definition	89
10.3	Compiler Functions	92
11	Editor entry Points	95
12	Error Control	97
13	Debugging in LISP	99
13.1	Tracing Functions	99
13.2	Tracing Memory Use	100
13.3	Timing Functions	101

14	Miscellaneous Functions	103
14.1	Graphics Functions	104
	Appendix A Error Messages	107
	Appendix B Bibliography	113

Part I

1 Introducing Acorn Cambridge LISP

The Acorn 32000 Cambridge LISP system runs under Panos, the operating system for Acorn Cambridge Series computers, with a minimum of 512K of memory. This section describes the main features of the Acorn Cambridge 32000 LISP, and comments on the facilities provided. Please note that this manual is not a tutorial; appendix B contains several references to such texts. Throughout this manual, except where an alternative meaning is obvious, 'LISP' refers to the Acorn 32000 implementation of Cambridge LISP.

1.1 Installing LISP

Details of installation are given in the *User Guide*, supplied with the system. Note that even though the LISP system is supplied on DFS formatted floppy disc, it must still be installed as described.

1.2 A Short Introduction to Acorn Cambridge LISP

Cambridge LISP was originally developed to provide support for an ongoing research project in computer algebra. It is intended for running experimental programs, and so it makes a policy of checking for exceptional cases (e.g. car or cdr of atoms) and tries hard to provide clear and concise diagnostics. The expectation that the system would be used for writing parts of algebra systems has led to the inclusion of an arithmetic package that puts consistency above efficiency: integers can grow to be any size, the normal arithmetic primitives accommodate rational numbers, and there is a well-defined interface between exact and floating point number representations. The system provides a number of character handling facilities, can select and use several input/output streams, and has a built-in LISP prettyprinter.

To a large extent, the system is compatible with a proposal for a LISP standard that was put forward by Professor A.C.Hearn and others of the University of Utah and the Rand Corporation. A short bibliography is provided in appendix B.

For users coming to Cambridge LISP from other dialects of LISP, your attention is drawn to the following points:

1. Function definition is performed with the functions `de` or `df`;
2. The function associated with an identifier is its value;
3. The distinct value and function definition cells of other LISP dialects are not supported.

1.3 Running LISP

To run LISP, type

```
-> Lisp -image <image file name>
```

to the Panos prompt. The keyword '-image' must be supplied, followed by the full (or relative) pathname of the directory which contains the image files. On the distributed LISP system, the image directory is ':0.i' on the DFS, and '\$.PanosLib.LispImage' on the AD/NFS.

Panos locates the executable LISP program 'Lisp-rif' by searching in the directories given as values to the global string variable 'Cli\$Path'. See the *Panos Guide to Operations* for a description of global variables. On the distributed system, 'Cli\$Path' is initialised to search firstly in PanosLib on the AD/NFS, (and each drive in turn, starting with drive 0 on the DFS) and then in the current working directory for all runnable programs.

If -identify is specified, you will be given an indication of the store in use, then an initial store image is loaded from the directory image. As more complex use is made of LISP, the various store preservation functions can be used to produce customised versions. For example, if the REDUCE system is available and is in the current directory, typing:

```
-> LISP -image reduce
```

will run REDUCE.

The process of customisation takes the form:

```
LISP -image oldimage -dump newimage
```

Then, when the LISP function (`preserve`) is called, the store image in `oldimage` is copied to `newimage`, together with updates made before the call to (`preserve`)

1.3.1 Start-up Options

The options available at the point of starting LISP are:

-from

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

-to

If present expects a 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 store image and fast load modules. The default is \$.i.

-dump

If present expects the name of a directory in which to put the store image and fast load modules generated in the run, by use of module and preserve. The default is the image directory.

-imageroot -dumproot

These options specify a directory for the root section of saved files. This is useful for use with the DFS where a complete image may overflow one disc.

-leave

Expects a number of bytes (in units of 1024) that LISP will leave for the Panos to use as workspace. This should normally not be needed. The default is 20.

-store

Expects a number of bytes (in units of 1024) that LISP will use for this run. It is useful to give exactly reproducible runs, or to see how well some program would behave if used with a smaller computer than the one you have.

-identify

This option enables a few lines of start-up information to be displayed on entering LISP. This gives the version number, the amount of space used up by LISP, the image size, the date and time the store image was created, and how much store was used.

-help

Displays help information: a brief synopsis of the start-up options.

1.4 Use of Space

LISP will attempt to acquire as much space as it can, leaving a little behind for other operating system activities. This behaviour can be changed by use of options as described above. If -identify has been specified, messages produced at the start and finish of each run give some indication of how much store is used and how much was available. If garbage collection becomes too frequent, more store is needed. There is no 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. Note also that the LISP compiler does not take up much space until it is used, and it can be removed using `excise` when it is finished with.

1.5 An example session with Cambridge LISP

The following example shows the dialogue during a short session with LISP. A few variables to configure the environment are set, a short LISP function is defined and tested, and some function definitions are read in from a file.

The prompt '->' signifies the Panos prompt.

The sign '>' is not a LISP prompt, but a signal that what follows is an evaluation returned by LISP.

```
->Lisp -image $.PanosLib.LispImage -identify
Acorn Cambridge Lisp entered in about 680 Kbytes
Store image was made at 11:38:27 on 10 May 85
Lisp version - 1.0 Apr 85 image size = 125244 bytes

Started at 15:33:45 on 14 Jun 85 after 0.01+9.15 secs - 31.8% store used

(setq !*comp nil)
> nil

(de flatten (L)(cond
```

```

(null L) nil)
(atom L)(list L))
(t{nconc (flatten (car L))
      (flatten (cdr L))})

> flatten

(flatten 'a)

> (a)

(compile '(flatten))
> **128 bytes 170 ms compiling flatten
> (flatten)

(flatten '(a))
> (a)

(flatten '((a)((1 2 3)1(((1)
> (a 1 2 3 1 1)

> *** End of file detected

> *** END OF RDF

(stop)

> End of Lisp run after 19.50+14.49 secs - 66.1% store used
->

```

1.6 Finding out what is available

There are three ways of finding out what is available in Cambridge LISP. The first is to try something and see if it works. Many of the functions provided have the same specifications as those in other LISP dialects as described in various textbooks and reference manuals. In particular, the Standard Lisp is close to this implementation. See the references in Appendix B.

The second suggestion for checking what might be available is to look at the object list. LISP keeps the names of all atoms - and hence all functions - that it knows about in a structure known as the object list.

The function `obl` `ist` which does not need any arguments, returns a list of all items held in this structure. It is a sort of index to the collection of available functions.

The third and best method is to consult Part II of this manual. Particular incompatible or especially useful features of the Cambridge system are noted below. See also appendix B for references to other documents which describe similar Cambridge LISP systems, for example, most of the functions provided have essentially the same specification as the versions defined in either the *LISP 1.5 users manual* or the *Stanford LISP reference manual*.

1.7 Compatibility with Acornsoft LISP

Acornsoft LISP (see appendix B) is a small version of LISP for the BBC Microcomputer. You will find that most Acornsoft LISP programs run under Cambridge LISP after modest changes are made.

Acornsoft uses upper case identifiers and accordingly `*lower` should be set to `t` in Cambridge LISP:

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

Acornsoft uses `defun` to define both `eval/spread` and `noeval/nospread` functions; the distinction being made by the format of the parameters in the definition. You may equate functions by typing:

```
(setq defun de)
```

Lastly, the hyphen '-' in Acornsoft LISP is not defined as a break character. You may wish to edit your source or to use `setsyntax` to redefine this. Using `PRIN` in Acornsoft LISP to tidy up the use of break characters in your program text is advised.

2 Preparing Programs

2.1 General Remarks

2.1.1 Special Characters

LISP is normally used in an interactive fashion with program material being entered on line. When it is desired to import program material from other LISP systems, or to generate material off line, it is important to remember that many characters have a meaning assigned to them by the read functions of the system.

See `setsyntax` for a description of the initial definitions. `!` used as a prefix causes the character immediately following to be accepted without special interpretation. This is useful, for example, in specifying pathnames which include a dot, e.g.

```
(rdf 'lspdir!.tsp)
```

Enclosing a file name in double quotes will also cause the following name to be taken literally, e.g.

```
(rdf "lspdir.tsp")
```

If an identifier is desired which includes a hyphen, there are two courses of action:

- (a) write it as a double!-barrelled!-word
- (b) call `(setsyntax "- " break!-character nil)` then write the double-barrelled-name without the need for the exclamation marks.

2.1.2 Case

Cambridge LISP is case sensitive and all supplied functions are named in lower case. When handling program material referencing store LISP functions it may be advantageous to set the variable `*lower` to `t` to enable continued references to the upper case versions of their names:

```
(setq !*lower t)
(SETQ F00 T)           this now equivalent to (setq foo t)
```

2.1.3 Defining Functions

The function `rdf` will read in a source text file and execute the statements found there. The particular functions used to define functions and macros e.g. `def` may not prove compatible with LISP source text from all other LISP systems.

The normal way of defining functions will be to use `(def ...)` and `(df ...)`. The format for these is:

```
(de <function name> (arg1 arg2)
  <body>)
```

as in

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

As well as functions that have their arguments spread out, it is possible to define functions which expect an evaluated list. These are sometimes known as `lexprs`. In Cambridge LISP they are defined (for example) by

```
(de mylfunc l
  (mapcar l (function print)))
```

If the LISP compiler is available in the image directory and the global flag `*comp` is non-nil, `de` automatically invokes the compiler. Note that `de` is a special form, and you do not have to put quote marks in front of its arguments.

