Simply Scheme

Brian Harvey
Matthew Wright

Foreword by Harold Abelson

Simply Scheme: Introducing Computer Science

SECOND EDITION

The MIT Press
Cambridge, Massachusetts
London, England

This book was typeset in the Baskerville typeface, using the JOVE text editor and the TEX document formatting system, for which we thank John Baskerville (1706–75), Jonathan Payne (1964–), and Donald Knuth (1938–), respectively.

Library of Congress Cataloging-in-Publication Data

Harvey, Brian, 1949–
Simply scheme : introducing computer science / Brian Harvey, Matthew Wright ;
foreword by Harold Abelson. — 2nd ed.
p. cm.
Includes bibliographic references and index.
ISBN 0–262–08281–0 (hc : alk. paper)
1. Scheme (Computer programming language) 2. Computer science.
I. Wright, Matthew. II. Title.
QA76.H3475 1999
005.13'3–dc21 99-10037
CIP

Contents

Appendices

Foreword

One of the best ways to stifle the growth of an idea is to enshrine it in an educational curriculum. The textbook publishers, certification panels, professional organizations, the folks who write the college entrance exams—once they've settled on an approach, they become frozen in a straitjacket of interlocking constraints that thwarts the ability to evolve. So it is common that students learn the "modern" geography of countries that no longer exist and practice using logarithm tables when calculators have made tables obsolete. And in computer science, beginning courses are trapped in an approach that was already ten years out of date by the time it was canonized in the mid-1980s, when the College Entrance Examination Board adopted an advanced placement exam based on Pascal.*

This book points the way out of the trap. It emphasizes programming as a way to express ideas, rather than just a way to get computers to perform tasks.

Julie and Gerry Sussman and I are flattered that Harvey and Wright characterize their revolutionary introduction to computer science as a "prequel" to our text *Structure and Interpretation of Computer Programs*. When we were writing *SICP,* we often drew upon the words of the great American computer scientist Alan Perlis (1922–1990). Perlis was one of the designers of the Algol programming language, which, beginning in 1958, established the tradition of formalism and precision that Pascal embodies. Here's what Perlis had to say about this tradition in 1975, nine years *before* the start of the AP exam:

> Algol is a blight. You can't have fun with Algol. Algol is a code that now belongs in a plumber's union. It helps you design correct structures that

* Since Hal wrote this Foreword, they've switched the AP exam to use C++, but the principle is the same.

> don't collapse, but it doesn't have any fun in it. There are no pleasures in writing Algol programs. It's a labor of necessity, a preoccupation with the details of tedium.

Harvey and Wright's introduction to computing emerges from a different intellectual heritage, one rooted in research in artificial intelligence and the programming language Lisp. In approaching computing through this book, you'll focus on two essential techniques.

First is the notion of *symbolic programming*. This means that you deal not only with numbers and letters, but with structured collections of data—a word is a list of characters, a sentence is a list of words, a paragraph is a list of sentences, a story is a list of paragraphs, and so on. You assemble things in terms of natural parts, rather than always viewing data in terms of its tiniest pieces. It's the difference between saying "find the fifth character of the third word in the sentence" and "scan the sentence until you pass two spaces, then scan past four more characters, and return the next character."

The second technique is to work with *higher-order functions*. That means that you don't only write programs, but rather you *write programs that write programs,* so you can bootstrap your methods into more powerful methods.

These two techniques belong at center stage in any beginning programming course, which is exactly where Harvey and Wright put them. The underlying principle in both cases is that you work with general parts that you extend and combine in flexible ways, rather than tiny fragments that you fit together into rigid structures.

You should come to this introduction to computing ready to think about ideas rather than details of syntax, ready to design your own languages rather than to memorize the rules of languages other people have designed. This kind of activity changes your outlook not only on programming, but on any area where design plays an important role, because you learn to appreciate the relations among parts rather than always fixating on the individual pieces. To quote Alan Perlis again,

> You begin to think in terms of patterns and idioms and phrases, and no longer pick up a trowel and some cement and lay things down brick by brick. The Great Wall, standing for centuries, is a monument. But building it must have been a bore.

