Lisp in Small Pieces

Lisp in Small Pieces

Christian Queinnec
Ecole Polytechnique

Translated by Kathleen Callaway

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York NY 10011–4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published in French as *Les Langages Lisp* by Interéditions 1994

First published in English 1996
First paperback edition 2003

Translated by
Kathleen Callaway, 9 allée des Bois du Stade, 33700 Merignac, France

A catalogue record for this book is available from the British Library

ISBN 0 521 56247 3 hardback
ISBN 0 521 54566 8 paperback

Table of Contents

To the Reader

EVEN though the literature about Lisp is abundant and already accessible to the reading public, nevertheless, this book still fills a need. The logical substratum where Lisp and Scheme are founded demand that modern users must read programs that use (and even abuse) advanced technology, that is, higher-order functions, objects, continuations, and so forth. Tomorrow's concepts will be built on these bases, so not knowing them blocks your path to the future.

To explain these entities, their origin, their variations, this book will go into great detail. Folklore tells us that even if a Lisp user knows the value of every construction in use, he or she generally does not know its cost. This work also intends to fill that mythical hole with an in-depth study of the semantics and implementation of various features of Lisp, made more solid by more than thirty years of history.

Lisp is an enjoyable language in which numerous fundamental and non-trivial problems can be studied simply. Along with ML, which is strongly typed and suffers few side effects, Lisp is the most representative of the applicative languages. The concepts that illustrate this class of languages absolutely must be mastered by students and computer scientists of today and tomorrow. Based on the idea of "function," an idea that has matured over several centuries of mathematical research, applicative languages are omnipresent in computing; they appear in various forms, such as the composition of UN⋆X byte streams, the extension language for the EMACS editor, as well as other scripting languages. If you fail to recognize these models, you will misunderstand how to combine their primitive elements and thus limit yourself to writing programs painfully, word by word, without a real architecture.

Audience

This book is for a wide, if specialized audience:

- to graduate students and advanced undergraduates who are studying the implementation of languages, whether applicative or not, whether interpreted, compiled, or both.
- to programmers in Lisp or Scheme who want to understand more clearly the costs and nuances of constructions they're using so they can enlarge their expertise and produce more efficient, more portable programs.

- to the lovers of applicative languages everywhere; in this book, they'll find many reasons for satisfying reflection on their favorite language.

Philosophy

This book was developed in courses offered in two areas: in the graduate research program (DEA ITCP: Diplôme d'Études Approfondies en Informatique Théorique, Calcul et Programmation) at the University of Pierre and Marie Curie of Paris VI; some chapters are also taught at the École Polytechnique.

A book like this would normally follow an introductory course about an applicative language, such as Lisp, Scheme, or ML, since such a course typically ends with a description of the language itself. The aim of this book is to cover, in the widest possible scope, the semantics and implementation of interpreters and compilers for applicative languages. In practical terms, it presents no less than twelve interpreters and two compilers (one into byte-code and the other into the C programming language) without neglecting an object-oriented system (one derived from the popular MEROON). In contrast to many books that omit some of the essential phenomena in the family of Lisp dialects, this one treats such important topics as reflection, introspection, dynamic evaluation, and, of course, macros.

This book was inspired partly by two earlier works: *Anatomy of Lisp* [All78], which surveyed the implementation of Lisp in the seventies, and *Operating System Design: the Xinu Approach* [Com84], which gave all the necessary code without hiding any details on how an operating system works and thereby gained the reader's complete confidence.

In the same spirit, we want to produce a precise (rather than concise) book where the central theme is the semantics of applicative languages generally and of Scheme in particular. By surveying many implementations that explore widely divergent aspects, we'll explain in complete detail how any such system is built. Most of the schisms that split the community of applicative languages will be analyzed, taken apart, implemented, and compared, revealing all the implementation details. We'll "tell all" so that you, the reader, will never be stumped for lack of information, and standing on such solid ground, you'll be able to experiment with these concepts yourself.

Incidentally, all the programs in this book can be picked up, intact, electronically (details on page xix).

