The Ten Commandments

The First Commandment

When recurring on a list of atoms, *lat*, ask two questions about it: (*null? lat*) and **else**.
When recurring on a number, *n*, ask two questions about it: (*zero? n*) and **else**.
When recurring on a list of S-expressions, *l*, ask three question about it: (*null? l*), (*atom?* (*car l*)), and **else**.

The Second Commandment

Use *cons* to build lists.

The Third Commandment

When building a list, describe the first typical element, and then *cons* it onto the natural recursion.

The Fourth Commandment

Always change at least one argument while recurring. When recurring on a list of atoms, *lat*, use (*cdr lat*). When recurring on a number, *n*, use (*sub1 n*). And when recurring on a list of S-expressions, *l*, use (*car l*) and (*cdr l*) if neither (*null? l*) nor (*atom?* (*car l*)) are true.

It must be changed to be closer to termination. The changing argument must be tested in the termination condition:

when using *cdr*, test termination with *null?* and

when using *sub1*, test termination with *zero?*.

The Fifth Commandment

When building a value with $+$,always use 0 for the value of the terminating line, for adding 0 does not change the value of an addition.

When building a value with $\times$, always use 1 for the value of the terminating line, for multiplying by 1 does not change the value of a multiplication.

When building a value with *cons*, always consider () for the value of the terminating line.

The Sixth Commandment

Simplify only after the function is correct.

The Seventh Commandment

Recur on the *subparts* that are of the same nature:

- On the sublists of a list.
- On the subexpressions of an arithmetic expression.

The Eighth Commandment

Use help functions to abstract from representations.

The Ninth Commandment

Abstract common patterns with a new function.

The Tenth Commandment

Build functions to collect more than one value at a time.

The Five Rules

The Law of Car

The primitive *car* is defined only for non-empty lists.

The Law of Cdr

The primitive *cdr* is defined only for non-empty lists. The *cdr* of any non-empty list is always another list.

The Law of Cons

The primitive *cons* takes two arguments. The second argument to *cons* must be a list. The result is a list.

The Law of Null?

The primitive *null?* is defined only for lists.

The Law of Eq?

The primitive *eq?* takes two arguments. Each must be a non-numeric atom.

The Little Schemer

Fourth Edition

Daniel P. Friedman

Indiana University
Bloomington, Indiana

Matthias Felleisen

Rice University
Houston, Texas

Drawings by Duane Bibby

Foreword by Gerald J. Sussman

The MIT Press
Cambridge, Massachusetts
London, England

This book was set by the authors and was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Friedman, Daniel P.
The little schemer / Daniel P. Friedman and Matthias Felleisen; drawings by Duane Bibby; foreword by Gerald J. Sussman.—4 ed., 1st MIT Press ed.
p. cm.
Rev. ed. of: The little LISPer. 3rd ed. ©1989.
Includes index.
ISBN 0-262-56099-2 (pbk: alk. paper)
1. Scheme (Computer program language) 2. LISP (Computer program language) I. Felleisen, Matthias. II. Friedman, Daniel P. Little LISPer. III. Title.
QA76.73.S34F75 1996
005.13'3—dc20 95-39853
CIP

To Mary, Helga, and our children

((**Contents**)

Foreword

This foreword appeared in the second and third editions of *The Little LISPer*. We reprint it here with the permission of the author.

In 1967 I took an introductory course in photography. Most of the students (including me) came into that course hoping to learn how to be creative—to take pictures like the ones I admired by artists such as Edward Weston. On the first day the teacher patiently explained the long list of technical skills that he was going to teach us during the term. A key was Ansel Adams' "Zone System" for previsualizing the print values (blackness in the final print) in a photograph and how they derive from the light intensities in the scene. In support of this skill we had to learn the use of exposure meters to measure light intensities and the use of exposure time and development time to control the black level and the contrast in the image. This is in turn supported by even lower level skills such as loading film, developing and printing, and mixing chemicals. One must learn to ritualize the process of developing sensitive material so that one gets consistent results over many years of work. The first laboratory session was devoted to finding out that developer feels slippery and that fixer smells awful.

But what about creative composition? In order to be creative one must first gain control of the medium. One can not even begin to think about organizing a great photograph without having the skills to make it happen. In engineering, as in other creative arts, we must learn to do analysis to support our efforts in synthesis. One cannot build a beautiful and functional bridge without a knowledge of steel and dirt and considerable mathematical technique for using this knowledge to compute the properties of structures. Similarly, one cannot build a beautiful computer system without a deep understanding of how to "previsualize" the process generated by the procedures one writes.