Hal Abelson
Cambridge, MA

Preface

There are two schools of thought about teaching computer science. We might caricature the two views this way:

- **The conservative view:** Computer programs have become too large and complex to encompass in a human mind. Therefore, the job of computer science education is to teach people how to discipline their work in such a way that 500 mediocre programmers can join together and produce a program that correctly meets its specification.
- **The radical view:** Computer programs have become too large and complex to encompass in a human mind. Therefore, the job of computer science education is to teach people how to expand their minds so that the programs *can* fit, by learning to think in a vocabulary of larger, more powerful, more flexible ideas than the obvious ones. Each unit of programming thought must have a big payoff in the capabilities of the program.

Of course nobody would admit to endorsing the first approach as we've described it. Yet many introductory programming courses seem to spend half their time on obscure rules of the programming language (semicolons go *between* the instructions in Pascal, but *after* each instruction in C) and the other half on stylistic commandments (thou shalt comment each procedure with its preconditions and postconditions; thou shalt not use `goto`). In an article that was *not* intended as a caricature, the noted computer scientist Edsger Dijkstra argues that beginning computer science students *should not be allowed to use computers,* lest they learn to debug their programs interactively instead of writing

programs that can be proven correct by formal methods before testing.*

If you are about to be a student in an introductory computer science course, you may already be an experienced programmer of your home computer, or instead you may have only a vague idea of what you're getting into. Perhaps you suspect that programming a computer is like programming a VCR: entering endless obscure numeric codes. Even if you're already a computer programmer, you may not yet have a clear idea of what computer *science* means. In either case, what we want to do in this book is put our best foot forward—introduce you to some new ideas, get you excited, rather than mold you into a disciplined soldier of the programming army.

In order to understand the big ideas, though, we'll also have to expend some effort on technical details; studying computer science without writing computer programs is like trying to study German grammar without learning any of the words in the language. But we'll try to keep the ideas in view while struggling with the details, and we hope you'll remember them too.

One Big Idea: Symbolic Programming

We said that our approach to teaching computer science emphasizes big ideas. Our explanation of symbolic programming in the following paragraphs is in part just an illustration of that approach. But we chose this particular example for another reason also. Scheme, the programming language used in this book, is an unusual choice for an introductory computer science course. You may wonder why we didn't use a more traditional language, such as Pascal, Modula-2, or C. Our discussion of symbolic programming is the beginning of an answer to that question.

Originally computers were about numbers. Scientists used them to solve equations; businesses used them to compute the payroll and the inventory. We were rescued from this boring state of affairs mainly by researchers in *artificial intelligence*—people who wanted to get computers to think more nearly the way people do, about ideas in general rather than just numbers.

What does it mean to represent *ideas* in a computer? Here's a simple example: We want to teach the computer to answer the question, "Was so-and-so a Beatle?" We can't quite ask the question in English; in this book we interact with the computer using Scheme. Our interactions will look like this:

* "On the Cruelty of Really Teaching Computer Science," *Communications of the ACM,* vol. 32, no. 12, December, 1989.

You type: `(beatle? 'paul)`
Computer replies: `#t` (computerese for "true")

You type: `(beatle? 'elvis)`
Computer replies: `#f` ("false")

Here's the program that does the job:

```
(define (beatle? person)
  (member? person '(john paul george ringo)))
```

If you examine this program with a (metaphoric) magnifying glass, you'll find that it's really still full of numbers. In fact, each letter or punctuation character is represented in the computer by its own unique number.* But the point of the example is that you don't have to know that! When you see

```
(john paul george ringo)
```

you don't have to worry about the numbers that represent the letters inside the computer; all you have to know is that you're seeing a *sentence* made up of four *words*. Our programming language hides the underlying mechanism and lets us think in terms more appropriate to the problem we're trying to solve. That hiding of details is called *abstraction,* one of the big ideas in this book.

Programming with words and sentences is an example of symbolic programming. In 1960 John McCarthy invented the Lisp programming language to handle symbolic computations like this one. Our programming language, Scheme, is a modern dialect of Lisp.