Functions which do not evaluate their arguments can be defined using `df` or `dcf`: for details see section 10.2.

2.1.4 Macros

As well as functions, there are also macros. The body of a macro is evaluated to give a form that is then evaluated. For example:

```
(dm if (u)
  (list 'cons
    (list (cadr u) (caddr u))
    (list 't 'caddr u))))
```


is an approximation to the definition of the `if` conditional.

The functions that define new functions (i.e. `de`, `df` and `dm`) will print a warning message when they redefine an existing function or macro.

Note that 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 is 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` which can cope with any number of arguments. An atomic variable list for a `lambdaq` is illegal; the only valid format for the parameter field is a list with a single entry.

2.1.5 Error Recovery

Some errors cause entry into an iterative break-loop, which prompts for a character which will determine a variety of ways of exiting the loop. These are explained in the prompt:

```
Lisp break>
Q to quit, A to Abort, C to Continue, or . <expression>
```

2.2 The LISP Editor

2.2.1 Introduction

The Cambridge LISP full-screen editor described in this document is a powerful structure editor written entirely in LISP. It may be invoked via two functions: the first of these, `sedit`, edits the s-expression given as its argument. The edited copy is returned. `fedit` edits the definition of a function given the function's name as an argument; the display portion of the editor has been especially tuned for editing function definitions.

As well as edit existing functions, `fedit`

can be used to create new ones, specifying the type of the function as an optional second argument, i.e.

```
(fedit myfunction fexpr)
```

The editor sets up a template for the function, and the user fills in its definition using the normal editor commands. If `myfunction` already exists, the second argument is ignored, and the editor entered as normal.

On exit from `fedit`, the edited function is compiled if the variable `*comp` is currently set to a non-nil value. The function is also checked to see if it has been changed from an `expr/lexpr` to a `fexpr` (or vice-versa), and if so a warning message is printed that unexpected effects might occur on calls to the function from compiled code.

2.2.2 General background

The editor commands are designed around the concept of the 'current s-expression', which is visible on the screen at all times, the first character of which is highlighted by an inverse-video block (the 'edit pointer'). The current s-expression can be any one of the following:

- (1) an atom;
- (2) a LISP list starting with an opening parenthesis.
- (3) The `cdr` (tail) of a LISP list; in this case the inverse-video block is over the blank immediately before the `car` (head) of the current s-expression.

Initially, on entry to the editor, the current s-expression is the whole of the structure being edited, and the screen will resemble figure 1.

```

h[ead] t[ail] u[p] b[egin] l[ook-f]or] ma[rk]-mo[ve] z[oom]i[n]-z[oom]o[ut]
d[ele]te] r[e]place] s[plice] i[n]sert] [u]n[do] c[hange] [e]x[plode] #[hash]
e[val] w[indup] q[uit]      ?[redraw]

```

```

(lambda(a b)
  (cond
    ((null a) b)
    (t
     (cons
      (car a)
      (append & b) ) ) ) )

```

Figure 1 The Lisp Editor

The three lines at the top of the screen form a menu containing the more important (mostly single character) editor commands; then the structure being edited is displayed (in this case a function definition). The ampersands (&) indicate detail that has had to be suppressed for the structure to fit on the vdu screen.

2.2.3 Editor commands

Elementary moving

When the arrow keys are pressed, the edit pointer does not move from character to character like a cursor in a text editor (e.g. the Panos editor), but jumps from one s-expression to the next. The best way to familiarise yourself with the edit pointer's style of movement is to experiment.

In addition to using the arrow keys, there are several commands to move the edit pointer around, and thus make other parts of the structure the current s-expression. They are:

- h - Move to the head of the current s-expression.
- t - Move to its tail.
- u - Move up one level; inverse of head and tail.
- b - Move back up to the beginning of the smallest enclosing list.

Find and move

The next four commands perform more complicated location changing, the first two prompting at the bottom of the screen for an s-expression for the command to find.

- lf - Look for and move to the next occurrence (in print order) of the user-specified s-expression. The area in which the search takes place starts at the s-expression immediately after the current one, and the search begins at the end of the function, as it appears on the screen.
- lk - Look for and move to the next occurrence (in reverse print order) of the user-specified s-expression, the search beginning with the s-expression immediately before the current one.
- l - Repeat the last look for command.
- ma - Insert a mark at the current edit position. The mark is a new GENSYM ed atom of the form `M00<n>`.
- mo - Move to a previously set up mark.

Structure modification

The following commands perform structure modifying operations:

- d - Delete the current s-expression.
- r - Replace current s-expression with a user-specified s-expression read from the bottom of the screen. See also section 2.2.4 below headed 'hash variables'.
- c - Repeat the last structure modification.
- i - Insert a user-specified s-expression at the edit pointer, performing a cons of the user's expression with the current s-expression.
- s - Splice in a user-specified list at the edit pointer. If the current s-expression is an atom, it is replaced by the elements in the new list.

n - Undo the last structure modification command. A record is kept of all the modifications made to the structure, and successive undo commands will restore the function to its state prior to editing.

x - Explode the current s-expression so that it can be modified using one of the commands above.

Reformatting the screen

The next three commands change the appearance of the screen:

? - Reinitialise screen, and redisplay (useful if the screen has been messed up for some reason).

zi - Zoom in and display current s-expression in more detail. The command will work only when the edit pointer is not at the top of the screen.

zo - Zoom out; opposite of zoom in. The zoom is not allowed if it would cause the current s-expression itself to be suppressed as a detail.

Eval loop and leaving the editor

The final three commands allow the user to enter a read-eval-print loop while still inside the editor, and allow the user to terminate the edit session and exit:

e - Enter a read-eval-print loop; typing 'fin' terminates the loop and causes the editor to be reentered.

w - Windup and update edited structure.

q - Quit and do not update structure.

2.2.4 Entering an s-expression to the editor

The 'find', and some of the structure modification commands, require the user to enter an s-expression; in these cases the user is prompted at the bottom of the screen by ?. The s-expression input can cover as many lines as desired, the **(DELETE)** key deleting the last character typed (even back across previous lines), and **(TAB)** inserting three spaces. Typing **(ESCAPE)** at any point completely abandons the current command.

Hash Variables

For convenience, in ‘cut and paste’ operations, the editor sets up the variable `#0` to contain the current s-expression, and, if it is a list, `#1`, `#2`,... to contain the values of the top-level elements of it. There is also the command ‘`#`’ which sets the value of the variable `#` to be the current s-expression at the time the command was issued. The values of these `#`-variables are substituted into each s-expression and typed wherever they occur in it. These `#`-variables are also available in the read-eval-print loop inside the editor invoked by the `eval` command.

Miscellaneous features

The editor stores the name of the last function it edited in the global variable `**edit-last-function`, and if the function editor is called without being given the name of a function, it re-edits the last one.

The menu of editor commands displayed at the top of the screen can be changed by altering the contents of the global variable `**edit-menu`. The variable must contain a list of strings, each not longer than the width of the screen, each string to go on one line. The position on the screen of the structure being edited is automatically adjusted depending on how much space the menu takes up.

Since there is the facility to enter a supervisor loop within the editor, it is possible (and sometimes very useful) to enter the editor within that loop and thus edit at a second (or even higher) level. To help the user keep track of which level he is in, and the name of the function he is editing at each level, the editor maintains this information in the variable `**edit-level`.

The editor maintains a list of all the functions that have been edited in the current LISP session in the global variable `**edit-functions`. Thus it is easy to keep track of which functions have been changed and so need to be written back out to disc at the end of a session.

2.2.5 Limitations

The editor cannot deal with reentrant LISP s-expressions. Currently it cannot express vectors or strings adequately for easy editing of their internal constituents.

3 Implementation Features

Cambridge 32000 LISP is a value cell system. This means that for each identifier in any LISP program there is a single word used to hold the current value of that variable. When variables get bound or unbound, the contents on the value cell will be updated appropriately. This scheme has the advantage of being efficient and lends itself to graceful error recovery. For interpreted code, this mechanism provides what in 32000 Cambridge LISP is referred to as `fluid` scope rules (this is known as `special` in some other LISP dialects). The visible manifestations of a value cell LISP implementation are:

- a) A name can only be associated with one value at once. If you try to use a local variable called `car` it will not be possible to get at the built-in function `car` while within the scope of your local variable.
- b) Function definition and `cset` are variants of `set`. If you assign to a name not bound as a local variable, your assigned value will be stored 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.

When LISP is given something to evaluate it has to decide what to do about the function it is given. The Cambridge system works in the following way:

- (i) 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;
- (ii) otherwise LISP replaces the function part by its value and goes back to step (i).

Thus functions get repeatedly evaluated until they make sense. All the LISP built-in functions are 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 sometimes go into a loop. It must be stressed that this loop has the same status as any other non-terminating program; this constitutes a user error

and not a failure in the LISP system. Some other LISP systems will only evaluate function parts once, or perhaps twice before demanding that a recognizable form has been reached.

The symbol `lambdaq` is used in Cambridge LISP when a function must avoid evaluating its arguments too early. The use of `lambdaq` is like that of `lambda` except that it inhibits the evaluation of its arguments. Thus a function rather like the built-in `quote` can be redefined by:

```
(setq my!-quote '(lambdaq (a) a))
```

Within the body of a function introduced in this way, `eval` can be called to get arguments explicitly evaluated.

