Robin Jones • Clive Maynard • Ian Stewart

The Art of LISP Programming

Springer-Verlag

Robin Jones Clive Maynard Ian Stewart

The Art of Lisp Programming

With 12 Illustrations

Springer-Verlag
London Berlin Heidelberg New York
Paris Tokyo Hong Kong

Robin Jones
Department of Mathematics, Science and Information Technology, South Kent College, Folkestone CT20 2NA, UK

Clive Maynard
School of Electrical and Computer Engineering, Curtin University of Technology, Perth, W. Australia

Ian Stewart
Mathematics Institute, University of Warwick, Coventry CV4 7AL, UK

The illustrations by Sir John Tenniel that decorate all chapters except for the Quick Reference Guide are reproduced from *Alice's Adventures in Wonderland* and *Through the Looking-Glass* by Lewis Carroll, with the permission of the publishers, MacMillan & Co. Ltd., London.

ISBN 3–540–19568–8 Springer-Verlag Berlin Heidelberg New York
ISBN 0–387–19568–8 Springer-Verlag New York Berlin Heidelberg

British Library of Cataloguing in Publication Data
Jones, Robin
The art of lisp programming.
1. Computer systems. programming languages: Lisp languages
I. Title II. Maynard, Clive, *1945–* III. Stewart, Ian, *1944–*
005.13'3
ISBN 3-540-19568-8

Library of Congress Cataloging-in-Publication Data
Jones, Robin
The art of Lisp programming/Robin Jones, Clive Maynard, Ian Stewart,
p. cm.
ISBN 0-387-19568-8 (U.S.)
1. LISP (Computer Program language) I. Maynard, Clive, *1944–* II. Stewart, Ian. III. Title.
QA76.73.L23J66 1989
005.13'3--dc20 89-21706 CIP

Filmset by Goodfellow and Egan
Printed by The Alden Press Ltd, Osney Mead, Oxford
543210 – Printed on acid-free paper.

The Art of Lisp Programming

"What does it live on?" Alice asked, with great curiosity.
"Sap and sawdust," said the Gnat. "Go on with the list."

Lewis Carroll, Through the Looking-Glass

Preface

Until recently, Lisp was a language used largely by the Artificial Intelligentsia on mainframe computers. The existing textbooks are consequently directed at a sophisticated audience. But over the last few years Lisp has become widely available on IBM PCs, Macintoshes and the like. Furthermore, Lisp is an immensely powerful general-purpose language. The aim of this book is to introduce the philosophy and features of the language to new users in a simple and accessible fashion.

The reader is likely to be familiar with several procedural languages such as Pascal or BASIC, but may not have met a functional language before. Lisp is a good starting point for learning about such languages. Functional languages in general are beginning to become popular because their mathematically oriented structure makes them more susceptible to formal methods of program proof than are procedural languages. It's an important practical problem to prove that software actually does what it's designed to do—especially if, as is not unusual, it's controlling an ICBM with a 50-megaton warhead. In addition, functional languages are attractive for parallel programming, the hardware for which is now becoming available at realistic costs.

Our main objective is to present the basic ideas of Lisp, without too many confusing frills, so that the reader gets used to the particular style of thinking required in a functional programming language. This is precisely why we *don't* use some of the standard techniques. For example, our program layout is unconventional: the aim is to emphasize the logical structure of the code rather than to save space. While developing the facility to program in Lisp we solve some problems before having introduced the most appropriate tools. To the experienced programmer the resulting code will look clumsy—and, we hope, to you when you've finished the book—but the sheer fact that we can do this illustrates the flexibility of the language. It also gives some insight into how the standard functions might be constructed.

This book is not a reference manual. Although it standardizes on Common Lisp, it discusses only a fraction of the full range of functions in that implementation of the language. It's a question of seeing the wood for the trees: the new user would find the full language bafflingly complex. We hope that, by the time the reader has finished this book, his manuals will have become accessible.

Where appropriate, chapters end with a series of exercises. We have given answers immediately after them, because the material forms an integral part of the development of the reader's understanding of the language.

The early chapters develop the language using short, easily understood functions. However, not all Lisp is short, or easily understood. To give a feeling for what a realistic project might look like, the last three chapters develop a Lisp-based interpreter for the language ABC.

Most Lisp systems have an interpreter and a compiler. We confine our attention to interpreters, because they provide instant feedback and, usually, simple editing features. The reader will get most out of the book by working through it while sitting in front of a warm computer running a Lisp interpreter.

