
CAMBRIDGE LISP

reference manual

ARM Evaluation System

Acorn OEM Products



Cambridge LISP

Part No 0448,011
Issue No 1.0
15 August 1986

© Copyright Acorn Computers Limited 1986

Neither the whole nor any part of the information contained in, or the product described in, this manual may be adapted or reproduced in any material form except with the prior written permission of the copyright holder. The only exceptions are as provided for by the Copyright (photocopying) Act, or for the purpose of review, or in order for the software herein to be entered into a computer for the sole use of the owner of this book.

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

- The manual is provided on an 'as is' basis except for warranties described in the software licence agreement if provided.
- The software and this manual are protected by Trade secret and Copyright laws.

The product described in this manual is subject to continuous developments and improvements. All particulars of the product and its use (including the information in this manual) are given by Acorn Computers in good faith.

There are no warranties implied or expressed including but not limited to implied warranties or merchantability or fitness for purpose and all such warranties are expressly and specifically disclaimed.

In case of difficulty please contact your supplier. Every step is taken to ensure that the quality of software and documentation is as high as possible. However, it should be noted that software cannot be written to be completely free of errors. To help Acorn rectify future versions, suspected deficiencies in software and documentation, unless notified otherwise, should be notified in writing to the following address:

Customer Services Department,
Acorn Computers Limited,
645 Newmarket Road,
Cambridge
CB5 8PD

All maintenance and service on the product must be carried out by Acorn Computers. Acorn Computers can accept no liability whatsoever for any loss, indirect or consequential damages, even if Acorn has been advised of the possibility of such damage or even if 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.

Econet® and The Tube® are registered trademarks of Acorn Computers Limited.

ISBN 1 85250 006

Published by:

Acorn Computers Limited, Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN, UK

Contents

Part I

1. Introduction	2
1.1 Installing LISP	3
1.2 Running LISP	3
1.3 Start-up options	4
1.4 Use of space	5
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 Special characters	9
2.2 Case	9
2.3 Defining functions	10
2.4 Macros	10
2.5 Error recovery	11
3. The LISP editor	12
3.1 Editor commands	13
3.2 Elementary moving	14
3.2.1 Find and move	15
3.2.2 Structure modification	15
3.2.3 Reformatting the screen	16
3.2.4 Eval loop and leaving the editor	16
3.3 Entering an s-expression to the editor	16
3.3.1 Hash Variables	17
3.3.2 Miscellaneous features	17
3.3.3 Limitations	17
4. Implementation features	18
4.1 Preserve	21
4.2 Load-on-call facility (FASL)	22
5. Input and output	24
5.1 I/O Routines	24
5.1.1 Open/Close	24
5.1.2 wrs/rds	24
5.1.3 Printing	24
5.2 Reading	26

Part II

6. Functions and variables	29
6.1 Argument types	30
6.2 Characters	32
6.3 Specialised variables	33
6.4 Atoms and values	34
6.5 Dotted pair functions	37
6.6 Tagged cons cells	40
6.7 List processing functions	42
6.8 List equality and searching	45
6.9 Pointer replacement functions	49
6.10 List manipulating functions	50
6.11 Creation of symbols	53
6.12 Flags and property lists	54
6.13 Function definitions as values	56
6.14 Vector operations	57
6.15 The AVL module	59
6.16 Arithmetic functions	60
6.17 Basic arithmetic operations	64
6.17.1 Modulo arithmetic functions	67
6.17.2 Rational arithmetic operations	69
6.17.3 Trigonometric calculating functions	70
7. Control structures	71
7.1 Common LISP control structures	74
8. Loops	76
9. Logic functions	78
9.1 Bit-level operations	78
10. I/O and file handling	81
10.1 Files	81
10.2 Printing	83
10.3 The programmable reader	87
10.4 Syntax	89
10.4.1 Character level syntax	91
10.5 Interacting with the LISP supervisor	92
10.6 Saving work	93
11. Evaluating functions	94
11.1 Declarations and binding	96
11.2 Function definition	98
11.3 Compiler functions	102

12. Editor entry points	104
13. Error control	105
14. Debugging in LISP	107
14.1 Tracing functions	107
14.2 Tracing memory use	109
14.3 Timing functions	111
15. Miscellaneous functions	112
15.1 Graphics functions	114
16. Appendix A	116
16.1 Error Messages	116
17. Appendix B	121
17.1 Bibliography	121

Part I

1. Introduction

This is a guide to Acorn Cambridge LISP running under the Executive on Acorn ARM computers. This section describes the main features of the Acorn Cambridge 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 ARM implementation of 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.

1.1 Installing LISP

Acorn Cambridge LISP for the ARM is distributed on ADFS floppy discs.

1.2 Running LISP

To run LISP, type

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

at the ARM 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 '\$.Lib.Image' on the ADFS.

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 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.

-leave

Expects a number of bytes (in units of 1024) that LISP will leave for the ARM 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.

```
Lisp -image $.Lib.Image -identify
```

```
Acorn Cambridge Lisp entered in about 4030 Kbytes
```

```
Store image was made at 09:55:48 on 12-Mar-86
```

```
Lisp version - 1.05\1.05\1.05 12-Mar-86 image size - 105716 bytes
```

```
Started at 15:33:45 on 14 Jun 86 after 0.01+9.15 secs - 7.7% store used
```

```
(setq !*comp nil)
```

```
> nil
```

```
(def 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
```

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

1.6 Finding out what is available

There are three ways of finding out what is available in Cambridge LISP. First, 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 `oblist` 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 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, for example,

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

Enclosing a file name in double quotes will also cause the following name to be taken literally, for example,

```
(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.

2.2 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 FOO T)
```

this now equivalent to `(setq foo t)`

2.3 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 for example, `de` `df` may not prove compatible with LISP source text from all other LISP systems.

The normal way of defining functions will be to use `(de ...)` 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.

2.4 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 (that is, `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.

2.5 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>
```

3. The LISP editor

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, that is,

```
(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.

3.1 Editor commands

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 resembles:

```
h[ead] t[ail] u[n] b[egin] l[ook]-f[or] m[ark]-m[ove] z[oom]i[n]-z[oom]o[ut]
d[elete] r[eplace] s[plice] i[n]s[er]t [u]n[d]o 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) ) ) ) )
```

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.

3.2 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, 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.