Preface

At first glance, the title of this book is an oxymoron. After all, the term *abstraction* refers to an idea or general description, divorced from physical objects. On the other hand, something is concrete when it is a particular object, perhaps something that you can manipulate with your hands and look at with your eyes. Yet you often deal with concrete abstractions. Consider, for example, a word processor. When you use a word processor, you probably think that you have really entered a document into the computer and that the computer is a machine which physically manipulates the words in the document. But in actuality, when you "enter" the document, there is nothing new inside the computer—there are just different patterns of activity of electrical charges bouncing back and forth. Moreover, when the word processor "manipulates" the words in the document, those manipulations are really just more patterns of electrical activity. Even the program that you call a "word processor" is an abstraction—it's the way we humans choose to talk about what is, in reality, yet more electrical charges. Still, although these abstractions such as "word processors" and "documents" are merely convenient ways of describing patterns of electrical activity, they are also *things* that we can buy, sell, copy, and use.

As you read through this book, we will introduce several abstract ideas in as concrete a way as possible. As you become familiar and comfortable with these ideas, you will begin to think of the abstractions as actual concrete objects. Having already gone through this process ourselves, we've chosen to call computer science "the discipline of concrete abstractions"; if that seems too peculiar to fathom, we invite you to read the book and then reconsider the notion.

This book is divided into three parts, dealing with procedural abstractions, data abstractions, and abstractions of state. A *procedure* is a way of abstracting what's called a computational process. Roughly speaking, a process is a dynamic succession of events—a happening. When your computer is busy doing something, a process

is going on inside it. When we call a process a *computational* process, we mean that we are ignoring the physical nature of the process and instead focusing on the information content. For example, consider the problem of conveying some information to a bunch of other people. If you think about writing the message on paper airplanes and tossing it at the other people, and find yourself considering whether the airplanes have enough lift to fly far enough, then you are considering a mechanical process rather than a computational one. Similarly, if you think about using the phone, and find yourself worrying about the current carrying capacity of the copper wire, you are considering an electrical process rather than a computational one. On the other hand, if you find yourself considering the alternative of sending your message (whether by phone or paper airplane) to two people, each of whom send it to two more, each of whom send it to two more, and so forth, rather than directly sending the message to all the recipients, then you are thinking about a computational process.

What do computer scientists do with processes? First of all, they write descriptions of them. Such descriptions are often written in a particular programming language and are called procedures. These procedures can then be used to make the processes happen. Procedures can also be analyzed to see if they have been correctly written or to predict how long the corresponding processes will take. This analysis can then be used to improve the performance or accuracy of the procedures.

In the second part of the book, we look at various types of data. *Data* is the information processed by computational processes, not only the externally visible information, but also the internal information structures used within the processes. First, we explore exactly what we mean by the term data, concentrating on how we use data and what we can do with it. Then we consider various ways of gluing small pieces of atomic data (such as words) into larger, compound pieces of data (such as sentences). Because of our computational viewpoint, we write procedures to manipulate our data, and so we analyze how the structure of the data affects the processes that manipulate it. We describe some common data structures that are used in the discipline, and show how to allow disparate structures to be operated on uniformly in a mix-and-match fashion. We end this part of the book by looking at programs in a programming language as data structures. That way, carrying out the computational processes that a program describes is itself a process operating on a data structure, namely the program.

We start the third part of the book by looking at computational processes from the perspective of the computer performing the computation. This shows how procedurally described computations actually come to life, and it also naturally calls attention to the computer's memory, and hence to the main topic of this part, state. *State* is anything that can be changed by one part of a computation in order to have an effect on a later part of the computation. We show several important uses for state: making processes model real-world phenomena more naturally, making processes that are more efficient than without state, and making certain programs

divide into modules focused on separate concerns more cleanly. We combine the new material on state with the prior material on procedural and data abstraction to present *object-oriented programming*, an approach to constructing highly modular programs with state. Finally, we use the objects' state to mediate interactions between concurrently active subprocesses.

In summary, this book is designed to introduce you to how computer scientists think and work. We assume that as a reader, you become actively involved in reading and that you like to play with things. We have provided a variety of activities that involve hands-on manipulation of concrete objects such as paper chains, numbered cards, and chocolate candy bars. The many programming exercises encourage you to experiment with the procedures and data structures we describe. And we have posed a number of problems that allow you to play with the abstract ideas we introduce.

Our major emphasis is on *how* computer scientists think, as opposed to *what* they think about. Our applications and examples are chosen to illustrate various problem-solving strategies, to introduce some of the major themes in the discipline, and to give you a good feel for the subject. We use sidebars to expand on various topics in computer science, to give some historical background, and to describe some of the ethical issues that arise.

Audience

This book is primarily intended as the text for a first (and possibly only) undergraduate course in computer science. We believe that every college student should have a trial experience of what it's like to think abstractly, the way mathematicians and computer scientists think. We hope that the tangible nature of the computer scientist's abstractions will attract some of the students who choose to avoid math courses. Because of this, we don't require that our readers have taken a college-level math course. On the other hand, mathematics is used in computer science in much the same way it is used in biology, chemistry, and physics. Thus we do assume that our readers have a knowledge of high school algebra.