Structure

This book is organized into two parts. The first takes off from the implementation of a naive Lisp interpreter and progresses toward the semantics of Scheme. The line of development in this part is motivated by our need to be more specific, so we successively refine and redefine a series of name spaces (Lisp_1, Lisp_2, and so forth), the idea of continuations (and multiple associated control forms), assignment, and writing in data structures. As we slowly augment the language that we're defining, we'll see that we inevitably pare back its defining language so that it is reduced to

Chapter	Signature
1	`(eval exp env)`
2	`(eval exp env fenv)`
	`(eval exp env fenv denv)`
	`(eval exp env denv)`
3	`(eval exp env cont)`
4	`(eval e r s k)`
5	`((meaning e) r s k)`
6	`((meaning e sr) r k)`
	`((meaning e sr tail?) k)`
	`((meaning e sr tail?))`
7	`(run (meaning e sr tail?))`
10	`(->C (meaning e sr))`

Figure 1 Approximate signatures of interpreters and compilers

a kind of λ-calculus. We then convert the description we've gotten this way into its denotational equivalent.

More than six years of teaching experience convinced us that this approach of making the language more and more precise not only initiates the reader gradually into authentic language-research, but it is also a good introduction to denotational semantics, a topic that we really can't afford to leap over.

The second part of the book goes in the other direction. Starting from denotational semantics and searching for efficiency, we'll broach the topic of fast interpretation (by pretreating static parts), and then we'll implement that preconditioning (by precompilation) for a byte-code compiler. This part clearly separates program preparation from program execution and thus handles a number of topics: dynamic evaluation (`eval`); reflective aspects (first class environments, auto-interpretable interpretation, reflective tower of interpreters); and the semantics of macros. Then we introduce a second compiler, one compiling to the C programming language.

We'll close the book with the implementation of an object-oriented system, where objects make it possible to define the implementation of certain interpreters and compilers more precisely.

Good teaching demands a certain amount of repetition. In that context, the number of interpreters that we examine, all deliberately written in different styles—naive, object-oriented, closure-based, denotational, etc.—cover the essential techniques used to implement applicative languages. They should also make you think about the differences among them. Recognizing these differences, as they are sketched in Figure 1, will give you an intimate knowledge of a language and its implementation. Lisp is not just one of these implementations; it is, in fact, a *family* of dialects, each one made up of its own particular mix of the characteristics we'll be looking at.

In general, the chapters are more or less independent units of about forty pages or so; each is accompanied by exercises, and the solutions to those exercises are found at the end of the book. The bibliography contains not only historically important references, so you can see the evolution of Lisp since 1960, but also

references to current, on-going research.

Prerequisites

Though we hope this book is both entertaining and informative, it may not necessarily be easy to read. There are subjects treated here that can be appreciated only if you make an effort proportional to their innate difficulty. To harken back to something like the language of courtly love in medieval France, there are certain objects of our affection that reveal their beauty and charm only when we make a chivalrous but determined assault on their defences; they remain impregnable if we don't lay seige to the fortress of their inherent complexity.

In that respect, the study of programming languages is a discipline that demands the mastery of tools, such as the λ-calculus and denotational semantics. While the design of this book will gradually take you from one topic to another in an orderly and logical way, it can't eliminate all effort on your part.

You'll need certain prerequisite knowledge about Lisp or Scheme; in particular, you'll need to know roughly thirty basic functions to get started and to understand recursive programs without undue labor. This book has adopted Scheme as the presentation language; (there's a summary of it, beginning on page xviii) and it's been extended with an object layer, known as MEROON. That extension will come into play when we want to consider problems of representation and implementation.

All the programs have been tested and actually run successfully in Scheme. For readers that have assimilated this book, those programs will pose no problem whatsoever to port!

Thanks and Acknowledgments

I must thank the organizations that procured the hardware (Apple Mac SE30 then Sony News 3260) and the means that enabled me to write this book: the École Polytechnique, the Institut National de Recherche en Informatique et Automatique (INRIA-Rocquencourt, the national institute for research in computing and automation at Rocquencourt), and Greco-PRC de Programmation du Centre National de la Recherche Scientifique (CNRS, the national center for scientific research, special group for coordinated research on computer science).