The Cambridge LISP interpreter implements the implied `progn` feature of many LISP systems, where the function `progn` can be omitted in nearly all circumstances. If a function is given too few arguments, then additional arguments of `nil` are supplied. If too many arguments are given, then excess arguments are ignored, after they have been evaluated in the case of `eval` functions.

Serious users of LISP will 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 can not be used indiscriminately. To use the compiler at all, an image directory containing the module `lspcomp` must be available to the LISP system.

If `f1`, `f2` ... are functions that have just been defined, a call:

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

will replace the definitions of the functions with compiled code that is essentially equivalent. Also functions defined using `de` or `df` can be compiled automatically if the variable `*comp` is set to `true`.

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

- a) In compiled code, variables bound either by `lambda` or by `prog` are treated as being local to the function binding them unless they have been specially declared `fluid`. Thus if use is made of free variables, those variables should be declared using a call


```
(fluid '(v1 v2...))
```

before compilation is attempted.

- b) The compiler freezes some assumptions and knowledge about routines called by the function being compiled. These assumptions can sometimes lead to inconsistencies if the user attempts to redefine functions later. The compiler will not treat calls to user-defined special forms (i.e. functions defined using `df` or `dm`) properly unless the special form has been defined before any attempt is made to compile a call to it.

These restrictions arise because the compiler generates open code or special calling sequences for many standard LISP functions: certain function names are treated specially and cause the compiler to generate fixed sequences of orders irrespective of what definition the user thinks should be associated with the name. These include `car`, `cdr`, `cons`, `putv`, `getv`, the integer arithmetic routines: `iplus`, `iminus` etc. and a few more. A complete list is given in the table below. If one of these is redefined, it will alter the behaviour of interpreted code but will not affect compiled code in any way. In all other cases, redefining a function will cause all references to that name to use the new definition.

Note again that if a variable is to be bound in one function but used in another (or if it is to be treated as a global variable), then before any function mentioning the variable is compiled, a declaration of the following form should be obeyed:

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

The declaration can be removed after compilation by a corresponding call to `unfluid`. Note that `fluid` takes the place of declarations known as `special` and `common` in some other LISP systems. When all user functions have been compiled, a substantial amount of space can be recovered by telling LISP to throw away the compiler. This is done by calling `excise` with argument `lspcomp`.

Functions fixed by the compiler

<code>apply</code>	<code>car</code>	<code>cdr</code>
<code>cdifference</code>	<code>cminus</code>	<code>cons</code>
<code>cplus</code>	<code>ctimes</code>	<code>equal</code>
<code>flagp</code>	<code>gensym</code>	<code>get</code>
<code>getd</code>	<code>getv</code>	<code>iadd1</code>

idifference	ileftshift	ilogand2
ilogor2	ilogxor2	iminus
iplus2	irightshift	isub1
itimes2	ncons	putv
rplaca	rplacd	rplacw
xcons		

3.1 Preserve

Particularly when developing a large program, it is useful to be able to dump the state of LISP to a disc file, then pick it up again in a later run. The function used for this is called `preserve`, and `(preserve)` writes a store image to the file with the name `lsproot` in the dump (or image, if no dump was specified) 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` will write to a member `lsproot` together with any other members defined by the FASL facility. If image and dump are the same, then the previous `lsproot` is renamed `bakroot` before the store is dumped.

Store images can only be loaded at the very start of a LISP run, and on entry to LISP the parameter image will give the directory in which to find the store image which is loaded. Thus:

```
-> LISP -IMAGE $.OLDIMAGE -DUMP $.NEWIMAGE
. . .
(preserve)
```

Dumps a store image to the directory `$.newimage`, and

```
-> LISP -IMAGE $.NEWIMAGE
. . .
```

reloads it and continues with the computation. If `preserve` writes out a file successfully, LISP will stop 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 is reloaded, the user-supplied function `initialsupervisor` will be called rather than the standard LISP supervisor. If `initialsupervisor` is exited, then LISP will stop.

3.2 Load-on-call Facility (FASL)

The following functions provide a load-on-call facility in LISP. Note that all code is loaded from members of the directory called 'image' (unless the call has designated another directory for that function) and any generated code is written to the same directory, unless a dump has been specified in the call to LISP. The file names `lsproot`, `bakroot`, and `lsperrs` should be avoided, as these are used by the basic system. Various other modules with names beginning with 'lsp' are part of the base system.

(module membername)

Until further notice the compiler will write Fast Load (FASL) data for functions it compiles to the file `membername` in directory 'dump'.

These files will be written as binary data. If `membername` is given as `nil`, or not given, `fasl` output is switched off and the compiler produces code in store.

(excise membername)

If `membername` is the name of a `faslfile` that has been loaded, (`excise membername`) replaces the definitions of all functions in that file with references to their disc versions. This frees the space that was consumed by the compiled code bodies. If `excise` is called with no arguments, it purges all loaded modules.

(preserve)

Write a file `lsproot` to `image` (or `dump` as described above) in a format suitable for LISP to reload it at the start of a subsequent run. If the store `image` is being written to `image`, then the previous `image` is renamed as `bakroot`. `preserve` will also close the output to a module thereby removing the necessity for a (module `nil`). After executing `preserve`, LISP stops.

The loading of modules is monitored if the appropriate value is given to `verbos`: after (`verbos 3`) a message is displayed each time a module is loaded from disc or removed from main memory.

4 Input and output

Input and output in LISP are based on the idea of selectable streams. At the start of a run, the stream designated `-from` (the default is the terminal) is opened and selected for input, and `-to` (default terminal) is opened and selected for output. All streams must be created by `open` before use: `open` returns as its result a LISP object that can be used to refer to a file; although this will often be a LISP atom that names the file, this fact should not be relied on.

4.1 I/O Routines

4.1.1 Open/Close

```
(open filename [input/output])  
(close file handle)
```

These `open` or `close` a stream. A stream must be opened before it can be selected. The second argument to `open` specifies whether the stream will be read from or written to. The value returned by `open` is a file handle for use with `close`, `rds`, `wrs`.

4.1.2 `wrs`/`rds`

```
(wrs filehandle):[select for writing]  
(rds filehandle):[select for reading]
```

`rds` and `wrs` affect all input and output (a possible exception being diagnostics which should always appear on the terminal); in particular, data to be read in, and LISP statements for execution are not kept separate.

4.2 Printing

There are four printing styles available in LISP. The first of these is intended to produce output that can be read back into LISP at a later stage, and so strings are printed with “ marks around them and any unusual

characters in atoms get prefixed with escape marks i.e. `!`. To use this print style:

`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 merely print the characters of atoms and strings without any extra markers.

`princ` basic print routine
`printc` as `princ` followed by call to `terpri`
`printcm` as `printc` but leaves a left margin of specified width.

On occasions it is useful to produce circular structures. The previous print styles will go into an infinite loop if they are asked to print such a list. This family however are safe, but more expensive.

`prinl` basic print for looped structure
`printl` as `prinl` terminated by a `terpri`
`printcl` as `prinl` with no escapes

The last style of printing is for the display of LISP programs, and provides a consistent style of indentation.

`superprint` print with indentation

For explicit control of layout on the page, the following functions are available:

`terpri` end a record (with a newline)
`eject` as `terpri` but prints 'end-of-page' (sends character with code 12, i.e. formfeed)
`linelength` sets the logical width of paper that can be used by `print`
`ttab` tab to specified column
`xtab` tab by a given number of columns

4.3 Reading

The read routines are:

- `tyi` get ascii code of next character
- `readch` get next character
- `read-token` gets next atom, and sets a variable `!*token!-type!*` to a classification code for it
- `read` get next s-expression or atom.

Characters obtained by calls to `readch` can be classified by the predicate functions:

- `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 effects of previous calls to `setsyntax`.

Characters can be packed in a buffer, and then assembled into complete atoms:

- `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

Notes

`numob` will fail if the buffer contents are anything other than a sensible string of digits, possibly with a leading sign.

`pack` must not be called before `clearbuff` has been used to set the buffer `boffo` to a standard initial state.

`read` either called explicitly by the user or called by the system to read in some more LISP program destroys the buffer contents. Therefore `pack` etc.

should only be called from within a nest of routines that both start and finish the atom assembly process.

`explode` takes any argument and returns a list of characters, the characters being those that `prin` would produce if it were handed the list or atom. In particular `explode` converts numbers into characters.

`explodec` is like `explode` but makes a list of the characters `prin` would have produced.

`compress` is an inverse of `explode`.

Part II

5 Functions and Variables

In this section, each function is provided with a prototypical header line, and each formal parameter is given a name and suffixed with its permitted 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
```

`putd` is a function with three parameters. The parameter `FNAME` is an identifier to be the 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
```

`is` is a function which 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 `apply` is being used as the user 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. The module of the system in which a function resides is also given. `Base` means it is part of the initial load, `core` means it is always loaded, but is defined in LISP. Other names are modules which are found in the image directory.

Argument Types

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

list

A list with each member being a dotted pair. i.e. ((a.b) (c.d) ...)

atom

Any type of number, string, id, vector, 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`.

ftype

The class of definable function types. The ids `expr`, `fexpr`, `subr`, `fsubr` and `macro`.

function

Anything that can be used as a function e.g. 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 i.e. less than 2^{24} . Arguments of this type can be passed to any function that expects a number or integer but the type of the result will not necessarily be a logical value.

sinteger

Signed integers that can be represented in one word i.e. modulus less than 2^{24} . Arguments of this type can be passed to any function that expects a number or integer but the type of the result will not necessarily be a logical value. They are used extensively by the compiler, and there are 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 “ ”.

