LISP
on the BBC Microcomputer

ARTHUR NORMAN
and GILLIAN CATTELL

```
(COND
   ((OR
      (ATOM X)
      (CHARCOUNT
         X
         (DIFFERENCE LINEWIDTH N)))
      (PRINC X))
   (T
      (PRINC LPAR)
      (SPRINT (CAR X) N)
      (SETQ N (PLUS N 3))
      (LOOP
         (SETQ X (CDR X))
         (COND
            ((AND X (ATOM X))
               (PRINC PERIOD X)))
         (UNTIL
            (ATOM X)
            (PRINC RPAR))
         (XTAB N)
         (SPRINT (CAR X) N)))))
```

LISP

on the BBC Microcomputer

ARTHUR NORMAN
and GILLIAN CATTELL

ACORNSOFT

Acknowledgements

The authors would like to acknowledge Jeremy Bennett at Acornsoft.

First published in 1983, by Acornsoft Limited.

FIRST EDITION

Second revision, September 2019

ISBN 0 907876 08 0

Published by:

Acornsoft Limited
4a Market Hill
Cambridge
CB2 3NJ
England

Contents

1 What is LISP?

LISP is a language of contrasts. It is one of the oldest computer languages, but is still forward looking. Many large programs have been written in LISP, and the language provides good support for the professional programmer. At the same time, LISP is an excellent language for beginners.

Most makes of mainframe and an increasing number of micros support LISP. As a result, there are now many LISP dialects. Despite the lack of a definitive LISP standard, adapting LISP code when moving it from one machine to another is usually easy and so LISP is portable.

The most publicised uses of LISP have been in artificial intelligence - programs that hold conversations, translate languages, control robots and prove theorems have all been written in LISP. There are LISP programs to design integrated circuits, support new computer languages, integrate and differentiate algebraic expressions and write stories for children. There are LISP-based computer games and interactive full-screen editors.

This book is an introduction to the LISP system that Acornsoft provides for use with the BBC Microcomputer. The first part introduces the various LISP functions, and shows each of them being used in isolation. The second part consists of a set of medium-sized example programs illustrating the way in which real LISP code is built up. Between them the sample programs contain much code that can be used, or at least adapted for use, in realistic LISP applications.

Much of LISP's strength comes from the fact that the language naturally supports a high-quality interactive programming environment. This means that it provides good facilities for trapping errors, tracing faulty programs and testing code as it is written. Learning to use LISP involves getting accustomed to these features and finding out how to adapt them to your particular needs. The sections here describing the LISP prettyprinter, editor and trace package show how to build the kernel of a LISP environment, and extending these and designing new tools will be important to serious users of the system.

This manual assumes that the reader is familiar with the BBC Microcomputer and/or has a basic but not necessarily thorough knowledge of computers. A knowledge of any other computer language is useful but not essential. An ability to type examples exactly as they appear in the text _is_ essential however, so too is the ability to count brackets!

Because LISP can be used in a desk-calculator mode, it should be easy to review the material presented by trying out examples on the computer. This is particularly recommended for the first few chapters, since they introduce the syntax of LISP and some basic functions which are used extensively later.

2 Preliminaries

LISP requires a BBC Microcomputer with 32K bytes of memory. This will normally be a Model B machine, but could be a store-extended Model A. Versions of LISP are available to exploit some of the more important options available for the micro.

In particular, versions exist for cassette- and disk-based systems, and for ones where a second processor is present. The main variations between these different versions relate to loading, input and output, speed and the amount of memory available to the user, otherwise they are compatible. ROM and second-processor versions of LISP have significantly more memory available to them than cassette and disk versions, and may use some of it to provide more built-in functions. The following sections explain how to load and start each of the versions of LISP.

2.1 LISP on cassette

To use LISP with a cassette-based system, connect the BBC Microcomputer to a television set or monitor and to a cassette player, following the instructions given in the BBC Microcomputer <u>User Guide</u>. Place the LISP cassette in the deck and rewind it. Set the volume and tone controls of the cassette player to the levels determined using the 'Welcome' cassette, type

```
CHAIN "LISP"
```

and press RETURN. If the cassette deck has automatic motor control it should start. If not, you will have to play the tape, stopping it when LISP announces that it has loaded. As usual, the Microcomputer displays a message 'Searching' while scanning the tape looking for LISP, 'Loading' when it finds the start of LISP on the tape, and then a sequence of numbers providing confirmation that loading is proceeding satisfactorily. Loading can be interrupted at any stage by pressing ESCAPE or BREAK on the computer, and the most straightforward response to any loading errors is to start the entire loading process again from the beginning.

Part way through the loading process LISP asks you to select a screen mode. In normal circumstances it will be best to reply by pressing 7 (i.e. Teletext 40 characters per line mode) followed by RETURN. When loaded, LISP displays a prompt:

```
Evaluate:
```

which indicates that it is ready to accept input.

2.2 LISP on floppy disk

Acornsoft LISP on disk can only be used with a 40-track disk drive.

With the relevant disk in place, entering LISP is achieved using the 'auto-boot':

1 Press, and hold down, the SHIFT key
2 Press and release BREAK
3 Release SHIFT

The LISP prompt should appear after a few seconds, rather than the minutes taken when loading from cassette.

2.3 LISP as a language ROM

After inserting the LISP ROM into one of the paged ROM (or 'sideways' ROM) sockets the computer will respond instantly to the command

```
*LISP
```

with the LISP prompt.

Since main memory is not used for the code making up LISP, the user will be able to run larger applications than are possible with either the cassette- or disk-based versions of LISP.

When installed as a language ROM, LISP does not prompt the user to select a screen mode - it simply utilises whichever mode was in use at the time it was entered. The mode can subsequently be changed by calling the LISP function MODE.

NOTE: If desired, it can be arranged that the computer enters LISP (rather than Basic) directly on power-up and when BREAK is pressed - see the ROM-fitting instructions provided with the ROM package, or ask your dealer.

2.4 LISP with a second processor

For LISP applications requiring more store than is available even with the ROM version of LISP, an auxiliary processor connected to the BBC Microcomputer via the Tube ® can be used. Acorn Computers offer a processor compatible with the main one in the BBC machine, but which is faster and has more memory. Loading LISP into this second processor is performed using the commands described previously for cassette and disk systems, preceded by the directives (discussed in the second processor manual) that enable the Tube.

When LISP is run in this way it is possible to select graphics or 80-column screen modes without encroaching on the store that LISP wishes to use. The ROM version of LISP automatically tests for the presence of a second processor, and if one is available copies itself to it.

2.5 Local editing

Except when the LISP programmer has explicitly arranged otherwise, LISP only takes account of what has been typed on a line when RETURN is pressed. Up to that point the BBC Microcomputer's local editing facilities may be used to correct typing errors and transcribe previously displayed text into the current input line.

DELETE

The DELETE key can be used to delete the character which is shown immediately before the cursor (this is normally the last character typed).

Use of the cursor control keys; →, ←, ↑ and ↓ causes the Microcomputer to display two flashing cursors, one at the normal place where typed characters are to be entered in the input line, the other free to move anywhere on the screen.

COPY

The COPY key moves the character identified by the free cursor down into the input line, and so makes it easy to repeat the entry of any text displayed on the screen. All characters are subject to auto-repeat; that is, if a key is depressed for more than a brief period the computer will act as though the key were being hit repeatedly. This makes for convenient use of the cursor controls, COPY and DELETE, as large amounts of text can be traversed or edited rapidly. The delay before auto-repeat starts, and the repeat rate once it has, can be set using the '*FX' calls described in the <u>User Guide</u> or by the equivalent use of CALL from within LISP.

Transactions with LISP generally take the form of strings of words enclosed in brackets. LISP cannot start any calculation until it has a complete expression to work on, and this involves both matched brackets and the use of the RETURN key. When ready for a new command, LISP prompts with the message

```
Evaluate:
```

RETURN

If the RETURN key is pressed before enough closing brackets have been typed to match the opening ones, LISP displays a prompt consisting of a sequence of arrows, the number of arrows corresponding to the number of brackets outstanding.

ESCAPE

An entire LISP transaction can be aborted by pressing the ESCAPE key. This can be used for abandoning incorrect input (regardless of whether it is on one or several lines) as well as interrupting unwanted calculations or printing.

BREAK

The BREAK key on the computer's keyboard should only be used when you have finished with LISP. It causes an immediate exit from LISP back to the most-recently selected language installed in ROM-form in your computer. The only safe way of restarting LISP after a BREAK is to reload it. Naturally this loses any user-introduced function definitions and all variable values. However, if LISP is uncorrupted it may be possible to recover from BASIC by using CALL PAGE+2.

CTRL-BREAK

If CTRL (control) is pressed at the same time as BREAK the computer will do a full reset, and select the language ROM plugged into the rightmost ROM socket.

CTRL-N

Sometimes LISP will generate more than one screenful of output in response to a single input command. Normally the first few lines of such an overlong display will disappear off the top of your television screen or monitor before you have a chance to read them. This effect can be avoided by setting the BBC Microcomputer's VDU driver into a mode where it pauses after each couple of dozen lines of output.

Typing CTRL-N (i.e. holding the CTRL key down while typing an N) followed by a RETURN will set this mode. It will also produce some spurious diagnostics from LISP which can be ignored for now.

CTRL-O

Typing CTRL-O will let the computer revert to its normal, unrestrained, scrolling mode. Following a CTRL-N the computer pauses whenever the screen becomes full, waiting until the user presses the SHIFT key. LISP automatically sets this 'paged' mode of operation when printing error messages, so that these messages should not be lost off the top of the screen before they can be read - if the system seems to have mysteriously stopped in the middle of printing a long and inscrutable message, try touching the SHIFT key!

CTRL-U

Typing CTRL-U will cause the computer to delete all the characters typed so far on the current line.

3 An introductory session with LISP

One of the particular strengths of LISP is its ability to store and retrieve structured data. A demonstration of a computerised address-book, using some of the facilities available in LISP, is given below.

In this simplified example it will be convenient to restrict people's names to single words (such as TARZAN or JANE). However, an address is clearly a structured object, composed of a number of lines of information, each of which may consist of several words.

To store this information using LISP, its structure is indicated by writing the whole address, and each of its major subfields, in brackets. This removes possible ambiguity by grouping together 'words' which belong together. For example, further copies of this manual may be ordered by writing to

```
((ACORNSOFT LTD) (4A MARKET
HILL) (CAMBRIDGE) (CB2 3NJ)(
ENGLAND))
```

Note that to maintain consistency between this address and others where the city or country require several words, both CAMBRIDGE and ENGLAND, which happen to be single word subfields, have been enclosed in brackets. Thus, for instance, the structure used can deal gracefully with addresses such as

```
((WHITE SANDS MISSILE
RANGE) (NEW MEXICO) (
88002) (UNITED
STATES OF AMERICA))
```

Here the ordering of one-word and multi-word fields is quite different from that in the Cambridge address. The two examples given have been written out on short lines, with line breaks at awkward places to stress the fact that all structure in LISP is indicated using brackets, and that limits based on (say) the number of characters which can be displayed across a television screen are irrelevant.

Two useful directives for associating pieces of information with one another, and later retrieving them for inspection, are called PUT and GET.

An address may be attached as a 'property' of a name by entering a LISP statement such as:

```
(PUT 'QUEEN 'ADDRESS '((BUCKINGHAM PALACE) (LONDON)
(ENGLAND)))
```

in response to the 'Evaluate:' prompt LISP displays. The quotation marks (') shown will be discussed further in the next chapter - they just direct LISP to treat what follows them as textual data rather than program needing to be obeyed. After PUT has been called, the saved information can be recovered by entering

```
(GET 'QUEEN 'ADDRESS)
```

This will find and display the stored address.

A succession of uses of PUT can build up a complete record of all the addresses to be stored. Should it be necessary to change the address attached to some name, the new address may be entered in the same way, and it will then replace the previously stored information.

Hyphens (-) are treated by the LISP system in just the same way as any alphabetic character. This means that we can have single-word names with more than one component by joining them with a hyphen:

```
(PUT
  'GOLDEN-WONDER
  'ADDRESS
  '((EDINBURGH-HOUSE)
    (ABBEY-STREET)
    (MARKET-HARBOROUGH)
    (LEICESTERSHIRE)
    (ENGLAND)
    (GREAT-BRITAIN)
    (EUROPE)
    (PLANET-EARTH)))
```

Having typed in a number of addresses, you will need to know how to ensure that they will be there to look up next time LISP is running on the computer. SAVE and LOAD directives can be used to allow a session to continue from the point where a previous one left off, with all the information in memory when SAVE was used still available.

A name is chosen for the file in which the session will be stored. For this example the name ADDRESS will be used, and then

```
(SAVE 'ADDRESS)
```

is entered. In the case of a cassette-based system, LISP will expect you to have put a blank cassette into the tape recorder, and will require that you set the deck to record the tones it emits. The saved session can be restored when LISP is running on some future occasion by entering

```
(LOAD 'ADDRESS)
```

It should be noted that when LOAD is used during a session the loaded file overwrites whatever LISP had in its store before the LOAD. Consequently if this information will be required later it should first be SAVEd.

You should try for yourself entering and looking up a few addresses using PUT and GET. Observe the effect of setting a new address for some individual, and verify that you can use SAVE and LOAD successfully. Also, try storing other properties with names, for example, putting BIRTHDAY or PHONE-NUMBER in place of the key

ADDRESS. Clearly it will be possible to record many items of information connected with a name using very simple pieces of LISP. For example:

```
(PUT 'JULIUS 'ADDRESS '((CAESARS PALACE) (ROME)))
(PUT 'JULIUS 'OCCUPATION 'EMPEROR)
(PUT 'JULIUS 'PHONE-NUMBER '(EX-DIRECTORY))
(PUT 'DVLC-SWANSEA 'PHONE-NUMBER '(0792 72151))
```

and so on!

Observe that the Driver and Vehicle Licensing Centre telephone number includes a leading zero in its area code. Try entering this, followed by (GET DVLC-SWANSEA 'PHONE-NUMBER). What gets printed is (792 72151). This is because LISP has treated the string of digits as a number, and so considered the zero irrelevant.

Later chapters in this manual show how to remove some of the limitations of the above database scheme. In particular, it will become apparent how the restriction that names be represented as single words can be overcome. Also, it will be shown how to handle arbitrary strings of characters, so as to avoid curiosities concerning punctuation marks and numeric conversion, and functions will be described which make it possible to print structures such as the above addresses using multi-line layout as normally seen on envelopes, rather than LISP's bracketed notation.

4 Variables and quotation

In the examples in chapter 3, all items of data were prefixed by a single quote mark ('). This symbol informs LISP that whatever follows immediately after, be it a word or a bracketed list of items, is literal data. In the absence of a quote mark LISP treats everything it sees as program. Thus if LISP is presented with

```
      'COLOURS
or    '(RED WHITE BLUE)
```

it will reply by printing the literal data given, that is

```
      COLOURS
or    (RED WHITE BLUE)
```

However, if it is given just

```
COLOURS
```

without the initial quote mark, LISP will treat the word given as a 'variable' called COLOURS and will attempt to print out its value. Initially most names do not have associated values, and so LISP will reply with

```
UNDEFINED
```

The assignment operator SETQ can be used to give a variable a value, and it is used as in

```
   (SETQ COLOURS '(RED WHITE BLUE))
or (SETQ WHERE (GET 'JULIUS 'ADDRESS))
```

As can be seen, the value to be stored in the variable can be either a literal expression (preceded by a quote mark) or a reference to some LISP function.

Thereafter (at least until the value is changed by a further SETQ operation) entering the name of a variable without a quote mark will produce the value assigned by the SETQ. So, for instance, COLOURS will expand into its LISP value (RED WHITE BLUE). Of course, regardless of how SETQ is used to give a value to the variable COLOURS, every time

```
'COLOURS
```

is entered the quote mark will cause LISP to leave the word COLOURS unevaluated, and so the result displayed will be just that one word.

There is a certain similarity between the actions of PUT and SETQ as the latter also attaches a value property to a name. However, here values are accessed via the name of the variable itself, so there is no need for any explicit retrieving function similar to GET.

Two things should be noted at this stage. First, when the item prefixed by a quote mark is a list, as in '(RED WHITE BLUE) above, the whole list (including both words and sublists within it) gets treated as literal data. Secondly, no quote mark preceded (GET 'JULIUS 'ADDRESS), hence LISP was required to evaluate this in its normal way, rather as if it had been entered directly from the keyboard as an expression on its own. It is quite often necessary to construct new lists partly out of literal data and partly out of the values of variables and the results of computations. This can be done using an operator called LIST.

Imagine that variables have already been assigned thus (or, better still, enter these assignments):

```
(SETQ FIRST 'JIM)
(SETQ SECOND 'JOE)
(SETQ THIRD 'JACK)
```

To assign the list (JIM JOE JACK) to a variable WINNERS using SETQ we can, of course, quote the list:

```
(SETQ WINNERS '(JIM JOE JACK))
```

But we could replace the literal list by either of the following constructed lists to achieve the same effect:

```
   (SETQ WINNERS (LIST 'JIM 'JOE 'JACK))
or (SETQ WINNERS (LIST FIRST SECOND THIRD))
```

LIST can be used to build multi-level lists with a mixture of literal and computed data included, as in the more complicated example

```
(SETQ SCORE-BOARD
  (LIST '(RESULTS OF COMPETITION)
         (LIST 'WINNER FIRST)
         (LIST 'RUNNER-UP SECOND)))
```

In the environment we have established it will generate the list

```
((RESULTS OF COMPETITION) (WINNER JIM) (RUNNER-UP
  JOE))
```

At least in terms of the LISP facilities introduced so far, this is a neatly formatted display of results!

Lists, both literal and constructed, have a very important role in LISP programming. They can be used as working data, as pieces of program, as things to store away information for some future purpose and as messages to be printed. Already you have begun to see these uses. For example, the list (RED WHITE BLUE) was working data assigned to the variable COLOURS, while (LIST FIRST SECOND THIRD) was treated as a call to the list-building operator LIST and references to three variables FIRST, SECOND and THIRD. The addresses in chapter 3 were lists saved first as properties of words using PUT, and then saved wholesale on tape or disk using SAVE. Later chapters will elaborate upon the nature and uses of lists, demonstrating the utility of having a <u>LIS</u>t <u>P</u>rocessing language such as LISP.

5 Simple function definitions

Programming, whatever language is chosen, regularly involves performing sets of closely-related actions, as anyone who stored even a modest number of addresses in the 'address book' of chapter 3 will have realised! Two illustrations of the desirability of a 'shorthand' way of carrying out frequently-needed actions are the necessary sequence of PUT directives, all referring to the same property (ADDRESS in our example) and the similarly restricted GET enquiries. Function definitions like the following meet this requirement:

```
(DEFUN FIND-ADDRESS (PERSON)
   (GET PERSON 'ADDRESS))
```

Using the directive DEFUN, a function has been defined to carry out one GET operation to retrieve a stored address. When this is entered LISP records the given definition under the chosen name, FIND-ADDRESS. Any previous definition FIND-ADDRESS happened to have is replaced. Nothing else is done until the function is called.

```
(FIND-ADDRESS 'JULIUS)
```

is a function call, and is equivalent to

```
(GET 'JULIUS 'ADDRESS)
```

handing back exactly the same result for printing.

The definition describes a class of actions (in this case the stored ADDRESS property) while the call specifies the details of a particular case (the address of JULIUS). In this example, JULIUS is passed as an 'argument' for FIND-ADDRESS. In the function definition, DEFUN is followed by the name of the function, and then a 'local variable list'. This is sometimes referred to as a parameter list or list of formal arguments). Local variables introduced in this way provide the means for referring to the values of arguments. They can also be used as working storage for intermediate results.

To evaluate a function call, LISP associates the value of the arguments with the variables named in the local variable list. This means in our example that the variable PERSON will be given the value JULIUS. LISP then evaluates the body of the function. The body can either be a single LISP form as shown above, or a sequence of operations. The value returned by the last (or only) part of the body is saved since this is the value LISP will hand back as the result of the function call. Before this value is returned, all local variables introduced by the function are restored to whatever state they had before the call started.

Clearly, FIND-ADDRESS as defined above is a very simple, short function, and using it does not result in a massive reduction in the amount of LISP code which must be entered to retrieve information. However, good LISP code often includes definitions as short as this, using them to keep program structure clean as well as short. It is sound practice for implementation details, for instance how addresses are stored and found in a particular scheme, to be enclosed within access functions which are easy to call. This leads to readable code, as mnemonic names can be used for the access functions. It is then easy to alter the program if it later becomes appropriate to change the way in which data is represented.

As an example of a function definition where one call causes multiple effects consider:

```
(DEFUN PUT-STATISTICS (NAME YEARS CM KILOS)
   (PUT NAME 'AGE YEARS)
   (PUT NAME 'HEIGHT CM)
   (PUT NAME 'WEIGHT KILOS)
   (LIST 'STATISTICS 'RECORDED 'FOR NAME))
```

Here a call might be

```
(PUT-STATISTICS 'CHARLIE 33 165 72)
```

The function has as its definition four parts. The first three are uses of PUT, the last simply builds a list, as in (STATISTICS RECORDED FOR CHARLIE) which will be returned as the value of PUT-STATISTICS and so will get printed when a call to the function is handed directly to LISP.

The links LISP makes between local variables and argument values are temporary ones, lasting only while the body of the function is being processed. This means that it is quite in order to use the same name as a local variable in several functions - the separate uses will not interfere with one another. It also means that if SETQ is used to change the value of a local variable the effect of the change only persists so long as the function that introduced that variable is being processed.

Two points should be noted here. The first is that the LISP functions LIST and PUT are being used inside PUT-STATISTICS. LISP function calls can be nested arbitrarily, and wherever quoted data could be written a function call will be valid. The second point is that when PUT-STATISTICS was called the numeric arguments were not preceded by quote marks - this of course meant that LISP will have evaluated the numbers 33, 165 and 72 in the call. The LISP evaluation mechanism is arranged so that this gives the expected numeric values.

A further illustration of a function definition which could result in less typing, and which shows the use of LIST with some of its arguments being constant and some computed, is

```
(DEFUN DESCRIBE (NAME)
   (LIST (GET NAME 'AGE) 'YEAR-OLD NAME
         'OF (FIND-ADDRESS NAME)
         'IS (GET NAME 'HEIGHT) 'TALL
         'AND 'WEIGHS (GET NAME 'WEIGHT) 'KILOS))
```

This collects a variety of pieces of information about the person named, both through the direct use of GET and by calling FIND-ADDRESS, and interleaves this information with descriptive words to make a long list.

Inherent in the ability to embed pieces of LISP inside one another to (almost!) any depth is a noticeable build-up of brackets - this is not necessarily a sign of poor LISP code. Indeed, although at first it appears a bit odd, it should soon become apparent that this bracketing aids understanding by making the structure of pieces of LISP code absolutely explicit. In all the examples given in this manual the bracket structure of LISP will be highlighted by the use of indentation, and a set of LISP functions displaying existing LISP code in this style is presented and explained later.

6 Reprise

In the examples of LISP programming already seen, words have been used both as literal data (generally preceded by ') and as variables which can store values. Similarly, lists have appeared both as literal data and as LISP's representation of pieces of program. These programs have been built up from functions which construct lists and make associations between names and values.

Words in LISP are known as 'identifiers' or 'atoms', and are the most elementary objects used when creating LISP datastructures and programs. The difference between them is that identifiers are solely those objects made up out of strings of characters (such as A, NIL and GOLDEN-WONDER) while atoms include all of these together with numbers (e.g. 2, 9999 and -77). All of the lists we have seen so far, whether as data or as program, have been written as a left bracket, then a sequence of atoms or sublists and finally a right bracket. In the next chapter we will see how these structures are represented within LISP, and will in the process find that lists are actually a special case of things called 's-expressions' which are built up out of primitive structures known as 'dotted pairs'.

So far, then, we have seen:

(a) How to store information as a property of an identifier using PUT:

```
(PUT identifier property-name property-value)
```

(b) How to inspect this property once it has been established:

```
(GET identifier property-name)
```

(c) How to assign values to identifiers:

```
(SETQ identifier value)
```

(d) How to reference the value of an identifier, and how to distinguish between the use of a variable and the specification of literal data.

```
'literaldata
variablename
```

(e) How to define simple functions using DEFUN, and how expressions built up using all the above facilities can be combined to make it possible to build, store and retrieve data and messages.

```
(DEFUN function-name (arg1 arg2 ... argn)
body of the function)
```

Eventually we will also need to know how to remove properties, dismantle structures or obtain copies of extracts of them and perform tests and comparisons on LISP data.

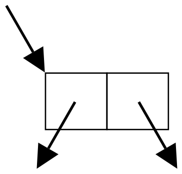
7 Datastructures

It is often useful to have a diagrammatic representation of the structures on which your programs operate. This chapter introduces one such notation for LISP, and aims to give some insight into the way LISP sees and manipulates what is presented to it.

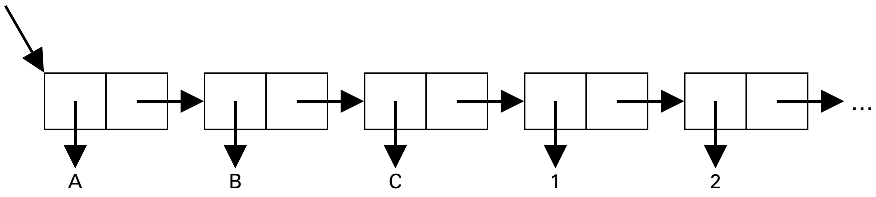
Consider a list: it consists of a number of items each of which, except the last, has a successor. LISP wishes to treat lists as a class of objects which can be referred to in a uniform way, irrespective of their length. The idea that lists can be traversed by stepping along them one element at a time helps greatly with this.

LISP also wishes to allow identifiers and numbers to be mixed freely with lists. It achieves its ends by handling everything via 'pointers':

In the case of lists, such a pointer will identify a cell, each half of which in turn contains another pointer:

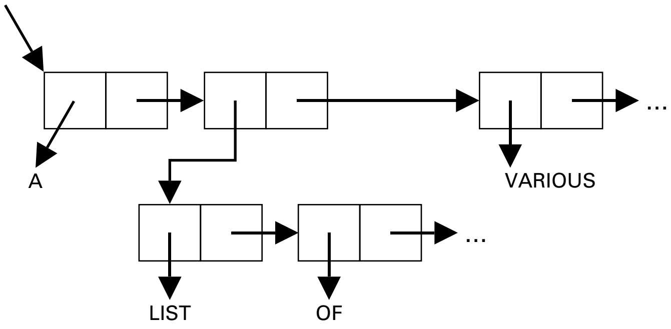


The lefthand pointer leads to the first element in the stored list, while the righthand one points the remainder. So for the list (A B C 1 2 ...) we get:

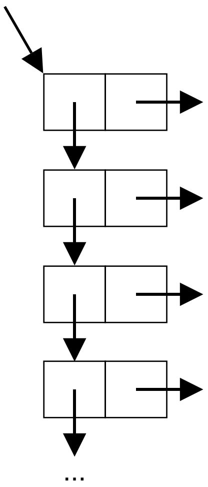


For the moment there is no need to consider the details of how LISP represents words and numbers (see chapter 21). Of course, the elements need not be atoms, so we might have

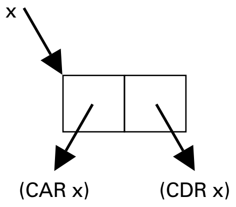
(A (LIST OF ...) VARIOUS ...)