Thus far we have referred to ourselves in the plural. However, only one of us wrote any particular chapter, so to continue to do so strikes us as stilted and formal. Henceforward 'we' therefore becomes 'I'. Any reader who dislikes this should invoke Lisp's power and

```
( defmacro I() 'we )
```

Folkestone, Kent	Robin Jones
Perth, W. Australia	Clive Maynard
Coventry, Warwickshire	Ian Stewart

Contents

1 **Some Basic Ideas** 1

Lists 2
Representing Lists 3
The Interpreter 4
Functions 5
Two More Primitives 5
Constants and Variables 6
S-Expressions 6
Postscript 6
Exercises 1 7
Answers 7

2 **List Functions** 9

Splitting Lists 10
car and cdr 10
Setting Up Lists 11
setq 11
A Piece of Pedantry 12
Head and Tail Revisited 12
cons 13
Postscript 13
Exercises 2 13
Answers 14

3 **Predicates** 17

cond 18
Boolean Functions 19
Exercises 3 21
Answers 21

4 **Recursion** 23
Powers 24
Boolean Functions Revisited 25
eq 25
Meanwhile Back at the Problem. 26
null 26
AND and OR 26
flat 27
first 28
rest 28
The Fly in the Ointment 29
append and list 29
Postscript 30
Exercises 4 30
Answers 31

5 **A Look Back (and Forward)** 33
Variables: a Sense of Place 34
Variable Scope 36
Variable Binding 36
Free Variables 37
Lexical Scoping 37
A Software Engineering Problem 38
setq 38
System Variables 39
Exercises 5 39
Answers 40

6 **Sets and Lists** 41
Sets of Numbers 42
The Set Functions 42
Intersection 43
disjoint-p 44
Subsets 44
Complement 44
Union 45
Exercises 6 46
Answers 47

7 **Input** 49
read 49
read-line 49

Single Characters 50
More About # 51
Yes or No Responses 52
Using the read Primitives 52
Exercises 7 53
Answers... 53

8 **Output**.. 55

format 56
The Control String 56
Printing Values.. 56
Format Directives 57
Using Arguments as Parameters 58
Tabs... 58
Other Output Functions 58
The Telephone Index Again 59
The read–eval–print Loop 59
The Program–Data Distinction.................................... 60
Exercises 8 60
Answers.. 61

9 **Iterative Constructs** 63

loop 64
return... 64
do.. 65
The Size of a List.. 65
Iteration v. Recursion 66
dotimes... 67
dolist 67
Mapping 69
mapcar 69
Exercises 9 70
Answers.. 70

10 **More About Program Control** 73

if 73
when 74
unless.. 74
case 74
Synonyms... 75
Defaults.. 76

Some New Control Constructs 76
let 76
flet 77
progl to prog*n* 78
The Lambda Notation 79
Exercises 10 80
Answers 80

11 **File Handling** 83

Pathnames 84
merge-pathnames 84
Streams 85
Opening and Closing Files 85
Keyword Arguments 85
The Body 86
Other Keyword Arguments 86
Getting the File Back 87
EOF-ERROR-P and EOF-VALUE 87
Exercises 11 88
Answers 89

12 **Data Structures** 91

Association Lists 92
pairlis 92
The Dot Notation 93
assoc 93
rassoc 93
acons 94
setf 94
Database Manipulation 95
Trees 96
Binary Trees 97
The Searching Problem 97
Implementing a Binary Tree 98
Growing a Tree 98
Searching the Tree 100
Stacks 100
push 101
pop 102
Exercises 12 103
Answers 103

13 **Arrays, Strings and Structures** 105

Arrays 105
Changing an Array Element 106
Multidimensional Arrays 106
Strings 107
char 107
string= 107
string < 108
Substrings 108
String Manipulation Functions 108
Structures 109
defstruct 109
What defstruct Does 110
Inheritance 110
Exercises 13 111
Answers 112

14 **A Pot-Pourri of Features** 113

Functions, Macros and Special Forms 114
defmacro 114
macroexpand 115
apply and funcall 115
More About Function Arguments 116
Undefined Numbers of Arguments 116
Optional Arguments 117
Keyword Arguments 118
Multiple-Valued Functions 118
Packages 119
Parallel and Sequential Assignments 121
psetq 121
let* 121
Comments 121
Exercises 14 122
Answers 122