5.1 Characters

The character `'` is used 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`.

(Parentheses are used in LISP input to form lists. To use the atom `(` see the entries under `!` and `lpar`.

{ Braces are used as super parentheses. A `}` will close sufficient opening parentheses 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 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. `(pr in 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`. The value of `*echo` is

*echo effect
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/printLoop) have *echo controlled by (prarg n). The initial value is nil. See prarg (section 9.5).

5.2 Specialised variables

Many variables with names beginning with * and & are for internal use and should not be modified. The variables documented below are the ones most likely to be useful to the general user.

*comp

Type: Variable

When the variable *comp is true, function definitions are compiled. See de, df and dm. The default is t.

*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.

*lower

Type: Variable

When true, all uppercase letters are translated to their lowercase equivalent.

*raise

Type: Variable As *lower but forces characters into upper case.

*plap

Type: Variable

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

5.3 Atoms and Values

`t`

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

`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 `(abc)` is 'really' `(abc . 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 will return `nil` for false (most will 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))`.

`(null U:any):boolean`

Type: eval, spread, Base

Returns `t` if `u` is `nil`.

`(setq variable:id value:any):any`

Type: noeval, nospread, Base

`setq` is the normal assignment operator in LISP. 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).

`(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.

(unset U:id):any

Type: eval, spread, Base

Is equivalent to (set U indefinite! value) but returns the old value of the variable.

(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.

csetq, pts

For compatibility with older LISP Systems, these are synonyms for setq and set respectively. pts is a synonym for set.

(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.

(idp U:any):boolean

Type: eval, spread, Base

Returns t if U is an id (i.e. an atom that is not a constant).

comma

Initial value of comma is the atom ','. See description of blank and dollar.

5.4 Dotted Pair Functions

Dotted pairs are the primitive data type in LISP. They are the product of functions from the cons class. The following are fundamental functions for their manipulation and creation

(cons U:any V:any):dotted-pair

Type: eval, spread, Base

Returns a dotted-pair (or cons-cell) 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

Check if U is a list created by cons.

(car U:dotted-pair):any
 Type: eval, spread, Base

```
(car(cons a b))
> a
```

The left part of U is returned. An error occurs if U is not a dotted-pair.

(cdr U:dotted-pair):any
 Type: eval, spread, Base

```
(cdr(cons a b))
> b
```

The right part of U is returned. An error occurs if U is not a dotted-pair.

caaaaar
 Type: eval, spread, core

Any name of the form cxxxxr where x represents 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 present implementation allows up to four letters between the 'c' and the 'r', so caaaaar to cddddr are provided for. These are the possible combinations:

car	caar	caaar	caaaaar	cdaddr
cdr	cadr	caadr	caaaadr	cddaar
	cdar	caddr	caaddr	cddadr
	cddr	cdddr	cadddr	cdddr
		cdaar	cdaaar	cddddd
		cddar	cdaadr	caadar
		cdadr	cdadar	cadadr
		cadar	caddar	cadaar

(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 is provided to inhibit this check. (`carcheck n`) directs the compiler to check at level `N`, where `N = 0` means no checking `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. `carcheck` also controls the level of checking performed by compiled versions of a few other low-level functions such as `putv` and `getv`. It is strongly recommended that (`carcheck 2`) be used even with programs that are thought to be fully debugged; the overhead is modest and the extra sensitivity substantial. If ultimate performance calls for (`carcheck 0`) this option should be used for just a few of the most time-critical functions.

(`profile U:integer`)

Type: eval, spread, lispcomp

After a call to `profile`, the LISP compiler will generate 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 gathered can be accessed by `readcount` or (more commonly) by `mapstore`.

(`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 MacLISP and InterLISP code to the Cambridge system.

5.4.1 Tagged Cons cells

(`acons U:any V:any`):list

Type: eval, spread, Base

32000 Cambridge LISP permits the creation of tagged cons-cells and their subsequent identification. This function is like `cons` but produces a different internal tag. This can be checked with the functions `aconsp` or `constype`. Similarly

```
(bcons U:any V:any):list
(ccons U:any V:any):list
(dcons U:any V:any):list
(econs U:any V:any):list
(fcons U:any V:any):list
(gcons U:any V:any):list
(hcons U:any V:any):list
```

(acons U:any):boolean
Type: eval, spread, Base

Checks to see that U is a cons-cell created by acons. Similarly

```
(bconsp U:any):boolean
(cconsp U:any):boolean
(dconsp U:any):boolean
(econsp U:any):boolean
(fconsp U:any):boolean
(gconsp U:any):boolean
(hconsp U:any):boolean
```

For example:

```
(acons (bcons 'a 'b))
> nil
```

(constype U:list):integer
Type: eval, spread, Base

constype returns an integer in the range [0..8] depending on whether U is a node created by cons, acons, , hcons. For example:

```
(constype (econs 'a 'b))
> 5
```

(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 type, t=1 gives an acons, . . . , t=8 gives an hcons. Values of t outside the range [0..8] give an error.

(xcons U:any V:any):dotted-pair
Type: eval, spread, Base

This is the same as See ncons and cons

(ncons U:any):dotted-pair

Type: eval, spread, Base

Is the same as (cons U nil). See cons, xcons.

5.5 List Processing Functions

(list [U:any]):list

Type: noeval, nospread, Base

A list of the evaluation of each element of U is returned.

```
(progn (setq x 5)
      (list 'the 'value 'of 'x 'is x ))
> (the value of x is 5 )
```

(list!* [U:any]):list

Type: MACRO

A list made up of the arguments given, with the last one used as the final tail of the list. Thus

```
(list!* p q r)
```

has the same effect as

```
(cons p (cons q r))
```

(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.

```
(append '(a b c) '(d e f))
> (a b c d e f)
```

(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.

```
(setq l1 '(a b c)
      l2 '(d e f)
      l3 '(g h i)
      (conc l1 l2 l3)
      > (a b c d e f g h i))
```

l2 now has value (d e f g h i)

(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.

(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 will fail if the list to be copied has been made cyclic through the use of rplaca or rplacd. . copy does not duplicate atoms or vectors: the copying operation is confined to lists as made up using cons.

(reverse U:list):list

Type: eval, spread, core

Returns a copy of the top level of U in reverse order.

```
(reverse '(a b (c d) e))
> (e (c d) b a)
```

(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. See reverse.

```
(progn (setq l1 '(a b c d e f)
        (setq l2 (reversewoc l1))
        l1)
      > (a)
```

l2 has value (f e d c b a)

(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.

(orderp U:any V:any):boolean

Type: eval, spread, Base

This is a predicate defining a self-consistent order relation between lists. That is if (orderp ab) and (orderp bc) are both true then.bl(orderp ac) is true. 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 upon.

5.6 List Equality and Searching

(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 upon.

(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. The test equal is close in meaning to requiring that the two expressions look alike when printed

(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.

(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.

```
(progn (setq key 2)
      (assoc key '((0 . key0)
                  (1 . key1)
                  (2 . "you got it")))
      > "you got it"
```

(atsoc U:any V:alist):(dotted pair nil)

Type: eval, spread, core

atsoc is exactly assoc except that it uses eq to test if a tag has been found instead of equal. See also sassoc.

(sassoc U:any V:alist FN:function):any

Returns the same result as (assoc U V) if U is present in V, otherwise the evaluation of FN is returned.

(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.

```
(member 'c '(a b c d e f))
> (c d e f)
```

(memq A:any B:list):extra-boolean

Type: eval, spread, core

Same as member but an eq check is used for comparison.

(delete 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.

`(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 will only occur once in the union list. See `append` for list merging/concatenation that does not remove duplicate entries, and `xn` below for list intersection.

`(setdiff U:list V:list):list`

Type: eval, spread, core

Returns the set difference between two lists; that is those members of U are not members of V using a member test.

`(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)`.

`(sortip U:any-list PREDICATE:id):any-list`

Type: eval, spread, core

As `sort`, but the input list is overridden with the result.

`(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.

```
(sublis '((a . A)(b. b)) '(a a bab b))
> (aab ab b)
```

`(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.

`(xn U:list V:list):list`

Type: eval, spread, core

Returns the set intersection of the lists U and V, i.e. those members of U that are also in V. See also `union`.

(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. But note:

```
(last '(a b c . e)
      > c
```

(length X:any):integer

Type: eval, spread, core

The top level length of the list X is returned.

```
(length '(1 (2 3) 4)
      > 3
(length '(1 2 .3)
      > 2
```

(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.

(neq U:any V:any):boolean

Type: eval, spread, core

neq is equivalent to (not (equal U V)).

5.7 Pointer Replacement Functions

The following group of functions operate directly on the data structures given as arguments rather than on copies. Accordingly, they do not cause memory to be exhausted, but if used unwisely will damage data and program irreparably.

(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. An error occurs if U is not a dotted-pair.

```
(setq l '(1 2 3 4))
(prog (ll)
  (cond [(setq ll (member 3 L))(rplaca ll 'a)
        (t nil)]))
```

leaves l with value (1 2 a 4)

(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)) In general the printl etc. forms of the printing functions should be used to print the results of these functions especially where circular structures may be created.

5.8 List Manipulating Functions

(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.

```
(compress '(A B C))
> ABC
```

(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 characters that appear in the list are those that would be produced if U were handed to `prin`.