Indeed, lists and sublists can be nested to arbitrary depths:



LISP provides two basic functions for following the pointer chains in list structures. CAR extracts the lefthand pointer of a pair, while CDR extracts the righthand one. For the lists considered here CAR selects the first item in the list, while CDR of the same list would be the remainder.



The names CAR and CDR, which LISP uses for these basic selectors, are acronyms related to the addressing structure of a long-defunct type of computer. Over the years they have become so thoroughly identified with the idea of left and right subfields in a simple cell that they are now simply the proper technical terms to be used for these functions.

By reference to the box diagrams for list structures (shown above) it can be seen that if a pointer x refers to a list, (CAR x) refers to the first item in the list. Similarly, (CDR x) is the sublist left over when this first element is discarded. Combinations of these two functions can extract any desired component of a list: for instance, after

```
(SETQ A '(A (LIST) (WITH SUBLISTS)))
```

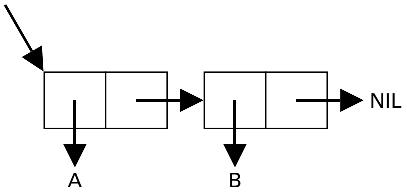
had set A to refer to a list of length three, items could be extracted as follows:

```
(CAR A)                = the identifier A
(CDR A)                = the list with two members,
                           ((LIST) (WITH SUBLISTS))
(CAR (CDR A))          = the list with one member
                           (LIST)
(CAR (CAR (CDR A)))    = the identifier LIST
(CDR (CDR A))          = the list with one member
                           ((WITH SUBLISTS))
(CAR (CDR (CDR A)))    = the list with two members
                           (WITH SUBLISTS)
```

and so on. By drawing out the pointer structures involved in the above examples it should become clear how CAR and CDR are being used.

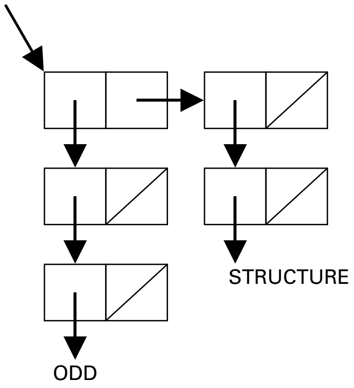
In the above A is a list of three items, (CDR A) has length two, and (CDR (CDR A)) is of length one. This leads to an expectation that (CDR (CDR (CDR A))) will be a list without any items in it at all, i.e. an empty list which could be written as just (). LISP chooses to represent the empty list by a special identifier called NIL, and when reading expressions it treats NIL and ()

as being equivalent. Thus the full box diagram for a simple list (A B) is:

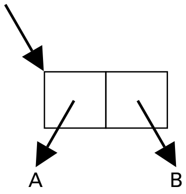


Since structures terminating with an empty list are so heavily used it has become conventional to fill in the final CDR field in the picture of a list not with a pointer to NIL but with a stroke:

(((ODD)) (STRUCTURE))



Throughout this book the word 'list' will be reserved for structures chained together as in the examples given so far, with NIL as a neat terminator for the CDR direction chaining. More general structures can be created by the LISP system. For example

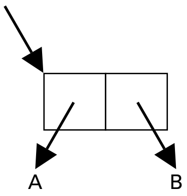


We have not (as yet) seen how to make the LISP reader accept such a form or the LISP printer display it. The term 'list structure', or 's-expression' will be used when there is a chance of such things occurring. The functions CAR and CDR are not, of course, at all concerned with the global properties of the structures on which they work. They can therefore be used to follow whatever chains of pointers a LISP program creates.

Other functions are not so understanding, and so the next chapter contains an explanation of ways of building both lists and s-expressions, of how general structures get printed and of how some of the clumsiness of deeply-nested calls to CAR and CDR can be avoided.

8 Further datastructures

In terms of the diagrams used in chapter 7, the primitive list-building function CONS can be thought of as allocating a fresh cell and filling in its CAR and CDR pointers. Thus (CONS 'A 'B) gives the structure



Similarly (CONS 'ITEM NIL) is the list (ITEM). CONS can obviously be used to push new items onto the front of an existing list. If the variable L refers to a list then (CONS p L) is a list one item longer, with p as its first element and the rest of the old L following on. From the box diagrams it should be clear that for any u and v

```
    (CAR (CONS u v)) = u
and (CDR (CONS u v)) = v
```

Nested calls to CONS can build any list that the LISP user might require, but for long lists its use gets rather clumsy. This is where the function LIST comes in - it can be thought of as a shorthand for nested CONSes For instance (LIST 'DATA 'FOR NAME) can stand for

```
(CONS 'DATA (CONS 'FOR (CONS NAME NIL)))
```

The number of CONSes which LIST expands into is equal to the number of arguments it is given. For consistency, if LIST is given no arguments it returns just the empty list, NIL, without doing any CONSing at all. It can be observed that LIST builds up its result from the tail end, each successive CONS being used to put another item on the front of the constructed list.

As well as the normal input notation for lists, LISP provides a format which stresses the way structures are built up out of CONS cells. This notation is mainly used in cases where a datastructure does not terminate (in the CDR direction) on a NIL, and so is not a proper list. In this notation, the result of (CONS 'A 'B) would be written (A . B). This is known as a dotted pair. The '.' in dotted pairs indicates a cell that may have a non-NIL atomic CDR. Dotted pair notation can be used for lists if the item after the '.' is a list or NIL. When using it, the list normally input as '(1 2 3) would be presented as

```
'(1 . (2 . (3 . NIL)))
```

where each pair of brackets corresponds to a single cell in the structure.

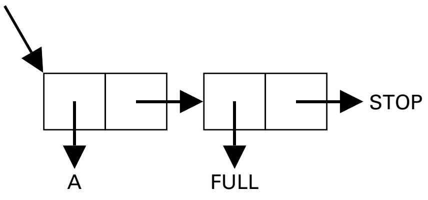
List and dot notation can be mixed, giving further ways of expressing the same structure:

```
'(1 . (2 3)), '(1 . (2 . (3)))
```

Furthermore, lists can be written in dot notation with several items before the dot, giving two more equivalents for '(1 2 3):

```
'(1 2 . (3)), '(1 2 3 . NIL).
```

The last of these examples shows the format which is generally preferred when working with data structures which do not end with NIL. The standard LISP print functions display, for instance, the structure

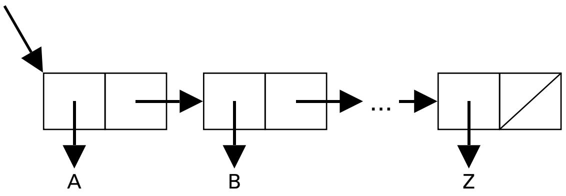


```
(A FULL . STOP)
```

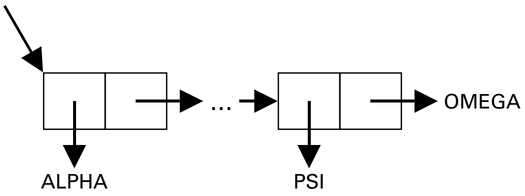
All of this flexibility over input formats can easily become confusing. It is suggested here that the route out of this confusion is the conversion of input strings into their equivalent box diagrams, where the only two rules needed are that lists like

(A B ... Z)

turn into



and that anything in the form (ALPHA ... PSI . OMEGA) becomes



where each letter in the above two examples can stand for either an atom or <u>any</u> other list.

In the majority of simple applications of LISP it will never be necessary to use dot notation for input and, since all datastructures built will be proper lists, dots will never appear in the output that LISP generates.

The function CONS makes it easy to join new objects to the start of a list. Consideration of the box diagrams should make it obvious that adding things to the end of an existing list is a much more complicated operation. In particular, it involves making a copy of the list that is to be extended, with the final NIL-pointer replaced by a pointer to the required extension. There is a LISP function which does this, and it is usually called APPEND (see Appendix B). However, most LISP programs will turn out to be both neater and faster if they arrange to build up lists by using CONS to add to the front, rather than APPEND to add to the end.

In particular, when it is necessary to add things to the end of an existing long list, it is useful to be able to alter the pointers in existing lists. This can (sometimes) drastically reduce the number of times that structures have to be copied while they are being converted from original to desired formats. The LISP functions which support this are called RPLACA and RPLACD, contractions of the word 'replace' and an 'A' or a 'D' to indicate whether it is the CAR or CDR field which needs changing. If L refers to a certain CONS cell, then (RPLACA L 'ASTON-MARTIN) overwrites the CAR field of that cell so that it refers to the atom ASTON-MARTIN. RPLACD acts in a similar way by overwriting the CDR field. Each of the two functions returns a reference to the cell specified by their first argument, so for instance

```
(RPLACD (RPLACA L new-car) new-cdr)
```

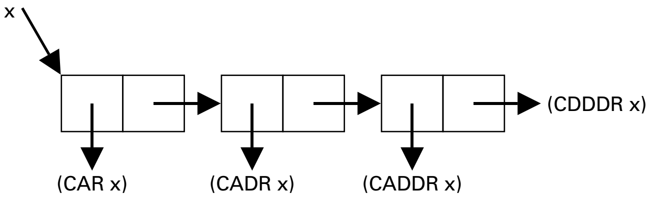
changes both fields within L. Updating existing lists using these two functions can have unpleasant consequences for the unwary. When a cell is overwritten, <u>all</u> structures sharing that cell reflect the change, so when RPLACA/D are used it is important to make copies of structures which may be needed later and are subject to overwriting. Also, such constructions as

```
       (RPLACA L L)
and    (RPLACD L L)
```

can build lists which are circular, thereby causing trouble to several built-in LISP functions (including the print ones). Modern LISP systems tend to discourage the use of the RPLAC functions, considering that in almost all circumstances the time savings they give are not worth the trouble they can cause.

LISP programs tend to be built around datastructures consisting of several CONS cells, where a group of say three or four cells is regarded as a unit. If these cells are arranged as a linear chain in the CDR direction, with a NIL at the end, the function LIST will build the structure in a painless way. In other cases it is useful to introduce user-provided routines which do all the necessary CONSes at once. Similarly, it is useful to have functions corresponding to the compositions of CAR and CDR which select components out of small pieces of structure. Built in to LISP is a collection of such access functions, corresponding to all possible combinations of up to three CARs and CDRs. The names for these functions are obtained in a systematic way: each starts with a 'C' and ends with an 'R', with an 'A' in-between for each CAR and a 'D' for each CDR. The following examples should make this clearer:

```
(CAAR x)       = (CAR (CAR x))
(CADR x)       = (CAR (CDR x))
(CDAR x)       = (CDR (CAR x))
(CDDR x)       = (CDR (CDR x))
(CADDR x)      = (CAR (CDR (CDR x)))
(CDAAR x)      = (CDR (CAR (CAR x)))
```



The longer names can become unpronounceable, and if any program is found to be making heavy use of them it probably indicates that the user should introduce more mnemonic names for the data access operations being done.

The remaining LISP function to be explained in this chapter is the primitive equality test, EQ. EQ works by comparing two pointers to see if they refer to the same object. It is arranged that LISP identifiers are kept in a special table, so that all references to an identifier are clearly the same. Thus EQ provides a direct way of checking whether two pointers refer to the same identifier. It is arranged that EQ also returns LISP's representation of 'true' if its two arguments are equal numbers. But EQ will always return 'false' if one argument is an identifier and one a number or list. For lists, EQ provides a test of whether two pointers identify the same CONS cell. Note that since CONS always makes a new cell when it is called, i.e. one different from all other existing CONS cells, it is always the case that

```
(EQ L (CONS (CAR L) (CDR L))  = false
```

even though the two list structures which are being compared have identical shape and print as the same collections of characters. Later in this book we will meet a function EQUAL that makes a full scan of its arguments, testing to see whether they are equivalent in shape and content. EQUAL will obviously be slower than EQ, but may sometimes be needed.

9 Arithmetic in LISP

In LISP, numbers are just a special sort of atomic (i.e. non-list) object. A number can be written anywhere it is needed in a fragment of LISP code - the quote marks required with other sorts of data are not necessary because it is understood that numeric values stand for themselves. Thus one can write (SETQ COUNT 0), and the effect is just the same as (SETQ COUNT '0). Since the interpretation of numeric values is fixed in this way it is not (of course) possible to use numbers as variable names. Such things as (SETQ 1 2) are illegal and will give rise to error messages.

The arithmetic built into Acornsoft LISP works on 16 bit integers. This means that numbers in the range -2^15 to (2^15)-1 can be handled, i.e. -32768 to +32767. Any attempt to generate a number outside this range will cause an error to be signalled. The fact that overflow is detected and reported provides some slight compensation for the small range of numbers that can be used.

The syntax of LISP does not provide infix operators such as '+' and '-' for combining numbers. Arithmetic operations are invoked in just the same way as all other LISP facilities, by explicit function calls. The function which adds numbers together is called PLUS, and so one has to write

```
(PLUS 2 2)
(PLUS 1 2 3 4 5)
```

As can be seen in the second example, PLUS can take many arguments, and its result will be the sum of them all. (Note: in Acornsoft LISP PLUS can have up to 28 arguments). Arithmetic can be mixed with assignment statements and variable reference:

```
(SETQ N 3)
(SETQ N (PLUS N N))
(SETQ N (PLUS N N 7))
```

sets N to 3, then to 6 and finally to 19. Where PLUS adds numbers, TIMES multiplies them, so

```
(TIMES 10 100)  = 1000
```

In normal written notation the character '-' is used in two ways: as a prefix for negating a single quantity and as an infix operator indicating subtraction. Compare -(3+7) with (2+3)-(1+4). In LISP these two usages are provided for by different functions. MINUS negates things, DIFFERENCE subtracts:

```
(MINUS (PLUS 3 7))
(DIFFERENCE (PLUS 2 3) (PLUS 1 4))
```

As should be expected, arbitrarily complicated expressions can be built up by the nested use of these functions. In normal mathematical notation it is necessary to learn that, for instance, 3 x 4 + 2 means (3 x 4) + 2 rather than 3 x (4 + 2). In LISP the bracketing structure is always explicit and there is never any ambiguity in how the terms in expressions should be grouped together.

Division is performed using the functions QUOTIENT and REMAINDER. It is carried out in such a way that remainders are always kept positive. For positive divisors and dividends this corresponds to truncating any inexact quotients towards zero.

To make counting programs a little more legible, LISP provides functions ADD1 and SUB1 which respectively increment and decrement their arguments. These are exactly equivalent to suitable calls to PLUS and DIFFERENCE, but save some typing and a small amount of time.

In LISP any variable can hold either a list, or an identifier or a number. Sometimes it is useful to be able to tell into which of these classes a value falls.

The function ATOM distinguishes between lists (which are not atoms) and all other classes of objects. Within the atoms, the function NUMBERP can recognise numbers. Further predicates provide for other tests on numbers. Appendix A explains EQ, GREATERP, LESSP, MINUSP, ONEP and ZEROP.

As a simple example of arithmetic in use, here is a function which updates the record of someone's age in a database:

```
(DEFUN BIRTHDAY (PERSON)
   (PUT PERSON 'AGE
      (ADD1 (GET PERSON 'AGE)))
   (LIST 'HAPPY 'BIRTHDAY PERSON))
```

On this micro-based LISP no great effort has gone into making arithmetic either super-flexible or especially efficient. Most of LISP's work is done using symbolic rather than numeric data. It may be useful to note, however, that large-scale modern LISPs can (if suitably directed) compile numeric code as efficiently as almost any more conventional-looking language.

One of the example packages given in chapter 23 shows how arbitrary precision arithmetic can be built on top of the base which has been described here. One of LISP's strengths is that if extensive use was going to be made of this extended arithmetic it would be possible to redefine PLUS, TIMES, NUMBERP and all the other relevant functions to use long arithmetic rather than the built-in variety. The effect would then be exactly as if the basic LISP had been provided with arbitrary precision arithmetic from the start. Similarly, floating point or complex arithmetic can be simulated (somewhat slowly!) in terms of what exists and then integrated so fully into the system that only performance marks them out as being user- rather than system-provided.

10 Tests and comparisons

The facility of being able to define a function, so that by means of a simple call, structured information may be built, stored or retrieved, or arithmetic performed, has already been demonstrated. However, far more interesting programs can be produced once functions are written capable of using some 'intelligence' - considering their argument data and making decisions about what to do next on the basis of their findings. In other words, the next thing we need in LISP is the ability to make tests on and comparisons between data.

In fact some of the tests available have already been mentioned: EQ, ATOM and NULL. These functions returned values which represent the Boolean quantities 'true' and 'false'. In LISP, NIL is used to stand for 'false' (as well as for the empty list), and where truth values are required anything non-NIL will be treated as 'true'. Conventionally the identifier T is used, and this variable initially has a non-NIL value (in fact 'T).

Also the variable F initially has the value NIL so that T and F evaluate to quantities that are 'true' and 'false'.

LISP functions that are thought of as returning truth values are known as predicates. Their main use is in conjunction with conditional expressions built up using the special function COND. The general form of such expressions is

```
(COND
   (predicate1 expression1)
   (predicate2 expression2)
   ...
   (T final-expression))
```

There can be as many (or as few) predicate-expression pairs as required. When LISP finds a COND expression it starts evaluating the given predicates in order. The value it returns is that of the expression which is paired with the first predicate to yield a 'true' (i.e. non-NIL) value.

In the example above the constant T is used as a final predicate. Since this always has the value 'true' the corresponding final-expression can be seen as a default result to be returned if all of the other predicates return 'false'. Thus, expressed without the brackets, the conditional form can be read as

```
IF      predicate1 is true
THEN    return the value of expression1
ELSE IF predicate2 is true
THEN    return the value of expression2
...
ELSE    if none of the above predicates were non-NIL
        return the value of the final-expression
```

A simple example of conditionals in action is found when defining an absolute value function. This will test the sign of its argument. Negative inputs will be negated to make them positive, while positive ones can be returned unaltered:

```
(DEFUN ABS (NUMBER) (COND
   ((MINUSP NUMBER) (MINUS NUMBER))
   (T NUMBER)))
```

Here predicate1 is (MINUSP NUMBER), expression1 is (MINUS NUMBER) and final-result is just NUMBER. The bracket structure may be easier to come to grips with if the skeleton of the conditional is written out in one colour, the predicates in another and the result expressions in a third!

In previous chapters there are several examples based on the use of PUT and GET to keep things on property lists. If GET is used in an attempt to retrieve a property which has never been established it returns

NIL, and since that is LISP's representation of false it is easy to test for. This leads to program fragments of the form

```
...
(COND
   ((GET NAME 'ADDRESS)
      (LIST 'ADDRESS 'IS (GET NAME 'ADDRESS)))
      (T '(ADDRESS UNKNOWN)))
...
```

It is sometimes more convenient to format code so that the case where a property is missing is dealt with first. LISP provides a function NOT which can be used to reverse the sense of a predicate. For any expressions p,q and r the following two are equivalent:

```
(COND (p q)                (COND ((NOT p) r)
      (T r))                     (T q))
```

NOT can, of course, be used anywhere in a LISP program, not just inside conditionals. From the rules given above for how LISP represents Boolean values, it can be deduced that (NOT NIL) will be treated as true, while (NOT anything-but-NIL) is false. Thus as well as its use for Boolean negation, NOT can act as a function that recognises NIL, i.e. empty lists. LISP provides a function called NULL that tests to see if its argument is an empty list: it should now be clear that NULL and NOT have exactly the same behaviour.

When LISP requires composite predicates it can build them up using AND and OR. Each of these functions can cope with an arbitrary number of arguments. They evaluate these arguments one at a time until the normal rules of logic allow them to deliver a result. Thus AND will evaluate arguments until it reaches the end of the list (and returns T) or until it finds that one of its arguments evaluates to NIL (false). OR will stop evaluating and return T as soon as it finds a non-NIL argument. Pieces of LISP code which seem to require complicated nestings of COND, AND, OR and NOT are usually best re-written as groups of smaller, simpler functions.

Scattered through this manual, and collected in Appendix A, are a collection of additional predicates known to LISP. These range from ones like ATOM which can test when an expression is atomic (rather than being a list structure), through EQ (the primitive equality test) and the arithmetic tests MINUSP and GREATERP, to items at the level of EOF which tests if an input file has reached its end.

The full form of COND allows a sequence of consequents to be selected by each predicate, as in

```
(COND
   (predicate1 r1a r1b r1c ... r1z)
   (predicate2 ... )
   ...
```

If this form is used and predicate1 is non-NIL, each of r1a through r1z will be evaluated in turn and the value delivered by the last one will be the result handed back by COND. This can be very useful if some of the consequents involve printing or the use of PUT or SETQ. Almost all the examples in later chapters of this manual will take advantage of this extended format.

11 Styles of LISP programming

The collection of LISP facilities introduced so far does not look particularly extensive or powerful. These facilities are, however, sufficient to support any computation performed by any computer. Indeed, even with just the functions ATOM, CAR, CDR, COND, CONS, EQ and DEFUN, LISP is a completely general-purpose language. This will be illustrated later in this manual as more advanced LISP functions are defined in terms of these elementary ones. There are important practical implications. For instance, many of the functions you find built into mainframe LISPs can be provided on even a micro-based system by constructing towers of definitions on top of the facilities initially available.

More surprisingly perhaps, experience within the LISP community shows that not only are restricted forms of LISP general-purpose in a theoretical sense, but they are convenient to use. This section contains some thoughts on LISP style, based on the avoidance of any facilities which go substantially beyond those introduced so far. This will involve code being developed in the form of many very small functions. The use of PUT and SETQ will be restricted to the initial setting up of datastructures for these functions to inspect.

If the use of PUT and SETQ is restricted in this way the behaviour of LISP functions can depend only on the arguments explicitly passed to them. It cannot be influenced by the side-effects of functions previously executed. A major advantage of this method of organising LISP code is that each function can be read in isolation from all others. This aids both designing and debugging, because it means that functions can be written, tested and corrected one at a time before they are integrated to form a complete package.

Alternatively we can view these types of LISP programs as schemes for transforming argument data into results. Then each individual function will be designed to make some definite progress in the transformation process, while being as simple as possible. The entire conversion is then made up of a long series of almost trivial steps, each of which will be very easy to understand.

Although, of course, any of the functions which make up a complete program can be called by the user, normally one (or a very small number) will be viewed as entry points to the package. The rest will then tend to fall into layers ranging from ones carrying out low-level manipulation of datastructures to ones concerned with the overall logic of the problem being tackled. When starting a fresh LISP project it can be useful to begin by planning this layer structure.

'Bottom-up programming' consists of starting with the layers closest to the facilities LISP already provides. It amounts to extending LISP until solving the given problem becomes a trivial process. At the other extreme comes 'top-down programming', where the first functions designed and coded are the ones which will be entrypoints to the package being written.

Since LISP functions can be called before all their subfunctions have been defined (an error message gets generated when one of the missing routines is required) LISP can provide a comfortable environment for interactive top-down programming. Because it allows for immediate testing of small pieces of code it is well suited to bottom-up development too! Of these two development strategies top-down programming has a particular advantage. If it leads to a completed program at all it will produce one which has the originally specified user-interface and which solves the problem initially posed. Each function produced will either be expected to deal with some simple case completely, or to split the problem being solved into two or more component problems. Each of these will get passed to lower levels of code. By calling the top level procedures with fairly simple arguments a partially completed design of this sort can be exercised - the first undefined function to be called gives a natural pointer to the region in which the next bit of programming is required. It is frequently useful to write 'stubs' of code to hold the place of some of these functions. A stub is a dummy function, which at some time will be replaced by a proper, fully-defined one, and which hands back some value in order to allow testing to proceed in an incomplete program. In LISP one could have a generalised stub which printed its own name and argument and required the user to enter the result to be returned:

```
(DEFUN GENERAL-STUB (A1 A2)
   (PRINT 'GENERAL-STUB A1 A2 '?)
   (READ))
```

The example programs included later in this book show a mixture of top-down and bottom-up design. They generally start with items which (and it is readily apparent) are necessary for any attack on the given problem, followed by a top-down presentation of the main logic of the solution.

12 The design of simple LISP functions

A surprisingly large proportion of the functions one writes in LISP are variants of just a few basic structures. In this chapter some of these structures are presented, and useful LISP operations implemented using them. The first set of examples will be built on the following pattern:

```
(DEFUN name (arguments) (COND
   (condition something very easy)
   (T further call to name using slightly modified
   arguments)))
```

In each case the condition checked will be a test that the arguments have a particularly simple form, indeed, quite often it will be a test for the empty list.

Consider the problem of taking two lists and concatenating them. In LISP the function which performs this operation is usually called APPEND. If one of the lists concerned is empty APPEND does not have very much work to do. You can exploit this at the start of the definition by writing

```
(DEFUN APPEND (A B) (COND
   ((NULL A) B)
   ...
```

This is certainly in accord with the pattern given earlier. If APPEND's first argument is not the empty list we can try to build up the complete result we need using the fact that it is a list with at least one item in it. This means that it will be appropriate to make use of the dismantling of A into (CAR A) and (CDR A). Assuming for the moment that a working version of APPEND already exists, observe that

```
(APPEND (CDR A) B)
```

will construct just the concatenated list required except that the first element (i.e. the one that is (CAR A)) will be missing. This missing element can be replaced using CONS to give a complete definition of the form:

```
(DEFUN APPEND (A B) (COND
   ((NULL A) B)
   (T (CONS (CAR A)
            (APPEND (CDR A) B)))))
```

where there are just enough final brackets so that the final closing one matches with the opening one before the DEFUN. This definition can be tried out on paper before it is put into the machine. Consider the call

```
(APPEND '(P Q R) '(X Y Z))
```

A test is made on the first argument to see whether it is the empty list - it is not. So APPEND goes on to the final clause with

```
A = (P Q R)
B = (X Y Z)
```

It calls itself again with arguments (CDR A) and B, which will have an effect similar to that of the top level call

```
(APPEND '(Q R) '(X Y Z))
```

This is a simpler case than the original one, because the length of the first argument to APPEND has been reduced. It will yield the list (Q R X Y Z) as its value. Then the outer version of APPEND CONSes the value of (CAR A) (i.e. P) onto this to give the final result (P Q R X Y Z).

