QL LISP Development Kit

Contents

Preface

Metacomco's QL Lisp Development Kit is a powerful package incorporating a full screen editor and a Lisp interpreter. This book is intended to be a guide for users of the kit and does not aim to be fully comprehensive on all related aspects of the QL or Lisp programming. It assumes that the reader has knowledge of the QDOS operating system.

If further detailed information is required, a full specification of the Motorola 68008 microprocessor can be found in *MC68000 16/32 Bit Microprocessor Programmer's Reference Manual* (4th edition, ISBN 1-356-6795X) published by Prentice-Hall. An introduction to Lisp programming can be found in various introductory texts several of which are listed at the end of this book. Particularly relevant is *Lisp for the BBC Micro* by A.C. Norman and G.E. Cattell published by Acornsoft, as this QL Lisp Development Kit is compatible with the Lisp available on the BBC micro. Further information about QDOS can be found in *QL Advanced User Guide* by Adrian Dickens (ISBN 0-947929-00-2) published by Adder Publishing.

Chapter 2: Language guide

2.1 Introduction

There are many paradoxes about the language LISP. It is one of t oldest computer languages. However, it is also one of the most forwa looking. Powerful and flexible enough for the professional programme it is also an excellent choice for a beginner.

Over the years LISP has developed a number of dialects. Howeve standardisation has not been too great a problem because of the relati ease in adapting and redefining code from one dialect to another. Th particular implementation contains only a subset of all possible LIS functions. It is not bound to these though, as functions can be added (removed!) by the user.

QL LISP has had the benefit of being a late-comer: the language h been refined and developed from the early Lisps to work efficiently on micro. QL LISP is also compatible with the Lisp used on the BB microcomputer, although it has been further extended to utilise many the QL features. When memory expansion for the QL is general available it will be possible to mount mainframe-proportio implementations of LISP on it (and versions of these larger scale LISI are already running on 68000s).

The *QL LISP Development Kit* is not meant to be a primer fo LISP. It is assumed that the reader has at least a limited knowledge the language. Good introductory books are: *LISP on the BB Microcomputer* by Norman and Cattell (Acornsoft, second editio 1984); *A Beginner's Guide to Lisp* by Tony Hasemer (Addiso Wesley, 1984); and *Lisp for Micros* by Steve Oakey (Newnes, 1984 (See also the bibliography at the end of this manual).

Definition of conventions and terms

Conventions:

In order to distinguish the names of LISP functions used in this manual from the rest of the text they have been written in bold. However, function names found in examples appear exactly as they are used and in the same typeface as the rest of the example. Certain standard abbreviations have been used in defining syntax:

arg	argument.
fn	function.
n	number.
var	variable.
< >	used to surround text describing the type of thing to be inserted rather than an actual name. i.e. (print <function name>) where <function name> should be replaced by the actual name of the function to be printed.
[ ]	used to surround anything that is optional.

Terms:

alist	an association list or list with each member being a dotted pair. i.e. ((a . b) (c . d)...).
arguments	things a function has to work with. (See 3.2 for a full list of argument types).
atom	the smallest data object that can be manipulated. It can be a character, a number, or an identifier.

bound variable	an atom that [illegible] to a function argument list
circular list	a list that contains a pointer back to itself.
dotted pair	The fundamental non-atomic data object in LISP. A dotted pair has two components, called car and cdr. The dotted pair whose car is 'a' and whose cdr is 'b' is written (a . b).
filenames	a name the format of which is system dependent
function	a named procedure which may be defined and called. It takes arguments as input and gives a value as output
garbage collection	returning old used cells to free storage
identifier	a named atom, which can have a value and various properties.
integer	fixed point number.
lambda	marker atom used in anonymous functions. lambda identifies a piece of LISP structure as representing a function.
list	groups of atoms surrounded by matching brackets. Lists can also be grouped together to form further lists. Thus (a), (a b c) and ((a) (a b c)) are all lists. Lists are, in fact, dotted pairs (a) being short-hand for (a . nil), and (a b c) for (a . (b . (c . nil)))

number	any type of number.
quotes (')	the quote symbol ' before an s-expression prevents its evaluation. This is equivalent to quote. A q at the end of certain (but not all) functions implies a quote symbol e.g. setq for set'.
s-expression	atoms and lists collectively form s-expressions (or symbolic expressions).
unbound variable	or free variable, is an identifier that is used in a function, but does not appear in the function's argument list.

2.2 Running QL LISP

Loading LISP

LISP is invoked using EXEC or EXEC_W as follows

```
EXEC_W mdv1_lisp
```

The difference between invoking a program with EXEC or EXEC_W is as follows. Using EXEC_W means that LISP is loaded and SuperBasic waits until the session is complete. Anything typed while LISP is running is directed to LISP. When LISP stops, keyboard input is directed at SuperBasic once more.

Using EXEC is slightly more complicated but is more flexible. In this case LISP is loaded into memory and is started, but SuperBasic carries on running. Anything typed at the keyboard is directed to SuperBasic unless the current window is changed. This is performed by typing CTRL-C, which switches to another window. A subsequent CTRL-C switches back to SuperBasic. When LISP is terminated a CTRL-C will be needed to switch back to SuperBasic once more.

Once the program is loaded it will ask if you wish to alter the window. The default window is normally the same as the window used in the initialisation although this may be altered if required. (See Appendix A for details of how to do this). The question will appear as:

```
Alter window (Y/N)?
```

If you type N or just press ENTER then the default window is used. If you type Y then you are given a chance to alter the window. (Note that if LISP was invoked using EXEC rather than EXEC_W it will be necessary to press CTRL-C before the Y or N so that your keyboard input is sent to LISP rather than SuperBasic). The current window is displayed on the screen and the cursor keys can be used to move the window around. The combination ALT and the cursor keys will alter the size of the window although there is a minimum size which may be used. Within this constraint you can specify a window anywhere on the screen, so that you can edit a file and do something else such as run a SuperBasic

program concurrently. When you are satisfied with the position of the window press ENTER.