(explodec U:any):id-list

Type: eval, spread, Base

As `explode` but no escape characters are produced in ids (c.f. `prin`).

(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.

(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.

arg	explode	explodec	explodecn	exploden
123	(!1 !2 !3)	(!1 !2 !3)	(49 50 51)	(49 50 51)
'cat	(c a t)	(c a t)	(99 97 116)	(99 97 116)
"cat"	(!" c a t ")	(c a t)	(99 97 116)	(34 99 97 116 34)

(pack U:any V:any):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 initialised 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 non-nil second argument, it forces all the characters it packs into lower case.

(packbyte U:any):nil

Type: eval, spread, Base

As pack but acting on internal codes.

(clearbuff):nil

Type: Base

This is a routine with no arguments that clears an internal buffer boffo which is used for constructing atoms. It must be called before an attempt is made to use pack etc. See pack, numob, mkatom.

(numob):number

Type: Base

If the characters placed in the buffer boffo (by pack etc.) represent a number, numob will form that number as a proper LISP object. If mkatom 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.

(mkatom):id

Type: eval, spread, Base

If a sequence of characters have been assembled in the buffer boffo (by calls to clearbuff and/or 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.

(mkstring):string

Type: Base

Creates a LISP string from the characters in boffo.

```
(progn (clearbuff) (pack 123) (pack 456) (numob) > 123456
(progn (clearbuff) (pack 'abc) (pack 'def) (mkstring) > "abcdef"
(progn (clearbuff) (pack 'abc) (pack 'def) (mkatom) > abcdef
```

5.9 Creation of symbols

(gensym):id

Type: eval, spread, Base

This creates an identifier which is not interned on the oblist and consequently not eq to anything else.

```
(eq (gensym) (gensym))
> nil
```

(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. See also symnam

(symnam U:id):id

Type: eval, spread, Base

symnam takes one argument which must be an identifier. Until symnam is called again, this identifier will be used to provide 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.

5.10 Flags and Property Lists

(flag U:id-list V:id):nil

Type: eval, spread, Base

U is a list of ids which are to be flagged with V. The effect of flag is that flagp will have the value t for those ids which 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

This returns t if U has been previously flagged with V, else nil. It returns nil if either U or V is not an id.

(get U:any IND:any):any

Type: eval, spread, Base

This 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).

`(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 will occur and no property will be placed. `put` cannot be used to define functions (use `putd` instead).

`(setplist U:id PLIST:alist):alist`

Type: eval, spread, Base

This replaces the property list of `U` with `PLIST`. The function returns the previous property list owned by `U`.

`(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`).

`(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.

`(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 will occur. `nil` is returned.

```
(progn
  (flag '(atm1 atm2) 'myFlag)
  (put 'atm1 'cost 1000)
  (put 'atm2 'price 2000)
  (print (plist 'atm1))
  (print (plist 'atm2)))
```

```
prints ((cost . 1000) myFlag)
      ((price . 2000) myFlag)
```

(obl ist):id-list

Type: Base

Returns a lexicographically 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 `obl ist` will have no effect on the object list.

(remob U:id):id

Type: eval, spread, Base

If `U` is present on the `obl ist` it is removed. This does not affect `U` having properties, flags, functions and the like. `U` is returned.

5.11 Function Definitions as Values

(putd FNAME:id TYPE:fype BODY:function):id

Type: eval, spread, Base

Creates a function with name `FNAME` and definition `BODY` of type `TYPE`. This should be one of the symbols `expr` or `fexpr`. If `putd` succeeds, the name of the defined function is returned. The effect of `putd` is that `getd` will return a dotted-pair with the function's type and definition. Likewise the `globalp` predicate will return `t` when queried with the function name. If function `FNAME` already exists (and messages are enabled by `(verbose t)`) a warning message will appear:

```
*** FNAME redefined
```

The function defined by `putd` will be compiled before definition (changing `exprs` to `subrs` and `fexprs` to `fsubrs`) if the `*comp global` variable is non-`nil`.

(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 `xsubr`, 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.

5.12 Vector Operations

(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 initialised to nil. An error will occur if UPLIM is less than zero, or if there is not enough space for a vector of this size.

```
(mkvect 4)
> %< nil,nil,nil,nil,nil %>
```

Note the printing convention used.

(vectorp U:any):boolean

Type: eval, spread, Base

Returns t if U is a vector.

(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.

(putv V:vector INDEX:integer VALUE:any):any

Type: eval, spread, Base

Stores VALUE into the vector V at position INDEX. VALUE is returned. A type mismatch error may occur, and if INDEX does not lie in 0... upbv(V) an error occurs.

(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.


```
(prog (V)
  (setq v(mkvect 3))
  (putv v0 "a b c")
  (putv v1 34)
  (putv v2 t)
  (putv v3 29)
  (print V)
  (print (upbv V))
```

prints

```
> %<"a b c",34,t,29 %>
> 3
```

(stringconcat U:string V:string):string

Type: eval, spread, Base

Returns the concatenation of the strings U and V.

```
(stringconcat "abc" "def")
> "abcdef"
```

(stringp U:any):boolean

Type: eval, spread, Base

Returns t if U is a string.

5.13 The AVL Module

The `avltree` module provides an efficient balanced tree lookup and deletion. It is used in the printing of circular lists (see `printl`, `princl`, `printcl`) and the maintenance of the LISP,bl oblist. The functions of interest to the user are documented below. The trees produced by this module make use of the `acons`, `bcons`, etc. functions, and are not general LISP trees.

(avl-add KEY:any TREE:avltree ORDER:function):avltree

Type: eval, spread, LISPAVLT

The `KEY` is added to the `avlb` balanced tree, using `orderas` the ordering predicate. `ORDER` must be an eval/spreadfunction 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 avltree.

(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 avlbaldnced 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:qvltree ORDER:function):avltree
Type: eval, spread, LISPAVLT

The KEY is deleted from the avltree.

(avlq-look{up KEY:any TREE:avltree ORDER:function):boolean
Type: eval, spread, LISPAVLT

The KEY is looked up on the tree using an eq test.

(values-in-tree T:avl-tree):list
Type: eval, spread, LISPAVLT

values-in-tree returns the list of value in the avltree in order. See also the functions avl-add and avlq-add.

5.14 Arithmetic Functions

Acorn Cambridge LISP supports arithmetic operations over four classes of numeric atoms:

small integers in the range -2^{24} to $+2^{24}$ integers of any size
floating point numbers rational numbers

It supports the usual operations over these, and additionally can perform modulus arithmetic over the small integers. A convenient naming convention has grown up whereby arithmetic operations are named e.g.

`add1` and `iadd1`; the `i-` prefix indicating the use of the function on integer arguments. `c-` and `m-` prefixes are likewise used to indicate functions specialised to handle arithmetic modulo `n` where `n` is a number set by `setmod`.

A number of functions are supplied for type testing and conversion.

`(fix U:number):integer`

Type: eval, spread, Base

Returns an integer which corresponds to the truncated value of `U`. The result of conversion retains 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).

`(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 implementation 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.

`(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. (i.e. `U = !2` returns `t` but `U = 2` returns `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. 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 required.

(geq U:number V:number):boolean

Type: MACRO, core

Returns t if $U \geq V$.

(evenp U:number):boolean

Type: eval, spread, Base

Returns t if U is an even integer.

(greaterp U:number V:number):boolean

Type: eval, spread, Base

Returns t if U is strictly greater than V, otherwise returns nil.

(igreaterp U:integer V:integer):integer

Type: eval, spread, Base

Similar to greaterp but only works for small integers. When appropriate, it may be faster than greaterp.

(ilessp U:integer V:integer):integer

Type: eval, spread, Base

Similar to lessp but only suitable for small integers.

(imax [U:integer]):integer

Type: noeval, nospread, Base

Similar to max but only suitable for small integers.

(imax2 U:integer V:integer):integer

Type: eval, spread, Base

Similar to max2 but only suitable for small integers.

(imin [U:integer]):integer

Type: noeval, nospread, Base

Similar to min but only suitable for small integers.

(iminusp U:integer):boolean

Type: eval, spread, Base

Similar to minusp but only suitable for small integers.

`(imin2 U:integer V:integer):integer`

Type: eval, spread, Base

Similar to `min2` but only suitable for small integers.

`(izerop U:integer):integer`

Type: eval, spread, Base

Similar to `zerop` but will not recognize floating point zero. See `iadd1`. In fact `(izerop xx) = (eq xx 0)`.

`(lessp U:number V:number):boolean`

Type: eval, spread, Base

Returns `t` if `U` is strictly less than `V`, otherwise returns `nil`.

`(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`.

`(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.

`(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`.

(numberp U:any):boolean

Type: eval, spread, Base

Returns t if U is a number (integer, rational or floating).

(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) .

(smallp U:number):boolean

Type: eval, spread, Base

Returns t if U is an integer that can be stored as a single word.

(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. See onep.

5.15 Basic Arithmetic Operations

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 will satisfy the same constraints. Failure to adhere to these constraints (e.g. 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 will at least never return a value that will 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.

(abs U:number):number

Type: eval, spread, Base

Returns the absolute value of its argument.

(add1 U:number):number

`(iadd1 U:integer):integer`

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`.

`(sub1 U:number):number`

`(isub1 U:integer):integer`

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.

`(difference U:number V:number):number`

`(idifference U:integer V:integer):integer`

Type: eval, spread, Base

Returns U - V.

`(minus U:number):number`

`(iminus U:integer):integer`

Type: eval, spread, Base

Returns - U.