I also want to thank those who actually participated in the creation of this book by all the means available to them. I owe particular thanks to Sophie Anglade, Josy Baron, Kathleen Callaway, Jérôme Chailloux, Jean-Marie Geffroy, Christian Jullien, Jean-Jacques Lacrampe, Michel Lemaître, Luc Moreau, Jean-François Perrot, Daniel Ribbens, Bernard Serpette, Manuel Serrano, Pierre Weis, as well as my muse, Claire N.

Of course, any errors that still remain in the text are surely my own.

Notation

Extracts from programs appear in **this type face**, no doubt making you think unavoidably of an old-fashioned typewriter. At the same time, certain parts will appear in *italic* to draw attention to variations within this context.

The sign → indicates the relation "has this for its value" while the sign ≡ indicates equivalence, that is, "has the same value as." When we evaluate a form in detail, we'll use a vertical bar to indicate the environment in which the expression must be considered. Here's an example illustrating these conventions in notation:

```
(let ((a (+ b 1)))
  (let ((f (lambda () a)))
    (foo (f) a) ) )|        ;the value of foo is the function for creating dotted
                   |b→ 3   ;pairs that is, the value of the global variable cons.
                   |foo≡ cons
≡ (let ((f (lambda () a))) (foo (f) a))|
                                       |a→ 4
                                       |b→ 3
                                       |foo≡ cons
                                       |f≡ (lambda () a)|
                                                         |a→ 4
≡ (foo (f) a)|
             |a→ 4
             |b→ 3
             |foo≡ cons
             |f≡ (lambda () a)|
                               |a→ 4
→ (4 . 4)
```

We'll use a few functions that are non-standard in Scheme, such as **gensym** that creates symbols guaranteed to be new, that is, different from any symbol seen before. In Chapter 10, we'll also use **format** and **pp** to display or "pretty-print." These functions exist in most implementations of Lisp or Scheme.

Certain expressions make sense only in the context of a particular dialect, such as COMMON LISP, Dylan, EULISP, IS-Lisp, Le-Lisp[1], Scheme, etc. In such a case, the name of the dialect appears next to the example, like this:

```
(defdynamic fooncall                                    IS-Lisp
  (lambda (one :rest others)
    (funcall one others) ) )
```

To make it easier for you to get around in this book, we'll use this sign [**see p.**] to indicate a cross-reference to another page. When we suggest variations detailed in the exercises, we'll also use that sign, like this [**see Ex.**]. You'll also find a complete index of the function definitions that we mention. [**see p. 495**]

1. Le-Lisp is a trademark of INRIA.

Short Summary of Scheme

There are excellent books for learning Scheme, such as [AS85, Dyb87, SF89]. For reference, the standard document is the *Revised revised revised revised Report on Scheme*, informally known as R^{4}RS.

This summary merely outlines the important characteristics of that dialect, that is, the characteristics that we'll be using later to dissect the dialect as we lead you to a better understanding of it.

Scheme lets you handle symbols, characters, character strings, lists, numbers, Boolean values, vectors, ports, and functions (or procedures in Scheme parlance).

Each of those data types has its own associated predicate: **`symbol?`**, **`char?`**, **`string?`**, **`pair?`**, **`number?`**, **`boolean?`**, **`vector?`**, and **`procedure?`**.

There are also the corresponding selectors and modifiers, where appropriate, such as: **`string-ref`**, **`string-set!`**, **`vector-ref`**, and **`vector-set!`**.

For lists, there are: **`car`**, **`cdr`**, **`set-car!`**, and **`set-cdr!`**.

The selectors, **`car`** and **`cdr`**, can be composed (and pronounced), so, for example, to designate the second term in a list, we use **`cadr`** and pronounce it something like *kadder*.