Example session

This is an example of a trivial session. It shows the simplest use of numbers, characters, variables and functions, as well as some basic errors. Lastly it shows how to get out of LISP.

```
QLLISP: heap size 54296 bytes

Evaluate: 56
Value: 56

Evaluate: oranges

*** Error 999: unbound variable oranges
End of backtrace

Evaluate: (setq fruit '(oranges lemons apples))
Value: (oranges lemons apples)

Evaluate: fruit
Value: (oranges lemons apples)

Evaluate: (car fruit)
Value: oranges

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

Evaluate: (plus 1 3)
Value: 4

Evaluate: (setq a 15)
Value: 15

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



```
Evaluate: (plus a fruit)

*** Error 21: Numeric argument required (oranges
lemons apples)
In PLUS
End of Backtrace

Evaluate: (setq flavours '(orange apple
                           vanilla E123))
Value: (orange apple vanilla E123)

Evaluate: (flavours)
*** Error 5: Undefined function orange
End of Backtrace

Evaluate: flavours
Value: (orange apple vanilla E123)

Evaluate: (stop)
```

After the atom 56 is entered, the corresponding value is returned:

```
Value: 56
```

The second example returns an error because the input 'oranges' is treated as an unset variable. The error then gives rise to a code number, message and backtrace. Note that this did not happen with the previous numerical example.

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

The next example demonstrates a very simple evaluation of the arithmetic function plus on the atoms 1 and 3. The following input shows how to set a variable with a numerical value and the next shows how this value can be used in a subsequent call. The next input shows how the mixing of symbolic and numerical expressions is nonsensical when using an arithmetic function.

```
(read)
```

reads a list or s-expression from the keyboard.

readline on its own reads characters from the keyboard. If a file-handle is specified, however, the contents of the corresponding file is read. All the characters from the current position in the specified file up to the end of the next line are then assembled into a single identifer which is returned as the value of readline. ordinal or explode may then be used to extract the characters from this long identifier. getchar, although normally used to read from the keyboard, can be given a file-handle as an argument, in which case it reads one character from the specified file.

The function rdf reads and executes the LISP code from a file. The format for invoking rdf is:

```
(rdf <filename>)
```

e.g. `(rdf 'mdv1_demos)`

If (stop) is obeyed from within the file control is passed back to rdf.

Apart from using the read-file rdf, streams can also be selected by the read-stream rds and write-stream wrs. However, before another stream can be selected using these commands, it has first to be opened using the function open:

```
(open <filename> t)
```

the numeric filehandle returned by open can then be used.

To select a stream to be written to:

```
(wrs <filehandle>)
```

wrs selects the file for future output to be written to, and returns a file-handle for the file that was selected when wrs was called.

To select a stream to be read from:


```
(rds <filehandle>)
```

rds selects the file for all future input, and returns the file handle for the file that was selected when rds was called. Note that linewidth is reset whenever a new output stream is selected. (The default line width is 80 for non-console files).

rds quits back to the normal state on end of file. If rds is used to change stream selection a stop or end-of file will cause LISP to quit back to SuperBasic. rds and wrs are used inside programs to read in data.

There are further commands which write to a file: write, writef, writec and writec0. These all take a file handle as one of their arguments. They then direct output to the file specified by that handle. In other ways write and writef0 correspond to print and prin. While writec and writec0 are the same as write and writef but work on single characters.

A file specified by a file-handle is opened [illegible] closes a file specified by a file-handle, writing out any buffers associated with it. Whenever input or output to a particular file is completed,

```
(close <filehandle>)
```

should be given. It is good practice to keep [illegible] open at any one time. If for some reason [illegible] is returned.

2.3 QL LISP structure editor

The *LISP* structure editor is invoked by

```
(edit <name>)
```

The function edit does not evaluate its argument, so that the function name does not need to be quoted (i.e. use (edit sed) rather than (edit 'sed)). edit prettyprints the definition associated with the function and then uses set to replace that definition by whatever sed returns. edit could have been defined in LISP as:

```
(defun edit name
    (superprint (eval (car name)))
    (set (car name) (sed (eval (car name))))
    (terpri)
    (car name))
```

sed defines the commands to which the editor responds. These are single characters, obtained by the call to getchar. The most important are a, d and b. a and d cause sed to recurse, entering itself to edit respectively the car and cdr of its previous expression. b causes it to return, thus backing up towards the top of the expression. If an a or a d command would take sed off the end of a list it prints a star and ignores the command:

```
(defun sed (a (q))
  (loop
    (setq q (princ (getchar)))
    (until (eq q 'b) a)
    (setq a
      (cond
        ((eq q 'r) (terpri) (read))
        ((eq q cr) (superprint a) a)
        ((eq q 'c) (terpri) (cons (read) a))
        ((atom a) (princ '*) a)
        ((eq q 'd) (cons (car a) (sed (cdr a))))
        ((eq q 'a) (cons (car a)) (cdr a)))
        ((eq q 'x) (cdr a))
        (t (princ '?) a)))))
```


The r command allows you to replace the expression that you are currently considering. c and x insert and delete the second element, and prettyprints the current expression at the end of each line of edit requests. There are, of course, many additional commands that would be wanted in a structure editor - commands for searching and performing global exchanges, and for moving up and down the tree in larger steps than the commands provided here. This basic editor is built into LISP so as to achieve good performance. A version coded in LISP is included as one of the demonstration programs, and can easily be extended to provide whatever additional commands are required.

2.4 Turtle graphics

QL LISP provides a simple graphics package known as turtle graphics. The 'turtle' is invisible, but can be imagined to be a small triangle which can be made to turn and move about the two-dimensional screen drawing lines, points and circles. Any shapes thereby produced may be then left clear, or filled and coloured.

Initially the turtle is set at its 'home' position. At any time the command home will return the turtle to this position. The position is at the coordinates 500 500 with the point of the turtle turned to 0 degrees: this is roughly the middle of the screen and pointing directly upwards.

The command move will move the turtle forward in the direction it is pointing for a specified distance. The turtle is always considered to carry a pen. When the turtle moves it does so with its pen up and no pen 'trail' is left. The command draw is similar to move, except that in this case the pen is down and a trail is drawn from the initial position to the new current position. For example:

```
(move n)
(draw n)
```

where n is a number representing the distance the turtle is to move/draw a line.

move and draw do so in the direction the turtle is pointing already. To make the turtle turn, the following two commands are given: turn and turnto. turn changes the turtle's heading by the given angle, working in degrees. A positive angle causes it to turn right, a negative one gives a turn to the left. turnto sets the heading of the turtle in an absolute way.

```
(turn n)
(turnto n)
```

where n is the angle of turn in degrees.

Although an accurate position for the turtle can be described by the combination of the commands turn, turnto, move and draw, it is not always convenient. The corresponding commands moveto and drawto allow an absolute position to be described. moveto takes an x and

y-coordinate, it then moves the turtle from the current position to that position with the pen up. drawto draws a line from the current position to the new position described by its coordinates. For example:

```
(moveto x y)
(drawto x y)
```

where x and y are the coordinates of the new position.

For all these commands the screen is treated as having a height of 1000 units - this can be changed by the scale function.

The command point also takes coordinates. It then causes a spot to be drawn at the position described by those coordinates. For example:

```
(point x y)
```

The command circle causes a circle to be drawn about the current position for a given radius. circleat allows a circle to be drawn at any other specified position for a given radius. It is equivalent to (moveto x y) (circle r). For example:

```
(circle r)
(circleat x y r)
```

where r is the radius of the circle, and x and y are the coordinates of the position of the centre of the circle.

ink sets or resets the colour used. For example:

```
(ink n)
```

The argument n is a numeric code (0 to 7) corresponding to the required colour. These numbers give clear, solid colours. Numbers higher than 7 result in various stipples.

fill takes t or nil. t sets fill mode; nil clears it.