Lisp and Radical Computer Science

Symbolic programming is one aspect of the reason why we like to teach computer science using Scheme instead of a more traditional language. More generally, Lisp (and therefore Scheme) was designed to support what we've called the radical view of computer science. In this view, computer science is about tools for expressing ideas. Symbolic programming allows *the computer* to express ideas; other aspects of Lisp's design help *the programmer*

* The left parenthesis is 40, for example, and the letter `d` is 100. If it were a capital `D` it would be 68.

express ideas conveniently. Sometimes that goal comes in conflict with the conservative computer scientist's goal of protection against errors.

Here's an example. We want to tell our computer, "To square a number, multiply it by itself." In Scheme we can say

```
(define (square num)
  (* num num))
```

The asterisk represents multiplication, and is followed by the two operands—in this case, both the same number. This short program works for any number, of course, as we can see in the following dialogue. (The lines with > in front are the ones you type.)

```
> (square 4)
16
> (square 3.14)
9.8596
> (square -0.3)
0.09
```

But the proponents of the 500-mediocre-programmer school* think this straightforward approach is sinful. "What!" they cry. "You haven't said whether `num` is a whole number or a number with a decimal fraction!" They're afraid that you might write the `square` program with whole numbers in mind, and then apply it to a decimal fraction *by mistake.* If you're on a team with 499 other programmers, it's easy to have failures of communication so that one programmer uses another's program in unintended ways.

To avoid that danger, they want you to write these two separate programs:

```
function SquareOfWholeNumber(num: integer): integer;
  begin
    SquareOfWholeNumber := num * num
  end;

function SquareOfDecimalNumber(num: real): real;
  begin
    SquareOfDecimalNumber := num * num
  end;
```

* Their own names for their approach are *structured programming* and *software engineering.*

Isn't this silly? Why do they pick this particular distinction (whole numbers and decimals) to worry about? Why not positive and negative numbers, for example? Why not odd and even numbers?

That two-separate-program example is written in the Pascal language. Pascal was designed by Niklaus Wirth, one of the leaders of the structured programming school, specifically to *force* programming students to write programs that fit conservative ideas about programming style and technique; you can't write a program in Pascal at all unless you write it in the approved style. Naturally, this language has been very popular with school teachers.* That's why, as we write this in 1993, the overwhelming majority of introductory computer science classes are taught using Pascal, even though no professional programmer would be caught dead using it.**

For fourteen years after the introduction of Pascal in 1970, its hegemony in computer science education was essentially unchallenged. But in 1984, two professors at the Massachusetts Institute of Technology and a programmer at Bolt, Beranek and Newman (a commercial research lab) published the Scheme-based *Structure and Interpretation of Computer Programs* (Harold Abelson and Gerald Jay Sussman with Julie Sussman, MIT Press/McGraw-Hill). That ground-breaking text brought the artificial intelligence approach to a wide audience for the first time. We (Brian and Matt) have been teaching their course together for several years. Each time, we learn something new.

The only trouble with *SICP* is that it was written for MIT students, all of whom love science and are quite comfortable with formal mathematics. Also, most of the students who use *SICP* at MIT have already learned to program computers before they begin. As a result, many other schools have found the book too challenging for a beginning course. We believe that everyone who is seriously interested in computer science must read *SICP* eventually. Our book is a *prequel;* it's meant to teach you what you need to know in order to read that book successfully.*** Generally speaking, our primary goal in Parts I–V has been preparation for *SICP,* while the focus of Part VI is to connect the course with the

---

* Of course, *your* teacher isn't an uptight authoritarian, or you wouldn't be using our book!

** Okay, we're exaggerating. But even Professor Wirth himself has found Pascal so restrictive that he had to design more flexible languages—although not flexible enough—called Modula and Oberon.

*** As the ideas pioneered by *SICP* have spread, we are starting to see other intellectually respectable introductions to computer science that are meant as alternatives to *SICP.* In particular, we should acknowledge *Scheme and the Art of Programming* (George Springer and Daniel P. Friedman, MIT Press/McGraw-Hill, 1989) as a recognized classic. We believe our book will serve as preparation for theirs, too.

kinds of programming used in "real world" application programs like spreadsheets and databases. (These are the last example and the last project in the book.)