15 **Debugging Techniques** 123

Using format 124
break 124
The Stepper 125
trace 126
apropos 126
time 127

16 **Object-Oriented Programming** 129
Objects 130
Classes 131
Objection! 132
Example 132
Inheritance 133
Functions 135
display 135

17 **ABC** 137
Examples of ABC Code 138
Description of the Language 138
Before We Start 139

18 **ABC Interpreter: Scanner** 141
Overall Structure 141
The Scanner 143
Symbol Table 144
Extracting Tokens 145
Extracting Syntax 147
Stacks Revisited 148
Optimization 148

19 **ABC Interpreter: Execution** 153
What the Interpreter Will Recognize 154
Template Matching 154
The Remaining Code 155
A Session with the Interpreter 159

Quick Reference Guide 161

Subject Index 167

· 1 ·
Some Basic Ideas

The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked.
"Begin at the beginning", the King said, very gravely, "and go on till you come to the end: then stop."

Alice's Adventures in Wonderland

There are those who will tell you that LISP is an acronym for LISt Processor and others who insist that it stands for Lots of Infuriatingly Silly Parentheses. Both camps have good arguments to back them up. Lisp *is* a language which deals almost exclusively in list structures, and there are a great many brackets in a typical Lisp program. Paradoxically, Lisp derives much of its power as a programming language from the fact that it *is* limited in this way, and, as we shall see, this philosophy leads automatically to the proliferation of similar symbols (which just happen to be brackets) which so incense Lisp's detractors.

Lists

Perhaps we should begin, then, by studying what is meant by the term 'list' in the current context, and try to see why a list is such a powerful construct.

Suppose that you are writing a word processor. A simple technique for storing the text in memory would be to use an array as shown in Figure 1. There are two main objections to this rather naive format. First, an attempt to add the word 'black' before 'cat' requires that the remaining words in the text are each

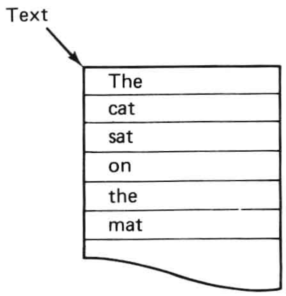

FIGURE 1. How not to store text.

shuffled one cell down the array. This is OK if we're talking about a six word sentence, as here, but if it's the first sentence of the great American novel, there's going to be a long wait before the extra adjective is successfully inserted. Second, there's an implicit assumption that all words are of the same length. Of course, a word may be padded with spaces, but that will waste memory, and there's certainly a limit to the size of word which can be held. So let's consider an alternative arrangement, shown in Figure 2. Here, each entry consists of a word and a pointer to the next entry. The physical order of the entries no longer matters, so that additional words can be added to the text simply by creating cells for them and altering a couple of pointers (see Figure 3).

That deals neatly with the editing problem, but it still leaves us with the difficulty of handling what are effectively fixed-length words. Suppose we revise the structure of Figure 3 so that each entry consists not of a datum plus a pointer, but *two* pointers. Now the left-hand pointer can point to a similar structure which identifies a word and of course it can be any length, because we can signal

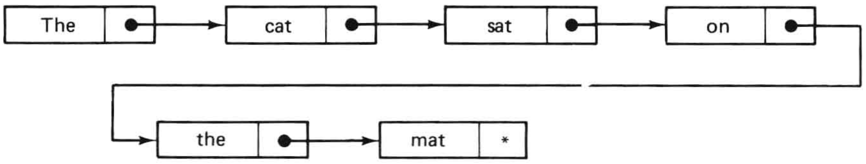

FIGURE 2. A better structure.

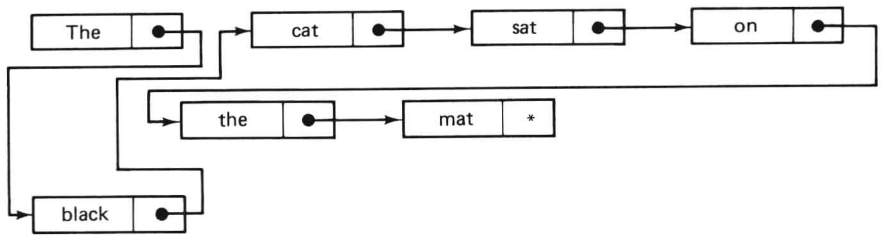

FIGURE 3. Adding the word 'black' now only entails some pointer manipulation.