These values can be implicitly named and created simply by mentioning them, as we do with symbols and identifiers. For characters, we prefix them by **`#\`** as in **`#\Z`** or **`#\space`**. We enclose character strings within quotation marks (that is, **`"`**) and lists within parentheses (that is, **`()`**). We use numbers are they are. We can also make use of Boolean values, namely, **`#t`** and **`#f`**. For vectors, we use this syntax: **`#(do re mi)`**, for example. Such values can be constructed dynamically with **`cons`**, **`list`**, **`string`**, **`make-string`**, **`vector`**, and **`make-vector`**. They can also be converted from one to another, by using **`string->symbol`** and **`int->char`**.

We manage input and output by means of these functions: **`read`**, of course, reads an expression; **`display`** shows an expression; **`newline`** goes to the next line.

Programs are represented by Scheme values known as *forms*.

The form **`begin`** lets you group forms to evaluate them sequentially; for example, **`(begin (display 1) (display 2) (newline))`**.

There are many conditional forms. The simplest is the form *if—then—else*—conventionally written in Scheme this way: **`(if`** *condition then otherwise*`)`. To handle choices that entail more than two options, Scheme offers **`cond`** and **`case`**. The form **`cond`** contains a group of *clauses* beginning with a Boolean form and ending by a series of forms; one by one the Boolean forms of the clauses are evaluated until one returns true (or more precisely, not false, that is, not **`#f`**); the forms that follow the Boolean form that succeeded will then be evaluated, and their result becomes the value of the entire **`cond`** form. Here's an example of the form **`cond`** where you can see the default behavior of the keyword **`else`**.

```
(cond ((eq? x 'flip) 'flop)
      ((eq? x 'flop) 'flip)
      (else (list x "neither flip nor flop")) )
```

The form **`case`** has a form as its first parameter, and that parameter provides a key that we'll look for in all the clauses that follow; each of those clauses specifies

which key or keys will set it off. Once an appropriate key is found, the associated forms will be evaluated and their result will become the result of the entire **case** form. Here's how we would convert the preceding example using **cond** into one using **case**.

```
(case x
  ((flip) 'flop)
  ((flop) 'flip)
  (else (list x "neither flip nor flop")) )
```

Functions are defined by a **lambda** form. Just after the keyword **lambda**, you'll find the variables of the function, followed by the expressions that indicate how to calculate the function. These variables can be modified by assignment, indicated by **set!**. Literal constants are introduced by **quote**. The forms **let**, **let***, and **letrec** introduce local variables; the initial value of such a local variable may be calculated in various ways.

With the form **define**, you can define named values of any kind. We'll exploit the internal writing facilities that **define** forms provide, as well as the non-essential syntax where the name of the function to define is indicated by the way it's called. Here is an example of what we mean.

```
(define (rev l)
  (define nil '())
  (define (reverse l r)
     (if (pair? l) (reverse (cdr l) (cons (car l) r)) r) )
  (reverse l nil) )
```

That example could also be rewritten without inessential syntax, like this:

```
(define rev
  (lambda (l)
    (letrec ((reverse (lambda (l r)
                        (if (pair? l) (reverse (cdr l)
                                               (cons (car l) r))
                            r ) )))
      (reverse l '()) ) ) )
```

That example completes our brief summary of Scheme.

Programs and More Information

The programs (both interpreted and compiled) that appear in this book, the object system, and the associated tests are all available on-line by anonymous **ftp** at:

(IP 128.93.2.54) **ftp.inria.fr:INRIA/Projects/icsla/Books/LiSP.tar.gz**

At the same site, you'll also find articles about Scheme and other implementations of Scheme.

The electronic mail address of the author of this book is:

Christian.Queinnec@polytechnique.fr

Recommended Reading

Since we assume that you already know Scheme, we'll refer to the standard reference [AS85, SF89].

To gain even greater advantage from this book, you might also want to prepare yourself with other reference manuals, such as COMMON LISP [Ste90], Dylan [App92b], EULISP[PE92], IS-Lisp [ISO94], Le-Lisp [CDD+91], OakLisp [LP88], Scheme [CR91b], T [RAM84] and, Talk [ILO94].

Then, for a wider perspective about programming languages in general, you might want to consult [BG94].