The recursion in this definition of APPEND (i.e. the way in which APPEND calls itself) does not result in never-ending chains of calls to the function, because successive calls always involve shorter lists as their first arguments. Eventually this length reduction will lead to a call where APPEND receives an empty first argument, and so the test (NULL A) will succeed and no further calls will be made. As each of the nested calls to APPEND is completed the required list is reconstructed using CONS until it is finally returned to the user. This process may be made clearer by following through the steps involved in the example call, where each succeeding line shows a transformation of an initial call to APPEND into LISP input that would compute the same value.

```
   (APPEND '(P Q R) '(X Y Z))
-> (CONS 'P (APPEND '(Q R) '(X Y Z)))
-> (CONS 'P (CONS 'Q (APPEND '(R) '(X Y Z))))
-> (CONS 'P (CONS 'Q (CONS 'R (APPEND '() '(X Y Z)))))
-> (CONS 'P (CONS 'Q (CONS 'R '(X Y Z))))
-> (CONS 'P (CONS 'Q '(R X Y Z)))
-> (CONS 'P '(Q R X Y Z))
-> '(P Q R X Y Z)
```

Now consider another useful LISP function, MEMBER. This will test to see whether a given item is one of the members of a list, for instance

```
(MEMBER 'B '(A B C))   = T
(MEMBER 'Z '(A B C))   = NIL
```

Observe that if the second argument to MEMBER is the empty list the function must return NIL, so its definition can start off as

```
(DEFUN MEMBER (A L) (COND
   ((NULL L) NIL)
   ...
```

This is a pattern of code very similar to that used in APPEND. Another case which can be detected easily is when A is the first element of L and MEMBER can immediately return T:

```
...
((EQ A (CAR L)) T)
...
```

In the final line of the definition there will be a further call to MEMBER with slightly altered arguments. In this case the first argument remains unchanged, but the second should be (CDR L) to indicate a search through the tail of the list. Thus the full definition of MEMBER is

```
(DEFUN MEMBER (A L) (COND
   ((NULL L) NIL)
   ((EQ A (CAR L)) T)
   (T (MEMBER A (CDR L)))))
```

and a trace similar to the one given for APPEND shows

```
    (MEMBER 'C '(A B C D))
->  (MEMBER 'C '(B C D))
->  (MEMBER 'C '(C D))
->  T

    (MEMBER 'C '(A B))
->  (MEMBER 'C '(B))
->  (MEMBER 'C NIL)
->  NIL
```

Minor variations on this pattern of recursion provide simple LISP functions for such diverse problems as computing factorials, finding the last item in a list and deleting items from the middle of lists. The following general points may be noted:

(a) It is often useful to write a function on the supposition that it already exists and can be used to deal with sub-cases of the original problem;

(b) The escape conditions at the top of a function definition can test for very simple cases;

(c) Quite complicated effects can be achieved without the use of anything more elaborate than function calls and conditionals. In particular, LISP does not strictly need either the function SETQ to update variables, or any built-in looping construction;

(d) The behaviour of a function or group of functions can be predicted or documented by tabulating the arguments and results from each nested call.

13 Debugging LISP functions

When you first write a set of LISP functions and try them out, you are, by the nature of computing, certain to have errors in what you have done. LISP will respond either by displaying incorrect and possibly ridiculous answers or by generating an error message followed by an obscure-looking collection of expressions (known as a backtrace) which are supposed to help you track down where the error occurred. In each case it is easy to get the impression that you have lost control, and that there is no way of discovering which error was responsible for the chaos that you see! When programming in LISP this pessimism is generally unreasonable - there are a number of reasons to expect debugging in LISP to be easier than debugging in almost any other language.

The first thing to do on encountering an error is to verify that it really is an error and not just unexpected but proper behaviour of your program. This may seem a silly thought, but it can help you decide whether you need to redesign your code to different specifications or simply to bring it into line with the specifications that you have. While considering this issue it will be useful to make sure that the error you have found can be provoked repeatedly; things which cannot may have been side effects of typing errors when you invoked your functions.

Exhibiting an error in definite, repeatable form is central to debugging whatever language you use. LISP's strengths show up during the process of isolating the error to one part of your code. The key to LISP's success here is that, with isolated exceptions, all the quantities ever passed to LISP functions and all the values that can ever get returned as results can be read and printed.

The only time this is not quite true is where problems occur with function entry-points and with re-entrant structures. The principle is nevertheless valid in most circumstances.

This can be exploited to allow one to home in from the initial (often large and ugly) manifestation of an error to a simple call to a single function which misbehaves. To start with, then, it is reasonable to ignore the fine details of LISP's backtraces and just try to spot in them some of the names of your functions in amongst the mess. Then these functions can be expunged one at a time by invoking them directly from the keyboard, giving them first trivial arguments, then progressively more elaborate ones until the error re-surfaces.

The function TRACE (see Appendix B) can be used to see how values get passed between functions in a complex calculation. Again, the object of using it is to isolate a single function returning incorrect results, and to exhibit arguments which lead it to this bad behaviour. If the bad function is as short as most well-written LISP ones, it should be straightforward to follow through its logic and discover either how its design was inadequate or where its behaviour does not correspond to your expectations. In the first case you will just need to construct a corrected version of the function taking account of your new knowledge of how it must behave. The second case is typical of misapprehensions on how LISP is supposed to behave. Breaking free of these can be hard. Perhaps the best suggestion that can be made is for the basic built-in LISP functions called by your code to be invoked from the keyboard to verify that they behave as required. Also the structure of your function (particularly its brackets) should be compared with that in existing working code - for example, the programs given in chapter 23.

When debugging, it is frequently useful to include print statements in some of your functions so that they report when they are called and what they are doing. It is possible for the output generated in this way to be made both more concise and more informative than that provided by the general-purpose TRACE facility. One good aim when putting in diagnostic printing is to make the trace output such that it would be intelligible to anyone to whom you were explaining your program. This involves thought, in that readable traces must be kept short. It also involves taking the trouble to print words of explanation between the raw LISP datastructures.

Finally if the messages that you generate for yourself cannot help it may be necessary to attempt complete deciphering of LISP's own diagnostics, i.e. backtraces. Even in LISP, where backtraces can be fairly informative, this should be viewed as a last resort. A backtrace is a display which shows exactly what the LISP interpreter was processing at the time that it noticed that something was wrong. At the top it explains what error provoked it.

In some versions of Acornsoft LISP errors are only identified by numeric codes. Appendix C lists these and gives some explanation of typical circumstances which provoke them. Then the backtrace lists the names of functions and values of variables most closely involved in the error. For instance, when an attempt is made to take CAR or CDR of an atom the offending atom will be displayed. Further down the backtrace shows which functions were in the process of being called when the error occurred. Finally there will usually be what amounts to a reflection of the request originally presented to LISP.

Large backtraces where deep recursion is combined with large arguments to functions are tedious to decipher - when chasing a bug always start by trying to find a compact manifestation of it.

14 Correcting LISP functions

The most obvious way of correcting erroneous LISP programs is by the redefinition of complete functions. Since typical LISP functions are quite short the amount of typing this involves is usually tolerable! On the BBC Microcomputer the cursor control keys and COPY can greatly increase the speed and reliability of transcribing those parts of a definition which do not need changing. In this context it is useful to know that defining a function (in Acornsoft LISP) is almost exactly the same as setting a variable. After defining a function, FN for instance, the definition can be accessed as the value of the variable FN. In particular, it can be displayed on the screen just by typing the name FN in response to the 'Evaluate:' prompt.

The use of cursor-based editing is available in almost any language implemented on the BBC Microcomputer. LISP naturally supports another style of working, based on the fact that LISP programs can be treated as datastructures. The use of LISP code to inspect and correct other pieces of LISP code is known as structure editing. The simplest example of its use is in the correction of simple typing or spelling mistakes. Imagine that a function called STRING-TOGETHER has been defined; it was supposed to call APPEND but by accident in one of the calls a letter was missed out, so that the word APPND appeared. The definition can be corrected by replacing APPND by APPEND wherever it occurs. A suitable function to perform such substitutions is SUBST, defined as

```
(DEFUN SUBST (A B S) (COND
   ((EQ S B) A)
   ((ATOM S) S)
   (T (CONS (SUBST A B (CAR S))
            (SUBST A B (CDR S))))))
```

This substitutes A for B wherever it appears in the datastructure S. Note that this definition follows the general pattern explained in chapter 12. The correction required can now be effected by

```
(SETQ STRING-TOGETHER
   (SUBST 'APPEND 'APPND STRING-TOGETHER))
```

For more elaborate corrections, and for that matter for program writing, it can be useful to define a LISP editor which is in effect a version of SUBST, but one which changes its argument on the basis of a continuous string of directives provided by the user. Thus rather than traversing the whole of a function definition making systematic changes, this editor will use READ to discover what alterations the user requires, PRINT to show what it has done, and a recursive structure like the one in SUBST to allow it to create a modified version of its input. An editor of this type is presented as one of the sample programs in chapter 23.

If there is not enough room in the computer for a tidy structure editor, some of the same effects can be achieved through the direct use of CAR, CDR, RPLACA and RPLACD. Imagine, for instance, that in the definition of SUBST given above the word CONS had been mistyped as COND. There would be two reasons why the error could not be repaired by use of

```
(SETQ SUBST (SUBST 'CONS 'COND SUBST))
```

First, SUBST cannot be used effectively while it still contains an error; secondly, even if it did work, it would change both the instances of COND into CONS, and only one needs altering.

The required change could be achieved (given a cool head!) as follows:

```
(SETQ W SUBST)
```

This sets W to have as its value the definition of SUBST, which is displayed as

```
          (LAMBDA (A B S) (COND ((EQ S
B) A) ((ATOM S) S) (T (COND (SUBST A B (
CAR S)) (SUBST A B (CDR S))))))
```

The word LAMBDA is just a marker indicating that this datastructure represents a function definition. A series of applications of CAR and CDR will move W down to refer to the component of SUBST needing adjustment:

```
(SETQ W (CDDDR (CADDR W)))
```

This leaves W with the value

```
((T (COND (SUBST A B (CAR S)) (SUBST A B (CDR S)))))
```

and then (SETQ W (CADAR W)) gets to the structure starting with (COND. Anyone lacking the confidence to leap directly to CDDDR of CADDR as the selector function used here can experiment at each stage by displaying (CAR W) and (CDR W) and then setting W to whichever of the two still contains the required component.

With W pointing at the piece of SUBST's definition needing correcting, a call

```
(RPLACA W 'CONS)
```

overwrites the pointer that led to COND with one to CONS. Inspection of the definition of SUBST will show that the adjustment is permanent, and not just on some copy of SUBST's definition. The technique is clearly primitive, but very powerful!

At least one large LISP system exploits the language's ability to inspect and change function definitions by allowing its error handler to propose and make changes to the user's code automatically. Ingenious people who might otherwise have to immerse themselves in machine code can try to design LISP functions which rely on making dynamic changes to themselves, while those concerned with human interfaces can work on the ultimate LISP editor coded (of course) in LISP.

15 Input and output in LISP

For a great many LISP applications only the most rudimentary input and output facilities are required. Indeed, effects which in other languages would involve reading and printing can very often be achieved in LISP simply by presenting a LISP function with some argument and letting the system display the structure it returns. There are, however, cases where it is useful to communicate with the user from within a piece of code. The simplest way of doing this is to use READ and PRINT which are just user-level entry points to the code LISP employs to read commands and print results. If READ is called without any arguments, i.e. by writing (READ), it waits until an expression has been typed at the keyboard and returns that expression as its value. As will be explained later, READ can be given an argument and used to read expressions from files rather than from the keyboard. READ ignores initial blanks and newlines; and can cope with numbers, identifiers and bracketed lists. It will generate an error message if it finds misplaced dots or righthand brackets.

PRINT can take several arguments. It displays these on the screen one after the other (with no intervening blanks) and then puts out a carriage return so that output from the next call to PRINT will appear on a fresh line.

There are, in fact, four closely related printing functions: PRIN, PRINC, PRINT and PRINTC. All of these display their arguments, and can deal with both atoms and lists. PRINT and PRINTC put out a carriage return at the end of their output, while PRIN and PRINC do not. PRIN and PRINT are distinguished from PRINC and PRINTC by their treatment of atoms containing punctuation characters. The differences will be explained here in terms of PRIN and PRINC, but PRINT and PRINTC behave similarly with just the addition of a final carriage return.

When PRINC displays an atom it simply prints the characters which make up the atom's name. For instance, the variables BLANK, LPAR and RPAR, have as their initial values identifiers the names of which are left and righthand brackets and a blank. Then (PRINC LPAR BLANK RPAR) displays the text '( )'. Note that if this were to be stored in a file and later read in, the brackets would be taken to indicate a list structure and the blank would be ignored. To get output which can be read back in again, one uses PRIN. (PRIN LPAR BLANK RPAR) prints the text '!(! !)', where each punctuation character is preceded by an exclamation mark. The LISP reader recognises exclamation marks as escape symbols, and builds atoms as if characters following !s were letters.

To summarise: PRINC is useful for printing messages, while PRIN can reduce confusion if there is any chance of having unusually-spelt atoms in use. To get these unusual atoms in the first place, either use predefined variables such as LPAR, or precede non-alphabetic characters in the atoms' names with exclamation marks. For instance, the variables LPAR and RPAR could have been set up by issuing the commands:

```
(SETQ LPAR '!()
(SETQ RPAR '!))
```

On the BBC Microcomputer it is sometimes useful to be able to generate a series of control codes for sending to the screen driver. In LISP this is done using the function VDU. VDU takes numeric arguments and sends them to the operating system. If they are codes for normal characters then those characters will appear on the screen, and if they are control codes they can cause the screen to change colour, plot pictures or draw double-height characters. Details of the available codes can be found in the BBC Microcomputer User Guide.

Individual character input is provided by GETCHAR. Its value is always an atom with a single character name, that character being the next one obtained from the input device. READLINE reads characters until it finds a carriage return and makes everything it finds into a single identifier. In Acornsoft LISP READ only allows for one expression per line of input, and so GETCHAR or READLINE must be used if user input is to have several words per line not enclosed in brackets.

It was indicated earlier that the read routines could collect their input from files rather than from the keyboard. Before files can be used they must be opened, and after they have been used but before LISP is left they should be closed. LISP's file-handling facilities are most useful in association with a disk- or Econet ® filing system, and will be described in relation to these. The functions will work with cassette provided that tapes are started, stopped and rewound at appropriate times.

OPEN takes the name of a file as argument, and locates that file on disk. It returns a value which is known as a 'file handle'. (OPEN name T) opens the named file for input, and complains if it does not already exist. (OPEN name NIL) opens the file for output, creating it if necessary.

Optionally, the input functions READ, GETCHAR and READLINE can all be given the handle of an input file as an argument from which they then read.

Functions WRITE and WRITE0 are versions of PRINT and PRIN which expect their first argument to be the handle of an output file. They print the rest of their arguments to that file. The predicate (EOF handle) can test if an input file is positioned at its end. (Note: EOF stands for End Of File.) When the input or output to the file is completed, it is necessary to call (CLOSE handle). If this is not done some of the data being sent to an output file may be lost. The operating system on BBC Microcomputers allows only a limited number of files to be open at once, so closing files as early as possible is recommended.

LISP provides operations which can be regarded as read and print facilities where files are represented as lists of characters. EXPLODE takes an identifier as argument and returns a list of the constituent characters, so that (EXPLODE 'EXPLODE) has the value (E X P L O D E). IMPLODE does the converse: given a list of characters (e.g. read one at a time using GETCHAR) it makes them into a single identifier. In this LISP, EXPLODE will work only on identifiers, not on numbers or lists. Similarly, IMPLODE creates identifiers regardless of the sequence of characters it is given, even if it includes digits or brackets.

For conversion between character codes and LISP identifiers the functions CHARACTER and ORDINAL are available. CHARS returns the number of characters which would be used when PRINCing its identifier argument.

The prettyprinter shown in section 23.5 displays most of the print functions in action together. It is worth noting that PRINT with no arguments can be used to generate blank lines. When using PRINC with multiple arguments it will often be useful to put in extra blanks, as in

```
(PRINC 'X= BLANK X)
```

Classifying characters as letters, digits and so on is best done using ORDINAL. For instance, a test for a letter can be coded as

```
(DEFUN LETTER (CH)
   (SETQ CH (ORDINAL CH))
   (AND (GREATERP CH 64)
        (LESSP CH 91)))
```

This relies on the fact that the internal codes for A to Z are 65 to 90.

The use of files via OPEN and CLOSE can often be circumvented by use of the operating system commands SPOOL and EXEC - see the function '*' in Appendix A for information on how to invoke them.

16 Optional arguments and local variables

Most LISP functions take a fixed number of operands and complain if attempts are made to call them with either excess or insufficient arguments. This chapter is concerned with writing functions where this constraint is relaxed. It also shows how LISP can introduce working variables which will be updated and referenced within some function, but will not exist outside it. The arrangements described are specific to the Acornsoft LISP system - other LISPs may well not provide for optional arguments and will introduce local variables using a construction called PROG.

To specify that a function will be prepared to accept less than its full complement of arguments the final few parameter names in the call to DEFUN should be enclosed in brackets:

```
(DEFUN FNAME (NEEDED (OPTIONAL 1) (OPTIONAL 2))
   ...
```

Only trailing arguments can be optional. In the example above FNAME becomes a function which will accept from one to three arguments. If it is given just one, then OPTIONAL1 and OPTIONAL2 will get bound to the default value NIL. If it is given two arguments they will specify values for NEEDED and OPTIONAL1, and OPTIONAL2 will default to NIL. A call with three arguments gives explicit values to all three parameters.

If a function header is written like this, but the function is then systematically called only with a minimal number of arguments the extra (optional) arguments can be viewed as local variables. They can be set and reset within the body of the function. Any optional argument can be supplied with a default value. Expressing the same thing in a different way, any local variable can be given an initial value. The declaration format shown above is just the special case where these default values are all NIL. The full specification of an argument list is that any item in it that is atomic indicates an argument that will be required. Items that are themselves lists show where optional arguments can occur, and provide default values for these arguments. In the list that marks an optional argument the CAR is the parameter name and the CDR the default value. Thus,

```
(DEFUN TAB ((N . 8))
   (LOOP
      (UNTIL (MINUSP (SETQ N (SUB1 N))))
      (PRINC BLANK)))
```

is a function with one optional parameter, N, and a default value of 8. A call (TAB) will, then, print 8 blanks, but by giving TAB a numeric argument it can be induced to print any other number of spaces.

This scheme of optional arguments allows user-defined functions to behave like, for instance, READ where a single argument (or perhaps a few) can be missed out. It does not allow the user to construct things like PLUS and LIST which accept a fully flexible number of arguments (see chapter 18).

17 Loops

In programs it is often useful to perform the same sequence of operations repeatedly until some terminating condition is satisfied. In LISP this is frequently arranged by making the actions part of the definition of some function, and the repetition is achieved by allowing the function to call itself. As an alternative to this recursive style of coding all LISP systems provide some direct way of writing loops. The support for repetition built into Acornsoft LISP is just one of a variety of schemes to have arisen. While function definitions and calls can be transferred easily from one dialect of LISP to another, the constructions described in this chapter will not be generally available in this exact form elsewhere. To compensate for this, they are clean, efficient and simple to use.

Repetition is specified using a function called LOOP. LOOP takes a series of expressions as arguments. It evaluates them, in order, repeatedly. Thus the regular LISP read-eval-print loop can be approximated by:

```
(LOOP
   (PRINC 'Evaluate:)
   (SETQ U (READ))
   (SETQ U (EVAL U))
   (PRINC 'Value:)
   (PRINT U))
```

This prints a prompt, reads an expression, evaluates it, prints a message and then the value, performing this cycle of operations until an error interrupts it.

Perhaps the ultimate use of LOOP is to write just

```
(LOOP)
```

which directs LISP to do nothing, but to do it over and over again, without end. The result will be that LISP goes very quiet! Indeed, on some versions of Acornsoft LISP the only way of escaping from this very tight loop is to press BREAK, thereby restarting LISP completely.

To produce loops which can terminate, the functions WHILE and UNTIL can be used. They are usually written directly within a loop body: the effect of using them elsewhere will be noted later.

In their simplest form both WHILE and UNTIL just specify exit conditions for the LOOP which encloses them. WHILE takes a predicate as argument, and the loop will terminate as soon as this evaluates to NIL (i.e. 'false'). UNTIL is similar but the loop exit occurs when the predicate becomes non-NIL. Thus for any predicate p the loop exits caused by (WHILE p) are the same as those from (UNTIL (NOT p)). Conversely, (UNTIL p) behaves rather like (WHILE (NOT p)). If there is only one instance of WHILE or UNTIL in a loop the statements above can be read as stating that WHILE allows the loop to proceed for so long as its predicate yields 'true', whereas UNTIL causes loop exit on a 'true' predicate.

When a loop is completed, the result returned is the value of the last expression evaluated. This will either be or will contain a WHILE or an UNTIL. Each of the exit functions can be provided with any number of expressions following the initial predicate. When evaluation of the predicate indicates that the loop is to be left these expressions get evaluated. The last of them is returned as the value of the WHILE (or UNTIL), and so typically gets returned as the value of the enclosing LOOP. If there are no expressions following the predicate the predicate value itself gets returned.

An example of this can be seen in the following definition of a function which finds the last element of a list:

```
(DEFUN LAST (L)
   (LOOP
      (UNTIL (NULL (CDR L)) (CAR L))
      (SETQ L (CDR L))))
```

The loop repeatedly obeys (SETQ L (CDR L)) to chain down a list, stopping when the predicate (NULL (CDR L)) is satisfied, and then returning (CAR L).

Exit statements can be placed anywhere in a loop, as the first, last or any intermediate place in the list of repeated expressions.

It is quite in order to have both WHILE and UNTIL clauses present, with several of each if that is useful. The loop terminates on the first occasion that any of the clauses indicate that it should, and the value returned is (as with simple exits) just that of the last expression processed. Thus the following LOOP expression scans down a list L, removing any initial positive numbers. If before finding a negative number it finds something non-numeric or reaches the end of the list then it returns an identifier indicating that fact. Otherwise, it hands back the tail of the list starting with the first negative number.

```
(LOOP
   (UNTIL (NULL L) 'NONE-LEFT)
   (WHILE (NUMBERP (CAR L)) 'NON-NUMERIC)
   (UNTIL (MINUSP (CAR L)) L)
   (SETQ L (CDR L)))
```

This is probably not a useful loop, but it does illustrate several uses of UNTIL and WHILE all active at once.

LOOPs can, of course, be nested to any depth required. WHILE and UNTIL each influence only the LOOP immediately enclosing them.

Any piece of LISP which can be expressed using LOOP can also be written without it by providing more function calls. Recursion gives the user enormously greater flexibility in the control of a program, and frequently leads to programs which are both shorter and clearer. Therefore it makes sense to reserve the use of LOOP for circumstances which directly and easily accommodate its structure.

18 Functions which do not evaluate their arguments

A number of LISP functions mentioned so far have been unusual in the way in which they process their arguments. COND, SETQ, DEFUN and LOOP do not obey LISP's normal rule that all arguments get evaluated before any function is called. The quote mark which has been used to introduce data comes in the same category. The form 'x is just a shorthand form, recognised by the LISP reader, for (QUOTE x), and the whole point of the QUOTE function is that its argument is not to be treated as executable LISP code. These built-in functions can be supplemented by user-written special functions. In Acornsoft LISP such functions are indicated by writing a non-NIL identifier in the place of the normal list of parameters in DEFUN. When the function is invoked this variable will be bound to the complete, unevaluated argument list found in the call. Thus if the only purpose of the function is to inhibit evaluation of a fixed number of arguments it will have to use CAR, CADR and so on to separate the arguments it was given. This can be illustrated by showing how the QUOTE function could have been defined if it did not already exist:

```
(DEFUN QUOTE X (CAR X))
```

In cases like SETQ, where only some of the arguments have to be kept unevaluated, it is necessary to call EVAL explicitly to evaluate the others. Here is SETQ defined in terms of SET:

```
(DEFUN SETQ X
   (SET (CAR X) (EVAL (CADR X))))
```

One fairly common use for functions with unevaluated arguments is to support arbitrary numbers of arguments, as found with the system functions TIMES and PRINT. In these cases it will be necessary to call EVAL on all members of the argument list. If LISP had only been provided with a function PRIN1 which could print just one expression at a time, PRIN could have been coded as

```
(DEFUN PRIN X
   (LOOP
      (WHILE X)
      (PRIN1 (EVAL (CAR X)))
      (SETQ X (CDR X))))
```

However it should be noted that this definition interleaves evaluation and printing, while the system provided completes evaluation of the arguments before doing any printing. This can be relevant if any of the actual arguments in a call to PRIN themselves involve printing.

The introduction of large numbers of user-defined special functions is viewed with some suspicion in the LISP world. This is partly because the need to call EVAL directly provides opportunity for confusion, and partly because the compilers in large-scale LISP implementations cannot generate efficient code for these functions. It is probably worth considering the use of unevaluated arguments for a few small functions which represent entry points into code (e.g. EDIT in section 23.8), otherwise they should be used sparingly.

19 Functions as objects

This chapter introduces the idea that, in LISP, functions can be passed as arguments to other functions, and they can also be returned as results. Such uses of functions should probably be viewed as being advanced applications of LISP. Most LISP programmers will not need to worry about all the rules given here relating to variable binding and scopes and consequently can skip this chapter at a first reading of the book.

Functional arguments are used to separate the control of LISP's recursion from the precise details of the operations being performed. For instance, it is often useful to perform a certain operation on all members of a list. This <u>can</u> be done by writing list-scanning code for each of the operations to be used. So to map initial lists like ((A B C) (1 2 3) (X Y)) into lists (A 1 X), showing the heads of all sublists, a function would be defined:

```
(DEFUN CARLIST (L) (COND
   ((NULL L) NIL)
   (T (CONS (CAAR L) (CARLIST (CDR L))))))
```

However, a more general way of achieving the same effect would be to define the single function MAPC (MAPC is, in fact, already built into Acornsoft LISP), as

```
(DEFUN MAPC (FN L) (COND
   ((NULL L) NIL)
   (T (CONS (FN (CAR L))
            (MAPC FN (CDR L))))))
```

(MAPC CAR x) can then be written in place of (CARLIST x).

MAPC can accept as its first argument either one of

LISP's built-in functions (such as CAR in the above example) or a user-defined function. Sometimes the operation to be performed is sufficiently simple for it to seem hardly sensible to define a new function for it. In this case it can be written directly in the call to MAPC using a special notation to mark it as an anonymous function. This notation introduces the function with the word LAMBDA, followed by a list of its arguments and then its body. Thus, to increment all the numbers in the list L, it would be possible to write

```
(MAPC '(LAMBDA (N) (PLUS N 1)) L)
```

Similarly to print the property lists of all atoms known to LISP it is sufficient to call

```
(MAPC '(LAMBDA (ID)
          (PRINT ID '= (PLIST ID)))
      (OBLIST))
```

It would be acceptable to write (MAPC '(LAMBDA (A) (CAR A)) x) in place of (MAPC CAR x) above - the effect would be the same but the efficiency slightly less.

Whenever a program has sets of structures with similar shapes but different stored items it is worth considering the introduction of mapping functions to capture all the necessary information on how to traverse them. In the sorting package in section 23.3, for instance, the ordering function used could be made a parameter of SORT, thereby giving a more general piece of code.

The use of functional arguments presented above involves use of the symbol LAMBDA which is part of LISP, but which is normally inserted automatically in definitions by DEFUN. If the function being passed is a simple one there are no further worries. If, however, the function uses any variables it does not itself bind there are several technical issues to be faced. These relate to the way in which these free variables get associated with bindings.

