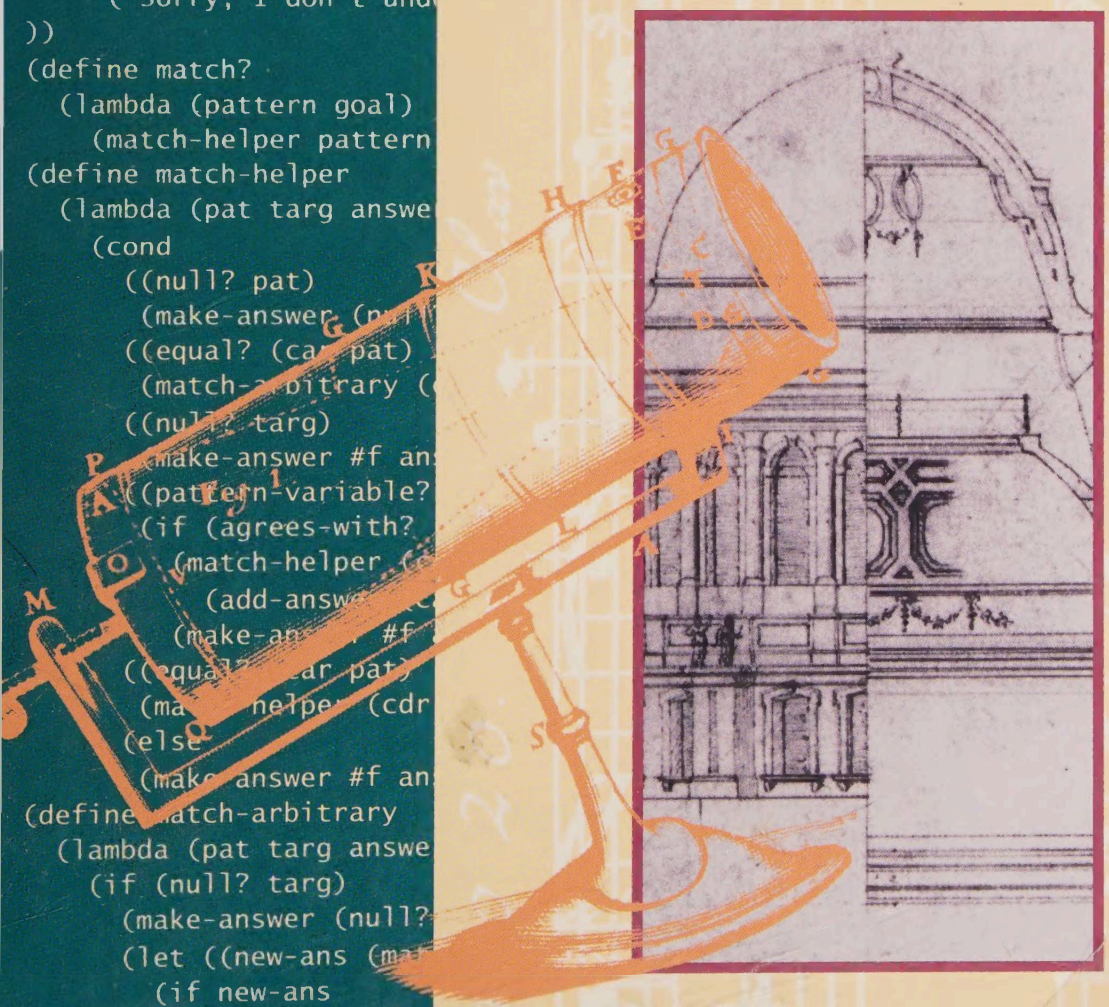
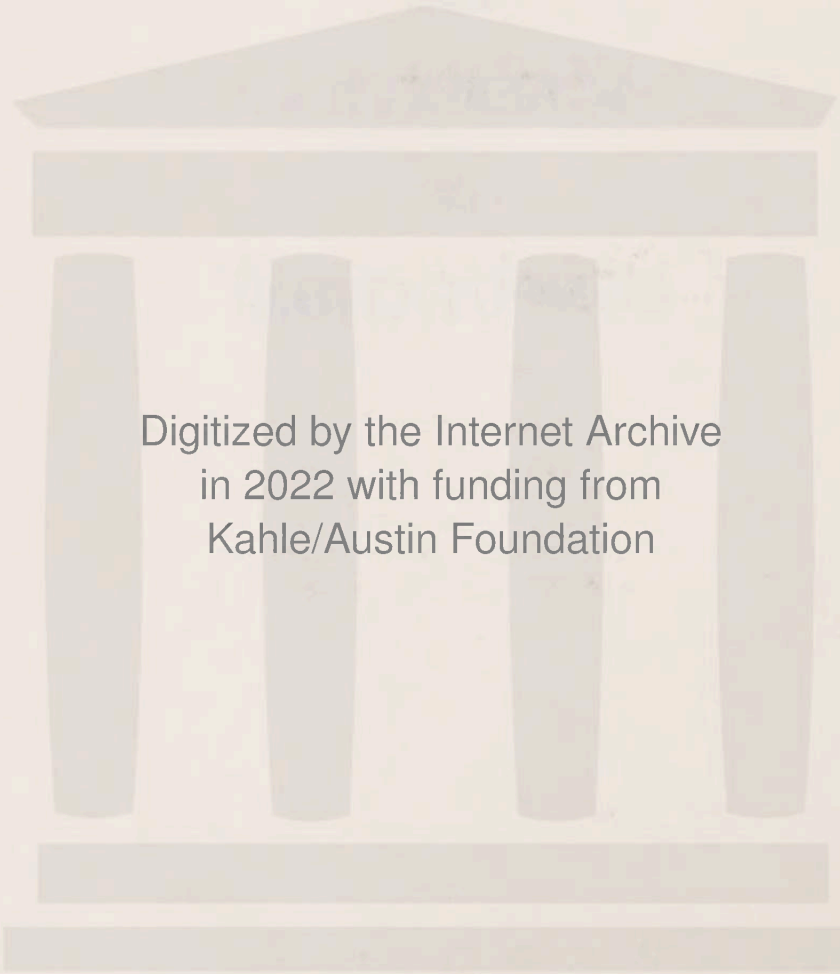
```
(define driving-rules
  '(
    ((?? right turn ?? re
     ("Yes, but you must
    ((?? right turn ??)
     ("Move into the righ
    ((?? left turn ??)
     ("Move into the left
    ((?? turn ??)
     ("You must signal be
    ((?? pass ?? hill ??)
     ("You can pass when
    ((?? pass ??)
     ("Always pass in the
    ((?? pedestrian ??)
     ("Yield. The pedestr
    ((?? class ?a ??)
     ("See the booklet GE
    ((??)
     ("Sorry, I don't und
))
(define match?
  (lambda (pattern goal)
    (match-helper pattern
(define match-helper
  (lambda (pat targ answe
    (cond
      ((null? pat)
       (make-answer (n
      ((equal? (ca pat)
       (match-arbitrary (
      ((nu targ)
       make-answer #f an
      ((pattern-variable?
       (if (agrees-with?
           (match-helper
             (add-answ
           (make-an #f
      ((equa ar pat)
      (ma helpe (cdr
      (else
      (make answer #f an
(define atch-arbitrary
  (lambda (pat targ answe
    (if (null? targ)
      (make-answer (null?
      (let ((new-ans (ma
        (if new-ans
          new-ans
          (match-arbitrar
(define make-answer
```