```
(fill t)
(fill nil)
```

To use fill, go (fill t), use some set of graphics functions to draw an enclosed figure, then go (fill nil). the figure drawn will be filled in with ink. For example:

```
(fill t) (circle 200) (fill nil)
```

Other graphics effects can be achieved using the functions window and screen as described in Chapter 3.

Chapter 3: Functions and Variables

3.1 Introduction

This chapter lists the functions and variables which are initially available in QL LISP. It should not be viewed as a full list of all possible functions and variables, rather it should be seen as a description of a core of basic definitions. The user may extend this list by using defun to define new functions, and setq to define new variables. Any function or variable may be added to, or removed from, this list. After a version of LISP has been saved, a full up to date listing of all functions and variables can be obtained by the function oblist.

Each function in this chapter is provided with a prototypical header line. Each formal parameter is given a name and suffixed with its allowed type. Lower case tokens are names of classes and upper case tokens are parameter names referred to in the definition. The type of the value returned by the function (if any) is suffixed to the parameter list. If it is not commonly used the parameter type may be a specific set enclosed in brackets (...). For example:

(apply FN:(id function) ARGS:any-list):any

where FN is name of the function being defined and ARGS is the list of arguments in a form ready to be bound to the formal parameters of FN.

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]):boolean

and is a function which accepts zero or more arguments which may be of any type.

Some indication of whether a name refers to a variable or to a function is given at the end of the header line. Variables are not named as such. Functions are indicated by one of the following words: fsubr,

Fsubr or Expr. These are function types. Subr means that a function is built-in and processes its arguments normally. Fsubr is also built-in, but has special argument processing. For example, it guarantees to process its arguments from left to right, or not to evaluate all its arguments. Expr is a function defined in LISP, not in machine code.

3.2 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 on.

ulist	A list with each member being a dotted pair. i.e ((a.b)(c.d) ...).
atom	Any type of number, character or id.
boolean	The set of global variables t and nil, or their values t and nil.
filename	A QDOS filename.
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 are treated specially as they do not need this full mechanism and so cannot be used where an id is specified.
number	Any number. On the QL numbers may be integers in the range -134217728 to 134217727.
byte,	A number in the range 0 to 255

3.3 Functions and variables

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.

(

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

!

! is the 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.

---

*, **, *** Varinbles

The three most recent results produced by LISP are saved in the variables *, ** and ***. Each time the user interacts with LISP these variables are updated. This makes it easy to re-use recently computed quantities, for instance

```
'(a b c)
(append * *)          => (a b c a b c)
(car **)              => a
```

The saved values may be discarded if LISP runs out of store. See +, ++, +++ and -.

-, +, ++, +++ Variables

Recently presented input expressions are saved in these variables. - is the form currently being evaluated, while +, ++ and +++ are previous ones. These values are most often useful as a reminder of just what was typed, but they can sometimes save the need for re-typing long expressions. Examples

```
(cdt '(a b c))                (meant to be cdr)
<error>
(subst 'cdr 'cdt +) => (cdr (quote (a b c)))
(eval *)            => (b c)
```

(add1 U:number):number Subr

Returns its numeric argument incremented by 1. Equivalent to, but faster than, a call to (plus U 1). See also sub1.

(and [U:any]):boolean Fsubr

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. Thus the value will be treated by LISP as true if and only if all its arguments are non-nil, and does not necessarily evaluate all its arguments. It goes through the list evaluating them one by one until:

i) the value of an argument is nil - the value returned by and is then nil.

ii) the end of the argument list is reached, in which case and returns the value of the last argument (which will, in this case, be non-nil).

For example,

```
(and (numberp n)
     (greaterp n 0)
     (lessp n 7))
```

yields t when the variable 'n' has as its value a number between 0 and 7. See also or, not, t and nil.

(appena U:list V:list):list Subr

If U and V are two lists, then (append U V) is the list obtained by putting all the elements of V after those of U. Thus (append '(p q) '(r s)) is the list (p q r s). append could have been defined in LISP as

```
(defun append (a b) (cond
      ((null a) b)
      (t (cons (car a) (append (cdr a) b)))))
```

but it is built into QL LISP in machine code.

(apply FN:{id function} ARGS:any-list):any Subr

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 (ie. 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.

(assoc U:any V:alist):{dotted-pair nil} Subr

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.

```
(defun assoc (a l) (cond
      ((null l) nil)
      ((equal a (caar l)) (car l))
      (t (assoc a (cdr l)))))
```


(atom U:any):boolean Subr

Returns t if U is an atom, i.e. an identifier, number or reference to machine code. If atom is true, then car or cdr would be illegal.

(band U:number [U:number]): Subr

band treats all its arguments as 28-bit binary quantities, and bitwise ands them. For example,

```
(band 5 9) = 1        (binary: 0101 & 1001 = 0001)
(band 31 -2) = 30
  (binary: 0000000000011111 & 1111111111111110
                          =   0000000000011110)
```

See also bor and bnot.

---

blank Variable

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

(bnot U:number):number Subr

bnot treats its numeric arguments as a 28-bit binary number and complements each bit. The resulting bit pattern is used as bnot's numeric result. The respresentation used by LISP for numbers means that for any number 'n', (bnot n) has the same value as (sub1 (minus n)), so (bnot 0) = -1, and (bnot -100) = 99. See also band and bor.

(bor U number [U:number]) number — Fsubr

bor is similar to band, except that it forms the bitwise inclusive or of all the numbers that are its arguments. So

```
(bor 12 6) = 14   (binary: 1100 | 0110 = 1110)
```

(car U:dotted-pair):any — Subr

car(cons a b) = = > a. The left part of U is returned. An error occurs if U is an atom.

(cdr U:dotted-pair):any — Subr

cdr(cons a b) = = > b. The right part of U is returned. An error occurs if U is an atom.

(character N:byte):id — Subr

The argument to character, N, should be an integer in the range 0 to 255. character treats N as the code for a character. It hands back an identifier that has this one character as its printname.

(charp U any) boolean — Subr

charp returns t if its argument is an identifier. Otherwise, it returns nil. Thus charp can be used to distinguish identifiers (sometimes known as character atoms) from other types of objects in LISP, i.e. numbers, code pointers and lists. For example:

```
(charp 'abracadabra)   = t
(charp 42)             = nil
(charp (cons a b))     = nil
```

(chars U:any):number — Subr

chars returns the number of characters that would be displayed if its argument was printed. So, for example,