In all the examples of LISP code given so far the variables used have been either global established by SETQ before the program was entered) or declared within the function using them. LISP does not restrict the programmer solely to these extremes - any function can access variables which are bound by any other function active when it gets called. In cases where several functions have variables sharing a common name the declaration that will be in force will be the one most recently made. The following functions illustrate this, albeit with a slightly unnatural example:

```
(DEFUN ADDN (N L)
   (MAPC ADD L))

(DEFUN ADD (P) (PLUS P N))

(ADDN 3 '(1 5 7))
```

Here the final call to ADDN returns the result (4 8 10). Note that the subfunction ADD references the variable N which was bound by ADDN. Consider now the following slightly different version of the above code, and assume that MAPC had the definition given earlier:

```
(DEFUN ADDL (L N)
   (MAPC ADD N))

(DEFUN ADD (P) (PLUS P L))
```

ADD gets called from within the body of MAPC, and references the variable L. Unfortunately, MAPC has bound a variable by that name, and ADD will access that rather than the L bound by ADDL. As a result PLUS gets invoked on a non-numeric argument and the program fails. Some LISP systems provide a complete correction to this problem of name clashes via a mechanism where the user always marks functional arguments with the word FUNCTION, as in (MAPC (FUNCTION ADD) N).

FUNCTION is not available in Acornsoft LISP, and so care is needed to avoid the re-use of names when functional arguments, user-defined mapping functions and free variables are being used simultaneously. The versions of MAP and MAPC built into the LISP interpreter have been arranged so as not to introduce any visible new local variables, and therefore their use cannot cause name-clash problems. The way DEFUN is defined using a local variable X means that it cannot be used to introduce a function that is just called X.

20 Summary

All the important features of LISP have now been introduced and enough information has been given to enable the reader to construct a range of short LISP functions and to test them. To come are two chapters on the internal organisation of LISP systems (which may be skipped at a first reading), and a collection of example programs and appendices containing reference information on the Acornsoft version of LISP for the BBC Microcomputer.

To review the basic LISP functions before going on to technical details and larger examples it is probably worth exercising the LISP system at this stage. In particular, it will be useful to set up various list structures, draw them out in box notation and then work out the chains of CARs and CDRs needed to reach various parts of them. The datastructures representing the functions APPEND, SUBST and so on which have been given will be quite adequate sample lists for this work. Building on this understanding of LISP data, a few small functions should be coded, following the suggestions in chapter 12 and perhaps looking ahead to chapter 23 for inspiration.

As well as list-manipulating functions it can be useful to code things which interface with the BBC Microcomputer's screen handler - for instance by using VDU to send codes that cause output to appear in double height flashing characters. The bracket structure of LISP can cause some confusion, especially to those used to writing function calls as name(arg) rather than (name arg). Reading, copying and making small changes to the examples in the text can help with this. It is also helpful to type programs into LISP using a regular indentation convention similar to the one adopted here - it makes bracket-matching much easier.

21 Inside LISP–the use of store

It is never necessary for the ordinary programmer to worry about the details of how LISP represents datastructures in terms of the bits, bytes and words in which the underlying computer deals. This chapter is, therefore, aimed at the curious, or at those who wish to become LISP system programmers rather than just users.

The description given here is not intended as a guide to the internal workings of any one particular LISP. Although it uses the system which runs on the BBC Microcomputer as an example, issues forced on it by the curiosities of the 6502 microprocessor chip are not discussed. One of LISP's strengths is that the face it shows to the external world can be implemented in a variety of slightly-differing ways, and these system-level differences can be kept absolutely invisible to the user. In view of this, the proprietors of any LISP system can quite properly reserve the right to change the internals of their system (perhaps radically) at any stage. Accordingly, details of store layout discovered by piecing together information from this chapter and dismantling of Acornsoft's code should not be viewed as part of the official specification of Acornsoft LISP for the BBC Microcomputer, and reliance on consistency between versions of this LISP may be ill-advised.

As noted much earlier, LISP works in terms of pointers. A pointer is kept as the address in memory of the object pointed at, and therefore on the BBC Microcomputer it takes up two bytes. It is then obvious that a CONS-cell (which contains two pointers) will be kept as a block of four bytes, with the first two containing the CAR pointer and the other two the CDR. This scheme would be all very well if the only things LISP ever had to deal with were CONS-cells, in which case every pointer would refer to one of these four byte blocks. To allow for atoms it is necessary to provide extra information with each pointer to indicate whether it is a pointer to a CONS cell, a pointer to an identifier, the address of a segment of machine code or just a number. On some machines the wordlength is sufficient for this information to be packed in with the pointer. On the BBC Microcomputer an extra byte is kept with each pair of words, and bits in it show how the 16-bit CAR and CDR fields have to be interpreted.

Identifiers start off as a block of five bytes very much like normal CONS cells. Conveniently CAR and CDR fields hold the current value of the identifier and its property list. Following on in memory will be a string of characters making up the name of the identifier. By keeping all identifiers in a compact block at one end of memory LISP is able to find the structure corresponding to any name it reads in.

The most complicated thing any LISP system has to hide from its users is the way new CONS cells and identifiers get space allocated for them, and how the memory released when a cell gets discarded is recycled. The key term in this process is 'garbage collection'. In this microprocessor implementation of LISP, CONS cells are simply allocated from consecutive bytes of memory as and when they are needed. At some stage the area of memory available will have been filled up; at this point the garbage collector (called RECLAIM) is invoked to tidy things up. Its first job is to identify all those cells which have become defunct. It does this by a process of elimination, i.e. by marking all the cells that may still be needed. Starting from all the places the LISP interpreter ever stores anything the system traces through lists setting a bit in each block of memory it threads through. The process of marking a list can be approximated by the code

```
(DEFUN MARK (L) (COND
   ((ALREADY-MARKED L) NIL)
   (T (SET-MARK-BIT L)
      (MARK (CAR L))
      (MARK (CDR L)))))
```

However, a real mark program will have to give atoms special treatment and will probably use clever tricks to avoid the need for recursive calls to itself and the corresponding use of stack.

When the mark phase is complete, any cells not marked can be treated as garbage and re-used. To make subsequent space allocation easier it is useful to move active space down into some of these gaps so that all the free space is at one end of memory. This can be done by finding an active and a free cell, and copying the pointers from the active to the free one. The adjustment is recorded by putting a 'forwarding address' into the old active cell. Cells can be copied in this way until all live data is in a compact block at one end of store. The garbage collector then takes a third pass over all LISP pointers. Anything pointing to a cell which has been relocated has to be corrected to point to the new location of that object. Forwarding addresses provide exactly the information needed for this.

If the garbage collector does not recover any store LISP has to stop and report an error. Normally, however, the only way in which the user is aware of garbage collection is through the short suspension it causes in LISP's activity. Even though garbage collection seems quite a complicated activity, it does not slow LISP down seriously until memory becomes really tight. As a program moves closer and closer to the limits imposed by the size of the computer, so garbage collection becomes more and more frequent and performance can suffer badly. LISP works best with plenty of memory, so if there are problems look for more space by discarding unwanted functions, moving from cassette or disk LISP to a language ROM system or by installing a second processor! To find out how necessary this is, use MESSON to instruct LISP to print a message each time it collects garbage.

Some of the books and papers referred to in the bibliography contain detailed descriptions of memory formats used in various mainframe LISP systems, and details of alternative garbage-collection strategies. Perhaps the most radical of these is seen with the LISP machines developed by the Massachusetts Institute of Technology, where with about 40 megabytes of store most calculations never fill up memory, and the garbage collector is normally left disabled!

22 Inside LISP–evaluation

A distinguishing feature of LISP is that it can be used to explain its own evaluation mechanism. This means that by building on an informal understanding of the language it is possible to gain very detailed insight into LISP's inner workings. To define a LISP system in LISP may seem like a circuitous way of doing things, but it has been used when implementing LISP on new machines as well as when explaining existing versions of the language.

The main part of LISP to be discussed here is the evaluator, i.e. the functions EVAL and APPLY. The explanation will be given partly in words and partly in the form of LISP code. Various minor details specific to the Acornsoft version of LISP for the BBC Microcomputer will be glossed over - the aim is to show how typical LISPs work, not to provide ultimately definitive documentation of this one. A number of low-level primitives will be used. Some of these will only exist within the code of the LISP interpreter, and normally will not be directly available to users.

Acornsoft LISP follows a scheme for keeping track of variable values known as 'shallow binding'. In this, each identifier has a cell associated with it, and that cell always holds the current value of the identifier. The basic function SET updates this cell. We require another function GTS to read it. If the value cell of an identifier is the first word in the block of memory representing the atom, we could almost have written

```
(DEFUN SET (VAR VAL)
   (RPLACA VAR VAL)
   VAL)

(DEFUN GTS (VAR)
   (CAR VAR))
```

The only thing needed to make these definitions realistic would be the use of versions of RPLACA and CAR which do not check that their arguments are CONS cells. Now EVAL can start

```
(DEFUN EVAL (X) (COND
   ((ATOM X) (COND
      ((CHARP X) (GTS X))
      (T X)))
   ...
```

This shows that identifiers are evaluated by accessing their value cells, and that all other sorts of atoms (e.g. numbers) evaluate to themselves. Any non-atomic argument to EVAL must denote a function application. The CAR of the list is the function, the CDR is the list of arguments:

```
(T (EVAL-CALL (CAR X) (CDR X)))))
```

In the interpreter given here, EVAL-CALL will repeatedly evaluate its first argument until it gets something that is recognisably a function. In Acornsoft LISP the built-in version of EVAL will stop and report an error unless it obtains a recognisable function within two evaluations. Pointers to machine code and lambda expressions are the basic forms of function required. EVLIS will be a function which evaluates all the expressions in the list that it is given.

```
(DEFUN EVAL-CALL (FN ARGS) (COND
   ((OR (SUBRP FN)
        (LAMBDAP FN)) (APPLY FN (EVLIS ARGS)))
   ((OR (FSUBRP FN)
        (FLAMBDAP FN)) (APPLY FN (LIST ARGS)))
   (T (EVAL-CALL (EVAL FN) ARGS)))
```

The tests SUBRP and FSUBRP recognise pointers to machine code representing normal and special functions respectively, while LAMBDAP and FLAMBDAP have to recognise normal and special lambda expressions:

```
(DEFUN LAMBDAP (FN) (AND
   (NOT (ATOM FN))
   (EQ (CAR FN) 'LAMBDA)
   (OR (NULL (CADR FN)) (NOT (ATOM (CADR FN))))))
```

```
(DEFUN FLAMBDAP (FN) (AND
   (NOT (ATOM FN))
   (EQ (CAR FN) 'LAMBDA)
   (ATOM (CADR FN))
   (NOT (NULL (CADR FN)))))
```

When APPLY has to invoke a function that is defined in machine code (i.e. built into the LISP system) it does not have to do anything very complicated. It just places the argument values in standard locations and performs a subroutine jump into the code. After a few tests which detect and cope with this case, APPLY will call something which in turn has to call a function defined by a lambda expression. Ignoring the issues of optional arguments, the code for this can be sketched as

```
(DEFUN APPLY-LAMBDA (FN ARGS)
   (BIND-VARIABLES (CADR FN) ARGS (CDDR FN)))

(DEFUN BIND-VARIABLES (BVL ARGS BODY (RESULT) (TEMP))
   (COND
    ((NULL BVL) (LOOP
       (WHILE BODY RESULT)
       (SETQ RESULT (EVAL (CAR BODY))
       (SETQ BODY (CDR BODY))))
    (T (SETQ TEMP (GTS (CAR BVL)))
       (SET (CAR BVL) (CAR ARGS))
       (SETQ RESULT
          (BIND-VARIABLES (CDR BVL) (CDR ARGS) BODY))
       (SET (CAR BVL) TEMP)
       RESULT)))
```

Each variable which has to be bound has its value saved away and a new value installed based on the argument value provided for it. When all variables have been bound in this way, a loop evaluates all the expressions making up the body of the lambda expression. Then each local variable has its value restored to the state it was in before the call. The elaboration of this code to deal with cases where the list of parameter names is not the same length as the list of actual arguments is quite easy.

A real LISP interpreter needs to keep a little more information around than the one given above does. This is because it will have to be able to restore the values of variables to their initial state not only when a function call is completed in the normal way, but also when something within the function fails and the error handler has to unwind the evaluator's recursion. The mechanism for doing this will naturally enable it to find out which variables have been bound and which expressions are being processed at any time. This is the information which is displayed in a backtrace. Generally, a compromise has to be reached between the ideal of keeping enough information to make backtraces easily readable and totally informative and the practical pressure to economise on space and time by saving a minimum of information.

The original LISP used a quite different variable binding strategy, one called 'deep binding'. Deep binding does not require that variables have value cells: instead it keeps all values in a list which pairs variables' names with their values. The result is a system that will usually be slower than one employing shallow binding, but one able to support functional arguments properly and which can easily be extended to provide coroutines, multi-tasking and all sorts of advanced control structures. The LISP 1.5 programmer's manual by McCarthy (see Bibliography) contains the description of an interpreter following this style.

23 Sample LISP applications

23.1 Introduction

The next eleven sections show LISP in action, developing a representative collection of LISP function definitions. The sample programs given are all fairly short - most are only a page or two long even when prettyprinted. As well as showing just how much can be done by rather compact pieces of LISP, some of these examples (e.g. the LISP editor presented in section 23.8) can be valuable additions to the programming environment. Most of them should be treated as being minimal implementations, demonstrating ideas but not exploiting them to the full. Thus, for instance, the games need expanding and refining before they really become worth playing.

Although presented as separate programs, many of the examples here can be treated as building blocks for constructing much larger LISP programs. For instance, the parser and the evaluator for arithmetic expressions would naturally fit together to make a calculator which would accept conventional input. Equally, the prettyprinter might be used within an extended editor, the graph drawing package with a game program and so on.

The code given here should not, therefore, be viewed as being fixed once and for all, but should be used as skeletal material for the development of new LISP programs. Similarly, limitations in these programs should be treated as incentives to write new and better code, so that you gradually build up a comprehensive library of useful LISP facilities.

23.2 The evaluation of arithmetic expressions

LISP can, of course, be used directly as a desk calculator. The normal read-eval-print loop will evaluate expressions such as

```
(PLUS (TIMES 2 3) 7)
```

without the need for any programming. This section considers the evaluation of expressions that do not follow precisely the conventions needed for the simple approach to work. The example taken will be of arithmetic expressions presented in the normal LISP way, except that the operators involved will be represented by the characters +, - and * rather than the LISP identifiers PLUS, DIFFERENCE and TIMES. What is required, then, is a function called EVALUATE, for instance, which can be called to compute arbitrary arithmetic values, as in

```
(EVALUATE '(+ (* 2 3) 7))
```

A generally useful way of developing LISP functions is to write code which tests for and processes trivial cases, handing harder ones down to a further function to be designed later. Here the easiest sort of expression to evaluate will be one that is just a number:

```
(DEFUN EVALUATE (EXPRESSION) (COND
   ((NUMBERP EXPRESSION) EXPRESSION)
   (T (EVALUATE-COMPOUND-EXPRESSION
       (OPERATOR-PART EXPRESSION)
       (ARG-1-PART EXPRESSION)
       (ARG-2-PART EXPRESSION)))))
```

This definition has set out a requirement for functions (as yet undefined) to extract the operator and each of the two operands from an expression, and which then evaluate a composite expression. Despite the fact that these functions do not yet exist, EVALUATE can be tested on the sorts of argument it should cope with:

```
(EVALUATE '3)
```

It will decide that 3 is a number and will return it unaltered! If, at this stage, EVALUATE is called with a compound argument LISP will generate an error message complaining about one of the currently undefined functions. For simplicity this code assumes that all operators will have exactly two arguments. With expressions represented by LISP lists it is easy to extract the parts required:

```
(DEFUN OPERATOR-PART (EXPRESSION)
   (CAR EXPRESSION))

(DEFUN ARG-1-PART (EXPRESSION)
   (CADR EXPRESSION))

(DEFUN ARG-2-PART (EXPRESSION)
   (CADDR EXPRESSION))
```

Once, again these can be tested immediately, for instance by entering

```
(ARG-1-PART '(+ (* 3 4) 7))
```

Note that the function to evaluate compound expressions has yet to be defined. It has two jobs to do: it must decide which arithmetic operation is being performed and then do it. LISP makes it easy to separate these steps by writing the function in terms of yet another undefined operator:

```
(DEFUN EVALUATE-COMPOUND-EXPRESSION
      (OPERATOR ARG1 ARG2)
   (PERFORM-ARITHMETIC
      OPERATOR
      (EVALUATE ARG1)
      (EVALUATE ARG2)))
```

PERFORM-ARITHMETIC can now be expected to receive an operator, i.e. one of the atoms +, - or * as its first argument, and numbers as its second and third.

The required definition for this, the final function in the evaluator, is one that tests its first argument to see which operation is required, and then uses the regular built-in LISP arithmetic functions to do the work indicated:

```
(DEFUN PERFORM-ARITHMETIC (OPERATOR ARG1 ARG2)
   (COND
      ((EQ OPERATOR '+) (PLUS ARG1 ARG2))
      ((EQ OPERATOR '-) (DIFFERENCE ARG1 ARG2))
      ((EQ OPERATOR '*) (TIMES ARG1 ARG2))
      (T (ERROR (LIST OPERATOR 'UNKNOWN)))))
```

The definition given here only knows about three operators - it would be very easy to extend it to cope with more. It also detects when an unknown symbol appears where an operator should, and reports its difficulty using the ERROR function.

If a sequence of arithmetic expressions is to be evaluated, much typing can be saved by making LISP successively read, evaluate and print them. This can be done without defining any more functions but simply by using the LISP LOOP operator.

```
(LOOP
   (PRINT (EVALUATE (READ))))
```

This causes LISP to read in expressions, find their value using the EVALUATE function and display the result using PRINT. The only way of escaping from the loop is by causing an error, either by pressing the ESCAPE key or by typing an invalid expression (for instance a word instead of a number). In each of these cases LISP will print an ugly backtrace: those who dislike this idea can devise for themselves code which quits the loop cleanly when, say, the word QUIT is typed in place of an expression.

In a sense the evaluator given so far is artificial - the three lines of LISP

```
(DEFUN + (A B) (PLUS A B))
(DEFUN - (A B) (DIFFERENCE A B))
(DEFUN * (A B) (TIMES A B))
```

would have extended LISP so that prefix-form arithmetic expressions of the sort discussed could be evaluated directly by LISP! Quite commonly, careful choice of datastructure makes LISP's built-in facilities solve problems that at first seemed to need special new code. The similarity between the behaviour of EVALUATE and of LISP itself could be built on by extending EVALUATE to deal with tests such as '=' and '>', with conditional expressions perhaps represented by lists such as

```
(IF condition value-if-true value-if-false)
```

and simple variables. None of the extensions are particularly hard, though the proper treatment of variables and user-defined functions requires care. The result rapidly becomes an interpreter for a new LISP-like language - and for that matter if names and structures were chosen correctly it could become a fresh implementation of LISP, itself written in LISP.

23.3 Sorting

The sort program presented here takes a list of atoms and prints them in alphabetical order. A simple use for it would be

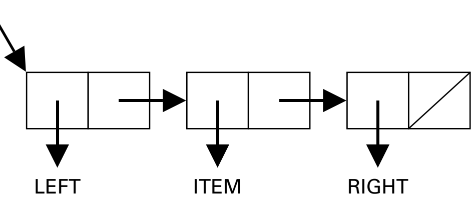
```
(SORT (OBLIST))
```

to get a tidy list of the functions that LISP knows. It could equally well be applied to lists of words in an index, to titles in a record catalogue or (with small changes to make the sort based on numeric rather than alphabetical comparisons) to scores in a computer game to generate a ranking of players.

The method used is known as 'tree-sort'. It is well suited to LISP's organisation, as well as being an efficient way of re-ordering initially chaotic lists. Some other sorting methods are faster if the items to be sorted start off in almost the correct order. Tree-sort works in two stages. In the first of these the items to be sorted are built into a structure reflecting their relative ordering and which is arranged so that adding a new item can be done quickly. The second stage then traverses this structure printing what it finds. The intermediate structure is known as a 'tree', and is built out of 'nodes'. Each tree-node has three things associated with it: one of the items which are being sorted, and a pair of subtrees, known as the left and right subtrees. A node which has empty left and right subtrees is known as a leaf. The bulk of the code doing the sorting need not be aware in any detail of the particular combination of LISP cells used to make up a tree-node - these details can be hidden away by defining a set of tree manipulation functions such as

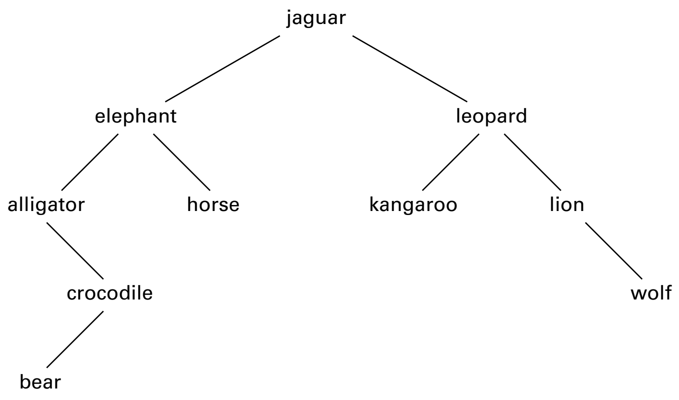
```
(DEFUN MAKE-NODE (VAL LEFT RIGHT)
   (LIST LEFT VAL RIGHT))

(DEFUN LEFT-SUBTREE (TREE)
   (CAR TREE))

(DEFUN RIGHT-SUBTREE (TREE)
   (CADDR TREE))

(DEFUN ITEM-IN-NODE (TREE)
   (CADR TREE))
```

This corresponds to a decision that nodes will be stored as



If a subtree is not wanted, the relevant field in the node can be filled with a NIL.

The main idea behind the use of trees is that it can be arranged, for any node in the tree, that all words stored in the left subtree come alphabetically before the word associated with that node, and all those in the right subtree come after.



With the representation for trees given above, when a tree is printed by LISP the words contained in it will appear in alphabetical order, but there will also be a great many brackets in the output. A clean list of the items in a tree can be displayed using:

```
(DEFUN PRINT-TREE (TREE) (COND
   ((NULL TREE) NIL)
   (T (PRINT-TREE (LEFT-SUBTREE TREE))
      (PRINT (ITEM-IN-NODE TREE))
      (PRINT-TREE (RIGHT-SUBTREE TREE)))))
```

Note how the use of long names, and of functions like LEFT-SUBTREE rather than CAR, make this definition immediately readable - particularly when it is spread out using the LISP prettyprinter.

PRINT-TREE can now be tested by using MAKE-NODE to build up some trees by hand, for instance:

```
(PRINT-TREE
    (MAKE-NODE 'SOCRATES
       (MAKE-NODE 'ARCHIMEDES NIL NIL)
       NIL))
```

The interesting use, however, is to apply it to trees made out of the items to be sorted. Rather than making all these into a tree at once, it will be convenient to start with an empty tree, and add them one at a time. This leads to the following definition of SORT:

```
(DEFUN SORT (ITEM-LIST (TREE))
   (LOOP
      (UNTIL (NULL ITEM-LIST) (PRINT-TREE TREE))
      (SETQ TREE (ADD-ITEM (CAR ITEM-LIST) TREE))
      (SETQ ITEM-LIST (CDR ITEM-LIST))))
```

Here ADD-ITEM is a function that has not yet been defined, but which will have to accept a word and a tree, and hand back a new tree obtained by merging the new word in with all the old ones. Words coming early in the alphabet are inserted in the left part of the tree, and ones coming late in the right. It also has to deal with the trivial case of adding an item to an empty tree:

```
(DEFUN ADD-ITEM (ITEM TREE) (COND
   ((NULL TREE) (MAKE-NODE ITEM NIL NIL))
   ((ORDERP ITEM (ITEM-IN-NODE TREE))
      (PUT-IN-LEFT-SUBTREE ITEM TREE))
   (T (PUT-IN-RIGHT-SUBTREE ITEM TREE))))
```

Remembering that ITEM-IN-NODE retrieves the word stored at the head of a tree, ADD-ITEM uses a test called ORDERP to decide whether the new item belongs to the left or right part of the tree. A definition of ORDERP which will give an alphabetical sort is shown later. If LESSP were used in place of ORDERP the function would sort numbers into ascending order. GREATERP would give a descending numeric sort, and so on for any ordering predicate you wish to use.

To complete the program, definitions are needed for PUT-IN-LEFT-SUBTREE and PUT-IN-RIGHT-SUBTREE, but these are very easy!

```
(DEFUN PUT-IN-LEFT-SUBTREE (ITEM TREE)
   (MAKE-NODE
      (ITEM-IN-NODE TREE)
      (ADD-ITEM ITEM (LEFT-SUBTREE TREE))
      (RIGHT-SUBTREE TREE)))
```

```
(DEFUN PUT-IN-RIGHT-SUBTREE (ITEM TREE)
   (MAKE-NODE
      (ITEM-IN-NODE TREE)
      (LEFT-SUBTREE TREE)
      (ADD-ITEM ITEM (RIGHT-SUBTREE TREE))))
```

Each of these copies a node of the tree, passing the item being added down to the relevant subtree. Provided ORDERP is defined, that it has been replaced by some other predicate, the entire sort package can now be tested. A dozen or so words or several dozen numbers should be sorted with little noticeable delay, while putting the object list into alphabetical order will take a little while and will involve several garbage collections.

Minor savings in both time and space could have been made in the programs given above by using CAR, CADR and CADDR directly rather than via ITEM-IN-NODE etc, and by writing PUT-IN-LEFT-SUBTREE's definition in place of its call, and perhaps even by using shorter names for variables and functions. Even on a microcomputer the resulting huge loss of clarity usually outweighs the savings in resources.

Acornsoft LISP does not provide a built-in function for deciding which of two words comes earlier in the alphabet. It does, however, provide a function EXPLODE that splits a word into its component letters, and another called ORDINAL that converts any single letter into a numeric code. Thus the following definition of ORDERP can be used:

```
(DEFUN ORDERP (A B)
   (ORDERP1 (EXPLODE A) (EXPLODE B)))

(DEFUN ORDERP1 (AL BL) (COND
   ((NULL AL) T)
   ((NULL BL) NIL)
   ((EQ (CAR AL) (CAR BL))
        (ORDERP1 (CDR AL) (CDR BL)))
   (T (LESSP (ORDINAL (CAR AL))
             (ORDINAL (CAR BL))))))
```

ORDERP simply spreads its two arguments out into lists of characters and hands the results down to ORDERP1.

When sorting into alphabetical order ORDERP1 places shorter words before longer ones starting with the same letters. It also strips any identical initial strings of characters from the two lists it is given. When it finally finds a difference in its input lists it converts the letters to numbers using ORDINAL and uses LESSP to do a comparison. The ordering produced is thus dependent on the sequence of codes generated by ORDINAL, but at least for letters that are all in the same case this corresponds to ordinary alphabetical order.