The SCHEMATICS of COMPUTATION



Vincent S. Manis • James J. Little

The SCHEMATICS of COMPUTATION



The SCHEMATICS of COMPUTATION

Vincent S. Manis
Langara College

James J. Little
University of British Columbia

An Alan R. Apt Book

Prentice Hall, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Manis, Vincent S.
The schematics of computation / Vincent S. Manis, James J. Little
p. cm.
"An Alan R. Apt book."
Includes bibliographical references and index.
ISBN 0-13-834284-9
1. Scheme (Computer program language) I. Little, James J.
II. Title.
QA76.73.S34M36 1995
005.13'3--dc20 94-21432
CIP

Publisher: Alan Apt
Project Manager: Mona Pompili
Developmental Editor: Sondra Chavez
Cover Designer: DeFranco Design Inc.
Copy Editor: Nick Murray
Production Coordinator: Lori Bulwin
Supplements Editor: Alice Dworkin
Editorial Assistant: Shirley McGuire

About the cover: Those who have advanced the sciences and arts have always used "schematic" methods to show their ideas. Christopher Wren's design for St Paul's Cathedral in London, Isaac Newton's reflector telescope, and Johann Sebastian Bach's score for the *Goldberg Variations* are all shown here in schematic form. Our computer programs are inspired by the schematic flavor of these works.

A Simon & Schuster Company
Englewood Cliffs, New Jersey 07632

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-834284-9

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada, Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda. *Rio de Janeiro*

This book is dedicated with love to my parents, Martin and Fay Manis. — VSM

I dedicate this book to my mother, Marguerite,
and to my wife, Debra, and my children, Brendan and Hanna,
who endured my long absences in my study. — JJL

Preface

The best laid schemes o' mice an' men
Gang aft a-gley.

— Robert Burns, "To a Mouse"

Students need an introductory course in computer science that exposes them to all of computer science. Computer science is not just about programming techniques. It rests on deep ideas and the nature of computation. We want students to understand these deep ideas, as well as grasp the practicality of computation and experience the pleasure of computing.

We wrote *The Schematics of Computation* for one reason—we wanted a book that presents the fundamental ideas of computer science in a way that students can understand. Introductory books in other fields such as physics introduce the areas of the field (mechanics, electricity and magnetism, and optics), while introductory computer science books often have chapters on how to use two-dimensional arrays.

We resolved to write a book that not only gives students essential programming skills, but also provides a vision of what computer science is about. We want to reach those individuals who are already committed to computer science, as well as those who plan to take computer science courses as part of a general education. We hope our enthusiasm for the field will encourage students to pursue advanced studies in computer science.

History and Objectives

This book had its genesis in a 1988 decision of the Department of Computer Science at the University of British Columbia to revitalize its introductory courses. A Scheme-based approach was adopted, and one of the authors (Little) set about teaching a prototype two-course sequence in 1989–90, using Abelson and Sussman's *Structure and Interpretation of Computer Programs*. We decided that we wanted a broader coverage of CS than Abelson and Sussman offers.

Since 1990, drafts of this book have been used at UBC, Langara, University College of the Okanagan, and the University College of the Cariboo, all in British Columbia, as well as at the University of Saskatchewan in Saskatoon, Saskatchewan. More recently, the book has been adopted by the University of Kansas and Vassar College.

Our model has been heavily influenced by the interim report of the ACM Task Force on the Core of Computer Science (CACM, January, 1989) and also by the ACM Curriculum '91 document.

The ACM Task Force Report identifies a number of areas in computer science, and develops three modes in which each of these areas can be studied, roughly corresponding to program design and implementation, empirical experimentation, and theoretical underpinnings. We have found this framework to be a useful way of designing a course that is effective in the classroom.

As we began planning this book, we identified several goals it had to satisfy.

- Whereas most texts give primary emphasis to programming skills, we give substantial weight to the analysis of programs and development of conceptual frameworks. We do not wish to shortchange the skills component of an introductory course. In fact, employers of students who have studied this material have told us that the students are *stronger* programmers than those who have taken the conventional Pascal course. However, we

also want students to learn how to read programs and understand critical issues such as efficiency.