```
(chars 'four)          = 4
(chars 'six)           = 3
```

(circle RADIUS:number): — Subr

circle draws a circle of radius RADIUS around the current position. See circleat.

(circleat X:number Y:number RADIUS:number): — Subr

circleat draws a circle around the position described by the coordinates X and Y with the radius RADIUS. It is equivalent to moveto followed by circle.

(clock).list Subr

This function returns a list of three numbers which represent the time, in hours, minutes and seconds, since the computer was last reset. See time, gctime and reset.

(close FILE:any):any Subr

close closes the file with file-handle FILE, writing out all buffers associated with it. An error occurs if the file cannot be closed.

(cls): Subr

cls clears the screen.

(concat NAME:id NAME:id):id Subr

This function creates an identifier the name of which is the concatenation of the two NAMEs given. It shows that identifiers can be used to support some sorts of string manipulation.

(cond [U:cond-form]):any Fsubr

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

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

Returns a dotted-pair which is not eq to anything preexisting and has U as its car part and V as its cdr part.

---

cr Variable

The value of cr is the identifier the name of which is a carriage return. Thus (princ cr) has the same effect as (print). cr = (character 10). See also blank.

---

cxxxr Subr

Any name of the form cxxxr, where the 'x's represent the characters 'a' or 'd', is treated as a combination of the basic functions car and cdr. Thus (caddr U) is equivalent to (car (cdr (cdr U))). The present implementation allows up to three letters between the c and the r, so caaar to cdddr are provided for.

(defun NAME:id PARAM:[id id-list] FN:any):id Fsubr

The function FN with the 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 then lost. defun is therefore a convenient way of defining functions. None of the arguments are evaluated. The use of defun is exactly equivalent to

```
(setq function-name
   '(lambda parameters body ... ))
```

The value returned by defun is the name of the function that has been defined. The second argument (PARAMeters) is a list of arguments and local variables that the function uses. Any number of actions can be given for the function to carry out.

```
(defun add2 (x) (plus x 2))
```

defines a function add2 by setting add2 to the value

```
(lambda (x) (plus x 2))
```

(See *Lisp for the BBC Micro* by Norman and Cattell, or the examples in Appendix B, for more sophisticated use.)

(delete U:any V:list):list — Subr

Returns V with the first top level occurrence of U removed from it.

(difference U:number V:number):number — Subr

Returns U - V.

(digit U:any):boolean — Subr

Returns t if U is a digit, otherwise nil. Note that a digit is a character and not a number. (ie. U = !2 returns t but U = 2 returns nil).

(draw U:number): — Subr

draw moves the graphics turtle forward U with pen down (i.e. it draws a line from the current position to U). See move.

(drawto X:number Y:number): — Subr

drawto takes the arguments X and Y which describe the coordinates of the absolute position for the turtle to move to with the pen down. See draw and moveto.

---

dollar — Variable

The initial value of dollar is the character $.

(edit FN:id):any — Fsubr

Details of edit together with notes on its use are given in chapter 2. The basic commands provided are:

A	move to car field
B	back up one level (ie inverse of A, D)
	To stop edit, repeat B to until edit is left.
C s	insert the expression s at front of current list by a cons
D	move to cdr field
R s	replace current expression with s
X	excise head of current list
<return>	prettyprint current expression

(eof FILE:any):boolean — Subr

Detects whether an end-of-file marker has been reached when reading. It returns t if so, and nil if not. The argument FILE is a file-handle obtained from open.

(eq U:any V:any):boolean Subr

Returns t if one of the following is true:

i) U and V are the same identifier.
ii) U and V are equal numbers.
iii) U and V are identical lists in LISP memory.

Otherwise, eq returns nil.

(equal U:any V:any):boolean Subr

Returns t if U and V are the same. Dotted pairs are compared recursively to the bottom levels of their trees. Function pointers must have eq values.

(error [MESSAGE:any]): Subr

error behaves like print in that it displays its argument, MESSAGE, on the screen. Having done that it generates error number 15 and the usual backtrace occurs. Here is an example of its use checking that 'w' is a list before attempting to find its cdr.

```
(cond
   ((atom w) (error (list w blank 'not 'list)))
   (t (cdr w)))
```

(errorset U:any FLAG:integer):any Fsubr

Normally when an error occurs in evaluating an expression, the backtrace works through all the function calls and halts the program. errorset is a means of preventing this and keeping control of the program. The argument to errorset is an expression to be evaluated and which might fail. If evaluation of this expression is successful, errorset acts just like list i.e. (errorset <expression>) is equivalent to (list <expression>). Note that in this case the value returned by errorset is never an atom. If evaluation of the protected expression fails, errorset returns as its value the number of the error that was detected. Thus the following loop will return an expression read from the keyboard but will trap the errors that could be provoked in read by misplaced brackets and dots:

```
(loop (setq x (errorset (read)))
      (until (listp x) (car x))
      (print '(try typing that again please)))
```

```
(errorset (car nil)) => 14
```

and an error message as controlled by messon/messoff

```
(errorset (cons 'a 'b)) => ((a . b))
```

See messon and messoff for control over the amount of diagnostic information printed when errors occur.

(eval U:any):any Subr

U is evaluated as a piece of LISP code with respect to the current collection of variable bindings. eval does almost all the work evaluating LISP expressions.

(explode U:any):id-list — Subr

Returned is a list of single-character identifiers representing the characters that print as the value of U. For example:

```
(explode 'myth)
```

returns the list

```
(m y t h)
```

See also implode.

---

f — Special identifier

The initial value of f is nil, and so f can be used as a name for 'false'. In this way it is similar to t. If f is used as a synonym for nil, it should be avoided as the name of an ordinary variable.

(fill U:boolean) — Subr

fill sets and unsets fill screen mode. t sets fill mode; nil clears it.

(flatten U:any-list):list — Subr

flatten takes a general list structure, and returns a single-level list of all the atoms found in it.

(fsubrp U:any):boolean — Subr

fsubrp tests whether its argument is an Fsubr atom. If so, it returns t; if not, it returns nil. Fsubr atoms represent entrypoints to those predefined LISP functions that process their arguments in special ways, and which are labelled as Fsubrs in this chapter. Thus

```
(fsubrp cond)   = t
(fsubrp cons)   = nil   (cons is defined as
                         a Subr, not a Fsubr)
(fsubrp 'cond)  = nil
```

where the last case gives a nil result because the argument handed to fsubrp is the identifier cond, which is not the same thing as the code-pointer defining the function associated with that identifier. See also subrp.

(gctime):integer — Subr

The value returned by gctime is the amount of time (in units of 1/100 second) spent in carrying out garbage collection. The recorded time is cleared to zero by reset. See also time.

(get U:any IND:any):any — Subr

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.

(getchar FIL:file).id Subr

getchar returns a single character identifier. This character is the next one read from the keyboard. getchar can be given a file handle (see open) as an argument, in which case it reads one character from the specified file. See also readline, read and ordinal.

(greaterp U:number V:number):boolean Subr

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

(home): Subr

home resets the graphics turtle to its original position and angle. It is equivalent to:

```
(moveto 500 500) (turnto 0)
```

(implode U:any):id-list Subr

The argument to implode must be a list of identifiers, where each item in this list is just a single character. implode returns the identifier whose name consists of these characters. The result of implode is an identifier even if all the characters in its argument are digits, and even if punctuation characters, brackets and blanks are present. See explode for the inverse operation. Examples:

```
(implode '(c a r))                = car
(implode (cdr (explode 'that)))   = hat
```

See also numob.

(ink U:byte):colour Subr

Sets screen colour

---

lambda Special identifier

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.

(last U:list):any Subr

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.

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

Returns t if U is strictly less than V, otherwise returns nil. Both arguments must be numeric.

---

linewidth Variable

linewidth is set by the system to reflect the width of the current window, and is reset when window is called.

(explode U:any):id-list — Subr

Returned is a list of single-character identifiers representing the characters that print as the value of U. For example:

```
(explode 'myth)
```

returns the list

```
(m y t h)
```

See also implode.

---

f — Special identifier

The initial value of f is nil, and so f can be used as a name for 'false'. In this way it is similar to t. If f is used as a synonym for nil, it should be avoided as the name of an ordinary variable.

(fill U:boolean) — Subr

fill sets and unsets fill screen mode. t sets fill mode; nil clears it.

(flatten U:any-list):list — Subr

flatten takes a general list structure, and returns a single-level list of all the atoms found in it.

(list [U:any]):list — Fsu

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

(listp U:any):boolean — Su

listp returns t if the argument is a list or dotted pair, and nil if th argument is an atom. It is the opposite of atom in that, for any x, (nu (atom x)) is equivalent to (listp x).

```
(listp (cons <anything> <anything>)) = t
(listp 3)                            = nil
```

(load U:filename) — Sub

The argument to load should be the name of a file created by save. loa reads the file into memory and in doing so restores all LISP's workspac to the state it was in when save was performed. This loses the values variables and the definitions of functions present before the load wa executed, replacing them with saved ones from the file.

(loop U:action [V:action]) — Fsub

This function is used in association with the functions until and while provides for the repetitive execution of a set of LISP commands. Fo example,

```
(loop
   (until (atom (errorset (print (eval (read)))))
          'done))
```

is a loop that obeys read, eval, print until an error is detected an trapped by errorset.

---

lpar

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

(map FN:function X:list):any

Applies FN to successive cdr segments of X. (ie. X, (cdr X), (cddr X), ...). It could have been defined as:

```
(defun map (fn l) (cond
        ((null l) nil)
        (t (cons (fn l)
                 (map fn (cdr l))))))
```

(mapc FN:function X:list):any

FN is applied to successive car segments of list X. (ie. (car X), (cadr X), (caddr X)...). It could have been defined as:

```
(defun mapc (fn l) (cond
        ((null l) nil)
        (t (cons (fn (car l))
                 (apc fn (cdr l))))))
```

(member A:any B:list):boolean

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.

(messoff U:byte) Subr

messon and messoff are used to control whether certain system messages are printed. messon will allow the message to be printed, and messoff will suppress it. Once the status of a message has been set this way it remains unchanged until a disastrous error occurs or another messon or messoff expression is evaluated. Each messages concerned corresponds to a single bit in the arguments to these functions, and is controlled using the following numbers:

Number	Message
1	Garbage collection bytes collected
2	Garbage collection number
4	Error number
8	Error top level arguments
16	Error backtrace
128	Read depth prompt

Thus (messoff 16) suppresses detailed error backtraces until further notice, (messoff 3) switches off all messages from the garbage collector while (messon 128) turns the '>' prompts back on (which indicate bracket nesting while reading). The control these functions give over error messages can be useful in association with errorset.

(messon U:byte) Subr

See messoff.

(minus U:number):number Subr

minus negates its argument, which must be a number. Note that subtraction is performed by the function difference.

(minusp U:number) boolean Subr

Returns t if U is a negative number, otherwise nil

(mode U:number): Subr

U should be 0 or 8; resets the screen mode to support 4 or 8 colours.

(move U:number): Subr

move moves the graphics turtle forward U with pen up (i.e. without drawing a line). See draw.

(moveto U:number V:number): Subr

moveto takes the arguments U and V which describe the coordinates of the absolute postion for the turtle to move to with the pen up (i.e. without drawing a line). See drawto and move.

---

nil Special identifier

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

(not U:any):boolean Subr

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

(null U:any):boolean Subr

Returns t if U is nil. It is sometimes used to test for an empty list. See also not.

(numberp U:any):boolean Subr

Returns t if U is a number. Otherwise the value is nil.

(numob U id-list):number Subr

numob is similar to implode in that it takes a list of characters as its argument. In the case of numob all the characters are expected to be digits, and the value returned is the number which has that sequence of digits as its decimal representation.

(oblist):id-list Subr

Returns a list of all the identifiers known to LISP except those having the value 'undefined' and or with empty property lists. These conditions eliminate those atoms which are being used as character strings rather than as atoms with interesting values. Inspection of the identifiers in oblist provides definitive information about what functions are available in any particular LISP image. oblist is short for object list.

(onep U:any):boolean Subr

Returns t if U is the number 1. There is no error if the item is not numeric. The effect is like (eq U 1). See also zerop.

(open FILE:any MODE:id):FILEHANDLE:integer Subr

open opens the named file, FILE, for input or output. If MODE is nil a new file will be created; otherwise, it will be expected to exist already. The value of open is a file-handle (itself a small integer) which will be used as an argument to such functions as readline, write and close.

(or [U any]) boolean Fsubr

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.

(ordinal U:id):ascii-code Subr

ordinal returns the numeric ASCII code for the first character in the printed form of the argument.

Examples:

```
(ordinal 'alphabet) = 97
(ordinal 'z)        = 90
(ordinal 33)        = 51
```

See also character.

(peek ADDRESS:number):byte Subr

peek returns a number representing the contents of the memory location of which the address is given in the argument. The address must be numeric.

peek should be used with care as variables may not be in the same positions in different systems. Programmers who rely on peek do so at their own risk and should not, in general, expect their programs to be appreciated.

period Variable

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

(plist U:id) plist Subr

The function plist called with one argument that is an identifier returns the property list of that atom. In the absence of a property list, it returns nil. Properties should normally be established and accessed using put and get, but plist can be useful when checking the behaviour of an errand program. Property lists in this LISP are made up of lists of dotted pairs having the format ((propertyname . value) (propertyname . value) ...). See also put, get.

(plus [U:number]):number Fsubr

Forms the sum of all its arguments.

(point X:number Y:number): Subr

point is given two numeric arguments, X and Y, which are the coordinates describing an absolute position on the screen. At this position, point draws a spot.

(poke ADDRESS:number V:byte): Subr

poke stores the single byte that is its second argument in the memory location specified by its first argument. See peek for the converse operation. poke can corrupt arbitrary locations in store and so destroy the integrity of LISP's datastructures. It should only be used with great care.

(prin [U:any]):any Fsubr

The value of U is printed with any special characters preceded by the escape character. The value of U is returned. prin can take any number of arguments.

(prine [U:any]):any Fsubr

The value of U is printed with no escape characters. The value of U is returned. prine can take any number of arguments.

(print [U:any]):any Fsubr

The value of U is printed, with escape characters, followed by a new line. print can take any number of arguments.

(printe [U:any]):any Fsubr

As for print but with no escape characters. printe can take any number of arguments.

(progn [U:any]):any Fsubr

U is a set of expressions which are executed sequentially. The value returned is the value of the last expression.

(put U:id IND:id PROP:any):any Subr

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. Values saved with put are normally retrieved by get. See also plist.

(quote U:any):any Fsubr

Returns U unevaluated. The shorthand 'x is expanded to (quote x) by the LISP read routines.

(quotient U:number V:number):number Subr

The quotient of U divided by V is returned. The result will be an integer and (remainder U V) will be the corresponding remainder. An error occurs if division by zero is attempted, or if U and V are not numbers.

(rdf [FILE:id]):nil Subr

Reads and executes the LISP code in the given file. If (stop) is obeyed from within the file then control is passed back to rdf.

(rds FILE:file):any Subr

rds selects the given file for future input, and returns the file-handle for file that was selected when rds was called. See wrs.

(read FILEHANDLE:any) any Subr

read reads one list or s-expression from the keyboard. It is a user-level entry into the code that LISP normally uses to read commands, and so all the conventions used there apply to expressions handled by read. If read is given an argument it should be a file-handle (see open), and then read takes its input from the indicated file. See also readline and getchar.

(readline FILE:any):id Subr

readline reads characters from the keyboard (or from a file specified by giving it a file-handle as an argument). All characters from the current position in the file up to the next carriage return are assembled into a single identifer, and this is returned as the value of readline. It may then be useful to use ordinal or explode to extract characters from this long identifier.

```
(cond
    ((eq (readline) 'yes) t))
```

tests if the next line typed by the user consists of exactly the characters 'y', 'e' and 's'. It is unlikely to be confused by people who type in brackets and punctuation marks instead of the expected word, whereas if read had been used this would have been a possibility.

(reclaim):number Subr

A user call to the function reclaim forces LISP to garbage-collect. The garbage collector used here is a classical mark-and-sweep one. The user may wish to call reclaim to see how much store LISP has left. reclaim returns as its value a number that shows roughly how much space LISP has free. See messon and messoff.

(remainder U:number V:number):number Subr

If both U and V are integers the result is the integer remainder of U divided by V. The sign of the remainder is always the same as the sign of V. An error occurs if V is zero. See quotient

---

(remprop U:any IND:any):any Subr

Removes the property with indicator IND from the property list of U. Returns the removed property or nil if there was no such indicator

---

(repeat N:number BODY:[any]):any Fsubr

The BODY is evaluated N times and the last evaluation of BODY is returned. For example

```
(repeat 4 (turn 90)
          (draw 100))
```

(reset) Subr

reset clears the counters used by time and getime. Thus to discover how long LISP takes to execute a function tedium (say), one can use

```
(progn
   (reset)
   (tedium)
   (list (time) (gctime)))
```

which returns a list showing the resources consumed by tedium

---

(reverse U list) list Subr

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

(reversewoc U list):list Subr

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

---

rpar Variable

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

(rplaca U:dotted-pair V:any):dotted-pair Subr

The car portion of the dotted-pair U is replaced by V. If the dotted-pair U is (a . b) then (V . b) is returned. The type mismatch error occurs if U is not a dotted-pair. It should be used with caution as it actually alters the list cell in memory. Therefore all datastructures using that cell will be changed. E.g.

```
(setq a (list nil))
(rplaca a a)
```

produces a structure that prints as

[illegible]

Eventually the print routines give up the attempt to display the whole of this structure!

[illegible]

[illegible]placed U:dotted pair V:any):dotted pair Subr

The cdr portion of the dotted-pair U is replaced by V. If the dotted pair U is (a . b) then (a . V) is returned. The type mismatch error occurs if U is not a dotted-pair. It is therefore similar to rplaca and the same problems apply.

(save U:filename):filename Subr

This function writes a copy of all of LISP's workspace to the file that is named as its argument. An environment that has been saved in this way can later be restored by using the function load. save is totally non selective: it writes out all variables and their values, all properties and all function definitions. It should normally only be invoked directly from the keyboard.