23.4 Arbitrary precision arithmetic

Many mainframe implementations of LISP take the view that the LISP functions PLUS, TIMES and so on work with numbers, and any restriction on the number of digits that can appear in a number would be an unwelcome concession to the limitations of computers. They therefore provide arithmetic limited only by the amount of time that the user is prepared to wait for results. In Acornsoft LISP normal arithmetic can only cope with 16-bit numbers, i.e. numbers in the range -32768 to 32767. The functions developed here indicate how mainframe-style unlimited arithmetic can be implemented using just the facilities already available in Acornsoft LISP. To demonstrate this, various large powers of two will be computed.

In this package, large numbers will be represented as list structures containing small numbers. Perhaps the most obvious representation to use would be holding big numbers as lists of digits, so that a million would be held as the list (1 0 0 0 0 0 0). The scheme used here differs from this simple representation in three ways:

(a) The small numbers, which are treated as digits in the long arithmetic package, are in the range 0 to 99 rather than 0 to 9. This grouping roughly doubles the speed of calculation, and does not introduce any serious complications in the algorithms needed.

(b) The least-significant parts of numbers are stored at the head of the big-number list, since this makes the programming of 'carry and borrow' operations easier.

(c) Numbers in the range 0 to 99 are represented without any enclosing list structure. This saves a small amount of store and avoids a large amount of confusion! Numbers up to 9999 will then naturally be stored using a single CONS cell - for instance, 1789 will be kept as (89 . 17), and larger numbers will lead to similar structures that print with a dot in them. The million considered earlier ends up as the list (0 0 0 . 1).

With a view to its use in debugging the rest of the functions to be written, the first function to be written prints big numbers in natural notation:

```
(DEFUN BIG-PRINT (N) (COND
   ((NUMBERP N) (PRINC N))
   (T (BIG-PRINT (CDR N))
      (PRINT-TWO-DIGITS (CAR N)) )))

(DEFUN PRINT-TWO-DIGITS (N)
   (PRINC (QUOTIENT N 10))
   (PRINC (REMAINDER N 10)))
```

The function PRINT-TWO-DIGITS has to be introduced to ensure that each number in the datastructure is printed as exactly two characters. Had BIG-PRINT just called PRINC to display these numbers some zero digits from the true answer could have been lost.

The next function to be implemented will be one to add big numbers. As is usual with LISP functions, this one starts by testing for simple cases, i.e. those where its inputs are not in fact multiple precision:

```
(DEFUN BIG-PLUS (A B)
   (COND
      ((NUMBERP A) (SMALL-PLUS-BIG A B))
      ((NUMBERP B) (SMALL-PLUS-BIG B A))
      (T (JOIN-DIGIT
            (PLUS (CAR A) (CAR B))
            (BIG-PLUS (CDR A) (CDR B))))))
```

A function BIG-NUMBER will be needed to convert a (basic LISP) number to the big representation, SMALL-PLUS-BIG will add a small number to a big one, and JOIN-DIGIT will attach a digit on the front of a big number, dealing with whatever carries are involved. Taking these in turn, we can write

```
(DEFUN BIG-NUMBER (N)
   (COND
      ((LESSP N 100) N)
      (T (CONS
            (REMAINDER N 100)
            (BIG-NUMBER (QUOTIENT N 100))))))
```

```
(DEFUN SMALL-PLUS-BIG (A B)
   (COND
      ((NUMBERP B) (BIG-NUMBER (PLUS A B)))
      (T (JOIN-DIGIT
            (PLUS A (CAR B))
            (CDR B)))))
```

Note that SMALL-PLUS-BIG has been coded so that it can cope with a second argument that is a small number. The fact that its code can be expressed neatly in terms of the JOIN-DIGIT function already required for BIG-PLUS is encouraging. JOIN-DIGIT itself turns out to be simple to define in terms of the functions already designed:

```
(DEFUN JOIN-DIGIT (N A)
   (COND
      ((LESSP N 100) (CONS N A))
      (T (CONS
            (REMAINDER N 100)
            (SMALL-PLUS-BIG (QUOTIENT N 100) A)))))
```

If no carry is needed, JOIN-DIGIT just behaves like CONS. Otherwise, it has to propagate a carry.

This completes the definitions concerned with adding big numbers. Multiplication is achieved by using a similar set, the first of which is mainly concerned with detecting and dealing with cases where one of its arguments turns out to be small. Note that in this package the small numbers are kept in the range 0 to 99, and so ordinary LISP TIMES can be used to form the product of two digits. If we had been more ambitious and tried to group digits into (say) fours in the datastructure this would not have been the case.

```
(DEFUN BIG-TIMES (A B)
   (COND
      ((NUMBERP A) (SMALL-TIMES-BIG A B))
      ((NUMBERP B) (SMALL-TIMES-BIG B A))
      (T ... )))
```

The final clause will be filled in after SMALL-TIMES-BIG has been written.

```
(DEFUN SMALL-TIMES-BIG (A B)
   (COND
      ((NUMBERP B) (BIG-NUMBER (TIMES A B)))
      (T (JOIN-DIGIT
            (TIMES A (CAR B))
            (SMALL-TIMES-BIG A (CDR B))))))
```

As with SMALL-PLUS-BIG this function starts by testing to see whether its second argument is the large number it threatens to be. If not, ordinary LISP TIMES can form the required product, and BIG-NUMBER spreads it out to standard big-number form. Otherwise, the leading digit of the product is obtained by multiplying the leading digit of B by A, and this is pushed on the front of the product of multiplying the higher order digits of B by A. The complete definition of BIG-TIMES can be obtained by completing the final clause by including code that does a very similar job to the general arguments A and B:

```
   (BIG-PLUS
      (SMALL-TIMES-BIG (CAR B) A)
      (CONS 0 (BIG-TIMES A (CDR B))))
```

Here the form (CONS 0 x) is used to align digits correctly in the answer for the addition.

The functions in this package are all defined in terms of one another, and a call to any one may exercise almost the whole of the code. Testing is still easy, because most of the functions have very simple behaviour when called on small arguments, and these basic routes through the code can be checked first.

The final part of this example arranges to call BIG-TIMES repeatedly to raise two to a large power. It then uses BIG-PRINT to display the result in readable form.

```
(DEFUN BIG-POWER-OF-2 (N)
   (PRINT (LIST 2 'TO 'THE 'POWER N 'IS))
   (BIG-PRINT (BIG-EXPT 2 N))
   'DONE)
```

The real result BIG-POWER-OF-2 produces is that displayed by BIG-PRINT: there is no very useful value it can hand back for the normal LISP evaluator to print. As shown above, the word DONE is passed back so that at least a predictable value is shown. BIG-EXPT raises a number to a power. It works by repeatedly squaring its argument, rather than simply by doing N multiplications - this improves its speed substantially.

```
(DEFUN BIG-EXPT (A N)
   (COND
      ((EQUAL N 0) 1)
      (T (SUBFUNCTION-FOR-BIG-EXPT A
            (BIG-EXPT A (QUOTIENT N 2))
            (REMAINDER N 2)))))

(DEFUN SUBFUNCTION-FOR-BIG-EXPT (A APOWER NREM)
   (COND
      ((EQUAL NREM 0) (BIG-TIMES APOWER APOWER))
      (T (BIG-TIMES
            (BIG-TIMES A APOWER)
            APOWER))))

(DEFUN EQUAL (A B)
   (COND
      ((EQ A B) T)
      ((OR (ATOM A) (ATOM B)) NIL)
      ((EQUAL (CAR A) (CAR B))
         (EQUAL (CDR A) (CDR B)))
      (T NIL)))
```

23.5 The LISP prettyprinter

If the LISP function PRINT is used to display a large piece of LISP structure, say a function definition, the result is not easy to read. This is because although LISP's brackets make the structure of the expression unambiguous, unformatted output does not give human readers any help in matching the brackets and recognising structure. A prettyprinter is a package of code which can be used in place of PRINT and which will spread out its output over many lines, using indentation to try to make it as legible as possible.

The prettyprinter used here knows two ways of laying out lists. For lists sufficiently short that they will fit on one line it uses the normal layout (a):

```
(function arg1 arg2 ... argn)
```

but for larger structures it splits lines and imposes indentation (b):

```
(function
   arg1
   arg2
   ...
   argn)
```

A simplified version of this program is built into the distributed version of Acornsoft LISP under the name SPRINT. The code here is larger, but deals more smoothly with quoted expressions and calls to various common functions (such as SETQ and DEFUN) that deserve special treatment. The prettyprinter starts work on a list by counting the number of character positions which would be needed to display the list in format (a). If this shows that the list can be printed on one line, the prettyprinter does so. Otherwise, a loop producing format (b) output is entered. Note that the arguments in format (b) are themselves prettyprinted and may be laid out using either format (a) or (b).

The first function required in this package is one that will measure the width of printed lists. There is no need to compute the exact length of lists which will not fit on one line, so the function given here takes as an argument a target width W, and subtracts the printed width of its other argument from W, quitting if this leads to a negative result. The LISP built-in function CHARS returns the number of characters in a LISP identifier, and forms the basis for the following:

```
(DEFUN WIDTH-OF-NUMBER (N)
   (COND
      ((MINUSP N)
         (ADD1 (WIDTH-OF-NUMBER (MINUS N))))
      ((LESSP N 10) 1)
      (T (ADD1 (WIDTH-OF-NUMBER (QUOTIENT N 10))))))

(DEFUN WIDTH-OF-ATOM (X)
   (COND ((NUMBERP X) (WIDTH-OF-NUMBER X))
         (T (CHARS X))))

(DEFUN SUBTRACT-WIDTH (X W)
   (COND
      ((ATOM X) (DIFFERENCE W (WIDTH-OF-ATOM X)))
      ((QUOTEP X)
         (SUBTRACT-WIDTH-OF-LIST (CADR X) (SUB1 W)))
      (T (SUBTRACT-WIDTH-OF-LIST X (SUB1 W)))))

(DEFUN SUBTRACT-WIDTH-OF-LIST (X W)
   (LOOP
      (UNTIL (OR (NULL X) (MINUSP W)) W)
      (UNTIL
         (ATOM X)
         (DIFFERENCE W (PLUS 2 (WIDTH-OF-ATOM X))))
      (SETQ W (SUBTRACT-WIDTH (CAR X) (SUB1 W)))
      (SETQ X (CDR X))))

(DEFUN WILL-FIT (X W)
   (NOT (MINUSP (SUBTRACT-WIDTH X W))))
```

The first two functions given are solely concerned with extending the LISP CHARS function to provide a function (WIDTH-OF-ATOM) which can measure numeric as well as symbolic atoms. SUBTRACT-WIDTH recognises three sorts of arguments. Anything atomic can be coped with by WIDTH-OF-ATOM. Quoted expressions are detected using a function QUOTEP which will be shown later, and proper lists are handed to a further subfunction, SUBTRACT-WIDTH-OF-LIST. This loops down its argument decreasing W by the width of the sublists found, and by extra amounts that allow for the spaces to be printed later between sublists. It exits either when W goes negative, or when the end of the list is found. If the list terminates with a non-NIL atom W has to be adjusted to account for the '.' which will be printed to reflect this.

The main prettyprint function can now be presented. SUPERPRINT has two arguments. The first is the list to be printed, the second, which defaults to zero, is a lefthand margin to be imposed on the printing. Most of SUPERPRINT's work is done by SUPERPRIN. When SUPERPRIN calls itself it will increase its second argument so that nested sublists appear further and further to the right. A variable called LINELENGTH is used to define the position of the righthand margin for printing:

```
(DEFUN SUPERPRINT (X (LEFTMARGIN . 0))
   (SUPERPRIN X LEFTMARGIN)
   (PRINT)
   X)

(DEFUN SUPERPRIN (X LEFTMARGIN (SUPER))
   (COND
      ((ATOM X) (PRIN X))
      ((QUOTEP X)
         (PRINC (CHARACTER 39))
         (SUPERPRIN (CADR X) (ADD1 LEFTMARGIN)))
      ((WILL-FIT X (DIFFERENCE LINELENGTH LEFTMARGIN))
         (SUPER-SUB X (PLUS LEFTMARGIN 3)))
      (T (SETQ SUPER T)
         (SUPER-SUB X (PLUS LEFTMARGIN 3)))))
```

SUPERPRIN does two things worthy of note. It detects quoted expressions (again using QUOTEP) and prints them with quote marks in front. The internal code for a quote mark is 39 on the BBC Microcomputer, so (PRINC (CHARACTER 39)) causes one to be displayed. An alternative way of obtaining the same output would be to introduce the quote mark with an escape character, as in (PRINC '!'). Then SUPERPRIN sets a flag SUPER which will instruct its subfunction SUPER-SUB whether it should use format (a) or format (b) printing. SUPER-SUB itself uses two more flags. SEPCHAR is used so that it prints '(' before the first sublist it processes, and blanks (or newlines) before all the rest. SPECIAL is used to reduce the number of newlines printed around various functions that usually have atomic first arguments. If the big COND in the middle of the loop is ignored it can be seen that SUPER-SUB represents a straightforward scan down the list X calling SUPERPRIN on all sublists, as shown below:

```
(DEFUN SUPER-SUB (X LEFTMARGIN (SEPCHAR) (SPECIAL))
   (COND
      ((CHARP (CAR X))
         (SETQ SPECIAL (GET (CAR X) 'SPECIAL))))
   (LOOP
      (UNTIL (NULL X) (PRINC RPAR))
     (UNTIL (ATOM X) (PRINC BLANK PERIOD BLANK X RPAR))
      (COND
         ((NULL SEPCHAR) (PRINC LPAR) (SETQ SEPCHAR T))
            (SPECIAL
               (PRINC BLANK)
               (COND
               ((MINUSP (SETQ SPECIAL (SUB1 SPECIAL)))
                     (SETQ SPECIAL NIL))))
            ((NULL SUPER) (PRINC BLANK))
            (T (PRINT) (SPACES LEFTMARGIN)))
         (SUPERPRIN (CAR X) LEFTMARGIN)
         (SETQ X (CDR X))))
```

The use of SPECIAL properties can be seen in the above displays of the code of SUPERPRINT, which were themselves formatted by SUPERPRINT. Observe that the format used with DEFUN is always

```
(DEFUN name args
   ... )
```

and not

```
(DEFUN
   name
   args
   ... )
```

The code for SUPER-SUB arranges matters so that if a function has had a number stored as its SPECIAL property that number inhibits some line breaks. A good start to the list of functions to be treated in this way is:

```
(PUT 'SETQ 'SPECIAL '0)
(PUT 'DEFUN 'SPECIAL '1)
(PUT 'T 'SPECIAL '0)
```

```
(PUT 'LAMBDA 'SPECIAL '0)
```

The only functions to remain undefined are SPACES (which prints a sequence of spaces) and QUOTEP which detects if a list should be displayed using a quote mark, i.e. if it is of the form (QUOTE something). These are both easy!

```
(DEFUN SPACES (N)
   (LOOP
      (UNTIL (MINUSP (SETQ N (SUB1 N))))
      (PRINC BLANK)))

(DEFUN QUOTEP (X)
   (AND
      (NOT (ATOM X))
      (EQ (CAR X) 'QUOTE)
      (NOT (ATOM (CDR X)))
      (NULL (CDDR X))))

(SETQ LINELENGTH '60)
```

That completes the prettyprinter. After installing it in LISP the call (SETQ SPRINT SUPERPRINT) will replace the system-provided formatter by this improved version, and the functions CHARCOUNT and XTAB can probably be discarded.

23.6 An animal guessing game

The game presented here allows you to think of an animal; the computer then tries to guess what it is. If it fails you tell it the question it should have asked in order to get the right answer. It remembers this question and your new animal, and can display this knowledge in later games. The program's database of zoological information can be stored on cassette or disk using SAVE and restored using LOAD, and so it can grow over a period of time. After friends have played with this program for a while you may find some very strange creatures defined! To invoke the package it will be necessary to call a function (ANIMAL) which is an endless loop:

```
(DEFUN ANIMAL NIL
   (LOOP
      (SAY-HELLO)
      (SETQ KNOWN-ANIMALS (GUESS KNOWN-ANIMALS))))

(DEFUN SAY-HELLO NIL
   (PRINT)
   (LPRI '(THINK OF AN ANIMAL - I WILL GUESS IT))
   (PRINT))
```

The animal database is being kept in the variable KNOWN-ANIMALS, and LPRI will print the words in a LISP list, missing out the ugly brackets. KNOWN-ANIMALS will be a tree, rather like the one used in the tree-sort example. At its leaves will be atoms naming animals; elsewhere there will be list structures containing:

(a) a question to ask;
(b) a subtree to scan if the player answers YES;
(c) a subtree to scan when the player answers NO;

When GUESS has finished it will return an updated copy of this tree incorporating any information it has gathered during the game. As coded here it uses RPLACA and RPLACD to update the tree, saving some space compared with the copying technique used in tree-sort:

```
(DEFUN GUESS (KNOWN-ANIMALS)
   (COND
      ((ATOM KNOWN-ANIMALS) (I-GUESS KNOWN-ANIMALS))
      (T (PRINTC (CAR KNOWN-ANIMALS))
         (COND
            ((YESP (READ))
               (RPLACA
                  (CDR KNOWN-ANIMALS)
                  (GUESS (CADR KNOWN-ANIMALS))))
            (T (RPLACD
                  (CDR KNOWN-ANIMALS)
                  (GUESS (CDDR KNOWN-ANIMALS)))))
         KNOWN-ANIMALS)))
```

When GUESS reaches a leaf of the tree of known animals it calls I-GUESS which reports that it believes it knows the animal being thought of. If the player confirms that it was right it prints 'hurrah', otherwise it has to give up.

```
(DEFUN I-GUESS (CREATURE)
   (LPRI (LIST 'IS 'IT 'A CREATURE))
   (COND
      ((YESP (READ)) (PRINTC 'HURRAH) CREATURE)
      (T (GIVE-UP CREATURE))))
```

GIVE-UP is straightforward except for the use of READLINE. The apparently superfluous call to it is needed to sweep up anything typed on the same line as the name of the new animal.

```
(DEFUN GIVE-UP (I-THOUGHT (NEW-ANIMAL) (NEW-QUESTION))
   (LPRI '(I GIVE UP))
   (LPRI '(WHAT WAS IT?))
   (SETQ NEW-ANIMAL (READ))
   (LPRI '(PLEASE TYPE IN A QUESTION THAT WOULD))
   (LPRI (LIST 'DISTINGUISH 'A NEW-ANIMAL
               'FROM 'A I-THOUGHT))
   (READLINE)
   (SETQ NEW-QUESTION (READLINE))
   (LPRI '(THANK YOU))
   (CONS NEW-QUESTION (CONS NEW-ANIMAL I-THOUGHT)))
```

So that the computer will accept either YES or just Y as an answer a function YESP is provided. It might be useful to use ORDINAL or EXPLODE to select the first character of the user's response and to test for both upper and lower case 'Y'....

```
(DEFUN YESP (A) (OR (EQ A 'YES) (EQ A 'Y)))
```

As mentioned earlier, LPRI prints the atoms in a list, separated by blanks but without surrounding brackets.

```
(DEFUN LPRI (L)
   (LOOP
      (UNTIL (NULL L) (PRINT))
      (PRINC (CAR L) BLANK)
      (SETQ L (CDR L))))
```

Finally, we have a possible initial setting for the variable KNOWN-ANIMALS. This one was created by running the game a few times and then printing out the database. Note that when entered via READ, atoms that contain blank characters in their names have to be typed with the blanks preceded by escape characters (!). Of course, when the game is being played READLINE is accepting input and these markers are not needed.

```
(SETQ KNOWN-ANIMALS
   '(DOES! IT! HAVE! A! LONG! NECK?
      (DOES! IT! LIVE! IN! AFRICA? GIRAFFE . SWAN)
      DOES! IT! HAVE! BIG! EARS?
   (DOES! IT! HAVE! A! BIG! NOSE? ELEPHANT . RABBIT)
      . CROCODILE))
```

23.7 A route-finding program

The package described here allows the user to create a database showing the distances between pairs of towns. Once these data have been entered, the package can find the shortest route between any given places. The method used is an example of an 'ink-blot' or 'breadth first' search. It starts from the source city, and fills in distance and direction information for the other towns in the order of their distance from the starting point. This can be likened to pouring gallons of ink into the centre of the starting city and watching it spread out uniformly along the road network colouring each place it reaches. The shortest route to the destination town will be the one taken by the river of ink to arrive first.

To apply this ink-blot method we will need to maintain a list of those roads the ink is currently flowing along, arranged so that the city reached first gets processed first. A list of this sort is referred to as a priority queue, and one way of representing it will use a datastructure known as a 'heap'.

Heaps are something like the trees used in the tree-sort package discussed earlier, but in a heap the smallest element stored is always at the top of the structure, and the only requirement on the two subheaps is that they remain roughly the same size. As for tree-sort, the heap package used here is programmed using small access functions to define the subfields in the LISP datastructures that make up heaps. ADD-TO-HEAP keeps the two halves of the heap it builds in balance by repeatedly swapping them over, thus adding items to the two subheaps alternately:

```
(DEFUN NEW-NODE (ITEM LEFT RIGHT)
   (CONS ITEM (CONS LEFT RIGHT)))

(DEFUN GET-ITEM (HEAP) (CAR HEAP))

(DEFUN LEFT-SUBHEAP (HEAP) (CADR HEAP))

(DEFUN RIGHT-SUBHEAP (HEAP) (CDDR HEAP))
```

```
(DEFUN ADD-TO-HEAP (ITEM HEAP)
   (COND
      ((NULL HEAP) (NEW-NODE ITEM NIL NIL))
      ((IS-SMALLER ITEM (GET-ITEM HEAP))
         (ADD-TO-HEAP
            (GET-ITEM HEAP)
            (REPLACE-ITEM HEAP ITEM)))
      (T (NEW-NODE
            (GET-ITEM HEAP)
            (RIGHT-SUBHEAP HEAP)
            (ADD-TO-HEAP ITEM (LEFT-SUBHEAP HEAP))))))
```

Note that the smallest item in a heap is immediately available since it is at the top. It represents the next city to be flooded with ink.

Once the top item has been processed it will have to be removed from the heap. This must be done in such a way that the two halves of the heap remain in balance, and the new smallest element appears at the top of the reduced heap. The code to achieve this looks complicated, but its costs grow only slowly with the size of the heap. First observe that it is easy to remove the element stored in the very rightmost branch of a heap by undoing the operations ADD-HEAP would have done to put it there:

```
(DEFUN REMOVE-RIGHTMOST (HEAP)
   (COND
      ((NULL (RIGHT-SUBHEAP HEAP))
         (SETQ RIGHTMOST (GET-ITEM HEAP))
         NIL)
      (T (NEW-NODE
         (GET-ITEM HEAP)
         (REMOVE-RIGHTMOST (RIGHT-SUBHEAP HEAP))
         (LEFT-SUBHEAP HEAP)))))
```

The item removed has been stored in a working variable RIGHTMOST. Of course, the rightmost item is not the one we want to remove. We want to dispose of the top item in the heap. Disregarding the need to keep small elements high in the heap, we can move towards this aim by overwriting the top item with a copy of the rightmost one just extracted! It will then be necessary to pass the smallest item to the top of the reformed heap:

```
(DEFUN REMOVE-TOP-ITEM (HEAP (RIGHTMOST))
   (COND
      ((NULL (RIGHT-SUBHEAP HEAP)) NIL)
      (T (SETQ HEAP (REMOVE-RIGHTMOST HEAP))
         (RESTORE-HEAP
            (REPLACE-ITEM HEAP RIGHTMOST)))))

(DEFUN REPLACE-ITEM (HEAP ITEM)
   (CONS ITEM (CDR HEAP)))
```

RESTORE-HEAP works by performing a three-way comparison between the item at the top of a heap and the items heading its left and right subheaps. The smallest of these three values must top the new heap. Obviously if the top item is already the smallest nothing more need be done. Otherwise, the three elements can be rearranged, leaving just one of the subheaps in need of further processing:

```
(DEFUN RESTORE-HEAP (HEAP)
   (COND
      ((TOP-ITEM-IS-SMALLEST HEAP) HEAP)
      ((LEFT-ITEM-IS-SMALLEST HEAP)
         (PERCOLATE-LEFT HEAP))
      (T (PERCOLATE-RIGHT HEAP))))
```

The three-way comparisons are very simple in concept, but can lead to code of unreasonable length. This is partly because they must deal with the degenerate cases arising when a heap has one or both of its subheaps empty. Here, this complexity is slightly reduced because all the functions are coded so that if a heap has an empty subheap it will be the left one:

```
(DEFUN TOP-ITEM-IS-SMALLEST (HEAP)
   (OR
      (NULL (RIGHT-SUBHEAP HEAP))
      (AND
         (IS-SMALLER
            (GET-ITEM HEAP)
            (GET-ITEM (RIGHT-SUBHEAP HEAP)))
         (OR
            (NULL (LEFT-SUBHEAP HEAP))
            (IS-SMALLER
               (GET-ITEM HEAP)
               (GET-ITEM (LEFT-SUBHEAP HEAP)))))))

(DEFUN LEFT-ITEM-IS-SMALLEST (HEAP)
   (AND
      (LEFT-SUBHEAP HEAP)
      (IS-SMALLER
         (GET-ITEM (LEFT-SUBHEAP HEAP))
         (GET-ITEM HEAP))
      (IS-SMALLER
         (GET-ITEM (LEFT-SUBHEAP HEAP))
         (GET-ITEM (RIGHT-SUBHEAP HEAP)))))
```

By using a freshly-named function (IS-SMALLER) for basic comparisons we have delayed the need to specify the exact format the item stored in the heap will take. Now the functions PERCOLATE-RIGHT and PERCOLATE-LEFT can be given. Each of them simply rebuild the top node of a heap in order to bring the smallest element to the top:

```
(DEFUN PERCOLATE-RIGHT (HEAP)
   (NEW-NODE
      (GET-ITEM (RIGHT-SUBHEAP HEAP))
      (LEFT-SUBHEAP HEAP)
      (RESTORE-HEAP
         (REPLACE-ITEM
            (RIGHT-SUBHEAP HEAP)
            (GET-ITEM HEAP)))))
```

```
(DEFUN PERCOLATE-LEFT (HEAP)
   (NEW-NODE
      (GET-ITEM (LEFT-SUBHEAP HEAP))
      (RESTORE-HEAP
         (REPLACE-ITEM
            (LEFT-SUBHEAP HEAP)
            (GET-ITEM HEAP)))
      (RIGHT-SUBHEAP HEAP)))
```

The heap manipulation functions described above could very easily be used as the basis for a general-purpose sorting package. One would expect this package to be somewhat slower than tree-sort, because the heap-deletion process costs more than the tree-traversal which terminates tree-sort. However, for typical input lists the two methods should differ in speed by at most a factor of two, regardless of the number of items being sorted. Furthermore, both methods will, at worst, be constant factor slower than _any_ general-purpose sort package based on the comparison of pairs of elements. Heap-sort does have one advantage over tree-sort that will sometimes make it preferable: the time taken is insensitive to the initial order of the items to be sorted. Tree-sort, by contrast, performs very badly for certain inputs, in particular, those where the input list starts off either in the correct order or its converse.