Who Should Read This Book

This book is intended as an introduction to computer programming and to computer science for two kinds of students.

For those whose main interest is in some other field, we provide a self-contained, one-semester experience with computer programming in a language with a minimum of complicated notation, so that students can quickly come in contact with high-level ideas about algorithms, functions, and recursion. The book ends with the implementation of a spreadsheet program and a database program, so it complements a computer application course in which the commercial versions of such programs are used.

For those who intend to continue the study of computer science but who have no prior programming experience, we offer a preparatory course, less intense than a traditional CS 1 but not limited to programming technique; we give the flavor of computer science ideas that will be studied in more depth later in the curriculum. We also include an extensive discussion of recursion, which is a stumbling block for many beginning students.

The course at Berkeley for which we wrote this book includes both categories of students. About 90% of the first-year students who intend to major in computer science have already had a programming course in high school, and most of them begin with *SICP*. The other 10% are advised to take this course first. But many of the students in this course aren't computer science majors. A few other departments (business administration and architecture are the main ones) have a specific computer course requirement, and all students must meet a broader "quantitative reasoning" requirement; our course satisfies these requirements. Finally, some students come just out of curiosity about computers.

We assume that you have never programmed a computer. On the other hand, we do assume that you can *use* a computer; we don't talk about how to turn it on, how to edit text, and so on, because those details are too different from one computer model to another. If you've never used a computer before, you may wish to spend a few days with a book written specifically for your machine that will introduce you to its operation. It won't take more than a few days, because you don't have to be an expert before you read our book. As long as you can start up the Scheme interpreter and correct your typing mistakes, you're ready.

We assume that you're not a mathematics lover. (If you are, you might be ready to read *SICP* right away.) The earlier example about squaring a number is about as advanced as we get. And of course you don't have to do any arithmetic at all; computers are good at that. You'll learn how to *tell* the computer to do arithmetic, but that's no harder than using a pocket calculator. Most of our programming examples are concerned with words and sentences rather than with numbers. A typical example is to get Scheme to figure out the plural form of a noun. Usually that means putting an "s" on the end, but not quite always. (What's the plural of "French fry"?)

How to Read This Book

Do the exercises! Whenever we teach programming, we always get students who say, "When I read the book it all makes sense, but on the exams, when you ask me to write a program, I never know where to start." Computer science is two things: a bunch of big ideas, as we've been saying, and also a skill. You can't learn the skill by watching.

Do the exercises on a computer! It's not good enough to solve the exercises on paper, even if you feel sure your solution is correct. Maybe it's 99% correct but there's some little detail you've overlooked. When you run such a program, you won't get 99% of the answer you wanted. By trying the exercise on the computer, you get unambiguous feedback. If your program is correct, you get the response you expected. If not, not.

Don't feel bad if you don't get things right the first time. Even the most experienced programmers have to *debug* their programs—that is, fix the parts that don't work. In fact, an important part of what you'll learn from the exercises is the *process* of debugging your solutions. It would be too bad if all of your programs in this course worked the first time, because that would let you avoid the practice in debugging that you'll certainly need when you write more complicated programs later. Also, don't be afraid or ashamed to ask for help if you get stuck. That, too, is part of the working style of professional programmers.

In some of the chapters, we've divided the exercises into two categories, "boring" and "real." The boring exercises ask you to work through examples mechanically, to make sure you understand the rules. The real exercises ask you to *invent* something, usually a small computer program, but sometimes an explanation of some situation that we present. (In some chapters, the exercises are just labeled "exercises," which means that they're all considered "real.") We don't intend that the boring exercises be handed in; the idea is for you to do as many of them as you need to make sure you understand the mechanics of whatever topic you're learning.

Occasionally we introduce some idea with a simplified explanation, saving the whole truth for later. We warn you when we do this. Also, we sometimes write preliminary, partial, or incorrect example programs, and we always flag these with a comment like

```
(define (something foo baz)                    ;; first version
  ...)
```

When we introduce technical terms, we sometimes mention the origin of the word, if it's not obvious, to help prevent the terminology from seeming arbitrary.