- The introductory course sequence offers a chance to recruit students into computer science. It should therefore offer a view of the questions a computer scientist studies, and the modes of inquiry that she or he would use. Even for students who plan no further studies in the field, we believe that a broad introduction to the ideas of computer science provides a strong foundation for acquiring further computing skills, as well as skills in other areas.
- Although it would be impossible to cover all of the areas identified by the ACM Task Force, we have given attention to important areas of computer science that are not studied in the traditional CS1/CS2 sequence. Databases, artificial intelligence, and logic programming are the most obvious examples.
- We wanted to show that a small number of powerful ideas underlies much of computer science, just as concepts such as "force" and "energy" underlie physics. We present a model of computation that can be used not only for Scheme, but for almost all languages.
- We wanted a presentation that was accessible to the average university or college student, requiring only Grade 12 mathematics.

Key Differences From Conventional Approaches

There are six ways in which this book differs from conventional introductions.

- **Features if necessary, but not necessarily features**
  Our philosophy is deliberately *minimalist*—we avoid introducing new features or concepts until they are absolutely necessary. This approach applies to our presentation of both the programming language and the applications.
- **Theory and Practice**
  We strongly believe in *balancing theory and practice*, and only presenting theory that is directly useful in solving problems. Each theoretical topic grows out of a practical problem that is being solved.
- **Experiential Learning**
  We emphasize *learning by doing*. This emphasis can be seen in the hundreds of exercises included in the book. These exercises have one major purpose—students should not read more than a page or two without being asked to apply what has been learned. The Laboratory Manual consists of a set of labs that ask students to apply a concept that has been covered in class.
- **System Prototypes**
  We follow a *schematic* approach by presenting prototypes of real, working software systems. It is not feasible for students to study a *real* relational database system or inference engine in an introductory course. But it *is* possible for students to understand a simplified database or inference engine. Our goal is not to study all of the complexity of the "real" versions of these systems. Instead, we want students to understand what such a system does, and the basic principles that underlie its implementation.

- **Broad View of Field**
  We present a *rich, broad view of computer science.* An introductory text is not a closed universe, but a starting-point. Sidebars present relevant applications, software tools, theoretical questions, and social issues.
- **Case Studies**
  The *Case Studies* located at the end of the chapters present extended examples of how software is developed and how it can be used to solve interesting, practical problems. Case Studies provide additional programming examples and demonstrate how the techniques covered in chapters have further application. We have chosen applications across a wide variety of endeavors including simulation, airline route planning, personal schedule planning, and client/server computing.

Instructional Support

The book is only a part of the comprehensive package we have developed to support the course. This package includes:

- A *Laboratory Manual* with approximately thirty different lab assignments keyed to the text, along with sidebars on such practical topics as debugging code.
- An *Instructor's Guide* that describes the pedagogy for each section of the book, as well as providing a number of resources such as a detailed Scheme manual and a tutorial on C++ for Schemers that can be distributed to students.
- A *Software Package* that includes a Microsoft Windows-based Scheme system, code to adapt other common Scheme systems to support this book, and all of the programs studied or used in the text or lab manual. The Software Package is provided in disk form with the Instructor's Guide, or it can be obtained in electronic form from the Internet (see Page *xvi* for details). The Internet version is preferred, because it may incorporate bug fixes and additions which have not yet reached the published version.

Choice of Programming Language

Almost nothing can arouse more controversy than the choice of introductory programming language (except perhaps which is the best text editor). Our approach is soundly based upon λ-calculus models that have been realized in many languages, and provide simple, consistent semantics. We have chosen to use *Scheme*, a lexically-scoped dialect of Lisp. Our choice was governed by several factors:

- minimal syntax to make learning fairly easy
- procedures as first-class objects
- support for a wide variety of programming paradigms including functional, imperative, object-oriented, and logic programming
- list processing capabilities (i.e., garbage collection)
- reasonably standardized implementations, especially for MS-DOS, Windows, Macintosh, and UNIX platforms

Even though Abelson and Sussman's *Structure and Interpretation of Computer Programs* influenced many of the ideas we explore, we certainly did not feel bound

to use Scheme. Most other languages are bound too tightly to a single computational paradigm to be suitable for some of the areas we proposed to explore. In particular, our study of evaluators would have been seriously complicated if we had used a language with a great deal of syntax. The bottom line is that Scheme is a simple language that can be used in many interesting ways, and also allows us to produce sophisticated, complex software systems.

