[redacted](**Contents**)

Foreword

If you give someone a fish, he can eat for a day.
If you teach someone to fish, he can eat for a lifetime.

This familiar proverb applies also to data structures in programming languages.

If you have read *The Little Lisper* (recently revised and retitled: *The Little Schemer*), the predecessor to this book, you know that lists of things are at the heart of Lisp. Indeed, "LISP" originally stood for "LISt Processing." By the same token, I suppose that the C programming language could have been called CHAP (for "CHAracter Processing") and Fortran could have been FLOP (for "FLOating-point Processing").

Now C without characters or Fortran without its floating-point numbers would be almost unthinkable. They would be completely different languages, perhaps almost useless. What about Lisp without lists? Well, Lisp has not only lists but functions that perform computations. And we have learned, slowly and sometimes laboriously over the years, that while lists are the heart of Lisp, functions are the soul.

Lisp must, of course, have lists; yet functions are enough. Dan and Matthias will show you the way. *The Little Lisper* was truly a feast; but, as you will see, there is more to life than food.

Have you eaten? Very good. Now you are prepared for the real journey.

Come, learn to fish!

—Guy L. Steele Jr.

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*.

The goal of this book is to teach the reader to think about the nature of computation. 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 the power of recursion, the subtlety of control, and the true role of state. The language of mathematics is the opposite of natural language: it can express powerful formal ideas with only a few symbols. We could, for example, describe the semantic content of this book in less than a page of mathematics, but conveying how to harness the power of functions in the presence of state and control is nearly impossible. The marriage of technology and mathematics presents us with a third, almost ideal choice: a programming language. Programming languages seem the best way to convey the nature of computation. 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 about the nature of computation is Scheme. Scheme is symbolic and numeric—the programmer does not have to make an explicit mapping between the symbols and numerals of his own language and the representations in the computer. Scheme is primarily a functional language, but it also provides assignment, set!, and a powerful control operator, letcc (or call-with-current-continuation), so that programmers can explicitly characterize the change of state. Since our only concerns are the principles of computation, our treatment is limited to the whys and wherefores of just a few language constructs: car, cdr, cons, eq?, atom?, null?, zero?, add1, sub1, number?, lambda, cond, define, or, and, quote, letrec, letcc (or call-with-current-continuation), let, set!, and if. Our language is an *idealized* Scheme.

The Little Schemer and *The Seasoned Schemer* will not directly 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.

Acknowledgments

We particularly want to thank Bob Filman for contributing to the TeXery and Dorai Sitaram for his incredibly clever Scheme program SLaTeX. Kent Dybvig's Chez Scheme made programming in Scheme a most pleasant experience. We gratefully acknowledge criticisms and suggestions from Steve Breeser, Eugene Byon, Corky Cartwright, Richard Cobbe, David Combs, Kent Dybvig, Rob Friedman, Gustavo Gomez-Espinoza-Martinez, Dmitri Gusev, Chris Haynes, Erik Hilsdale, Eugene Kohlbecker, Shriram Krishnamurthi, Julia Lawall, Shinnder Lee, Collin McCurdy, Suzanne Menzel, John Nienart, Jon Rossie, David Roth, Jonathan Sobel, George Springer, Guy Steele, John David Stone, Vikram Subramaniam, Perry Wagle, Mitch Wand, Peter Weingartner, Melissa Wingard-Phillips, Beata Winnicka, and John Zuckerman.

Hints 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 five 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. Try the examples while you read. Schemes and Lisps are readily available. While there are minor syntactic variations between different implementations (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)))))
```

Those readers who have read *The Little LISPer* need to understand that the empty list, `()`, is no longer an atom. To find out whether your Scheme has the correct definition of `atom?`, try `(atom? (quote ()))` and make sure it returns `#f`. 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. The framenotes in this book, especially those concerning Common Lisp, assume knowledge of the framenotes in *The Little Schemer* or of the basics of Common Lisp.

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 Commandments thoroughly before passing them by. The key to programming is recognizing patterns in data and processes. The *Commandments* highlight the patterns. 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 (chapter 19 is not Lisp), the language incorporates more than needs to be covered in a text on the nature of computation.

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., **letrec**, **letcc**, **let**, **if**, **set!**, **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 completely set in a `typewriter` face. The typeface distinctions can be safely ignored until chapter 20, 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.

Once again, food appears in many of our examples, and we are no more health conscious than we were before. We hope the food provides you with a little distraction and keeps you from reading too much of the book at one sitting.

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