Having arranged to support heaps so that we can maintain queues of events, the route-finder program becomes fairly simple. It seeds an event queue with those cities adjacent to the source, each keyed by their distance from it. Its main loop extracts the nearest city from this queue, and if it has not already been visited (indicated by using the name of the source city as a property via PUT and GET) route information can be recorded. Furthermore, all cities directly accessible from the newly-encountered one are entered in the event-queue. The search normally terminates when the destination city is reached. If the event-queue becomes empty before then the program indicates that a route from source to destination does not exist:

```
(DEFUN FIND-ROUTE (SOURCE DESTINATION (PRIORITY-QUEUE)
                  (CITY) (DISTANCE) (VIA) (W))
   (SETQ PRIORITY-QUEUE (ADD-TO-QUEUE SOURCE 0 NIL))
   (LOOP
      (UNTIL
         (GET DESTINATION SOURCE)
         (REPORT-ROUTE SOURCE DESTINATION))
      (UNTIL (NULL PRIORITY-QUEUE) (REPORT-FAILURE))
      (SETQ W (GET-ITEM PRIORITY-QUEUE))
      (SETQ CITY (CAR W))
      (SETQ DISTANCE (CADR W))
      (SETQ VIA (CADDR W))
      (SETQ PRIORITY-QUEUE
         (REMOVE-TOP-ITEM PRIORITY-QUEUE))
      (COND
         ((NULL (GET CITY SOURCE))
            (PUT CITY SOURCE (CONS DISTANCE VIA))
            (SETQ PRIORITY-QUEUE
               (ADD-TO-QUEUE
                  CITY
                  DISTANCE
                  PRIORITY-QUEUE))))))
```

Only at this stage does it becomes necessary to decide how to record the database that identifies adjacent cities and gives the distances between them. The scheme chosen here is that each place has a property NEIGHBOURS, and stored under this property is a list associating close towns and the distances to them. It will, naturally, be useful to provide a function for setting up such properties:

```
(DEFUN NPUT (SOURCE NEIGHBOURS)
   (PUT SOURCE 'NEIGHBOURS NEIGHBOURS))
```

Examples of neighbour lists will be given later. The code that scans a neighbour list adding records to an event-queue is given below:

```
(DEFUN ADD-TO-QUEUE (CITY BASE-DISTANCE QUEUE
                     (NEIGHBOURS))
   (SETQ NEIGHBOURS (GET CITY 'NEIGHBOURS))
   (LOOP
      (WHILE NEIGHBOURS QUEUE)
      (SETQ QUEUE
         (ADD-TO-HEAP
            (LIST
               (CAAR NEIGHBOURS)
               (PLUS BASE-DISTANCE (CDAR NEIGHBOURS))
               CITY)
            QUEUE))
      (SETQ NEIGHBOURS (CDR NEIGHBOURS))))
```

This makes an item in the queue a list (city distance via) where the road from via to city forms the last leg of the journey covering the whole distance from source to city. Referring back to the heap package, we see that this format calls for a comparison operator of the form

```
(DEFUN IS-SMALLER (A B) (LESSP (CADR A) (CADR B)))
```

Apart from the functions to disentangle and print results this is the end of the package. Since the ink-blot search spread out from the source city, and each place visited was given a pointer (under the indicator VIA) of the town on the route just before it, the final route has to be reconstructed in reverse, from destination to source. By changing the search so that it spread out from the destination until it found the source this minor inconvenience could have been avoided. When searching large networks it would probably be useful to spread simultaneously from both source and destination (imagine pouring red ink for one and blue for the other) and stopping when the scanned regions overlap (become purple?):

```
(DEFUN REPORT-ROUTE (SOURCE DESTINATION (VIA))
   (PRINT 'DISTANCE= (CAR (GET DESTINATION SOURCE)))
   (LOOP
      (SETQ VIA (CONS DESTINATION VIA))
      (UNTIL (EQ SOURCE DESTINATION)
         (PRINT 'VIA: VIA) (PRINT))
      (SETQ DESTINATION
         (CDR (GET DESTINATION SOURCE)))))

(DEFUN REPORT-FAILURE () '(NO ROUTE EXISTS))
```

The code given above is designed to cope efficiently with large networks although the example of its use given here involves just six towns. With scant regard for geographical accuracy, the database is set up using a sequence of calls to NPUT:

```
(NPUT 'CAMBRIDGE '((BEDFORD . 15) (ROYSTON . 20)))
(NPUT 'ROYSTON
   '((CAMBRIDGE . 20) (WATFORD . 30) (LONDON . 50)))
(NPUT 'LONDON '((ROYSTON . 20) (WATFORD . 25)
   (OXFORD . 50)))
(NPUT 'BEDFORD '((CAMBRIDGE . 15) (WATFORD . 30)))
(NPUT 'WATFORD
   '((BEDFORD . 30) (ROYSTON . 30)
     (LONDON . 25) (OXFORD . 40)))
(NPUT 'OXFORD '((ROYSTON . 50) (WATFORD . 25)
   (LONDON . 50)))
```

Routes can then be found by calling FIND-ROUTE:

```
(FIND-ROUTE 'CAMBRIDGE 'OXFORD)

DISTANCE=85
VIA:(CAMBRIDGE BEDFORD WATFORD OXFORD)
```

The program leaves various properties attached to identifiers, and all these should really be removed between calls of FIND-ROUTE. They certainly need to be removed if any of the distance information established by NPUT is changed - a function to scan all the atoms in (OBLIST) cleaning up property lists is not hard to design.

23.8 The LISP editor

The LISP editor is invoked by a call (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:

```
(DEFUN EDIT L
   (SPRINT (EVAL (CAR L)))
   (PRINTC)
   (SET (CAR L) (SED (EVAL (CAR L)))))
```

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) (PRINTC) (READ))
            ((EQ Q CR) (SPRINT A) (PRINTC) A)
            ((EQ Q 'C) (PRINTC) (CONS (READ) A))
            ((ATOM A) (PRINC '*) A)
            ((EQ Q 'D) (CONS (CAR A) (SED (CDR A))))
            ((EQ Q 'A) (CONS (SED (CAR A)) (CDR A)))
            ((EQ Q 'X) (CDR A))
            (T (PRINC '?) A)))))
```

The R command allows you to replace the expression that SED is currently considering. C and X insert and delete items from lists. SED prettyprints the current expression at the end of each line of edit requests. There are, of course, many additional commands that could 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 can be used on itself, and thus extended to provide whatever extra commands are required.

23.9 Graphical output from LISP

If pictures can be represented by list structures, LISP can be used as a language within which a graphics package can be embedded. On the BBC Microcomputer all forms of plotting can be achieved by sending sequences of control characters to the VDU driver. Consequently by using the function VDU all of the microcomputer's graphics capabilities can be exploited from within LISP. In the demonstration graphics package described here the user writes LISP functions which build datastructures describing pictures, and a special print routine displays them, in colour, on the lower part of the screen.

To use this code, LISP must have been entered with a graphics screen mode active. The space required for the screen map means that a 32K machine is required, and unless a second processor is in use only graphics modes 4 and 5 will be usable. Note that this code changes graphics modes by sending control codes to the VDU driver - in normal circumstances it would be better to use MODE but that is not convenient here because it has a delayed effect.

The first two functions given simply call the VDU function to send a few bytes to the Microcomputer's VDU driver. SET-GRAPHICS establishes a text window restricting character output to the top few lines of the display, and sets mode 5 to enable 4-colour drawing. END-GRAPHICS sets mode 4, thereby clearing the screen and allowing for 40 character lines:

```
(DEFUN SET-GRAPHICS ()
   (VDU 22 5 28 0 5 19 0))

(DEFUN END-GRAPHICS ()
   (VDU 22 4))
```

The read-eval-print loop for graphics use is called GSUPER. It uses ERRORSET to ensure recovery from all kinds of errors. The only significant difference between GSUPER and the loop normally executed by LISP is that

GSUPER calls a graphics print function GPRINT (yet to be defined) to display the results of computations:

```
(DEFUN GSUPER ((U))
   (SET-GRAPHICS)
   (LOOP
      (LOOP
         (LOOP
            (PRINT 'Eval:)
            (SETQ U (ERRORSET (READ)))
            (WHILE (ATOM U)))
         (SETQ U (ERRORSET (EVAL (CAR U))))
         (WHILE (ATOM U)))
      (UNTIL (EQ (SETQ U (CAR U)) 'FIN))
      (ERRORSET (GPRINT U)))
   (END-GRAPHICS))
```

The escape clauses (WHILE (ATOM U)) in the inner loops reflect the fact that ERRORSET returns an atomic result if the protected evaluation fails, otherwise the result of the inner calculation can be recovered as CAR of the value returned by ERRORSET.

GPRINT will have to interpret list structures as pictures. In this package any list which has the atom PIC as its first element will be expected to represent a graphic:

```
(DEFUN PICP (X)
   (AND (NOT (ATOM X))
        (EQ (CAR X) 'PIC)))

(DEFUN GPRINT (X) (COND
   ((PICP X) (CLEAR-SCREEN) (DRAW (CDR X)))
   (T (PRINT X))))

(DEFUN CLEAR-SCREEN ()
   (VDU 18 0 7 18 0 128 16 25 4 0 2 0 2))
```

You will find from the BBC Microcomputer User Guide that the control codes given in CLEAR-SCREEN set the VDU driver to write in white on a black background, with a clear screen and the graphics cursor positioned roughly in the centre of the screen. DRAW is, of course, a further function we will have to design in LISP, and it will have to traverse datastructures emitting VDU control codes in order to live up its name. DRAW is coded here in a way that is very characteristic of LISP:

```
(DEFUN DRAW (X)
   (COND
      ((ATOM X) NIL)
      (T  (  (GET (CAR X) 'DRAW)
                (CDR X) )         )))
```

Atoms are treated as null pictures and are not displayed. Other things are expected to have a picture type as their CAR, and DRAW recovers a function from the property list of this type by using GET. This function is then applied to the rest of the picture data. Extra spacing has been used in the prettyprinting of the above example to try and show that in the final clause of the conditional the expression (GET (CAR X) 'DRAW) is appearing as a function. A typical argument to DRAW will be something like (BOX 100 200) - the code stored in BOX's property list is as follows:

```
(PUT 'BOX 'DRAW
  '(LAMBDA (X (Y))
      (SETQ Y (CADR X))
      (SETQ X (CAR X))
      (PLOT 0 (MINUS (QUOTIENT X 2))
              (MINUS (QUOTIENT Y 2)))
      (PLOT 1 X 0)
      (PLOT 1 0 Y)
      (PLOT 1 (MINUS X) 0)
      (PLOT 1 0 (MINUS Y))
      (PLOT 0 (QUOTIENT X 2) (QUOTIENT Y 2))))
```

The lambda expression gets called with X bound to (100 200), and the first step is to unpick this list into its two constituent numbers and store these in X and Y. A collection of calls to PLOT sends suitable bytes to the screen handler to draw a rectangle of the required size, with its centre near the position of the graphics cursor. Again, the main BBC Microcomputer manual should be consulted for further details of the control codes used. PLOT generates these:

```
(DEFUN PLOT (N X Y)
   (VDU 25 N
      (REMAINDER (SETQ X (PLUS X 16384)) 256)
      (DIFFERENCE (QUOTIENT X 256) 64)
      (REMAINDER (SETQ Y (PLUS Y 16384)) 256)
      (DIFFERENCE (QUOTIENT Y 256) 64)))
```

The adding of 16384 (=2^14) is to make sure that the remainders computed have the required sign. Further DRAW properties for objects other than rectangles and for changing and combining pictures can be provided. A few follow:

```
(PUT 'ADD 'DRAW '(LAMBDA (A)
   (LOOP (WHILE A) (DRAW (CAR A)) (SETQ A (CDR A)))))

(PUT 'SHIFT 'DRAW
  '(LAMBDA (A)
      (PLOT 0 (CADR A) (CADDR A))
      (DRAW (CAR A))
      (PLOT 0 (MINUS (CADR A)) (MINUS (CADDR A)))))

(PUT 'CIRCLE 'DRAW
  '(LAMBDA (X (W) (Z))
      (SETQ X (CAR X))
      (SETQ W (QUOTIENT (TIMES X 7) 10))
      (SETQ Z (DIFFERENCE X W))
      (PLOT 0 X 0)
      (PLOT 1 (MINUS Z) W)
      (PLOT 1 (MINUS W) Z)
      (PLOT 1 (MINUS W) (MINUS Z))
      (PLOT 1 (MINUS Z) (MINUS W))
      (PLOT 1 Z (MINUS W))
      (PLOT 1 W (MINUS Z))
      (PLOT 1 W Z)
      (PLOT 1 Z W)
      (PLOT 0 (MINUS X) 0)))

(PUT 'COLOUR 'DRAW
  '(LAMBDA (X)
      (VDU 18 0 (CADR X)) (DRAW (CAR X))))
```

These properties enable DRAW to cope with arguments such as:

```
(ADD (CIRCLE 200) (BOX 100 300) (BOX 300 100))
```

To interface this facility smoothly with GSUPER it is necessary to provide some user-level functions for adding, shifting and colouring boxes and circles. These spend most of their energy checking that their arguments are indeed flagged as being pictures - they then build the datastructures that DRAW requires:

```
(DEFUN ADD (A B)
   (COND
      ((PICP A) (COND
         ((PICP B)
            (MKPIC (LIST 'ADD (CDR A) (CDR B))))
         (T A)))
      ((PICP B) B)
      (T (LIST A B))))

(DEFUN SHIFT (P X Y)
   (COND
      ((PICP P) (MKPIC (LIST 'SHIFT (CDR P) X Y)))
      (T P)))

(DEFUN CIRCLE (A) (MKPIC (LIST 'CIRCLE A)))

(DEFUN BOX (A B) (MKPIC (LIST 'BOX A B)))

(DEFUN MKPIC (X) (CONS 'PIC X))

(DEFUN WHITE (X) (COLOUR X 3))
(DEFUN YELLOW (X) (COLOUR X 2))
(DEFUN RED (X) (COLOUR X 1))
(DEFUN BLACK (X) (COLOUR X 0))

(DEFUN COLOUR (X N)
   (COND
      ((PICP X) (MKPIC (LIST 'COLOUR (CDR X) N)))
      (T X)))
```

As a demonstration of the graphics language just implemented, try:

```
(MODE 4)
(GSUPER)
(DEFUN SPIDER (N) (COND
   ((MINUSP N) NIL)
   (T (ADD (CIRCLE N)
           (SPIDER (DIFFERENCE N 20))))))
(SETQ WEB (SPIDER 120))
(ADD (YELLOW WEB) (RED (SHIFT WEB 30 40)))
'FIN
```

Given (much) more memory and computing power a graphics extension to LISP in this form could grow into a quite general-design language, for either engineering or artistic use.

23.10 Parsing computer languages

A parser is a program that reads in a string of characters and works out how they should be grouped. Its output will generally be a tree-structure. Computers use parsers to convert programs from the form in which they are typed to one more useful for internal processing. The simple parser given here can cope with arithmetic expressions using the operators +, -, * and /. It knows about brackets, and produces as output prefix-form tree-structures similar to those discussed in section 23.2. When describing its action it is useful to have an exact definition of the class of expressions to be recognised. The rules used here are that an 'expression' is a collection of one or more 'terms' separated by + or - signs. A 'term' is a list of 'factors' separated from each other by * or / signs. A 'factor' is either a 'symbol' or an 'expression' enclosed in brackets. Programmers write rules of this sort in a shorthand form:

```
<expression> =  <term>     |   <expression> + <term>
                           |   <expression> - <term>
<term>       =  <factor>   |   <term> * <factor>
                           |   <term> / <factor>
<factor>     =  <symbol>   |   ( <expression> )
```

where the vertical bar (|) should be read as 'or'.

A parser can be constructed by taking each of these rules and writing it as a function. Thus the following procedure captures the idea that an expression starts with a term, and it can continue absorbing more terms so long as it finds either a + or a -. The function NEXTSYM will have to set the variable CURSYM to be the next item in the input text, and return the current item as its result:

```
(DEFUN EXPRESSION ((TREE))
   (SETQ TREE (TERM))
   (LOOP
      (WHILE (OR (EQ CURSYM '+) (EQ CURSYM '-)) TREE)
      (SETQ TREE (LIST (NEXTSYM) TREE (TERM)))))
```

The last line of EXPRESSION relies on LIST evaluating its arguments from left to right, so that (NEXTSYM) reads past the + or - sign before (TERM) is called to process the following operand. This practice should be viewed with some suspicion because some LISPs reserve the right to evaluate arguments in whatever order seems most convenient to them.

The functions which read terms and factors are remarkably similar in style:

```
(DEFUN TERM ((TREE))
   (SETQ TREE (FACTOR))
   (LOOP
      (WHILE (OR (EQ CURSYM '*) (EQ CURSYM '/)) TREE)
      (SETQ TREE (LIST (NEXTSYM) TREE (FACTOR)))))

(DEFUN FACTOR ((TREE))
   (COND
      ((EQ CURSYM LPAR)
         (NEXTSYM)
         (SETQ TREE (EXPRESSION))
         (NEXTSYM)
         TREE)
      (T (NEXTSYM))))
```

If memory were not at a premium, FACTOR should check that the symbol following the expression it reads is a right bracket, otherwise it should generate an error. As coded here it will accept operator symbols as operands and ignore whatever symbol is found where a closing bracket is wanted. NEXTSYM has to read a symbol and save it in CURSYM, returning the previous value of CURSYM. A full-scale parser would have to read multi-character words and distinguish between names and numbers. For simplicity, the code here treats every character (except blanks) in the input as a separate symbol. Note that an implication of this is that items one might have expected to be numbers, e.g. 6, get read as identifiers and so LISP cannot do arithmetic on them directly:

```
(DEFUN NEXTSYM ((PREV))
   (SETQ PREV CURSYM)
   (LOOP
      (SETQ CURSYM (GETCHAR))
      (PRINC CURSYM)
      (WHILE (EQ CURSYM BLANK) PREV)))
```

The function EXPRESSION is the main entrypoint to this parser, but for it to work CURSYM must have been set to be the first symbol of the input string. The driver function PARSER can do this.

```
(DEFUN PARSER ((CURSYM))
   (NEXTSYM)
   (EXPRESSION))
```

PARSER can then be tested as follows:

```
(PARSER)
2+(A-B)*(4/X+7)
```

There is obviously a big gulf between this parser and the ones that process complete computer languages. However, by roughly trebling its size this code can be converted into one that reads in a somewhat Algol or Pascal-like language and converts it into the structure LISP uses to represent programs. Probably the most unpleasant piece of coding is the one concerned with collecting together sequences of characters to make words and numbers: heavy use will have to be made of CHARACTER, IMPLODE, ORDINAL and the like. Writing parsers that recover well after they have detected syntax errors in their input is hard. When using a small interactive computer it will probably be best to respond to mistakes by calling ERROR to break right out of the parsing process.

23.11 Generating machine code

One of the early surprises the LISP developers had was that a LISP compiler could be written in LISP. The language still remains one in which it is particularly easy to demonstrate the idea of compilation. The compiler described here processes simple arithmetic expressions to generate from them 6502 assembly code which will do the 8-bit computations described. The code generated will not be very efficient, but the program is very short!

To make sense of this code it will be necessary to have some understanding of the machine code that is to be generated, and to recognise the assembler syntax used. The opcodes which can be produced by this compiler are LDA, STA, PHA, PLA, ADC, SBC, AND and ORA. The input to the compiler will be tree-structures such as (+ (& A 8) (- 2 B)).

In the case where its argument is an atomic expression, CG just prints a line of text, either

```
   LDA # argument      if its argument is a number
or LDA argument        otherwise.
```

Otherwise, it assumes that it has a binary operator to deal with. It calls itself recursively to load the first operand into the 6502 accumulator. It saves that value on the stack using PHA while it evaluates the second operand. It then shuffles its registers slightly and finishes by printing an instruction the opcode of which depends on the expression being processed. PUT and GET are used to establish the link between, for example, the operator & and the 6502 opcode AND:

```
(DEFUN CG (X)
   (COND
      ((NUMBERP X) (PRINTC 'LDA BLANK '# BLANK X))
      ((ATOM X) (PRINTC 'LDA BLANK X))
      (T (CG (CADR X))
         (PRINTC 'PHA)
         (CG (CADDR X))
         (PRINTC 'STA BLANK 'TEMP)
         (PRINTC 'PLA)
         (PRINTC (GET (CAR X) 'OPCODE) BLANK 'TEMP))))

(PUT '+ 'OPCODE 'ADC)
(PUT '- 'OPCODE 'SBC)
(PUT '& 'OPCODE 'AND)
(PUT '| 'OPCODE 'ORA)
```

Those familiar with 6502 assembly code will soon spot that this compiler generates incorrect code, in that it does not set or clear the carry flag before carrying out additions or subtractions. Extending it to correct this and to reduce the number of redundant pushes and pops is left to the reader. By making the compiler recognise operators with names like COND and LOOP it can be extended into one for a complete programming language. A complete optimising LISP compiler can be found in the paper by Griss and Hearn mentioned in the Bibliography - versions of it have been used to generate code for a range of machines from micros up to large mainframes.

23.12 Mazes and dungeons

LISP's ability to work with words and complex linked structures makes it a natural language for coding maze-threading adventure games. The skeleton game here shows a possible organisation of a dungeon, and deals with movement and with collecting and discarding property. It does not include magic, special conditions on some actions, or scores. Adding any of these is, however, just a matter of extra programming, together with extra items to be put on property lists. ADVENTURE starts by calling READLINE to flush out the keyboard input buffer. It prints a banner and puts you at the starting position in the maze with your hands empty. It then loops obeying your commands until you reach the end (or centre) of the maze.

```
(DEFUN ADVENTURE NIL
   (READLINE)
   (PRINTC '(WELCOME TO THE MAZE))
   (SETQ LOCATION 'START)
   (SETQ CARRYING NIL)
   (LOOP
      (DISPLAY-POSITION)
      (UNTIL (EQ LOCATION 'FINISH) 'CONGRATULATIONS)
      (MAKE-MOVE (READLINE))))
```

When displaying a position it is necessary to print information describing the location, indicating what is being carried, explaining what further objects can be seen and pointing out which directions it will be possible to move in. Much of this information can be associated with a location by the use of PUT and GET. Coping with exits is done by using the name of a direction (e.g. NORTH) as a property name, and storing with it the name of the place to which that particular direction leads. DISPLAY-POSITION unpicks all these datastructures, making use of MAPC to traverse some of the lists involved.

```
(DEFUN DISPLAY-POSITION NIL
   (PRINT)
   (PRINC 'AT: BLANK)
   (LPRI (GET LOCATION 'DESCRIPTION))
   (MAPC
     '(LAMBDA (ITEM)
         (LPRI (LIST 'YOU 'ARE 'CARRYING ITEM)))
```

```
      CARRYING)
   (MAPC
     '(LAMBDA (ITEM) (LPRI (LIST 'YOU 'SEE ITEM)))
      (GET LOCATION 'OBJECTS))
   (PRINC 'EXITS:)
   (MAPC
     '(LAMBDA (DIR)
         (COND ((GET LOCATION DIR) (PRINC BLANK DIR))))
     '(NORTH SOUTH EAST WEST UP DOWN))
   (PRINT))

(DEFUN LPRI (L)
   (LOOP
      (WHILE L (PRINT))
      (PRINC (CAR L) BLANK)
      (SETQ L (CDR L))))
```

This maze game accepts only single-word commands. MAKE-MOVE tests if this word is the name of an object being carried, the name of an object that can be picked up or the name of a direction that can be used to move from the current location:

```
(DEFUN MAKE-MOVE (WORD)
   (COND
      ((MEMBER WORD CARRYING) (DROP WORD))
      ((MEMBER WORD (GET LOCATION 'OBJECTS))
         (PICK-UP WORD))
      ((AND
         (GET LOCATION WORD)
         (CHARP (GET LOCATION WORD)))
         (MOVE-TO WORD))
      (T (LPRI (LIST WORD 'NOT 'UNDERSTOOD)))))
```

DROP and PICK-UP are then easily written. DROP removes the object being dropped from the list of things that are being carried and stores it as another item present at the current location. PICK-UP does the converse. Both print messages to confirm what they have done:

```
(DEFUN DROP (WORD)
   (SETQ CARRYING (DELETE WORD CARRYING))
   (PUT LOCATION 'OBJECTS
      (CONS WORD (GET LOCATION 'OBJECTS)))
   (LPRI (LIST 'DROPPED WORD)))

(DEFUN PICK-UP (WORD)
   (PUT LOCATION 'OBJECTS
      (DELETE WORD (GET LOCATION 'OBJECTS)))
   (SETQ CARRYING (CONS WORD CARRYING))
   (LPRI (LIST 'GOT WORD)))

(DEFUN MOVE-TO (DIRECTION)
   (SETQ LOCATION (GET LOCATION WORD)))
```

Apart from the service functions DELETE, MEMBER and EQUAL, which are defined in Appendix B, the remaining requirement is for a maze. Here is a very small one. It only has one side-branch, so there is not much real opportunity to get lost in it! First come the descriptions of the rooms in the maze:

```
(PUT 'START 'DESCRIPTION '(THE ENTRY TO A MAZE))
(PUT 'HALL 'DESCRIPTION '(A FINE GOTHIC HALLWAY))
(PUT 'TWIST 'DESCRIPTION '(A TWISTY PASSAGE))
(PUT 'DEAD 'DESCRIPTION '(DEAD END))
(PUT 'CAVE 'DESCRIPTION '(ALADDINS CAVE))
(PUT 'FINISH 'DESCRIPTION '(CASTLE SPLENDID))
```

Now the pathways linking these chambers:

```
(PUT 'START 'NORTH 'HALL)
(PUT 'HALL 'EAST 'TWIST)
(PUT 'TWIST 'UP 'CAVE)
(PUT 'CAVE 'EAST 'DEAD)
(PUT 'CAVE 'WEST 'FINISH)
(PUT 'HALL 'SOUTH 'START)
(PUT 'TWIST 'NORTH 'HALL)
(PUT 'CAVE 'DOWN 'TWIST)
(PUT 'DEAD 'DOWN 'CAVE)
```

Finally, a collection of objects scattered around the dungeon waiting to be found:

```
(PUT 'START 'OBJECTS '(LANTERN LASERGUN))
(PUT 'TWIST 'OBJECTS '(KEYS MAP))
(PUT 'CAVE 'OBJECTS '(TREASURE))
```

Even without the introduction of magic this program can be developed into a challenging game by having a LISP function set up a random maze with a few dozen rooms in it. Following the tradition set by an earlier adventure game, each of these should have as its description

(A MAZE OF TWISTY PASSAGES - ALL ALIKE)

making the task of finding a route through to the finish quite challenging.

Appendix A

Summary of functions provided in Acornsoft LISP

This Appendix lists the identifiers initially available in LISP. More can be added using DEFUN for functions or SETQ for variables. Since the user can add or remove functions and then SAVE a copy of the resulting system, this list should be viewed as a description of a core of basic definitions, and not as a complete listing of everything that could be wanted. In any particular SAVEd version of LISP, the function OBLIST can be used to get a definitive and up-to-date list of the names actually available. The language ROM versions of LISP distributed by Acornsoft can be expected to include all of these functions, and future releases may include yet more facilities. Disk- and cassette-based copies of LISP have much more severe limits on the amount of store available to them, and may not include some of the more specialist functions.

For each identifier listed here there is a header indicating whether it is the name of a function or a variable, and giving an indication of the arguments expected by functions. The words Subr, Fsubr and Expr are used to mean:

Subr:	A built-in function which processes its arguments normally.
Fsubr:	A built-in function with special argument processing, e.g. it guarantees to process arguments from left to right, or it sometimes does not evaluate all of its arguments.
Expr:	A function defined in LISP, not in machine code.

ADD1 Subr

(ADD1 number)

ADD1 increments its argument, which must be numeric. It will fail, recording an arithmetic overflow, if its result would be out of range (i.e. if its input is the largest positive integer LISP can handle, 32767). (ADD1 x) is exactly equivalent to (PLUS x 1), but its use can perhaps result in (slight) reductions in program size. See also SUB1.

AND Fsubr

(AND predicate predicate)

AND can have any number of arguments. It returns T if and only if all its arguments are non-NIL. If any of AND's arguments are NIL then AND itself returns NIL (the LISP representation of 'false'). AND does not necessarily evaluate all its arguments. It goes through the list evaluating them one by one until:

(a) the value of an argument is NIL - the value returned by AND is then NIL;

(b) the end of the argument list is reached, in which case AND returns the value T.

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.

ADVAL Subr

(ADVAL number)

If given a positive argument this function returns information about one of the computer's analogue to digital converters. The arguments 1,2,3 and 4 cause it to return the value from the corresponding converter. If the argument is zero the value returned indicates whether the 'fire' buttons are pressed on any joysticks/games paddles plugged into the machine. The least significant bit is 1 if the left button is pressed, the 2-bit is for the right button. Small negative arguments cause ADVAL to return information about various queues within the computer. Of these the most useful is probably (ADVAL -1) which returns the number of characters that are waiting in the keyboard input buffer.

APPLY Subr

(APPLY function argument-list)

The first argument to APPLY must be either a code-pointer (found as the value of one of the LISP built-in functions) or a lambda expression, such as that stored as a definition set up using DEFUN. The second argument is a list of the arguments to be handed to the given function. APPLY is useful either when the function that is to be called is itself the result of some computation, or when the number or form of arguments to a function have to be computed. The TRACE package in Appendix B illustrates the use of APPLY in a typical setting. The example here shows what APPLY does but cannot illustrate its usefulness.