What about using a functional language? Most functional languages enforce a functional paradigm everywhere, whether or not it is appropriate. We agree that a database system can be modeled in a purely functional way, but don't agree that an introductory student will find such a model convincing. One outstanding exception, supporting all the paradigms we study in this book, is ML. The first six chapters of our book would suffer almost no changes if ML were used instead of Scheme. ML's syntax would most definitely complicate our discussion of evaluators. Most functional languages, including ML, have elaborate type systems. However useful these type systems may be to the experienced programmer, they are substantial barriers to the introductory student, and hence destroy the concept of a simple underlying model.

What about using an object-oriented language? We renew our objections to single-paradigm languages. More importantly, we want students to see that the language we study is well-founded (in a logic sense). It is easier to understand objects in terms of procedures and state than the reverse.

What about using an imperative, more "conventional" language like Pascal, Modula-2, or C++ as our teaching vehicle? Everything we do in this book could be done in one of these languages, though perhaps with great difficulty. Procedures as first-class citizens, and garbage collection make code much more perspicuous. We do not deny the importance of these languages (especially C++) for building sophisticated applications. However, we think that learning Scheme helps a student learn one of these languages.

We believe the ideas behind Scheme are not just academic curiosities, but are immensely practical. Scheme and Scheme-like languages have some industrial application. Further, many newer languages, ranging from ML and PostScript to NewtonScript and Dylan, use concepts that Scheme elegantly demonstrates. Scheme is already widely used as an introductory programming language.

Coverage and Topics

Although we were inspired by Abelson and Sussman's book, we chose different topics and a different organization. We include a sequence of topics that makes sense to students, provides strong programming skills, and gives a coherent picture of the field. We therefore adopted the following structure.

- Introduction to Programming (Chapters 1 through 3)
  - basic data types and programming concepts
  - computation as substitution
  - procedures and procedural abstraction
  - recursion
  - algorithmic complexity
  - distinction between value and effect
  - program organization
  - data abstraction

- List Processing (Chapter 4)
  - — pairs and symbols
  - — using pair structure to represent compound values
  - — lists and list recursion
- State, Mutation, and Objects (Chapters 5 and 6)
  - — mutating existing data structure
  - — variables and assignment
  - — how the environment model explains computation
  - — how environments model objects
  - — classes
  - — delegation
  - — object-oriented design
- Programming Languages (Chapter 7)
  - — defining the syntax and semantics of programming languages
  - — tree representation of programs
  - — interpreters and compilers
  - — a Scheme evaluator based upon the environment model
- Databases (Chapter 8)
  - — relational table model
  - — fundamental relational operators, select, project, and join
  - — implementation of a simple database system
  - — database analysis and design
- Algorithms (Chapter 9)
  - — algorithms for searching (linear search, binary search trees) and sorting (insertion sort, Quicksort)
  - — stacks and queues
  - — application of stacks to implementing a graphics language based upon Postscript
- Rule-Based Computing (Chapter 10)
  - — patterns and pattern matching
  - — predicate calculus
  - — logic programming languages
  - — backward and forward chaining
- Machines and Evaluators (Chapters 11 and 12)
  - — representation of data
  - — a simple computer
  - — programming in assembly language
  - — building a Scheme evaluator for a conventional machine
  - — introduction to operating systems

How To Use This Book

We have used this book in a number of ways at the University of British Columbia and at Langara College. Our original design was a two-course—CS1 and CS2—introductory sequence for science students (not necessarily CS majors). We have found little difference in the performance of students who have a programming background and those who do not.

We have also used this text in a one-semester course for students who have already taken CS1. Students in this course are primarily CS or Computer Engineering majors. Therefore, the course can be more intensive.

This text has been used in a course for students who are concurrently taking a conventional CS2 course. This course emphasizes the interrelations between the Scheme-based material and the C++-based content of the CS2 course. A similar course could be designed at the second year level for students with CS1 and CS2.

Regardless of the course design, *The Schematics of Computation* contains more material than can be covered. We have endeavored to provide a rich enough package of materials that instructors can tailor courses to their needs. We also want students to use this text as a reference, as well as a springboard to further study in CS.