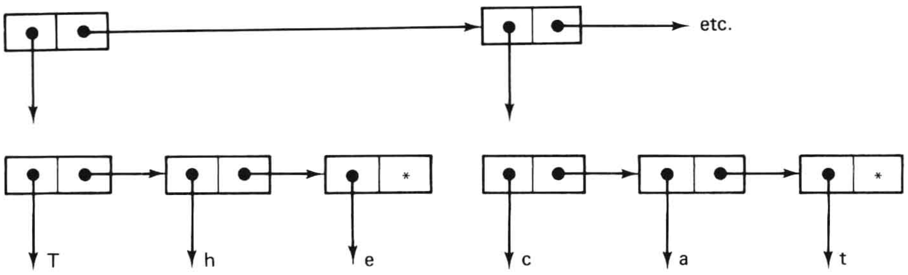

FIGURE 4. A true list. Now words can be indefinitely long.

its end with some appropriate delimiting value for the pointer. We now have the organization shown in Figure 4, where the delimiting pointer is an asterisk.

It's clear that this is an immensely flexible structure, because there is nothing in principle to prevent us from extending it indefinitely. Each pair of pointers may point to other structures of the same general type, or to terminating symbols. In this example, such symbols are either letters or null pointers, but there are all sorts of other possibilities. Note, however, that while both pointers can point to other structures, only left pointers can point to terminal symbols and only right pointers can be null.

The structures we have been discussing are called lists. The terminal symbols (here letters) are called *atoms*, because clearly they cannot be divided further. (Yes, I *know* physicists are always splitting them, but that's their fault for choosing the word before finding out that they could do so. We will use the word with its original meaning.)

Representing Lists

Clearly, the diagrammatic notation for a list of Figure 4 is inappropriate to a computer language, although there are occasions when it can be a useful prop on which to hang an idea and I shall use it from time to time in this book. Lisp employs a simple symbolic notation. For example, the list of letters 'A,B,C,D' is written (A B C D) and can be visualized as Figure 5. Similarly, Figure 6 shows the list ((a b) (c d e)). The Lisp form of the complete list of Figure 4 would be:

```
( ( The ) ( cat ) ( sat ) ( on ) ( the ) ( mat ) )
```

Note that spaces are used to separate atoms, but are otherwise insignificant. Thus the spaces between brackets and those between brackets and atoms are not necessary but do no harm.

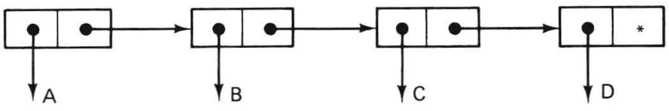

FIGURE 5. (A B C D).

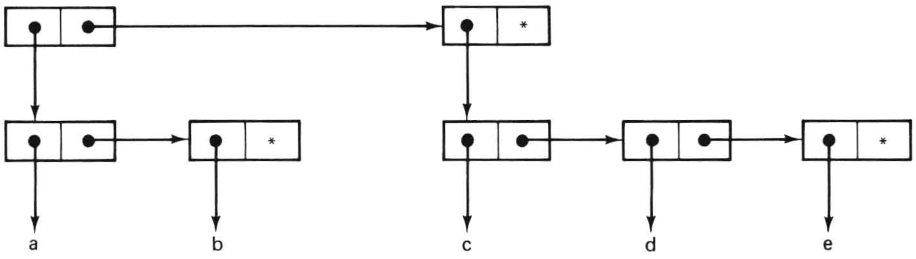

FIGURE 6. ((a b) (c d e)).

The Interpreter

We're very nearly in a position actually to try something out. Enter your Lisp interpreter. The syntax for this will be implementation dependent, but it will probably involve invoking an executable file from disk, perhaps coupled with an environment file. In any event, you should be greeted with a prompt which is likely to be '>'.

It should come as no surprise that the interpreter will only accept atoms or lists as input. So let's give it a list. Type:

```
( )
```

The interpreter will probably respond as soon as it sees the closing bracket (but some implementations require you to hit RETURN):

```
NIL
```

What's happened is this. The interpreter has accepted the list, which is empty, and immediately *evaluated* it. Its value is NIL, which is a reserved Lisp atom meaning *either* the empty list or the logical value FALSE. Since we're at the top level of the interpreter, Lisp has returned this value to us.