Some photographers choose to use black-and-white 8×10 plates while others choose 35mm slides. Each has its advantages and disadvantages. Like photography, programming requires a choice of medium. Lisp is the medium of choice for people who enjoy free style and flexibility. Lisp was initially conceived as a theoretical vehicle for recursion theory and for symbolic algebra. It has developed into a uniquely powerful and flexible family of software development tools, providing wrap-around support for the rapid-prototyping of software systems. As with other languages, Lisp provides the glue for using a vast library of canned parts, produced by members of the user community. In Lisp, procedures are first-class data, to be passed as arguments, returned as values, and stored in data structures. This flexibility is valuable, but most importantly, it provides mechanisms for formalizing, naming, and saving the idioms—the common patterns of usage that are essential to engineering design. In addition, Lisp programs can easily manipulate the representations of Lisp programs—a feature that has encouraged the development of a vast structure of program synthesis and analysis tools, such as cross-referencers.

The Little LISPer is a unique approach to developing the skills underlying creative programming in Lisp. It painlessly packages, with considerable wit, much of the drill and practice that is necessary to learn the skills of constructing recursive processes and manipulating recursive data-structures. For the student of Lisp programming, *The Little LISPer* can perform the same service that Hanon's finger exercises or Czerny's piano studies perform for the student of piano.

Gerald J. Sussman
Cambridge, Massachusetts

Preface

To celebrate the twentieth anniversary of Scheme we revised *The Little LISPer* a third time, gave it the more accurate title *The Little Schemer*, and wrote a sequel: *The Seasoned Schemer*.

Programs accept data and produce data. Designing a program requires a thorough understanding of data; a good program reflects the shape of the data it deals with. Most collections of data, and hence most programs, are recursive. Recursion is the act of defining an object or solving a problem in terms of itself.

The goal of this book is to teach the reader to think recursively. Our first task is to decide which language to use to communicate this concept. There are three obvious choices: a natural language, such as English; formal mathematics; or a programming language. Natural languages are ambiguous, imprecise, and sometimes awkwardly verbose. These are all virtues for general communication, but something of a drawback for communicating concisely as precise a concept as recursion. The language of mathematics is the opposite of natural language: it can express powerful formal ideas with only a few symbols. Unfortunately, the language of mathematics is often cryptic and barely accessible without special training. The marriage of technology and mathematics presents us with a third, almost ideal choice: a programming language. We believe that programming languages are the best way to convey the concept of recursion. They share with mathematics the ability to give a formal meaning to a set of symbols. But unlike mathematics, programming languages can be directly experienced—you can take the programs in this book, observe their behavior, modify them, and experience the effect of these modifications.

Perhaps the best programming language for teaching recursion is Scheme. Scheme is inherently symbolic—the programmer does not have to think about the relationship between the symbols of his own language and the representations in the computer. Recursion is Scheme's natural computational mechanism; the primary programming activity is the creation of (potentially) recursive definitions. Scheme implementations are predominantly interactive—the programmer can immediately participate in and observe the behavior of his programs. And, perhaps most importantly for our lessons at the end of this book, there is a direct correspondence between the structure of Scheme programs and the data those programs manipulate.

Although Scheme can be described quite formally, understanding Scheme does not require a particularly mathematical inclination. In fact, *The Little Schemer* is based on lecture notes from a two-week "quickie" introduction to Scheme for students with no previous programming experience and an admitted dislike for anything mathematical. Many of these students were preparing for careers in public affairs. It is our belief that *writing programs recursively in Scheme is essentially simple pattern recognition.* Since our only concern is recursive programming, our treatment is limited to the whys and wherefores of just a few Scheme features: car, cdr, cons, eq?, null?, zero?, add1, sub1, number?, and, or, quote, lambda, define, and cond. Indeed, our language is an *idealized* Scheme.

The Little Schemer and *The Seasoned Schemer* will not introduce you to the practical world of programming, but a mastery of the concepts in these books provides a start toward understanding the nature of computation.

What You Need to Know to Read This Book

The reader must be comfortable reading English, recognizing numbers, and counting.

Acknowledgments

We are indebted to many people for their contributions and assistance throughout the development of the second and third editions of this book. We thank Bruce Duba, Kent Dybvig, Chris Haynes, Eugene Kohlbecker, Richard Salter, George Springer, Mitch Wand, and David S. Wise for countless discussions that influenced our thinking while conceiving this book. Ghassan Abbas, Charles Baker, David Boyer, Mike Dunn, Terry Falkenberg, Rob Friedman, John Gateley, Mayer Goldberg, Iqbal Khan, Julia Lawall, Jon Mendelsohn, John Nienart, Jeffrey D. Perotti, Ed Robertson, Anne Shpuntoff, Erich Smythe, Guy Steele, Todd Stein, and Larry Weisselberg provided many important comments on the drafts of the book. We especially want to thank Bob Filman for being such a thorough and uncompromising critic through several readings. Finally we wish to acknowledge Nancy Garrett, Peg Fletcher, and Bob Filman for contributing to the design and TEXery.