Therefore, the book is divided into three parts:

- Chapters 1 through 7 are an integrated sequence that covers common programming paradigms: functional, imperative, and object-oriented.
- Chapters 8 through 10 provide an introduction to algorithms and data structures, through applications in data bases and logic.
- Chapters 11 and 12 introduce computer hardware and assembly-language programming, practical Scheme implementations, and operating systems.

In a two-semester course sequence, it is reasonable to cover the material right up to the imperative evaluator in Chapter 12. A one-semester course can go through the first two chapters much more quickly, and then cover Chapters 3 through 7 in depth. From there, the instructor has a number of choices:

- Proceed to Chapters 8, 9, and 10 if *algorithms* will be emphasized.
- Proceed to Chapters 11 and 12 if *machines* will be emphasized.
- A hybrid approach will go through Chapter 8 very quickly, and then cover Chapters 9 and 11 in depth, ending with the imperative evaluator in Chapter 12.

Many instructors will want to cover an additional language such as Modula-2, Pascal, C/C++, or Turing in the latter part of a two-course sequence. The Instructor's Guide contains a tutorial called *A Schemer's Guide to C++*, which can be distributed to students and used to introduce C++. The first part of the tutorial can be used with Chapter 7; the second part refers to Chapter 11.

To The Student

That lyf so short, the craft so long to lerne,
Th'assay so hard, so sharp the conquerynge.

— Geoffrey Chaucer, *The Parliament of Fowls*

We wrote *The Schematics of Computation* because we wanted to share our excitement about computer science with you. Programming languages, algorithms, and data structures are things of beauty. They are also immensely useful. Industries related to computing—such as information technology and software engineering—are in the process of becoming one of the biggest sectors in world economies. Things of beauty that can help us solve practical problems are rare indeed!

We want you to see how effective problem-solving techniques can help you plan useful programs, how a programming language can help you write those programs, how algorithms and data structures can help make those programs more efficient, and how conventional computers can help run programs.

We also want to introduce many of the current research areas of computer science. There are interesting and useful questions to ask in every area of the field: theory of algorithms, artificial intelligence and robotics, graphics, operating systems and networks, programming languages, data bases, and logic. Each of these areas has applications from designing sophisticated electronic circuits to making the computer systems in business, government, education, and elsewhere more responsive to the needs of their users.

It is important for you to understand that computer science, like all other fields of study, is based upon a few unifying ideas that recur over and over in computer science. One of our unifying ideas is the programming language we use—Scheme. This language is based upon a few fundamental ideas that can be combined in countless ways to produce interesting programming ideas. We will see a number of different programming styles, including functional, imperative, object-oriented, and logic programming.

* * *

The only way to learn computer science is to do it! We have scattered exercises throughout the text to help reinforce what has been covered. When you reach one of these exercises, *do it*! Then take a look at the answer in Appendix A.

At the end of each chapter, we have provided a list of the key words and Scheme features that have been covered in that chapter. Study these words and features until you understand them. We have included these lists not as something to be memorized, but rather as a measure of the important concepts you should be learning. We have also included some Problems for you to work on, a Self-Assessment (with answers in Appendix A) so you can see how effectively you have learned the basic chapter concepts, and some Programming Projects for you to investigate on your own.

Most chapters also end with a Case Study that explores ways in which the concepts you are learning can be used to solve complex problems. These Case Studies will repay your study of them.

Your instructor will give you lab assignments and programming projects to complete. The ideas and skills we show here will help you do interesting things and solve useful problems.

All of the programs we study are available electronically. Play with them! Try them out and modify them. You can learn a great deal about programming by studying the programs.

Although this is a large book, we only scratch the surface of computer science. Each chapter ends with a set of Suggested Readings that will let you explore further the topics raised in the chapter. The Bibliography at the end of the book lists all the books mentioned in the Suggested Readings, as well as many others that teach interesting programming languages, or cover other more advanced topics.