```
(APPLY CONS '(A B))
```

yields the result of applying the CONS function to the two arguments A and B (literally), and so is equivalent to (CONS 'A 'B). See EVAL.

ASSOC Subr

(ASSOC key association-list)

An association list is a list of dotted pairs of the form ((kl . vl) (k2 . v2) ...), where the keys kl, k2 ... are identifiers, and the corresponding values vl, v2 ... are arbitrary LISP datastructures. ASSOC scans such a list looking for the first instance of a given key in it. If it finds a match, it returns as its value the key-value pair; otherwise, it returns NIL. Thus

```
(ASSOC 'B '((A . 1) (B . 2) (C . 3)))
```

has the value (B . 2).

ATOM Subr

(ATOM object)

ATOM returns T if its argument is any atom, i.e. if it is an identifier, a number or the entry-point of a piece of machine code. Otherwise, i.e. if its argument is a list structure, ATOM returns NIL. Thus

```
   (ATOM 'IDENTIFIER) -> T,
   (ATOM CAR) -> T
        (because the value of CAR is the entrypoint of
        the machine code within LISP that implements
        the CAR function),
   (ATOM (CONS anything anything)) -> NIL,
   (ATOM '(some list)) -> NIL
```

See SUBRP, FSUBRP, NUMBERP, LISTP.

BAND   Subr

```
(BAND number ... number)
```

BAND treats all its arguments as 16-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

BLANK = a space

The initial value of the variable BLANK is the character with a print name containing a single space character.

BLANK is often useful when printing, as in

```
(PRINT A BLANK B BLANK C)
```

which is probably more readable than the equivalent

```
(PRINT A '!  B '!  C)
```

See also LPAR, RPAR, PERIOD, DOLLAR, CR for other variables having as their initial values characters which might be hard to talk about directly.

BNOT Subr

(BNOT number)

BNOT treats its numeric arguments as a 16-bit binary number and complements each bit. The resulting bit pattern is used as BNOT's numeric result. The representation 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 Subr

(BOR number ... number)

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)
```

CALL Subr

(CALL address accumulator)

CALL provides a way of using machine code subroutines from LISP. The first argument is numeric, and supplies the address of the subroutine. If the address is larger than 32767 the equivalent two's-complement negative number must be used. The second argument defines the contents of the microprocessor's accumulator on entry to the routine, and is optional. This argument must also be numeric. The low byte of the number is loaded into the accumulator. CALL returns a number giving the contents of the accumulator when the routine returns. For example:

```
(CALL -32)
```

calls OSRDCH within the operating system, and hence returns a code related to the next character typed. See also '*' and USR for alternative ways of using operating system facilities from within LISP.

CAR   Subr

(CAR list); note also CDR, CAAR, ... CDDDR

The function CAR returns part of the structure that is its argument. It is an error for CAR's argument to be atomic.

If the argument is viewed as a list, then CAR selects its first element. If the structure is being thought of as a general piece of datastructure CAR is the function that follows the first of the two pointers in a node. CDR finds the second pointer in a pair, which corresponds to the tail of a list. The composites CAAR, CADR and so on exist for all combinations of up to three A's and D's, and provide for compound uses of CAR and CDR. For example:

```
(CADR X)  = (CAR (CDR X))
(CDDAR Z) = (CDR (CDR (CAR Z)))
```

For building list structures see CONS and LIST.

CDR   Subr

(CDR list)

See CAR, and chapter 7.

CHARACTER   Subr

(CHARACTER number)

The argument to CHARACTER should be a number in the range 0 to 127. CHARACTER treats this number as the code for a character. It hands back an identifier that has this one character as its printname. Character codes can be found in the BBC Microcomputer User Guide but, for instance, (CHARACTER 13) is a carriage return, (CHARACTER 48) is the character '0', and (CHARACTER 65) is 'A'. See ORDINAL.

CHARCOUNT Expr

(CHARCOUNT item number)

Returns the value of the second argument minus the number of characters required to print 'item', or NIL if it is negative. Numbers are assumed to require six characters. CHARCOUNT is used by the version of SPRINT included in the distributed system. Its definition is as follows:

```
(DEFUN CHARCOUNT (X LEFT) (COND
   ((ATOM X) (COND
      ((GREATERP LEFT (CHARS X))
         (DIFFERENCE LEFT (CHARS X)))))
   (T (LOOP
         (UNTIL (ATOM X) (CHARCOUNT X
            (DIFFERENCE LEFT (COND (X 4) (T -2)))))
         (WHILE (SETQ LEFT (CHARCOUNT (CAR X)
            (SUB1 LEFT))))
         (SETQ X (CDR X))))))
```

CHARP Subr

(CHARP object)

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 Subr

```
(CHARS identifier)
```

returns the number of characters in the identifier given. So, for example,

```
(CHARS 'FOUR)  = 4
(CHARS 'SIX)   = 3
```

CLOCK Subr

```
(CLOCK)
```

This function returns a list of three numbers which represent the time, in hours, minutes and seconds, since either the computer was last reset using CTRL-BREAK, or since the LISP function RESET was called. See TIME and GCTIME.

CLOSE Subr

```
(CLOSE handle)
```

Closes a file, referring to the file by the handle that OPEN generated for it. CLOSE writes out all buffers associated with the given file, and on disk systems causes the disk directory to be updated to reflect any changes that have been made to the length of the file. The special case (CLOSE 0) causes all currently open files to be closed.

COND Fsubr

```
(COND (predicate action action ... ) ... )
```

COND is the main structure provided in LISP for testing and acting on conditions. COND can have any number of arguments. These arguments are treated in a special way.

The LISP interpreter looks at the arguments of COND one by one in the order in which they appear. Each argument is a list. The value of the first member of the list, i.e. the condition, determines the action taken. If the value is NIL, the rest of the list is ignored and LISP goes on to the next argument. If the value is T or anything other than NIL, the remaining members of the list are evaluated one by one. The value of the last member is returned as the value of the COND expression. There is no limit on the number of members of the list.

There are two special cases:

(a) All of the conditions evaluate to NIL. In this case the value of the COND is NIL.

(b) An argument list has no actions, only a condition. Here the value of the condition is returned if it is not NIL.

For example:

```
(COND
   ((EQ X 3) (PRINT 'THREE) 'YES)
   (T 'NO))
```

This will print THREE and return the value YES if the value of X is 3. Otherwise, it will return the value NO. COND expressions very often finish with a T condition to cope with all remaining possibilities. Another example is

```
(COND
   ((OR (MINUSP HOUR) (GREATERP HOUR 23))
      (ERROR '(HOUR NOT TIME OF DAY)))
   ((LESSP HOUR 7) 'NIGHT)
   ((LESSP HOUR 12) 'MORNING)
   ((LESSP HOUR 18) 'AFTERNOON)
   ((LESSP HOUR 22) 'EVENING)
   (T 'NIGHT))
```

This returns the time of day depending on the value of HOUR. There is an error message if HOUR does not represent a valid time.

CONS Subr

(CONS arg1 arg2)

This function CONStructs a new dotted pair out of its arguments. Often the second argument will be a list (maybe the empty list NIL). Then the result of CONS is also a list, and it is obtained by adjoining CONS's first argument to the front of the list. Thus

```
(CONS 'A '(B C D))  = '(A B C D).
```

Another way to describe CONS is in terms of CAR and CDR. If X is a list then (CONS (CAR X) (CDR X)) is another list that will look the same as X when printed, but which uses a new piece of memory to hold its first link. For example,

```
(CONS 'ALPHA 'OMEGA)  = (ALPHA . OMEGA)
(CONS 'XXX NIL)       = (XXX . NIL) = (XXX)
(CONS 'C '(D))        = (C D)
(CONS '(D) 'C)        = ((D) . C)
```

CR Variable

CR = a newline character

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 13). See also BLANK.

DEFUN Expr

(DEFUN function-name parameters body ...)

DEFUN is 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))
```

DIFFERENCE Subr

```
(DIFFERENCE number number)
```

The value returned is that obtained by subtracting the second argument from the first. For instance, if A has the value 77, then (DIFFERENCE A 23) has the value 54. An error will be reported if either argument is non-numeric or if the result would be outside the range of numbers that LISP can represent. See PLUS and MINUS.

DOLLAR Variable

DOLLAR = the character $

See BLANK.

EDIT Expr

```
(EDIT function-name)
```

Details of EDIT together with notes on its use are given in section 23.8. The basic commands provided are:

A	move to CAR field
B	back up one level (i.e. inverse of A, D)
C s	insert the expression s in front of current list
D	move to CDR field
R s	replace current expression with s
X	remove head of current list
<return>	prettyprint current expression

ENVELOPE Subr

(ENVELOPE <14 numbers>)

ENVELOPE is used with SOUND to control the characteristics of notes played on the four sound channels available on the BBC Microcomputer. It takes 14 arguments, all of which should be numbers in the range 0 to 255. A detailed discussion of the meaning of these numbers can be found in the User Guide supplied with the computer, where the following suggestion for a police siren is given:

```
(ENVELOPE 2 1 2 -2 2 10 20 10 1 0 0 -1 100 100)
```

EOF Subr

(EOF handle)

Detects whether an end-of-file marker has been reached when reading. It returns T if so, and NIL if not. The argument must be a file handle obtained from OPEN.

EQ Subr

(EQ arg1 arg2)

EQ will return T if one of the following conditions applies to its two arguments:

a) They are the same identifier.
b) They are equal numbers.
c) They are identical lists in LISP memory.

Otherwise, EQ returns NIL. For example:

```
(EQ 4 4)          = T
(EQ 'FRED 'FRED)  = T
(EQ '() NIL)      = T   (because () and NIL are the
                         same object)
(EQ 'FRED 4)      = NIL
```

For a pair of lists to be EQ they must share the same memory as well as being identical in form. Usually EQUAL (see Appendix B) is more useful on lists.

ERROR   Subr

(ERROR message message ...)

ERROR behaves like PRINT in that it displays its argument 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 L is a list before attempting to find its CDR.

```
(COND
   ((ATOM L) (ERROR L BLANK 'NOT BLANK 'LIST))
   (T (CDR L)))
```

ERRORSET   Fsubr

(ERRORSET expression)

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)))
```

See MESSON and MESSOFF for control over the amount of diagnostic information printed when errors occur.

EVAL Subr

(EVAL expression)

EVAL returns its argument evaluated one extra time. The argument can either be an identifier, in which case EVAL treats it as the name of a variable and looks up its value, or a list, in which case EVAL interprets it as a function application. Thus after

```
(SETQ EXPRESSION '(PLUS 2 3))
```

the value of EXPRESSION is the list (PLUS 2 3) and the value of (EVAL EXPRESSION) is 5.

EXPLODE Subr

(EXPLODE identifier)

The argument of EXPLODE should be an identifier, and the result will be a list of the characters making up its name. For instance, (EXPLODE 'BOMB) is the list (B O M B). See IMPLODE for the converse operation, and CHARS for a way of just counting the characters in an identifier's name.

F Variable

F = NIL

The initial value of F is NIL. It is intended that F is used for the logical value 'false', just as T is used for 'true'. Those who use F in this way should avoid using it as an ordinary variable.

FSUBRP Subr

(FSUBRP object)

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 Appendix. 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 Subr

(GCTIME)

Returns the time, in units of 1/100 second, which has been spent carrying out garbage collection. The recorded time is cleared to zero by RESET. See also TIME.

GET Subr

(GET identifier property-name)

GET searches the property list of the identifier for the given property name. If the name is found, the property is returned. If there is no such property the value of GET is NIL. Properties are placed on property lists using PUT. So for instance, after (PUT 'GET 'DESCRIPTION '(RECOVER ITEM FROM PROPERTY LIST)) the value of (GET 'GET 'DESCRIPTION) would be (RECOVER ITEM FROM PROPERTY LIST). Note that it is not possible to distinguish between an absent property and one that does exist but which has the value NIL.

GETCHAR Subr

(GETCHAR)

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, ORDINAL.

GREATERP Subr

(GREATERP number number)

The two numeric arguments are compared, and GREATERP returns T if the first is strictly larger than the second, and NIL otherwise. An error will be generated if either argument is non-numeric. See also LESSP and MINUSP.

IMPLODE Subr

(IMPLODE list-of-characters)

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
```

LAMBDA special identifier

(LAMBDA arg-list body ...)

LAMBDA is used as the first member of Expr function definitions. A list that starts with the identifier LAMBDA can be treated as a function, and such unnamed functions may be useful in association with the functions MAP and MAPC.

LESSP Subr

(LESSP number number)

LESSP returns T if its first argument is strictly less than its second. Otherwise, it returns NIL. Both arguments must be numeric. For example:

```
(LESSP 5 10)  = T
(LESSP 6 6)   = NIL
```

LINEWIDTH Variable

LINEWIDTH = 31

LINEWIDTH is a variable used by SPRINT as the limit to the number of characters that can be displayed on one line. It may be useful to change its value if an 80-column screen mode is selected or if output is being sent to a printer.

LIST Subr

(LIST arg1 arg2 ...)

LIST can have up to 28 arguments. It makes up a list that has the argument values as its members. If there are no arguments LIST returns the empty list, i.e. NIL. For example:

```
(LIST 'A 3 '(X Y))  = (A 3 (X Y))
(LIST)              = NIL
(LIST (LIST 'O))    = ((O))
```

The structures LIST builds could also be built using CONS. So, for example, the effect of (LIST 'A 3 '(X Y)) could be achieved by writing

```
(CONS 'A (CONS 3 (CONS '(X Y) NIL)))
```

where the direct use of CONS necessitates the mention of the NIL that terminates the list being built.

LISTP Subr

(LISTP object)

LISTP returns T if the argument is a list or dotted pair, and NIL if the argument is an atom. It is the opposite of ATOM in that, for any x, (NULL (ATOM x)) is equivalent to (LISTP x).

```
(LISTP (CONS <anything> <anything>))  = T
(LISTP 3)                             = NIL
```

LOAD Subr

(LOAD filename)

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

LOOP Fsubr

(LOOP action action ...)

This function is described in chapter 17, and is used in association with the functions UNTIL and WHILE. It provides for the repetitive execution of a set of LISP commands. For example,

```
(LOOP
   (UNTIL (ATOM (ERRORSET (PRINT (EVAL (READ)))))
      'DONE))
```

is a loop that obeys (PRINT (EVAL (READ))) until an error is detected and trapped by ERRORSET.

LPAR Variable

LPAR = the character (

LPAR has the identifier '(' as its value. See BLANK and RPAR.

MAP Subr

(MAP function list)

Each item in the list is handed to the function as an argument, the resulting values are discarded. MAP could have been defined in LISP as:

```
(DEFUN MAP (FN L) (COND
   ((NULL L) NIL)
   (T (FN (CAR L)) (MAP FN (CDR L)))))
```

MAP always returns a NIL value. It is therefore only of use with functions that have side-effects. Usually MAPC is more useful than MAP. For example, to print each list item on a separate line:

```
(MAP PRINT '(ONE TWO THREE))
```

MAPC Subr

(MAPC function list)

Each item in the list is handed to the function as an argument, and the resulting values are made up into a list. MAPC could have been defined in LISP as:

```
(DEFUN MAPC (FN L) (COND
   ((NULL L) NIL)
   (T (CONS (FN (CAR L)) (MAPC FN (CDR L))))))
```

For example, to double all the numbers in a list, one could use

```
(MAPC '(LAMBDA (X) (TIMES 2 X)) '(1 3 4 2 0))
```

MESSOFF Subr

(MESSOFF number)

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 message 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 Subr

(MESSON number)

See MESSOFF.

MINUS Subr

(MINUS number)

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

MINUSP Subr

(MINUSP number)

MINUSP returns the value T if its argument is a negative number, and NIL otherwise. For example, (MINUSP -4) = T, (MINUSP 7) = NIL. MINUSP returns NIL if its argument is non-numeric, so (MINUSP 'IDENTIFIER) = NIL.

MODE Subr

(MODE number)

Some versions of LISP provide this function. It is called with an argument which is an integer in the range 0 to 7, and arranges to reset the computer's screen mode accordingly. The mode change is delayed until LISP is next ready to display its 'Evaluate:' prompt. LISP takes account of the potential change in storage requirements for the new screen mode. If an attempt is made to select a high-resolution screen mode when too much LISP data is active an error will occur. When available, the use of MODE is much safer than the direct use of VDU for changing screen modes.

NIL special identifier

NIL = NIL

NIL is a special identifier, which has the value NIL and is used in LISP to represent empty lists and the value 'false'. The input syntax () is a synonym for NIL.

NOT Subr

(NOT truth-value)

NOT returns T if its argument is NIL (i.e. 'false') and NIL otherwise. It is generally used where a truth value is to be complemented. See NULL.

NULL Subr

(NULL object)

NULL returns T if its argument is NIL, and NIL otherwise. It could therefore have been defined as

```
(DEFUN NULL (X) (EQ X NIL))
```

NULL is frequently used as a test for an empty list. Note that NULL and NOT are identical in behaviour - the two functions exist solely to allow the user to indicate whether a test is being thought of as one on list structures or on truth values. Examples:

```
(NULL T)         = NIL
(NOT NIL)        = T
(NOT (NULL 55))  = T
```

NUMBERP Subr

(NUMBERP object)

NUMBERP returns T if its argument is a number. Otherwise the value is NIL. For example,

```
(NUMBERP (PLUS 3 4))  = T
(NUMBERP 'NUMBER)     = NIL
```

and in Acornsoft LISP

```
(NUMBERP '12345678901234567890)  = NIL
```

because the reader treats overlong strings of digits as identifiers, not numbers.

OBLIST Subr

(OBLIST)

OBLIST 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. The order of the atoms in the list is related to the order in memory. This in turn is dependent on the order in which they were created. Inspection of the identifiers in (OBLIST) provides definitive information about what functions are available in any particular LISP image. The word OBLIST is short for OBject LIST.

ONEP Subr

(ONEP number)

ONEP tests if its argument is the number 1, and if so returns T. Otherwise, it returns NIL. Thus (ONEP x) is exactly equivalent to (EQ x 1). See also ZEROP for a shorthand test for the number 0.

OPEN Subr

```
(OPEN filename mode)
```

The OPEN function opens the named file for input or output. If the 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 Fsubr

```
(OR arg1 arg2 ... )
```

OR returns T provided that at least one of its arguments is non-NIL. It evaluates its arguments one at a time from the lefthand end until it finds one the value of which is not NIL. OR then returns T, without evaluating the remaining arguments. If OR reaches the end of the argument list without finding a non-NIL value it returns NIL. For example,

```
(OR T (QUOTIENT 1 0) (CAR NIL))
```

has the value T, and the calls to QUOTIENT and CAR (which would cause errors if evaluated) never get performed.

```
(OR (LISTP 33) (ATOM '(P Q R)))  = NIL
(OR)                             = NIL
(OR (NOT x) (NOT y))             = (NOT (AND x y))
```

See also NOT and AND.

ORDINAL Subr

(ORDINAL identifier)

ORDINAL returns the numeric ASCII code for the first character in the name of the identifier that is its argument. It is an error for the argument to be anything other than an identifier. An identifier with an empty name can be created by, for example, (IMPLODE NIL). ORDINAL returns the value 0 if applied to this special atom. Examples:

```
(ORDINAL 'ALPHABET)  = 65
(ORDINAL CR)         = 13
(ORDINAL 33)         gives an error
```

See also CHARACTER.

PEEK Subr

(PEEK address)

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. Addresses greater than 32767 must be represented by their equivalent negative two's-complement numbers. Acornsoft make no guarantee that different releases of their system will use the same addresses to hold the same quantities, and so programmers who rely on PEEK do so at their own risk and should not, in general, expect their programs to be appreciated.

PERIOD Variable

PERIOD = the character .

PERIOD has as its value the identifier consisting of a single full stop. See BLANK, LPAR and RPAR.

PLIST Subr

```
(PLIST identifier)
```

PLIST returns the property list of an identifier. If the identifier has no properties it will return NIL. Properties should normally be established and accessed using PUT and GET, but PLIST can be useful when checking the behaviour of an errant program. In Acornsoft LISP property lists are structured as lists of dotted pairs of the form

```
((propertyname . value) (propertyname . value) ... )
```

So, for instance, after

```
(PUT 'CELLAR 'EAST 'TROLL-ROOM)
(PUT 'CELLAR 'SOUTH 'LOW-PASSAGEWAY)
```

the property list of CELLAR is given as

```
(PLIST 'CELLAR)  = ((SOUTH . LOW-PASSAGEWAY) (EAST .
TROLL-ROOM))
```

PLUS Subr

```
(PLUS number number ... )
```

PLUS returns the sum of all of its arguments, which must all be numbers. PLUS can have up to 28 arguments, and it will report an error if its calculations lead to an arithmetic overflow. For instance,

```
(PLUS 6 2 -3)  = 5
(PLUS)         = 0
```

See DIFFERENCE, MINUS and TIMES.

POINT Subr

(POINT x-coord y-coord)

POINT reports to its caller on the status of the computer's display. It is given numeric arguments: an x-coordinate in the range 0 to 1279 and a y-coordinate in the range 0 to 1023. Its result is the logical colour present at the indicated point on the screen. If the arguments are out of range, POINT returns -1.

POKE Subr

(POKE address number)

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.

PRIN Subr

(PRIN arg1 arg2 ...)

PRIN behaves just like PRINC, except that it attempts to make its output re-readable to LISP by inserting an escape character (!) before any special characters present in the identifiers it is given. Thus (PRIN LPAR BLANK RPAR) prints the message "!(! !)".

PRINC Subr

(PRINC arg1 arg2 ...)

PRINC prints its arguments one by one, starting from the current printing position. All the arguments get evaluated before any of the printing occurs. PRINC does not put any blanks between the objects it displays. Layout is the responsibility of the user, and can be arranged by making some of the items printed CR and BLANK. The value PRINC returns is that of its last argument. The output PRINC generates consists of just those characters present in the identifiers it prints.

Thus (PRINC LPAR BLANK RPAR) prints "( )". See also PRIN, PRINT and PRINTC.

PRINT Subr

```
(PRINT arg1 arg2 ... )
```

PRINT behaves as PRIN, but it forces a fresh line of output after the output it produces. The simple call (PRINT) has the same effect as (PRINC CR), i.e. it starts a fresh line of output, or prints a blank line.

PRINTC Subr

```
(PRINTC arg1 arg2 ... )
```

PRINTC is like PRINC except that it prints a newline at the end of its output.

PROGN Fsubr

```
(PROGN action action ... )
```

PROGN evaluates its arguments one by one and returns the value of the last one. It is useful in that it allows several expressions to be evaluated where the syntax of LISP would otherwise allow for just one. For example,

```
(PROGN (PRINT 'MESSAGE) 'RESULT)
```

prints MESSAGE and returns the value RESULT.

```
(COND
   ((PROGN
      (SETQ X (GET Y 'PROPERTY))
      (PRINT 'PROPERTY '= X)
      (ATOM X)) ... )
   ...
```

uses a PROGN in the predicate of a COND expression so that a sequence of calculations and trace print statements can precede the predicate (ATOM X).

PUT Subr

(PUT identifier property-name value)

PUT places the value on the property list of the given identifier, referenced by the property name. The value of the PUT expression is the value stored, i.e. the third argument. An identifier never has two properties with the same name: a second PUT with the same property name will replace the old property. Values saved using PUT are normally retrieved using GET, but see also PLIST.

QUOTE Fsubr

(QUOTE arg)

QUOTE returns its single argument unevaluated. The form (QUOTE anything) is LISP's internal representation for what is normally written as 'anything.