```
(save 'mdv1_extend)
```

gives a file called 'extend' on microdrive 1, which might perhaps contain a collection of commonly-used definitions which will be wanted at the start of most subsequent LISP sessions. If a file produced by save is present on microdrive1 with the name image when LISP starts it will be loaded automatically. (Please make sure you have a safe copy of the distributed version of mdv1_image before you try this out)

[illegible]scale N integer):integer Subr

scale resets the scale of the current window so that the height of the window is N. The default is (scale 1000).

---

screen OP:number U:number V:number): Subr

screen provides access to a range of QDOS facilities: see TRAP3 in the QL *Advanced User Guide* for full details; here are some of the most useful:

[illegible]

(screen 12 colour width)	reset border
(screen 16 x y)	repositions cursor to character position (x,y)
(screen 17 x nil)	tab to given colour
(screen 18 nil nil)	start a new line
(screen 19 nil nil)	backspace cursor
(screen 23 x y)	position cursor (pixel co-ordinates)
(screen 24 h nil)	scroll window by h
(screen 27 d nil)	pan window by d
(screen 32 nil nil)	clear window
(screen 39 col nil)	set paper colour
(screen 40 col nil)	set strip colour
(screen 41 col nil)	set ink colour
(screen 42 flash nil)	set flash mode 0 or 1
(screen 44 mode nil)	-1 = xor ink into background 0 = normal writing mode 1 = transparent strip
(screen 45 width height)	set character size

See also window.

(sed U:expression) Subr

sed is a subfunction called by the LISP structure editor edit.

(set EXP:id VALUE:any):any Subr

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. EXP must not evaluate to t or nil otherwise an error occurs because t or nil cannot be changed. (set '<anything>) is the same as (setq <anything>).

(setq VARIABLE id VALUE any) any Subr

The value of VARIABLE is replaced by the value of VALUE. VARIABLE must not be t or nil or an error occurs. setq is the normal assignment operator in LISP. See set.

(stop) Subr

When called normally stop exits from LISP (back to SuperBasic). From within rdf, stop exits to wherever rdf was called from.

(sub1 U:number):number Subr

sub1 returns a value of its argument decremented by 1. See also add1

(subrp U:any):boolean Subr

subrp tests whether its argument is the entry-point of a piece of machine code corresponding to a normal LISP function. If so, it returns t, otherwise, it returns nil. See fsubrp for a test identifying those special functions that do not process their arguments in the usual way. Any object that passes the subrp test is also an atom

---

(subst U:any V:any W:any):any Subr

The value returned is the result of substituting U for all occurrences of V in W.

(superprint U:any):nil — Subr

'Prettyprints' 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 standardised places.

A LISP-coded version of superprint is included as one of the demonstration programs distributed with this LISP.

---

t — Special identifier

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 variable.

(terpri):nil — Subr

The current print line is terminated (ie. a new line is started). See also print and superprint.

(time):integer — Subr

Returns the elapsed time (in 1/100 seconds) spent in LISP (excluding time taken in garbage collection) since reset was last called. On the QL times are recorded to a resolution of 1 second. See also clock and getime

---

(times [U:number]):number — Fsubr

Returns the product of all its arguments.

[redacted]race U:id id — Fsubr

Sets up tracing for the function U. For example after the call

```
(trace append)
```

[redacted] the user types

```
(append '(a b c) '(1 2 3))
```

[redacted]e system will respond

```
append ((a b c) (1 2 3))
append = (a b c 1 2 3)
```

where the first line displays the list of arguments that append is being called with and the second line shows the value that it returns. See [redacted]ntrace Note that attempts to trace those system functions used within trace and untrace and tracing of functions that do not evaluate their arguments can lead to trouble.

[redacted]turn U:number):number — Subr

turn adjusts the angle U for the graphics turtle. E.g.

```
(turn -90)      turn left by a right angle
(turn 180)      about face
```

(turnto U:number):number — Subr

turnto sets the turned angle to U

```
(turnto 0)      point directly up the screen
```

---

undefined — Special identifier

When a new identifier is created by read, getchar, character, readline or implode it is given the value undefined. The value of undefined is UNDEFINED. Atoms having neither the value UNDEFINED nor the properties are special in that they do not appear in the list of identifiers returned by oblist, and they may get removed by the garbage collector if no datastructure refers to them.

(until COND.any [V.action]):any — Fsubr

until is used in conjunction with loop. The condition is evaluated, and if it is nil, until behaves as if it had been a quite ordinary function returning the value nil. If the value is not nil the following actions occur:

(a) The values given after the condition are evaluated one by one, and the last of them is returned as the value of until. If there are no such values given, the value of the predicate is returned.

(b) A flag is set to terminate the instance of loop immediately surrounding the until. The loop does not terminate until it is ready to evaluate its next top-level argument. See loop. For example:

```
(loop
      (until (atom x) (print 'atom x))
      (setq x (cdr x)))
```

(untrace U:id) — Expr

Undoes the effect of trace for the function U.

C number [V.number]):

expects all its arguments to be numbers, and sends them sequentially to the screen-handler of your microcomputer. Thus, since 65 is the character code for the letter 'A', (vdu 65 66 67) will print the message ABC.

---

ile COND any [V action]):any — Fsubr

while is used in conjunction with loop. The condition is evaluated, and if non nil the enclosing loop is allowed to continue. If the condition is the values V are evaluated and it is arranged that the last one given is returned as the value of loop. See until

---

ndow CODE:number ARGS:(w h x y)
WIDTH:number COLOUR:number): — Fsubr

CODE is a decimal number. The hexadecimal equivalent may be found in *QL Advanced User Guide*. The list (w h x y) contains four numbers 'w' being the width; 'h' the height of the window; 'x' and 'y' the coordinates describing the new window. These are used to alter a window within LISP. The final two arguments are optional and are used to the border width and colour. For example:

```
ndow 10 '(0 0 0 0))          returns current window
                             size and cursor position
                             (in pixels)
(window 11 '(0 0 0 0))       ditto, but in character
                             coordinates
(window 13 '(w h x y) w c)   redefine whole window
(window 46 '(w h x y) nil c) fill a rectangular block
```

(write FILE:any [ARG:any]):any Fsubr

write is like print, except that it directs its output to the file the handle of which is specified by its first argument, FILE. See open and close. Note that the write functions, like the print ones, can take any number of arguments.

(write0 FILE:any [ARG:any]):any Fsubr

write0 is like prin, except that it directs its output to the file specified by its first argument. See prin andwrite.

(writec FILE:any U:any):any Fsubr

As write but with no escape characters. See write

(writec0 FILE:any U:any):any Fsubr

As write0 but with no escape characters. See write0.

(wrs U:file):any Subr

wrs selects the file U for future output and returns the file-handle of the previously selected output file. A file must be opened before it is selected. open creates a file-handle for use by wrs. See open and rds.

(xtab U:integer):nil Expr

Prints U spaces at the start of a new line.

Appendix A: Installation

Changing the default window

Both the editor and LISP allow the window which is to be used t
altered as part of the initialisation sequence. If this option is not requ
then the default window is used. This is initially the same as the win
used during the start of the program, but if required the default win
may be altered permanently by patching the programs. This is u
where a certain window size and position is always required and ma
that the window does not have to be positioned correctly each time
program is run.

Changing the default drive name

For those users who upgrade their QLs with disc drives, there is
possibility of changing the default drive to something other than mdv
This means that LISP and its image can be copied from the suppl
microdrive to disc, so that the interpreter can be EXECed from the ex
device. This option will not be given when installing the editor ED so
it can be EXECed from any device.

The INSTALL program

The program INSTALL is supplied on the distribution microdrive
perform both of the above tasks. It is run by the command

```
LRUN mdv1_install
```

The program starts by asking whether the default window is to be
up for TV or monitor mode. The minimum window size is greater in
mode because the characters used are larger. You should answer T if y
are setting the default for use with TV mode and M if you are setting
for use with monitor mode. Note that the current mode in use is of
consequence.

The standard window will appear on the screen and can be moved f
means of the cursor keys and altered in size by means of ALT curs
keys. This is similar to the mechanism used when altering the windo
during normal program initialisation. Once the window is in the rig

place and of the desired size, press ENTER.

The program now asks for the name of the file which is to be modified. If you wished to alter the editor then the file would probably be something like 'mdv1_ed'. The next item requested is the name of the program. When a new job such as the editor or LISP is running on the QL, it has a name associated with it. This can be inspected by suitable utilities. The name is six characters long, and whatever is typed here is used as the name and forced to the correct length. The name is of little importance except for job identification.

In the case of LISP the program will then go on to ask for a default drive name where it should look for its image. If you do not wish to change the default drive name the reply should be