`(plus [U:number]):number`

`(iplus [U:integer]):integer`

Type: noeval, nospread, Base

Forms the sum of all its arguments.

`(plus2 U:number V:number):number`

`(iplus2 U:integer V:integer):integer`

Type: eval, spread, Base

Returns the sum of U and V. This function is used in the expansion of `plus`.

`(times [U:number]):number`

(*itimes* [U:integer]):integer

Returns the product of all its arguments.

(*itimes2* U:integer V:integer):integer

(*times2* U:number V:number):number

Type: eval, spread, Base

Returns the product of U and V.

(*quotient* U:number V:number):number

(*iquotient* U:integer V:integer):integer

Type: eval, spread, Base

The quotient of U divided by V is returned. If U and V are integers the result will be an integer and (*remainder U V*) will be the corresponding remainder. An error occurs if division by zero is attempted.

(*remainder* U:number V:number):number

(*iremainder* U:integer V:integer):integer

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.

(*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.

(*random*):integer

Type: Base

Returns a random integer in the range 0 to 2**24-1. The pseudo-random sequence used is initialised to a number based on the

time of day LISP was run. The seed can be fixed by an option at load time: “-opt Rnnn” sets the initial seed to nnn.

(sqrt U:number):number

(isqrt U:integer):integer

Type: eval, spread, Base

Returns the square root of U. If U is an integer that is a perfect square the result will be an integer, otherwise floating point.

(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.

5.15.1 Modulo Arithmetic Functions

The functions supplied for modulo arithmetic are described below. A naming convention exists whereby functions which begin with a c perform arithmetic mod p in the range $[0, p-1]$; those functions which begin with an m perform arithmetic mod p yielding results in the range $[-p/2, p/2]$. The integer p is usually but not always a prime number. It is set by `setmod`.

(setmod U:integer):integer

Type: eval, spread, Base

(setmod p) sets the modulus for the `cplus` and `mplus` families of modulus arithmetic functions. (setmod 0) returns the current modulus without resetting it.

(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 $[0, p-1]$. See also `mdifference`.

(mdifference U:n-mod-p V:n-mod-p):n-mod-p

Type: eval, spread, Base

Returns $U - v \bmod p$ in the range $[-p/2, p/2]$. See also `cdifference` etc.

(cminus U:n-mod-p):n-mod-p

Type: eval, spread, Base

Returns $-U \bmod p$.

(mminus U:n-mod-p):n-mod-p

Type: eval, spread, Base

Returns $-U \bmod p$.

(cmod U:integer):n-mod-p

(mmod U:integer):n-mod-p Type: eval, spread, Base

These reduce the integer $U \bmod p$ in the range $[0, p-1]$ or $[-p/2, p/2]$ respectively.

(cplus U:n-mod-p V:n-mod-p):n-mod-p

(mplus U:n-mod-p V:n-mod-p):n-mod-p

Type: eval, spread, Base

Return $U + V \bmod p$.

(cquotient U:n-mod-p V:n-mod-p):n-mod-p

(mquotient U:n-mod-p V:n-mod-p):n-mod-p

Type: eval, spread, Base

Return the quotient of U and $V \bmod p$.

(crecip U:n-mod-p):n-mod-p

(mrecip U:n-mod-p):n-mod-p

Type: eval, spread, Base

Return the reciprocal of $U \bmod p$.

(ctimes U:n-mod-p V:n-mod-p):n-mod-p

(mtimes U:n-mod-p V:n-mod-p):n-mod-p

Type: eval, spread, Base

Return $U * V \bmod p$.

5.15.2 Rational Arithmetic Operations

`(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.

`(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.

`(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 will be converted to rational number form before the division is attempted.

```
(setq pi (rational 22 7)) > (22/7)
(numq pi) > 22
(denq pi) > 7
(times 4 pi) > (88/7)
```

`(rationalp U:any):boolean`

Type: eval, spread, Base

Returns t if U is a rational number, and nil otherwise.

`(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 will be zero.

5.15.3 Trigonometric Calculating Functions

(arccos U:number):number

Type: eval, spread, Base

(arcsin U:number):number

(atan U:number):number

(cos U:number):number

(cot U:number):number

(sin U:number):number

(tan U:number):number

(exp U:number):number

Type: eval, spread, Base

exp calculates the exponential of the argument which must be numeric. The result is a floating point number

(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. (i.e. 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.

(log U:number):number

Type: eval, spread, Base

As exp except that the function is the natural logarithm.

(log10 U:number):number

Type: eval, spread, Base

As log except that the function is logarithm to the base 10.

6 Control Structures

Acorn Cambridge LISP supports a wide variety of control structures both for transfer of control at a local level and at a global level. The conditional form is the primitive local operator.

(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.

(select U:any [V:pair] W:any):any

Type: MACRO, core

V is an association list. U is compared for equality with the successive cars 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.

(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 initialised 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" i.e. the flow of execution is not affected by any transfers to labels by go.

(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.
- 2) As the consequent of a cond item of a cond appearing on the top level of a prog.
- 3) As the consequent of a cond item which appears as the consequent of a cond item to any level.
- 4) 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 2 and 3.
- 5) 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 2, 3 and 4.

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.

(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.

```
(de iterativeFactorial (n)
  (prog (x)
    (setq x 1)
    L
    [cond ((zerop n)(return x ))
          ( t (setq x (times x n))
              (setq n (sub1 n))
              (go L)
```

(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.

(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.

(prog1 [U:any]):any

Type: eval, spread, Base

Evaluates its arguments in order and returns the value of U.

(prog2 [U:any]):any

Type: eval, spread, Base

prog2 is like progn except that it may only be used to combine two expressions. It is provided for compatibility with other LISP systems.

The catch and throw functions in Acorn Cambridge LISP provide the ability to transfer control to an enclosing function other than by the normal process of function evaluation and return.

(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 within this a throw is evaluated with the tag TAG, then catch will exit with the value of the throw (or the value of FAIL if given).

(throw TAG:id VAL:any)

Type: MACRO, core

Used in conjunction 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`.

6.1 Common LISP Control Structures

The following functions are based on Common LISP functions. Refer to a Common LISP Manual for extra detail.

`(let ((v1 val1) (v2 val2...)) body1..bodyn)`
 Type: MACRO

Equivalent to lambda expressions binding variable1 to value1 and then evaluating the sequence body. `let` performs all the bindings in parallel. Thus

`(let ((x y) (y x)) ...)`

temporarily swaps the value of `x` with the value of `y`.

`let*`

Type: MACRO As `let` but performs bindings sequentially.

`(do ((var1 init1 inc1)(var2 init2 inc2)..)
 (exitcondition resultvalue)
 body1...)`

Type: MACRO

`do` binds `var2` to `init2` and obeys the body until the condition is true. The expression `(inc2..)` is used to establish new values for `var2`. `do` updates values in parallel.

`do*`

Type: MACRO

As `do` but updates variables sequentially.

`(loop body1 .. body n)`

Type: MACRO

The body iterates until a return is obeyed. Within a `loop`, the forms


```
(while condn val)
```

```
(until condn val)
```

may be used to provoke exits. These are not part of common LISP but are extensions based on Acornsoft LISP.

```
(if cond value)
```

```
(if cond then-value else-value)
```

```
(when cond body1 .. bodyn)
```

```
(unless cond body1 .. bodyn)
```

```
(dolist (var init-list result) body1 ...)
```

Type: MACRO

```
(dotimes (var count result) body1...)
```

Type: MACRO

Obey the body within the variable bound to 0,1,..[count - 1].

```
(dolist (x '(0 1 2) 'done) (print (times x x)))
```

```
(dotimes (x 3 'done) (print (times x x)))
```

have the same effect.

7 Loops

The map functions of LISP provide the capacity for iterative operations over data structures. The functions `mapc`, `mapcar` and `mapcan` apply a given function to successive cars of a given list, thereby processing each top level element of the list; the functions `map`, `maplist` and `mapcon` apply the given function to successive cdr's of the given list. The functions `maplist` and `mapcar` return copied lists of the results of these multiple applications; `mapcon` and `mapcan` use `replacd`'s to modify the list of accumulated results.

There are also a collection of similar functions which have arguments in MacLISP order and support mapping over multiple lists. These are:

```
cl:mapc, cl:mapcan, cl:mapcar, cl:mapcon, cl:mapl,
cl:maplist.
```

(map x:list FN:function):any
Type: eval, spread, core

Applies FN to successive cdr segments of X, i.e. X, (cdr X), (cddr X) ... The list X is returned.

(mapc X:list FN:function):any
Type: eval, spread, core

FN is applied to successive car segments of list X (i.e. (car X), (cadr X), (caddr X) ...). The list X 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.

```
(mapcar '(1 2 3) (lambda (x)(plus x x))
> (2 4 6)
```

`(mapcon X:list FN:function):any`

Type: eval, spread, core

A concatenated list of FN applied to successive cdr segments of X is returned. (i.e. 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.

```
(maplist '(a b c d e f) (lambda (x)(compress x))
> (abcdef bcdef cdef def ef f)
```

8 Logic Functions

The functions `and`, `or`, and `not` are commonly used to implement a control structure analogous to some `cond` form. For bit level comparisons see `logand` and `logor`

(`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.

(`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.

(`not` U:any):boolean

Type: `eval`, `spread`, `Base`

If U `nil`, return `t` else return `nil` (same as `null` function).