```
(QUOTE (ARBITRARY LIST))  = (ARBITRARY LIST)
(QUOTE QUOTE)   = 'QUOTE  = QUOTE
'''X                      = (QUOTE (QUOTE X))
```

QUOTIENT Subr

(QUOTIENT number number)

QUOTIENT returns the integer part which results from dividing its first argument by its second. Both arguments must be numbers, and the second must be non-zero. For example,

```
(QUOTIENT 22 7)    = 3
(QUOTIENT -400 6)  = -66
```

See also REMAINDER.

READ Subr

(READ)

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 READLINE and GETCHAR.

READLINE Subr

(READLINE)

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 identifier, 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 Subr

(RECLAIM)

Calling RECLAIM forces LISP to perform a garbage collection. The value returned is always NIL. If garbage collection messages are enabled (see MESSON) this will give an indication of how much free memory is left. RECLAIM is called automatically when LISP finds that its memory has become cluttered. There should not be any circumstances where user calls to it alter the amount of data that LISP is able to handle. See GCTIME.

REMAINDER Subr

(REMAINDER number number)

REMAINDER returns the remainder from the division of its first argument by its second. Its result is always positive, and for any numeric P and Q it will always be the case that

```
P = (PLUS (TIMES Q (QUOTIENT P Q))
          (REMAINDER P Q)).
```

For example,

```
(REMAINDER 22 7)    = 1
(REMAINDER -400 6)  = 4
```

REMPROP Subr

(REMPROP identifier property-name)

REMPROP will remove a property from the property list of an identifier. The value is T if there was originally such a property on the list, and NIL if there was not. For example, after

```
(PUT 'EAGLE 'IS-A 'BIRD)
(PUT 'CROCODILE 'IS-A 'REPTILE)
```

the value of each of (REMPROP 'EAGLE 'IS-A) and (REMPROP 'CROCODILE 'IS-A) would be T the first time they were evaluated and NIL thereafter. Furthermore, after even the first call (GET 'EAGLE 'IS-A) would return NIL and the property IS-A would not be seen in (PLIST 'EAGLE).

RPAR Variable

RPAR = the character)

RPAR has as its value the identifier ')'. See BLANK and LPAR.

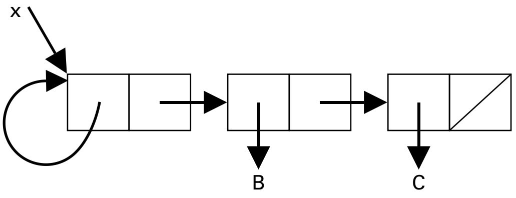
RPLACA Subr

(RPLACA list replacement-car)

RPLACA overwrites the CAR field of its first argument with its second argument. It returns the updated list. An error will occur unless the first argument is a list or dotted pair, but there are no restrictions on the second argument. RPLACA should be used with caution as it actually alters the list cell in memory. Therefore all datastructures using that cell will be changed. Furthermore, RPLACA can be used to create looped structures which will give the LISP print routines trouble. Examples:

```
(SETQ X '(A B C))
   (RPLACA X 'NEW)        X now has the value (NEW B C)
   (RPLACA X X)
```

After the last call to RPLACA, X refers to the list structure



and when LISP attempts to print it it will display

```
((((((((((((((((((((((((((((((((((((((((((((((
```

and so on, followed eventually by an error message.

RPLACD Subr

(RPLACD list replacement-cdr)

RPLACD overwrites the CDR field of its first argument. It is therefore similar to RPLACA, and the same cautions apply. For example:

```
    (SETQ X '(A B C))
    (RPLACD (CDR X) '(1 2))
leaves X with the value (A B 1 2), and then
    (RPLACD (CDDR X) X)
gives a cyclic structure that will print as
    (A B 1 A B 1 A B 1 A B 1 ...
```

RESET Subr

(RESET)

RESET clears the counters used by TIME and GCTIME. 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.

SAVE Subr

(SAVE filename)

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 'EXTEND)
```

saves a file called EXTEND, which might perhaps contain a collection of commonly-used definitions which will be wanted at the start of most subsequent LISP sessions.

SED Expr

```
(SED expression)
```

SED is a subfunction called by the LISP structure editor EDIT. Chapter 23 contains details of how it is used.

SET Fsubr

```
(SET identifier value)
```

SET evaluates both of its arguments. The first must evaluate to an identifier, and the value of this identifier is changed to whatever SET was given as its second argument. This value is returned. In most circumstances LISP programmers will want to assign values to known, not computed, variables, and SETQ will be more useful than SET. For example:

```
(SET 'ONE 'THIS)
(SET ONE 'THAT)
```

leaves a variable ONE with the value THIS, and a variable THIS with the value THAT.

SETQ Fsubr

```
(SETQ identifier value)
```

SETQ is the normal LISP assignment operator. Its first argument is an identifier to be updated, and the second is an arbitrary expression. SETQ returns as its value the quantity assigned to the variable. For example:

```
(SETQ X (PLUS 25 -20))
```

has value 5 and also sets X to the value 5.

SOUND Subr

(SOUND channel envelope pitch duration)

All four arguments for SOUND should be numbers. Calling the function causes one of the computer's four sound generator channels to become active - the first argument to SOUND specifies which. If the second argument is positive it selects one of the four possible envelopes, which must have been set up using ENVELOPE. Negative second arguments specify tones of uniform loudness, with -15 being loud and -1 being barely audible. The pitch of a note is specified in units of a quarter tone: middle 'C' corresponds to the number 53. The duration of sounds is given in units of 1/100 second. Further details of how sounds can, for instance, be controlled to synchronise the start of two notes, can be found in the BBC Microcomputer User Guide.

SUBRP Subr

(SUBRP object)

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.

SPRINT Expr

(SPRINT expression indentation)

SPRINT prints its first argument with the list structure neatly displayed. The second argument is optional and, if present, specifies an indentation to be applied to the display. It should be zero or a positive number. The version of SPRINT distributed with LISP is a variant on the code explained in chapter 23, and it uses the auxiliary functions XTAB and CHARCOUNT. SPRINT is defined as:

```
(DEFUN SPRINT (X (N . 0)) (COND
   ((OR (ATOM X)
        (CHARCOUNT X (DIFFERENCE LINEWIDTH N)))
           (PRIN X))
   (T (PRINC LPAR)
      (SPRINT (CAR X) N)
      (SETQ N (PLUS N 3))
      (LOOP
         (SETQ X (CDR X))
         (COND ((AND X (ATOM X)) (PRINC PERIOD X)))
         (UNTIL (ATOM X) (PRINC RPAR))
         (XTAB N)
         (SPRINT (CAR X) N)))))
```

T   special identifier

T = T

T has the value T. It is used to represent the logical value 'true'. Those LISP functions that return Boolean results all return T for 'true' and NIL for 'false'. Thus

```
(NUMBERP 2)  = T
(NULL NIL)   = T
(NOT T)      = NIL
```

TIME   Subr

(TIME)

TIME returns the time spent in LISP (excluding any time spent in garbage collection) since RESET was last called. The result is in units of 1/100 second. The restricted range of LISP arithmetic means that the count maintained by TIME wraps round after 5 or 10 minutes, and so if the function is being used for performance measurements it is best to call RESET immediately before the test run starts, and to arrange that the experiment completes itself inside a minute or two. The function CLOCK can be used to measure periods too long for the direct use of TIME.

TIMES Subr

(TIMES number number ...)

TIMES returns the product of all of its arguments. The arguments must all be numeric, and the evaluation of their product must not give rise to overflow. There can be up to 28 arguments. For example:

(TIMES 28 4 -2) = -224

See also PLUS, DIFFERENCE, QUOTIENT and REMAINDER.

UNDEFINED special identifier

UNDEFINED = UNDEFINED

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 any 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. To remove an identifier from LISP effectively, use (SETQ UNWANTED-ATOM UNDEFINED) after having used REMPROP to remove all properties from the unwanted atom.

UNTIL Fsubr

(UNTIL condition value ...)

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 an example.

USR Subr

(USR address a x y p)

USR calls the machine code function found at the specified address, with the computer's registers set to values provided in the four numeric arguments a, x, y and p. It returns a list as its result. The list always has exactly four members, and they will all be numbers in the range 0 to 255 representing the state of the processor's registers when it returned from the subroutine. USR is a generalised form of CALL, and can be used to access many of the BBC Microcomputer's operating system facilities.

VDU Subr

(VDU number number ...)

VDU expects all its arguments to be numbers, and sends them sequentially to the screen-handler of the BBC Microcomputer. If the numbers are in the range 0 to 31 they will have special effects, such as defining new soft characters, changing screen colours and plotting points, lines and triangles. Numbers in the range 32 to 127 correspond to printable characters, according to the same code used with CHARACTER and ORDINAL. See the BBC Microcomputer <u>User Guide</u> for further details of control codes available. In the following examples it is assumed that LISP is being run with a graphics screen mode, say mode 5:

(VDU 12)	clear the screen
(VDU 22 4)	change screen mode to 4
(VDU 19 1 3 0 0 0)	change meaning of colour number 1

If VDU is used to change screen mode to one that requires more memory than LISP has set aside for it, LISP may get overwritten. See MODE.

WHILE   Fsubr

(WHILE condition value ...)

WHILE is used in conjunction with LOOP. It is similar to UNTIL, except that the predicate must evaluate to NIL for the values to be processed and the LOOP to terminate.

WRITE   Subr

(WRITE handle arg1 arg2 ...)

WRITE is like PRINT, except that it directs its output to the file the handle of which is specified by its first argument. See OPEN and CLOSE.

WRITE0   Subr

(WRITE0 handle arg1 arg2 ...)

WRITE0 is like PRIN, except that it directs its output to the file specified by its first argument.

XTAB   Expr

(XTAB number)

Prints the specified number of spaces at the start of a new line. XTAB is used by SPRINT, and its definition is as follows:

```
(DEFUN XTAB (N)
   (PRINT)
   (LOOP
      (UNTIL (MINUSP (SETQ N (SUB1 N))))
      (PRINC BLANK)))
```

ZEROP   Subr

(ZEROP number)

ZEROP returns T if its argument is the number 0. Otherwise, it returns NIL. (ZEROP x) is exactly equivalent to (EQ x 0). See ONEP.

*   Subr

(* operating-system-command)

The function '*' can be used to invoke any command known by the BBC Microcomputer. Its argument is an identifier the name of which is used to specify the command. If the operation requested corrupts the memory used by LISP then LISP cannot continue operation after execution of the *. For instance:

(* 'BASIC)	leave LISP, selecting BASIC as next language.
(* 'CAT)	Inspect file directory.
(* 'EXEC! FILE)	Read contents of FILE as if characters in it had been typed at keyboard.

!   special character

!(!! !) !.

The character ! (exclamation mark) is used to allow the input of identifiers with names containing punctuation characters. Any unusual character can be prefixed by a !, and then the reader includes it in a name just as if it had been a letter. Thus to talk about brackets, write !(and !) (or use LPAR and RPAR). !! represents a single exclamation mark.

Appendix B

Some useful functions that are not built into Acornsoft LISP

Large-scale LISP implementations contain many hundreds, sometimes thousands, of built-in functions. A microcomputer implementation cannot be expected to provide all of them, even though each of the functions will be useful in some applications. Fortunately, many of the fairly standard utilities can be coded in LISP: this chapter provides definitions for some frequently-used functions. All of these definitions are short, and no attempt will be made to explain them. In many cases reconstructing the code for these operations from the required behaviour can provide useful practice in designing short fragments of LISP.

APPEND

If x and y are two lists, (APPEND x y) is the list obtained by putting all the elements of y after those of x. Thus (APPEND '(p q) '(r s)) is the list (p q r s).

```
(DEFUN APPEND (A B) (COND
   ((NULL A) B)
   (T (CONS (CAR A) (APPEND (CDR A) B)))))
```

CONCAT

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.

```
(DEFUN CONCAT (A B)
   (IMPLODE (APPEND (EXPLODE A) (EXPLODE B))))
```

DELETE

The first argument to DELETE is expected to be the same as some member of the list that is given as the second argument. The result returned is a list with the first instance of this member removed.

```
(DEFUN DELETE (A L) (COND
   ((NULL L) NIL)
   ((EQUAL A (CAR L)) (CDR L))
   (T (CONS (CAR L) (DELETE A (CDR L))))))
```

DIGIT

DIGIT tests if its argument is a digit. If it is, it returns the corresponding decimal value; otherwise, it returns NIL.

```
(DEFUN DIGIT (CH)
   (SETQ CH (ORDINAL CH))
   (COND
      ((OR (LESSP CH 48)
           (GREATERP CH 57)) NIL)
      (T (DIFFERENCE CH 48))))
```

This code uses the fact that on the BBC Microcomputer the internal codes for the characters '0', '1', ... '9' are 48, 49 ... 57.

EQUAL

The basic LISP function EQ compares two atoms for equality. When applied to list structures it checks if the pointers to them are identical. EQUAL compares list structures to see whether they have the same shape and the same atoms as leaves.

```
(DEFUN EQUAL (A B) (COND
   ((EQ A B) T)
   ((OR (ATOM A) (ATOM B)) NIL)
   ((EQUAL (CAR A) (CAR B)) (EQUAL (CDR A) (CDR B)))
   (T NIL)))
```

FLATTEN

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

```
(DEFUN FLATTEN (A (L)) (COND
   ((NULL A) L)
   ((ATOM A) (CONS A L))
   (T (FLATTEN (CAR A) (FLATTEN (CDR A) L)))))
```

LAST

(LAST x) returns the last item in the list x.

```
(DEFUN LAST (L) (COND
   ((NULL (CDR L)) (CAR L))
   (T (LAST (CDR L)))))
```

MEMBER

This function checks if its first argument is one of the items in the list that is its second argument. It uses EQUAL as a test for equality.

```
(DEFUN MEMBER (A L) (COND
   ((NULL L) NIL)
   ((EQUAL A (CAR L)) T)
   (T (MEMBER A (CDR L)))))
```

NUMOB

NUMOB is rather like 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:

```
(DEFUN NUMOB (L (N . 0))
   (LOOP
      (WHILE L N)
      (SETQ N (PLUS (TIMES N 10)
                    (DIFFERENCE (ORDINAL (CAR L)) 48)))
      (SETQ L (CDR L))))
```

RDF

(RDF filename) reads and evaluates all the expressions in the given file, stopping when it hits an error.

```
(DEFUN RDF (NAME (HANDLE))
   (SETQ HANDLE (OPEN NAME T))
   (LOOP
      (UNTIL (ATOM (ERRORSET (EVAL (READ HANDLE)))))
      (CLOSE HANDLE))
```

REVERSE

(REVERSE '(A B C D)) has the value (D C B A). For any list x, (CAR (REVERSE x)) = (LAST x).

```
(DEFUN REVERSE (X (W))
   (LOOP
      (WHILE X W)
      (SETQ W (CONS (CAR X) W))
      (SETQ X (CDR X))))
```

SUBST

(SUBST a b c) substitutes a for b throughout the expression c.

```
(DEFUN SUBST (A B C) (COND
   ((EQ B C) A)
   ((ATOM C) C)
   (T (CONS (SUBST A B (CAR C))
            (SUBST A B (CDR C))))))
```

TRACE

After a call (TRACE fname), any time the function fname is called a message will be printed showing what arguments it has been given, and what result it returns. This can be invaluable when it is suspected that the function is misbehaving in some way, but the pattern of calls to it is hard to predict. UNTRACE restores the function to its original state:

```
(DEFUN TRACE FN
   (SETQ FN (CAR FN))
   (PUT FN 'OLDDEF (EVAL FN))
   (SET FN (SUBST FN 'FN
     '(LAMBDA *X
          (SETQ *X (MAPC EVAL *X))
          (PRINTC 'FN BLANK *X)
          (SETQ *X (APPLY (GET 'FN 'OLDDEF) *X))
          (PRINT 'FN '= *X)
          *X)))
   FN)
```

An attempt to trace the same function twice, or to trace any of the functions used internally in this trace package, will produce confusing and perhaps disastrous results.

UNTRACE

UNTRACE undoes the effect of TRACE, and is called in a similar way.

```
(DEFUN UNTRACE FN
   (SETQ FN (CAR FN))
   (SET FN (GET FN 'OLDDEF))
   (REMPROP FN 'OLDDEF)
   (LIST FN 'UNTRACED))
```

Appendix C

Error codes

In the cassette- and disk-based version of LISP errors may be reported in terms of numeric error codes. The language ROM version of the system is less short of space, and is therefore able to provide textual messages. Even then, the values returned by ERRORSET when it traps an error will be the numeric codes, so this section lists these and describes circumstances which may provoke them.

<u>0,1,2 - Insufficient memory</u>

0	No space for stack
1	No space for variable binding
2	No space for new cell

These three errors all mean that LISP has filled the available workspace, invoked the garbage collector, but has been unable to recover enough space to continue with the calculation. This situation can arise for a number of reasons:

(a) The program needs more memory than is available.

(b) The program is in a loop, repeatedly allocating more store.

(c) The workspace is filled with a large datastructure or program from earlier work.

No backtrace is printed, because LISP would not be able to find the workspace needed by the print routines. Some suggested solutions are:

(a) Trace selected functions and check the program looking for non-terminating loops or uncontrolled recursion.

(b) Remove all unnecessary function definitions, variables and properties by setting identifiers to UNDEFINED and calling REMPROP.

(c) Replace some recursions by equivalent use of LOOP. This saves some memory.

(d) Reduce the size of datastructures used in the program, perhaps by redesigning it to be more store-efficient.

(e) Upgrade from cassette- or disk- LISP to a ROM based version, or from a ROM based version to one running on a second processor. Each of these steps makes a very large increase in the space available for user code, and brings many more LISP applications within reach of your computer.

3 - Not enough arguments for an Fsubr

This has been caused by an incorrectly written program. In the backtrace the second ARG will show the offending expression. Check Appendix A to see how many arguments the function should have, and correct the program.

4 - Interrupt during evaluation

This error is provoked by pressing ESCAPE while LISP is running. The backtrace shows what was happening at the time.

5 - Expression in function position not a function

The first member of a list to be evaluated should be a function. This error occurs either when a function has not been defined or when a badly-constructed piece of program is found.

The first ARG in the backtrace shows the stage LISP had reached when it gave up trying to produce a function: this will be UNDEFINED for undefined functions. The second ARG shows the list being evaluated, and the first item in this will be the invalid function.

6 - Wrong number of arguments for Expr or Subr

There are a number of ways that this error can occur:

(a) An attempt was made to call a function with more than 28 arguments.

(b) A predefined function (Subr) has been called with fewer arguments than it requires.

(c) A function defined in LISP has been called without enough arguments to satisfy all the non-optional parameters in its bound variable list.

The second ARG in the backtrace shows the offending expression.

7 - Syntax error in lambda expression

This error occurs either while binding the arguments of a function or while executing it. In either case it indicates that the function definition is malformed and should be corrected.

8, 9 - Syntax error in READ

These errors are provoked by READ finding an initial full stop or right bracket, or by mistakes in the format of expressions using dots within them, for example:

```
(THIS . WILL . FAIL TO READ IN)
```

If input was from the keyboard the evidence will still be on the screen.

10 - Word too long

LISP can only handle identifiers of up to 249 characters, and this error is signalled if an attempt is made to go beyond that limit.

11 - Interrupt during PRINT

This is similar to error 4, but ESCAPE was pressed while LISP was printing rather than evaluating. Since escapes during printing are often used to break into long unwanted expressions, this error suppresses the backtrace.

<u>12 - Internal failure in PRINT</u>

The LISP system has become corrupted (e.g. through the use of POKE). It may prove possible to provoke this error by changing the values of enough special variables, such as NIL and T.

<u>13 - Syntax error in COND expression</u>

This is a programming error - a COND expression has been written incorrectly. This error corresponds to an atom being found where a condition/action list was expected. The first ARG in the backtrace will show the atom that was found, and the second will be the complete, incorrect COND expression.

<u>14 - Attempt to take CAR or CDR of atom</u>

CAR, CDR and their compounds can only be applied to lists or dotted pairs. The first ARG in the backtrace will show which atom which was handed to CAR or CDR. If the error was detected in a compound function like CDADR this may be an intermediate result rather than the original argument to CDADR. Check function definitions and insert suitable print statements to find out how your program misbehaves and leads to this effect.

<u>15 - LISP ERROR function called</u>

The ERROR function generates this error number after it has printed its arguments.

<u>16 - Bad first argument for SET or SETQ</u>

The first argument of SET must evaluate to an identifier, while the first argument of SETQ must be an identifier. When detected in SET error 16 can be caused by a logical mistake in the functions being executed. When found in SETQ, it usually reflects a direct coding error, such as the use of (SETQ 3 12).

17 - Identifier expected

This error code is used by a number of functions within LISP requiring identifiers as arguments. The first ARG in the backtrace shows the non-identifier that was found, and the second ARG the expression in which the error arose.

18 - Arithmetic overflow

An arithmetic expression has produced a result greater than 32767 or less than -32768, or there was an attempt to divide by zero. The first ARG in the backtrace normally gives the value of the last argument that was evaluated in the offending expression. The expression itself is shown as the second ARG.

19 - First argument of RPLACA or RPLACD not a list

The first argument to RPLACA and RPLACD must be a list or dotted pair. This error is very similar to error 14 (illegal argument to CAR or CDR).

20 - Property list corrupted

Inappropriate use of RPLACA or RPLACD on values returned by PLIST can leave property lists with non-standard structures. If the normal property list access functions are then used, this error may occur.

21 - Numeric value required

Certain functions expect their arguments to be numeric, and generate this error if a non-number is found. The first ARG in the backtrace shows the non-numeric value that was found.

22 - First argument of APPLY not a function

The first argument of APPLY must either be a list starting with the word LAMBDA, or it must be a code pointer (Subr atom). This error occurs otherwise. Thus (APPLY CAR '((A))) is correct, but (APPLY 'CAR '((A))) gives error 22.

23 - Bad argument list for APPLY

The second argument of APPLY must be a list. This error occurs if it is not, i.e. if it is any atom other than NIL.

24 - Bad argument for MAPC

Typically provoked by the second argument for MAPC not being a proper list - for instance it may be UNSET or some non-NIL atom.

25 - String too long to IMPLODE

IMPLODE creates an identifier, and there is a limit to the length allowed. This error occurs when an attempt is made to breach that limit.

26 - Filing system error

Some function interfacing with the computer's filing system has failed. The failure was such as to cause the file system to issue a BRK. This can happen if a badly malformed file name is handed to OPEN - say something with several dots in it.

27 - File not found

An attempt was made to access a file that could not be opened.

Further error numbers may be allocated in future releases of the system, for instance as new special functions are added.

Glossary

A short Glossary of technical terms

Access function

Function whose job is to select a component out of a datastructure. Mainly exists to separate the logical purpose of a datastructure from its implementation.

Algorithm

Systematic procedure for solving problems.

Argument

A value that is passed to a function when the function is called.

Assembler

Program to convert mnemonics for computer instructions into the bit-patterns used inside the machine.

Atom

A LISP object that is not a list. A word or a number.

Backtrace

Information showing what functions were being evaluated at the time that an error was detected.

Binary operator

Symbol such as '+' or '*' that comes between two expressions to combine them.

Bottom-up

Programming style that starts a project by designing simple facilities which can be coded at once and which are expected to make later work easier. See Top-down.

Breadth first

Search strategy that explores all alternatives more or less in parallel.

Carriage return

The RETURN key on a keyboard. The effect that key produces.

Carry

Keep a digit of the result of an arithmetic operation in range (often 0 to 9) by adjusting the next most significant place.

Chain down

Search through the elements of a list.

Concatenating

Joining end to end.

Conditional expression

Construction that tests a condition and then yields one of a number of possible results.

Co-routines

Functions that co-operate without any one of them being subordinate to the others.

Cyclic structure

List structure where some pointers go round in closed loops.

Database

Collection of information stored on a computer, together with procedures for extracting parts of it.

Debugging

Finding and removing errors (bugs) from a program.

Dotted pairs

LISP datastructures written in the form (x . y) where x and y stand for substructures.

Econet ®

Acorn Computers' local area network.

Editor

Program that makes it possible to alter existing text or programs.

EOF

Acronym for 'End of File'.

Expression

Formula that is to be evaluated.

Factorial

Product of the first few integers. For example the factorial of 6 is 1 x 2 x 3 x 4 x 5 x 6 = 720

File handle

Value returned by OPEN that provides a reference to a file.

Formal arguments

Names used as part of a function definition to show what arguments it will expect.

Forwarding address

Indication of where something has been moved to.

Function call

Invocation of a function.

Global exchanges

Systematic replacement of one symbol by another throughout a text or datastructure.

Identifier

Word or symbol.

Ink blot

Another word for breadth-first search, suggesting that the search spreads like a blot over the area to be investigated.

Interactive

Computing style where user requests and machine responses interleave while a calculation progresses.

Keyboard input buffer

Place where the computer remembers characters that have been typed by the user but not yet used by a program.

Leaves

The ends of a tree-shaped datastructure.

List

Sequence of items. A datastructure that can easily represent such a sequence.

Local variables

Variables that are used just within one function.

Multi-tasking

Facility in computer operating system that allows several activities to be under way at once.

Operand

Value used by an operator.

Parser

Program that turns a sequence of words into a datastructure that reflects the way they group together to make sense.

Predicate

A function that yields a Boolean value (i.e. True or False).

Prettyprinter

Program that formats other programs in an attempt to make them easier to read.

Primitives

Functions that are built into a language that could not have been defined in terms of other existing lower level functions.

Priority queue

Queue that is so organised that some measure of priority determines which item comes first, regardless of the order in which items join the queue.

Property

Information associated with an object.

Recursion

A function calling itself.

ROM

Acronym for Read Only Memory.

S—expression

General LISP datastructure.

Scope

Region of a program where a given variable can be used.

Structure editing

Changing a LISP program by exploiting its representation as LISP data.

Stub

Placeholder fragment of code.

Syntax

Rules by which a program must be composed for the computer to accept it.

Top-down

Programming strategy where high level objectives are progressively divided to give smaller (and hence more manageable) subtasks.

Trace

Indication of which parts of a program are being executed.

Tree

LISP datastructure where each part refers to two disjoint subtrees.

Tube ®

Acorn Computers' scheme for connecting extra processors to a BBC Microcomputer.

Bibliography

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 1980 special issue on Lisp.

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

Griss, M L and Hearn, A C, A portable LISP compiler, Software Practise and Experience 11(6), p541, 1981.

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, 1976.

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

Proc 1982 Lisp Conference, Carnegie Mellon University, 1982.

Sandewall, E, Programming in an interactive environment - the Lisp experience, ACM Computing Surveys 10(1), 1978.

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

Steele, G, The Common Lisp manual, Carnegie Mellon University, 1982.

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

Symbolics, The Lisp Machine Manual, Symbolics Inc., Boston, Mass, 1980.

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

Winston, P and Horn B, Lisp, Addison Wesley, 1981.

Index

LISP

on the BBC Microcomputer

About this book

This book describes the Acornsoft LISP system for the BBC Microcomputer. It provides a complete introduction to LISP and assumes no previous knowledge of the language.

LISP is the fundamental language of artificial intelligence research and provides more flexibility in data and control structures than traditional languages. LISP is easy to learn and is widely used for writing substantial and sophisticated programs with practical applications including design of education systems and medical research.

The Acornsoft LISP system features a number of functions not found in other LISP systems such as the VDU function which provides an easy interface to the BBC Microcomputer machine operating system. Use of this additional function is completely explained in this book and illustrated with many example programs.

The second half of this book is devoted to many example programs. These include a tree-sorting program, an arbitrary arithmetic package, an animal guessing game, a route finding program, a graphics package, a simple compiler and an adventure game.

About the authors

Dr Arthur Norman is a lecturer in computer science at the University of Cambridge, specialising in research into LISP and other list processing languages and their application to algebraic manipulation. He has worked closely with Acornsoft on a number of occasions including advising on the Acornsoft implementation of LISP.

Gillian Cattell did research into the LISP language at Cambridge University and is now working at the National Physical Laboratory at Teddington.

Acornsoft Limited, 4a Market Hill, Cambridge, CB2 3NJ, England

The LISP system described in this book is available on disk, cassette, or ROM.

ISBN 0 907876 08 0

SBD04