This book starts easy but gets harder, in two different ways. One is that we spend some time teaching you the basics of Scheme before we get to two hard big ideas, namely, function as object and recursion. The earlier chapters are short and simple. You may get the idea that the whole book will be trivial. You'll change your mind in Parts III and IV.

The other kind of difficulty in the book is that it includes long programming examples and projects. ("Examples" are programs we write and describe; "projects" are programs we ask you to write.) Writing a long program is quite different from writing a short one. Each small piece may be easy, but fitting them together and remembering all of them at once is a challenge. The examples and projects get longer as the book progresses, but even the first example, tic-tac-toe, is much longer and more complex than anything that comes before it.

As the text explains more fully later, in this book we use some extensions to the standard Scheme language—features that we implemented ourselves, as Scheme programs. If you are using this book in a course, your instructor will provide our programs for you, and you don't have to worry about it. But if you're reading the book on your own, you'll need to follow the instructions in Appendix A.

There are several reference documents at the end of the book. If you don't understand a technical term in the text, try the Glossary for a short definition, or the General Index to find the more complete explanation in the text. If you've forgotten how to use a particular Scheme primitive procedure, look in the Alphabetical Table of Scheme Primitives, or in the General Index. If you've forgotten the name of the relevant primitive, refer to the inside back cover, where all the primitive procedures are listed by category. Some of our example programs make reference to procedures that were defined earlier, either in another example or in an exercise. If you're reading an example program and it refers to some procedure that's defined elsewhere, you can find that other procedure in the Index of Defined Procedures.

To the Instructor

The language that we use in this book isn't exactly standard Scheme. We've provided several extensions that may seem unusual to an experienced Scheme programmer. This may make the book feel weird at first, but there's a pedagogic reason for each extension.

Along with our slightly strange version of Scheme, our book has a slightly unusual order of topics. Several ideas that are introduced very early in the typical Scheme-based text are delayed in ours, most notably recursion. Quite a few people have looked at our table of contents, noted some particular big idea of computer science, and remarked, "I can't believe you wait so long before getting to *such and such*!"

In this preface for instructors, we describe and explain the unusual elements of our approach. Other teaching issues, including the timing and ordering of topics, are discussed in the Instructor's Manual.

Lists and Sentences

The chapter named "Lists" in this book is Chapter 17, about halfway through the book. But really we use lists much earlier than that, almost from the beginning.

Teachers of Lisp have always had trouble deciding when and how to introduce lists. The advantage of an early introduction is that students can then write interesting symbolic programs instead of boring numeric ones. The disadvantage is that students must struggle with the complexity of the implementation, such as the asymmetry between the two ends of a list, while still also struggling with the idea of composition of functions and Lisp's prefix notation.

We prefer to have it both ways. We want to spare beginning students the risk of accidentally constructing ill-formed lists such as

```
(((((() . D) . C) . B) . A)
```

but we also want to write natural-language programs from the beginning of the book. Our solution is to borrow from Logo the idea of a *sentence* abstract data type.* Sentences are guaranteed to be flat, proper lists, and they appear to be symmetrical to the user of the abstraction. (That is, it's as easy to ask for the last word of a sentence as to ask for the first word.) The `sentence` constructor accepts either a word or a sentence in any argument position.

We defer *structured* lists until we have higher-order functions and recursion, the tools we need to be able to use the structure effectively.** A structured list can be understood as a tree, and Lisp programmers generally use that understanding implicitly. After introducing `car-cdr` recursion, we present an explicit abstract data type for trees, without reference to its implementation. Then we make the connection between these formal trees and the name "tree recursion" used for structured lists generally. But Chapter 18 can be omitted, if the instructor finds the tree ADT unnecessary, and the reader of Chapter 17 will still be able to use structured lists.

Sentences and Words

We haven't said what a *word* is. Scheme includes separate data types for characters, symbols, strings, and numbers. We want to be able to dissect words into letters, just as we can dissect sentences into words, so that we can write programs like `plural` and `pig-latin`. Orthodox Scheme style would use strings for such purposes, but we want a sentence to look `(like this)` and not `("like" "this")`. We've arranged that in most contexts symbols, strings, and numbers can be used interchangeably; our readers never see Scheme characters at all.*** Although a word made of letters is represented internally as a symbol, while a word made of digits is represented as a number, above the abstraction line they're both words. (A word that standard Scheme won't accept as a symbol nor as a number is represented as a string.)