The fourth and latest edition greatly benefited from Dorai Sitaram's incredibly clever Scheme typesetting program SLATEX. Kent Dybvig's Chez Scheme made programming in Scheme a most pleasant experience. We gratefully acknowledge criticisms and suggestions from Shelaswau Bushnell, Richard Cobbe, David Combs, Peter Drake, Kent Dybvig, Rob Friedman, Steve Ganz, Chris Haynes, Erik Hilsdale, Eugene Kohlbecker, Shriram Krishnamurthi, Julia Lawall, Suzanne Menzel Collin McCurdy, John Nienart, Jon Rossie, Jonathan Sobel, George Springer, Guy Steele, John David Stone, Vikram Subramaniam, Mitch Wand, and Melissa Wingard-Phillips.

Guidelines for the Reader

Do not rush through this book. Read carefully; valuable hints are scattered throughout the text. Do not read the book in fewer than three sittings. Read systematically. If you do not *fully* understand one chapter, you will understand the next one even less. The questions are ordered by increasing difficulty; it will be hard to answer later ones if you cannot solve the earlier ones.

The book is a dialogue between you and us about interesting examples of Scheme programs. If you can, try the examples while you read. Schemes are readily available. While there are minor syntactic variations between different implementations of Scheme (primarily the spelling of particular names and the domain of specific functions), Scheme is basically the same throughout the world. To work with Scheme, you will need to define `atom?`, `sub1`, and `add1`. which we introduced in *The Little Schemer*:

```
(define atom?
  (lambda (x)
    (and (not (pair? x)) (not (null? x)))))
```

To find out whether your Scheme has the correct definition of `atom?`, try `(atom? (quote ()))` and make sure it returns `#f`. In fact, the material is also suited for modern Lisps such as Common Lisp. To work with Lisp, you will also have to add the function `atom?`:

```
(defun atom? (x)
  (not (listp x)))
```

Moreover, you may need to modify the programs slightly. Typically, the material requires only a few changes. Suggestions about how to try the programs in the book are provided in the framenotes. Framenotes preceded by "S:" concern Scheme, those by "L:" concern Common Lisp.

In chapter 4 we develop basic arithmetic from three operators: *add1*, *sub1*, and *zero?*. Since Scheme does not provide *add1* and *sub1*, you must define them using the built-in primitives for addition and subtraction. Therefore, to avoid a circularity, our basic arithmetic addition and subtraction must be written using different symbols: + and −, respectively.

We do not give any formal definitions in this book. We believe that you can form your own definitions and will thus remember them and understand them better than if we had written each one for you. But be sure you know and understand the *Laws* and *Commandments* thoroughly before passing them by. The key to learning Scheme is "pattern recognition." The *Commandments* point out the patterns that you will have already seen. Early in the book, some concepts are narrowed for simplicity; later, they are expanded and qualified. You should also know that, while everything in the book is Scheme, Scheme itself is more general and incorporates more than we could intelligibly cover in an introductory text. After you have mastered this book, you can read and understand more advanced and comprehensive books on Scheme.

We use a few notational conventions throughout the text, primarily changes in typeface for different classes of symbols. Variables and the names of primitive operations are in *italic*. Basic data, including numbers and representations of truth and falsehood, is set in sans serif. Keywords, i.e., **define**, **lambda**, **cond**, **else**, **and**, **or**, and **quote**, are in **boldface**. When you try the programs, you may ignore the typefaces but not the related framenotes. To highlight this role of typefaces, the programs in framenotes are set in a `typewriter` face. The typeface distinctions can be safely ignored until chapter 10, where we treat programs as data.

Finally, Webster defines "punctuation" as the act of punctuating; specifically, the act, practice, or system of using standardized marks in writing and printing to separate sentences or sentence elements or to make the meaning clearer. We have taken this definition literally and have abandoned some familiar uses of punctuation in order to make the meaning clearer. Specifically, we have dropped the use of punctuation in the left-hand column whenever the item that precedes such punctuation is a term in our programming language.

Food appears in many of our examples for two reasons. First, food is easier to visualize than abstract symbols. (This is not a good book to read while dieting.) We hope the choice of food will help you understand the examples and concepts we use. Second, we want to provide you with a little distraction. We know how frustrating the subject matter can be, and a little distraction will help you keep your sanity.

You are now ready to start. Good luck! We hope you will enjoy the challenges waiting for you on the following pages.

Bon appétit!

Daniel P. Friedman
Matthias Felleisen