8.1 Bit-level Operations

Acorn Cambridge LISP supports bit level logical operations over integers. These functions are constrained to work on small integers in the range `[0,2**24]`. Versions of these functions with names prefixed by `i` may compile into in-line code, but do not check their arguments as thoroughly as the more general versions.

(`logand` [U:logical]):logical

(`i` `logand` [U:logical]):logical

Type: `noeval`, `nospread`, `Base`

The result returned is the logical AND of all the arguments. `Ilogand` accepts only quantities with 24-bit values as arguments.

(logand2 U:logical V:logical):logical

(i logand2 U:logical V:logical):logical

Type: eval, spread, Base

logand2 behaves like logand except that it expects exactly two arguments. Compiled references to logand get converted into sequences of calls to logand2.

(logor [U:logical]):logical

(i logor [U:logical]):logical

Type: noeval, nospread, Base

logor forms the logical (i.e. bitwise) OR of a sequence of 24 bit values, and is otherwise similar to logand.

(logor2 U:logical V:logical):logical

(i logor2 U:logical V:logical):logical

Type: eval, spread, Base

Like logor, but expecting exactly two arguments. See logand2.

(logp U:number):boolean

Type: eval, spread, Base

Returns t if U is an integer in the range 0 to 2**24-1, i.e. if the binary representation of U is at most 24 bits long and so U can be used directly in logand, logor etc.

(logxor [U:logical]):logical

(i 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

(i logxor2 U:logical V:logical):logical

Type: eval, spread, Base

Like logxor but expecting exactly two arguments.

(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`.

(ileftshift U:integer V:integer):integer

Type: eval, spread, Base

Similar to `leftshift` but will not accept a negative second argument. `irightshift` is provided for right shifts. 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`.

9 I/O and File Handling

9.1 Files

At any one time, Acorn Cambridge LISP has one file selected for input and one for output. Reading and printing use these two files. Functions are provided to open and to select new streams.

(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. After calls to pds input and pds output, 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 if the file cannot be opened.

input,output

flag values, used as the second argument to open.

(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. The functions rds and wrs provide a temporary diversion for input and output streams.

(**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.

(**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.

A default directory for connection is specified by pdsinput and pdsoutput

(**pdsinput** FILE:any):nil

Type: eval, spread, Base

Causes calls to open, rds and close to refer to members of the directory with name FILE. This returns nil.

(**pdsoutput** FILE:any):nil

Type: eval, spread, Base

As pdsinput but for open, wrs and close.

9.2 Printing

(**prettyprint** U:any):any

Type: eval, spread, LISPPRI

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 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 `princ` but ensures that printing starts at the beginning of a line.

(`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` will fail if given cyclic structures, and there is no guarantee that the output it produces will be acceptable to the `read` function - in particular gensym `s` and binary code print legibly, but not in a way where they can be re-input.

```
(print '!*comp) prints !*comp
```

(`printc U:any`):any

Type: eval, spread, Base

As for `print` but with no escape characters.

```
(printc '!*comp) prints *comp
```

(`printcm U:any N:integer`):any

Type: eval, spread, Base

As for `printc` but leaving a left margin of size `non` line overflow.

`(printl U:any):any`

Type: eval, spread, LISPPRI

Prints circular lists without looping for ever. Reference points are labelled 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.

`(superprintm U:any N:integer)`

Type: eval, spread, LISPPRI

Same as `superprint` but leaves a left margin of width `N`.

`(superprint U:any):any`

Type: eval, spread, LISPPRI

Prints `U` in an indented format (if it will 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 (e.g. `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, LISPPRI

Same as `superprint` but leaves a left margin of width `N`, and terminates with a number of newlines.

`(tyo U:integer)` Type: eval, spread, Base

Prints the character with internal code `U`.

`(tyo 65)` prints an A

`(terpri):nil`

Type: Base

The current print line is terminated (i.e. a new line is started).

`(eject):nil`

Type: Base

Causes a skip to the top of the next output page if the destination supports carriage controls.

(line length 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 are not permitted.

(lposn):integer

Type: Base

lposn always returns zero. It is intended to record the line on the page.

(out radix RADIX:integer):nil

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.

(posn):integer

Type: Base

Returns the number of characters in the output buffer (ie. position in output line). When the buffer is empty, 0 is returned.

(ttab U:integer):nil

Type: eval, spread, Base

Enough spaces are printed to move the next character position in the line to U.

(xtab U:integer):nil

Type:eval, spread, Base

Prints U spaces on the current line.

9.3 The Programmable Reader

(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. i.e. a subsequent call to tyi will return this character.

(tyiq):integer

Type: Base

As tyi but does not echo what it reads.

(read):any

Type: LIsprad

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: LIsprad

Reads one symbol and sets the variable *token-type* to one of number, symbol or break-character.

(read-tokenq):atom

Type: LIsprad

As read-token but does not cause echo.

(readch):id

Type: LIsprad

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: LIsread

As readch but does not cause echo.

(readq):any

Type: LIsread

As read but does not cause echo.

(ascii CODE:integer):character

Type: eval, spread, Base

Returns the character corresponding to the given internal code e.g. (ascii 48) returns the character !0.

(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 by character. It translates from internal code to ASCII, and allows synonyms.

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 will stop on end-of-file, but will print a warning message to this effect.

9.4 Syntax

The LISP system read functions are driven by the syntactic properties of the characters encountered. The syntax properties are defined by a read-syntax table. A copy of the existing one can be made by the function (copy-syntax-table <table>) and the free variable read-syntax-table rebound.

(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. The value returned is that of U.

escape	enables 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!
lowercase	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.
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).

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

For example, to rename the characters * and - from the set of break-characters, evaluate (setsyntax "--*" 'break!-character)

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.

(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.

(liter U:any):boolean

Type: eval, spread, Base

Returns t if U is a character of the alphabet, nil otherwise.

(internalcode U:id):integer

Type: eval, spread, Base

If U is a single character id its ascii code is returned. See ascii.

9.4.1 Character level syntax

% Comments are by default introduced 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).

blank The atom blank has an initial value that is either the character blank or space. To test if 'ch' is a space, you can either type:

```
(eq ch blank)
or
(eq ch (quote !)).
```

See entry under ! for further explanation of the above.

- dollar The initial value of dollar is the character '\$'.
- 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 will override the global value.
- lpar The initial value of lpar is the atom '{', a left parenthesis.
- period The initial value of the atom period is the atom '.'.
- rpar The initial value of the atom rpar is the atom '}', a right parentheses.
- tab Variable having the value of the character tab.
- eqsign The initial value of eqsign is =.

9.5 Interacting with the LISP supervisor

(prarg U:mode):mode

Type: eval, spread, core

mode is one of nil, t, pretty or expanded. After (prarg nil), the LISP supervisor does not echo what it reads and interprets. (prarg t) causes the supervisor to print arguments; (prarg 'pretty) causes it to superprint them; and (prarg 'expanded) causes it to superprint them after expansion of readmacros. See prval and prmsg. The initial state is no echo. The function returns the previous value.

(prmsg U:boolean):boolean

Type: eval, spread, core

If U nil, then the printing of messages by the LISP supervisor is inhibited. The function returns the previous value.

(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) or (prval nil), which are the default, use print to display the result of a computation; while after (prval 2) the function printc is used. If U is 3, or pretty, values are superprinted. The function returns the previous value.

9.6 Saving Work

(preserve U:function)

Type: eval, spread, Base

a LISP core-image can be created by executing `preserve`. 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. Store 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 `LSPROOT`.

(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 omitted (the normal use), then the terminal is used.

(setreturncode U:integer):integer

Type: eval, spread, Base

Sets the eventual OS return code. `setreturncode nil` reads the return code currently set up.

(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. There is no exit.

(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.

10 Evaluating Functions

(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. eval is an inverse of quote, hence (eval 'a) is equivalent to requesting the value of a.

(evalis U any):any

Type: eval, spread, Base

evalis returns a list of the evaluation of each element of U.

(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 (i.e. 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.

```
(progn
  (setq x car)
  (setq y'(( a . b)))
  (apply x y)
> a
```

(mkquote U:any):any-list

This is a function of one argument and its definition is equivalent to (lambda (X) (list 'quote X)).

(quote U:any):any

Type: noeval, nospread, Base

Returns U unevaluated.

`(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`. This function is like `quote` (indeed, the interpreter does not distinguish between them), but if `(function (lambda ...))` occurs in code that is being compiled, the `lambda` expression will be compiled, and indeed may be expanded into in-line code, rather than into a separate `subr`.

`(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.

10.1 Declarations and Binding

`(fluid IDLIST:id-list):nil`

Type: `eval`, `spread`, `core`

The `ids` in `IDLIST` are declared as `fluid` type variables. `Ids` not previously declared are initialised 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.

`(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 initialised 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, otherwise nil is returned.

(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.

(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 initialised 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”. The VARS introduced by prog may have to be declared fluid if it is required to reference a binding made in this prog within a function called from within the prog

10.2 Function Definition

(def 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.

```

(de structEqual ( x y)
  (cond ((eq x y) t)
        ((atom x) nil)
        ((structEqual (car x)(car y))
         (structEqual (cdr x)(cdr y))

```

(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

Like deflist, but tries to compile the properties before placing them on the property list. No error occurs if a property is not compilable.

(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.

(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.