* Speaking of abstraction, even though that's the name of Part V, we do make an occasion in each of the earlier parts to talk about abstraction as examples come up.

** Even then, we take lists as a primitive data type. We don't teach about pairs or improper lists, except as a potential pitfall.

*** Scheme's primitive I/O facility gives you the choice of expressions or characters. Instead of using `read-char`, we invent `read-line`, which reads a line as a sentence, and `read-string`, which returns the line as one long word.

There is an efficiency cost to treating both words and sentences as abstract aggregates, since it's slow to disassemble a sentence from right to left and slow to disassemble a word in either direction. Many simple procedures that seem linear actually behave quadratically. Luckily, words aren't usually very long, and the applications we undertake in the early chapters don't use large amounts of data in any form. We write our large projects as efficiently as we can without making the programs unreadable, but we generally don't make a fuss about it. Near the end of the book we discuss explicitly the efficient use of data structures.

Overloading in the Text Abstraction

Even though computers represent numbers internally in many different ways (fixed point, bignum, floating point, exact rational, complex), when people visit mathland, they expect to meet numbers there, and they expect that all the numbers will understand how to add, subtract, multiply, and divide with each other. (The exception is dividing by zero, but that's because of the inherent rules of mathematics, not because of the separation of numbers into categories by representation format.)

We feel the same way about visiting textland. We expect to meet English text there. It takes the form of words and sentences. The operations that text understands include `first`, `last`, `butfirst`, and `butlast` to divide the text into its component parts. You can't divide an empty word or sentence into parts, but it's just as natural to divide a word into letters as to divide a sentence into words. (The ideas of mathland and textland, as well as the details of the word and sentence procedures, come from Logo.)

Some people who are accustomed to Scheme's view of data types consider `first` to be badly "overloaded"; they feel that a procedure that selects an element from a list shouldn't also extract a letter from a symbol. Some of them would prefer that we use `car` for lists, use `substring` for strings, and not disassemble symbols at all. Others want us to define `word-first` and `sentence-first`.

To us, `word-first` and `sentence-first` sound no less awkward than `fixnum-+` and `bignum-+`. Everyone agrees that it's reasonable to overload the name `+` because the purposes are so similar. Our students find it just as reasonable that `first` works for words as well as for sentences; they don't get confused by this.

As for the inviolability of symbols—the wall between names and data—we are following an older Lisp tradition, in which it was commonplace to `explode` symbols and to construct new names within a program. Practically speaking, all that prevents us from representing words as strings is that Scheme requires quotation marks around them. But

in any case, the abstraction we're presenting is that the data we're dissecting are neither strings nor symbols, but words.

Higher-Order Procedures, Lambda, and Recursion

Scheme relies on procedure invocation as virtually its only control mechanism. In order to write interesting programs, a Scheme user must understand at least one of two hard ideas: recursion or procedure as object (in order to use higher-order procedures). We believe that higher-order procedures are easier to learn, especially because we begin in Chapter 8 by applying them only to named procedures. Using a named procedure as an argument to another procedure is the way to use procedures as objects that's least upsetting to a beginner. After the reader is comfortable with higher-order procedures, we introduce `lambda`; after that we introduce recursion. We do the tic-tac-toe example with higher-order procedures and `lambda`, but not recursion.

In this edition, however, we have made the necessary minor revisions so that an instructor who prefers to begin with recursion can assign Part IV before Part III.

When we get to recursion, we begin with an example of embedded recursion. Many books begin with the simplest possible recursive procedure, which turns out to be a simple sequential recursion, or even a tail recursion. We feel that starting with such examples allows students to invent the "go back" model of recursion as looping.

Mutators and Environments

One of the most unusual characteristics of this book is that there is no assignment to variables in it. The reason we avoid `set!` is that the environment model of evaluation is very hard for most students. We use a pure substitution model throughout most of the book. (With the background they get from this book, students should be ready for the environment model when they see a rigorous presentation, as they will, for example, in Chapter 3 of *SICP*.)