* * *

The only limit to what you can do with what you learn is your own creativity. There is an infinite variety of programs that have yet to be written. We can help to give you a start, but what you do with what you learn is up to you.

Acknowledgments

We owe thanks to those computer scientists who over the past fifty years have searched for a clear, understandable, basis of programming. In particular, we are indebted to those who have tried to use these models to introduce computing. The developers of Logo as a teaching tool showed us that students can learn to do interesting things with computers, when given a powerful programming environment. The culmination of this approach is Abelson and Sussman's *Structure and Interpretation of Computer Programs*.

We have also learned from Dennis Ritchie and Ken Thompson, whose original design for UNIX showed that power and simplicity need not be conflicting goals. The work of Donald Knuth has showed us the close relationship between mathematics and computer science, and between theory and practice.

* * *

The notes on which this book are based have been tested with many students, teaching assistants, and colleagues. These poor souls have reported numerous errors, obfuscations, and downright lies to us. We have endeavored to accommodate as many of their suggestions as possible.

Don Acton, Carl Alphonce, Art Boehm, Craig Boutilier, Roelof Brouwer, Dave Forsey, Murray Goldberg, Norm Hutchinson, David Kirkpatrick, Gerald Neufeld, Nick Pippenger, and George Tsiknis, the other instructors who have used this book, deserve our thanks. We owe a special debt to Nick, who several times drew attention to ways in which the manuscript could be substantially improved.

Our summer student helpers deserve much credit: Jason Holmes, Yaron Kiflawi, Cedric Lee, Carson Leung, Greg Reid, and Marko Riedel. Additional corrections and suggestions were provided by Adrienne Drobnies, Rick Gee, Joseph Manning, and Art Pope. We owe a particular debt of gratitude to Philip Greenspun who read and commented on an early draft of the book.

Dan Friedman provided us with a large number of suggestions and improvements.

Several reviewers for this book have influenced the current shape in ways too numerous to describe. One reviewer deserves special mention. Brian Harvey produced massive, detailed critiques of our work. Even that minority of his suggestions we couldn't adopt caused us to think carefully about what we were trying to do and why.

Our publisher, Alan Apt, and editor, Sondra Chavez, have pushed us to produce what we are convinced is a much better book than before. Mona Pompili, our project manager, worked with us very patiently as we typeset the book, and provided a number of suggestions that improved both the presentation and the content.

We owe all of these people our sincere thanks, but of course cannot hold them responsible for the final shape of the book.

We are particularly grateful to Maria Klawe and Bob Woodham of the Department of Computer Science, The University of British Columbia, and to Judy Boxler and Habib Kashani, Department of Computer Science, Langara College, for their strong moral support and encouragement of this project.

Support for this project was also provided by the Department of Computer Science, The University of British Columbia, and by Langara College.

How to Get the Software Package

The software covered in this book can be obtained from your instructor, or via Internet ftp. To obtain a copy via the Internet, you will require an Internet connection, and appropriate software. This discussion assumes you are using a character-based ftp program.

The Software Package consists of three files:

- `schmtics.zip`: the Scheme code presented in this book and the lab manual
- `winscm.zip`, a Microsoft Windows-based Scheme system for 386, 486, Pentium, and compatible computers
- `vslib.zip`, the "Schematics Adaptor Kit", a package that can be loaded into a standard Scheme system to add the extra features used in this book.

There is also a file `!Readme`, with late-breaking information about the software.

To get the software, you will need an Internet connection and an `ftp` program. The instructions given here assume a character-based `ftp` program; consult your documentation if you are using a Windows or Macintosh ftp program.

Start up your ftp program, and carry out the following steps (`ftp>` is a prompt from the program).

```
ftp> open ftp.cs.ubc.ca
ftp> cd pub/local/schematics
ftp> get !Readme
ftp> binary
ftp> get schmtics.zip
ftp> get winscm.zip
ftp> get vslib.zip
ftp> close
```

Once you have obtained these files, use PKZip or a similar program to unpack them, each into a separate directory. Each directory will contain a `!Readme` file with information on how to install the software.

Contents