For example, if a macro to enable writing (push xL) with the effect of (setq l (cons xL)) is required:


```
(dm push(LL)
  (list 'car (list 'setq (caddr LL)
    (list 'cons (cadr LL)(caddr LL))
```

(expand L:list FN:function):list
 Type: eval, spread, lispcomp

FN is a defined function of two arguments to be used in the expansion of a macro. expand returns a list in the form:

```
(fnL[0] (fnL[1] ... (fnL[n-1] L[n]) ... )
```

where “n” is the number of elements in L, L[i] is the ith element of L.

(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 will be 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 will accept an indefinite number of arguments. If given a non-functional or malformed argument, some atomic value is returned.

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 will receive its arguments unevaluated. Except for suppressing argument evaluation, lambdaq behaves exactly like lambda. The lambdaq facility in Cambridge LISP takes the place of fexpr/fsubr activity in some other systems.

macro

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

(constantp U:any):boolean

Type: eval, spread, Base

Returns t if U is a constant (a number, string, function-pointer, or vector).

(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.

(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.

10.3 Compiler Functions

(codep U:any):boolean

Type: eval, spread, Base

Returns t if U is a pointer to compiled code.

(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, profile.

Example:

```
(compile '(FUN1 FUN2 FUN3))
```

(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.

`(module U:id):nil`

Type: eval, spread, Base

This directs the compiler to put the code that it generates into a module for later use. The code will be written directly to the `DUMP` directory if present, otherwise it will be written to the `IMAGE` directory, as well as being kept in store in case it is needed in a bootstrapping process. The module selected by `module` will replace any previous one with the same name. Empty modules are eventually purged from the system. The name `nil` and any starting with the characters 'LSP' should be avoided. `(endmodule)` or another call to `module` terminates a module.

`(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 'LSPCOMP)` purges the relevant functions. Note that after `excise` has been called, further reference to the excised module will result in it being reloaded.

`(endmodule)`

See `module`.

11 Editor entry Points

(`edit FN:id`):any

Type: `noeval`, `spread`, `lispedit`

`edit` enters the structure editor to edit the function defined with the name `ID`. For more details of the editor see chapter 2. 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 chapter 2 and `edit`.

(`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 chapter 2 and `edit`.

12 Error Control

(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 will cause a complete backtrace of all the functions being executed. `errorset` is a variant of `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` for further details. 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.

```
(progn
  (setq inputVariable (errorset (read) 0))
  (cond ((numberp inputVariable) (setq inputVariable "error"))
        (t inputVariable))
```

is a first attempt to cope with errors in input

`*emsg`

is set by errors to the error message produced, and so is useful after `errorset` catches the error.

(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.

(backgag U:integer):integer

Type: eval, spread, core

The amount of information printed after an error can be controlled 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 6:

0 means no notification of errors at all;

1 gives 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.

13 Debugging in LISP

13.1 Tracing Functions

`(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.

`(tracesetq U:id-list):id-list`

Type: eval, spread, Base

After a call to `tracesetq` in interpreted code, the use of `set` or `setq` to update the variables named in U leads to a message being printed.

Unlike `trace`, `tracesetq` has one list of traced variables, and to stop tracing a call `(tracesetq nil)` should be used.

`(untrace U:id-list)`

Type: eval, spread, Base

Undoes the effect of `trace` for the variables named in 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 before a problem occurs.

`(embed NAME:id NEWDEF:function)`

Type: eval, spread, core

`embed` is used to provide more detailed tracing than is available with `trace`, or to control the behaviour of system routines. 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))
  (changeX)
  (print (list "changed by change to:" GLOB))))
```

(unembed U:id)

Type: eval, spread, core

Undoes effect of embed.

13.2 Tracing Memory Use

(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.

(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 bcp1stack 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 will nullify the effect of this function. In general, users are recommended to use the backgag function which changes the backtrace characteristics at a higher, and hence more predictable level.

(count N:integer):integer

Type: eval, spread, Base

After (count n) has been obeyed, LISP will allow about n cons es to occur before causing an error:

‘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. count is intended for use with errorset and makes it possible to limit the amount of computation that a piece of code can perform. See speak.

(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.

13.3 Timing Functions

(tempus-fugit)

Type: Base

A call to tempus-fugit will result in LISP printing a line of text containing timing and store use figures.

```
(tempus!-fugit)
      prints
tempus-fugit after 3.21+41.11 secs - 40.9% store used
```

(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.

```
(timeofday)  
> "20:16:36"
```

(date):string

Type: eval, spread, Base

This returns a string giving the current date.

14 Miscellaneous Functions

(mapstore U:boolean)

Type: eval, spread, Base

mapstore prints details of all compiled functions present in the LISP heap. If any of these were compiled with the statistics option (see profile), the counts will be displayed and reset to zero. (mapstore t) prints a similar map of the core LISP system for use by system programmers.

(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.

gcdaemon

Type: User provided function

If a function gcdaemon is defined by the user, it will get called at the end of each garbage collection if it is compiled and not traced. gcdaemon will be handed one argument, which will be the serial number of the garbage collection just completed. The gcdaemon facility will be inhibited within execution of itself, so garbage collections triggered by work done within gcdaemon will not normally lead to re-entry to the function. Erroneous exits from the gcdaemon function will lead to the function's definition being removed, and so after the deliberate creation of such an error state gcdaemon should be redefined.

(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.

(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.

14.1 Graphics Functions

Acorn Cambridge LISP provides a set of graphics routines. In order to use them, a graphics screen mode must be selected by calling `(mode n)` with $n = 0, 1$, or 2 . Mode 0 provides $640 * 256$ resolution in monochrome, mode 1 is $320 * 256$ with 4 colours, and mode 2 is $160 * 256$ with 16 colours. The co-ordinate system for the screen can be reset by `(scale h)`, but by default the screen has height 1024. After a call to `(scale h)`, the top of the screen has y co-ordinate h . The width of the screen is always $5/4$ times its height. `(cls)` clears the screen, and `(home)` moves a notional graphics cursor to the centre of the screen. Two styles of graphics are supported:

- (1) simple cartesian graphics
`(moveto x y)`, `(drawto x y)`
- (2) turtle graphics
`(turn n)`, `(turnto n)`
`(move l)`, `(draw l)`

If a closed convex figure is drawn on the screen between calls `(fill t)` and `(fill nil)` the area will be filled in. `(circle r)` and `(circleat x r y)` draw circles, and with `(fill t)` set, they draw filled in circles.

The effect of `ink` depends upon the screen mode; small arguments will lead to solid colours, and larger values will give a variety of shaded effects.

<code>(circle r)</code>	draw circle at current position
<code>(circleat x y r)</code>	draw circle at position x, y
<code>(fill <flag>)</code>	set/clear area - fill mode
<code>(ink n)</code>	establish colour
<code>(mode n)</code>	set screen mode
<code>(move l)</code>	Turtle graphics
<code>(draw l)</code>	Turtle graphics
<code>(drawto x y)</code>	Cartesian graphics
<code>(moveto x y)</code>	Cartesian graphics
<code>(cls)</code>	clear screen

(home)	go to mid-screen
(scale h)	set logical screen height
(paper n)	establish colour
(turnto n)	turns through a specified angle:
(turnto 0)	points to 12 o' clock, and positive angles turn clockwise.

Appendix A 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 nor 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/expr
- 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
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	Function type assumptions wrong in compiled code
181	Bad use of car, cdr or 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
- 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
- 243 File already open as a partitioned dataset
- 244 Attempt to open file for both reading and writing
- 245 Attempt to read and write same pds member
- 246 Lost file in wrs
- 247 Lost pds on wrs
- 248 Bad argument for wrs
- 249 Lost file in rds
- 250 Lost pds in rds
- 251 Bad argument for rds
- 252 Fast loading failed-system may be in inconsistent state
- 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

Appendix B Bibliography

A good introductory text for LISP programming:

LISP, AC Norman and G Cattell, Acornsoft Ltd 1983.

The following texts explore many of the areas where LISP is found to be the most convenient programming tool.

Artificial Intelligence Programming, E Charniak C Riesbeck
D McDermott, Lawrence Erlbaum Associates 1980.

LISP, P Winston and B K Horn, 2nd edition Addison-Wesley 1984.

LISP 1.5 users manual
or the
Stanford LISP reference manual.

This publication describes a version of LISP which is fairly close to
32000 Cambridge Lisp:

ACM SIGPLAN Notices Vol. 14 No. 10 Oct 1979

Index

A

Acornsoft LISP 8

B

Bakroot 21
Bound variable list 11
Buffer 25

C

Case 8, 9, 32
Clearbuff 25
Common 19
Compilation 18
Customisation 4

D

Diagnostics 23

E

Editor 11
Eval 18
Excise 6, 19
Explode 26

F

Fedit 11
Fexpr 11
Fluid 19
Freestore 6
Function
 definition 4

G

Garbage collection 6

H

Hash tables 46

I

Implied progn 18
Initialsupervisor 21

Interpreter 18

L

Lambda 11, 18
Lambdaq 11, 18
Lexprs 10
List 11
 object 7
Load-on-call 21
Loading 21
Lspcomp 18, 19
Lsproot 20

M

Macros 10

N

Name lists 46
Numob 25

O

Object list 7, 49
Oblist 8
Options 5

P

Pack 25
Panos 3
Plus 11
Preserve 20
Prin 26
Progn 18

Q

Quote 18

R

Rdf 10
Read 25
Readch 25

S

Sedit *11*
Setsyntax *9*
Special *19*
Stack *6*
Store *6*
Stream *23*
Supervisor *21*

U

Unfluid *19*

V

Value cell *17*

Acorn Computers Limited
Scientific Division
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
Telephone 0223 245200