As the last topic in the book, we do introduce a form of mutation, namely `vector-set!`. Mutation of vectors is less problematic than mutation of lists, because lists naturally share storage. You really have to go out of your way to get two

pointers to the same vector.* Mutation of data structures is less problematic than assignment to variables because it separates the issue of mutation from the issues of binding and scope. Using vectors raises no new questions about the evaluation process, so we present mutation without reference to any formal model of evaluation. We acknowledge that we're on thin ice here, but it seems to work for our students.

In effect, our model of mutation is the "shoebox" model that you'd find in a mainstream programming language text. Before we get to mutation, we use input/output programming to introduce the ideas of effect and sequence; assigning a value to a vector element introduces the important idea of state. We use the sequential model to write two more or less practical programs, a spreadsheet and a database system. A more traditional approach to assignment in Scheme would be to build an object-oriented language extension, but the use of local state variables would definitely force us to pay attention to environments.

* We don't talk about `eq?` at all. We're careful to write our programs in such a way that the issue of identity doesn't arise for the reader.

Acknowledgments

Obviously our greatest debt is to Harold Abelson, Gerald Jay Sussman, and Julie Sussman. They have inspired us and taught us, and gave birth to the movement to which we are minor contributors. Julie carefully read what we thought was the final draft, made thousands of suggestions, both small and large, improved the book enormously, and set us back two months. Hal encouraged us, read early drafts, and also made this a better book than we could have created on our own.

Mike Clancy, Ed Dubinsky, Dan Friedman, Tessa Harvey, and Yehuda Katz also read drafts and made detailed and very helpful suggestions for improvement. Mike contributed many exercises. (We didn't take their advice about everything, though, so they get none of the blame for anything you don't like here.)

Terry Ehling, Bob Prior, and everyone at the MIT Press have given this project the benefit of their enthusiasm and their technical support. We're happy to be working with them.

The Computer Science Division at the University of California, Berkeley, allowed us to teach a special section of the CS 3 course using the first draft of this book. The book now in your hands is much better because of that experience. We thank Annika Rogers, our teaching assistant in the course, and also the thirty students who served not merely as guinea pigs but as collaborators in pinning down the weak points in our explanations.

Some of the ideas in this book, especially the different approaches to recursion, are taken from Brian's earlier Logo-based textbook.* Many of our explanatory metaphors, especially the "little people" model, were invented by members of the Logo community.

---

* *Computer Science Logo Style, volume 1: Intermediate Programming,* MIT Press, 1985.

We also took the word and sentence data types from Logo. Although this book doesn't use Logo itself, we tried to write it in the Logo spirit.

We wrote much of this book during the summer of 1992, while we were on the faculty of the Institute for Secondary Mathematics and Computer Science Education, an inservice teacher training program at Kent State University. Several of our IFSMACSE colleagues contributed to our ideas both about computer science and about teaching; we are especially indebted to Ed Dubinsky and Uri Leron.

We stole the idea of a "pitfalls" section at the end of each chapter from Dave Patterson and John Hennessy.

We stole some of the ideas for illustrations from Douglas Hofstadter's wonderful *Gödel, Escher, Bach*.

David Zabel helped us get software ready for students, especially with compiling SCM for the PC.

We conclude this list with an acknowledgment of each other. Because of the difference in our ages, it may occur to some readers to suspect that we contributed unequally to this book—either that Matt did all the work and Brian just lent his name and status to impress publishers, or that Brian had all the ideas and Matt did the typing. Neither of these is true. Almost everything in the book was written with both of us in front of the computer, arguing out every paragraph. When we did split up to write some sections separately, each of us read and criticized the other's work. (We're a little surprised that we still like each other, after all the arguments!) Luckily we both like the Beatles, Chinese food, and ice cream, so we had a common ground for programming examples. But when you see an example about Bill Frisell, you can be pretty sure it's Matt's writing, and when the example is about Dave Dee, Dozy, Beaky, Mick, and Tich, it's probably Brian's.