```
MDV1
```

(Note - the reply must not be MDV1_). If you do wish to change the default drive name the reply should be the device name, for example

```
FLP1
```

In this latter case LISP will append 'FLP1_' to its image before attempting to load it.

The INSTALL program will then modify the file specified. INSTALL can be run as many times as you like to alter the default window of the editor or the interpreter. It is unlikely to be useful with programs other than those distributed by Metacomco that provide user selection of an initial window such as this Lisp, Metacomco's Assembler and BCPL.

Appendix B: Example programs

The demonstration programs

A number of sample LISP programs are included on the microdrive. Most of them are versions of the ones described in detail in the book *Lisp for the BBC Micro* by Norman and Cattell: in some cases changes have been made to exploit either details of the QL graphics support or the extra memory available on the QL. The programs are intended to form a basis for future work, and so it is important to look at them (using, for instance, the screen editor) as well as run them. For quick reference here is a list of the files and sample function calls to issue after loading each so as to show what is available.

```
animal    (animal)
arith     (evaluate '(+ 2 2))
bignum    (big-power-of-2 100)
compile   (cg '(+ x 2))
demos     (flake) (target) (flower) (snow)
          (target) (path) (path 17) (squiral)
          (squiral 91) (flower)
edit      normally loaded into initial image file
graph     (gsuper) then (circle 100)
          (red (fill (box 100 200)))
          (add (yellow (circle 100))
               (red (box 100 200)))
maze      (adventure)
parse     (parser) then 2 + x * 7
pretty    LISP version of built-in superprint function
route     (find-route 'cambridge 'oxford)
sort      (sort '(zebra aardvark wolf snake hippo))
```

Other files

```
ed
install
lisp
image
```

A file can be loaded by using a LISP command such as:

```
df 'mdv1_demos)
```

efinition of example programs

Some of the example programs listed above are defined below. These finitions have been included for two reasons: to show how a function is fined in this Lisp for the QL and to show how the examples were signed using the graphics features. The graphics package is peculiar to e QL Lisp Development Kit and is not covered in any of the commended primers.

quiral

his example draws a simple spirograph design.

```
defun squiral ((a . 121) (w . 800))
  (cls)
  (moveto 500 10)
  (repeat
       100
       (draw w)
       (turn a)
       (setq w (difference w 2))))
```

ath

his causes a twisting line, or path, to be drawn across the screen which s then repeated.

```
(defun path ((n . 7) (a . 0))
     (cls)
     (repeat
         1000
        (draw 20)
        (turn a)
        (setq a (plus a n))))
```


Target

The third example draws superimposed circles of different colours around a specified centre. The result looks similar to a target board if its centre is visible on the screen. Otherwise it can look like a rainbow.

```
(defun target ((n . 20))
   (cls)
   (repeat n
    (ink n)
    (fill t)
    (circle (times n 25))
    (fill nil)
    (setq n (subl n)))
    (ink 4))
```

Flower

The name of this example function is self explanatory: it draws a flower design! Each row of 'petals' is in a different colour.

```
(defun flower ((a . 0) (r . 0))
     (cls)
     (repeat
          10
       (setq r (add1 r))
       (repeat
            36
          (home)
          (ink 4)
          (turnto a)
          (draw (times 30 r))
          (ink r)
          (fill t)
          (circle 30)
          (fill nil)
          (setq a (plus a 10)))))
```

Books on LISP

Abelson, H. and Sussman, G.J., *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

Acorn User, series of articles by "Stan Froco", March, April, May 1984

Allen, J.R., Anatomy of Lisp, McGraw Hill, 1977.

Artificial Intelligence - an MIT perspective, Vol 3.

Berkeley, E.C. and Bobrow, D., *The programming language Lisp - its operation and applications*, MIT Press, 1964

Byte, August 1979 special issue on Lisp.

Charniack, Riesbeck and McDermott, D., *Artificial intelligence programming*, Lawrence Erlbaum Associates, 1980.

Dickens, A., *QL Advanced User Guide*, Adder, 1984.

Griss, M.L. and Hearn, A.C., *A portable LISP compiler*, Software Practise and Experience 11(6), p. 541, 1981.

Hasemer, T., *A beginner's guide to Lisp*, Addison Wesley, 1984

Hearn, A.C., *Standard Lisp* SIGPLAN Notices 4(9), 1966, ACM

Marti, J. *et al.*, *Standard Lisp Report* SIGPLAN Notices 14(10), 1979, ACM.

Maurer, W.D., *A programmer's introduction to Lisp*, MacDonald/American Elsevier Computer Monographs, 1972.

McCarthy, J. *et al.*, *Lisp 1.5 programmer's manual*, MIT Press, Cambridge Mass., 1962.

Moon, D., *MacLisp Reference Manual*, MIT Press, 1976

Norman, A.C. and Cattell, G.E., *Lisp for the BBC Mic* Acornsoft, 2nd edition 1984.

Oakey, A., *Lisp for Micros*, Newnes Programming Books, 198

Personal Computer World, *Teach yourself Lisp*, series of arti by Dick Pountain, July to December 1984.

Proc 1980 Lisp Conference, Stanford University. The Lisp Comp 1980.

Proc 1982 Lisp Conference, Carnegie Mellon University, 1982.

Proc 1984 Lisp Conference, Austin Texas, 1984.

Sandewall, E., *Programming in an interactive environmen the Lisp experience*, ACM Computing Surveys 1 1978.

Schank, R.C. and Riesbeck, C.K., *Inside Compu Understanding*, Lawrence Erlbaum Associates, 1981

Siklossy, L., *Let's talk Lisp*, Prentice Hall, 1976.

Steele, G., *The Common Lisp manual*, Digital Press, 1984.

Stoutemyer, D. and Rich, A., *MuMath Users Manual*, SoftWarehouse, Honolulu, Hawaii, 1980.

Symbolics, *The Lisp Machine Manual*, Symbolics Inc., Bo Mass., 1980.

Touretzky, D.S., *Lisp - a gentle introduction to symbo computation*, Harper and Row, 1984.

Weissmann, C., *Lisp 1.5 primer*, Dickenson Press, 1967.

Wilensky, R., *Lispcraft*, W.W. Norton & Co., 1984.

Winston, P. and Horn, B., *Lisp*, Addison Wesley, (2nd edition) 1 (Note that the 2nd edition is significantly different from first).