Although we've tried to reach a broad audience, this is *not* a watered-down text unsuitable for serious students planning to major in computer science. We reject the notion than an introduction for majors should be different from an introduction for non-majors. Beyond the obvious difficulty that most students will not have any reasonable basis for categorizing themselves without having taken even a single course, we feel strongly that the most important need of a prospective major is the same as that of a non-major: a representative trial experience of what it is like to think the computer science way. Those who part company with us after this book will have an appreciation for what we do; those who stay with us will know what lies ahead for them.

Like most introductory college-level books, we make some assumptions about the readers' backgrounds. As we have said before, we assume that the readers understand

the material that is typically taught in a high school algebra course. We also make some assumptions about the readers' attitudes towards mathematics; in short, they should be willing to use mathematics as a tool for analyzing and solving problems. We occasionally use some mathematical tools that aren't typically taught in high school. When we do this, we present the relevant material in the text and the students need to be willing to learn this material on the fly.

Similarly, we also assume that our readers may not have had much computing or programming experience, beyond playing an occasional computer game or using a word processor. However, we do not describe how to start a computer, how to use a Scheme programming environment, or similar mechanics. This kind of information varies greatly from machine to machine and is best taught by a person rather than a book. Again, keeping an open mind about learning is probably more important than any prior experience.

Additionally, we assume that students have had some experience in writing. When we teach a course based on this book, we rely heavily on writing assignments. Students are expected to be able to write descriptions of what their procedures do and need to be able to articulate clearly the problems they may have in order to get help in solving them. Most of our students find that their writing skill improves considerably over the semester.

Finally, although we attempt to be reasonably gentle toward those with little prior mathematical or computer programming experience, in our experience even those students who think of themselves as experts find much here that is not only unfamiliar, but also challenging and interesting.

In short: this is an introduction for everyone.

Technical Assumptions

To make full use of this book, you will need access to a computer with an implementation of the Scheme programming language; for the final chapter, you will also need an implementation of the Java™ programming language, version 1.1 or later. Most of our examples should work on essentially any modern Scheme, since we have used constructs identified in the so-called "R^{4}RS" standard for Scheme—the *Revised*^{4} *Report on the Algorithmic Language Scheme*, which is available on the web site for this book, located at `http://www.pws.com/compsci/authors/hailperin`. The following materials are available:

- all code shown in this text, together with some additional supporting code;
- information on obtaining various Scheme implementations and using them with this text;
- Java applets that provide instructional support, such as simulations;
- manipulatives (i.e., physical materials to experiment with);
- the Scheme language specification;

- bug-reporting forms and author contact information;
- a list of errata; and
- tips for instructors.

One notable exception is that we use graphics, even though there are no graphics operations defined in R^{4}RS. Nearly every modern Scheme will have some form of graphics, but the details vary considerably. We have provided "library" files on our web site for each of several popular Scheme systems, so that if you load the library in before you begin work, the graphics operations we presume in this book will be available to you. The nonstandard Scheme features, such as graphics, that we use in the book are explained in the Appendix, as well as being identified where they are first used.

Teaching with This Book

Enough material is here to cover somewhere in the range from two quarters to two semesters, depending on your pace. If you want to cut material to fit a shorter format, the dependencies among the chapters allow for a number of possibilities beyond simply truncating at some point:

- Chapter 10 has only weak ties to the later chapters, so it can be omitted easily.
- Chapter 11 is primarily concerned with computer organization and assembly language programming; however, there is also a section introducing Scheme's vectors. It would be possible to skip the machine-level material and cover just the vectors with only minor adverse impact.
- Chapter 12 can be omitted without serious impact on the later chapters.
- Chapter 13 divides roughly into two halves: elementary data structures (stacks and queues) and an advanced data structure (red-black trees). You can stop after the queues section if you don't want the more advanced material.
- Chapter 14 has a large section on how object-oriented programming is implemented, which can be omitted without loss of continuity.
- You can skip straight from Chapter 7 to the vector material in Chapter 11, provided you stop after the queues section in Chapter 13. (Chapter 8 is crucial for the red-black tree material in Chapter 13, and Chapter 9 is crucial for Chapter 14.)

All exercises, other than those in the separate "review problems" section at the end of each chapter, are an integral part of the text. In many cases skipping over them will cause loss of continuity, or omission of some idea or language feature introduced in the exercise. Thus as a general rule, even when you don't assign the exercises, you should consider them part of the reading.

Acknowledgments

Because the project of writing this book has extended over several years, many extremely helpful people have had the opportunity to give us a hand, and three slightly disorganized people (the authors) have had the opportunity to lose track of a few of them. So, before we get to all the people we managed to keep track of, we'd like to thank and apologize to the anonymous others who have slipped through the cracks. Also, we'd like to make the standard disclaimer: the people listed here deserve much of the credit, but absolutely none of the blame. If nothing else, we chose to ignore some of the good advice they offered.

This text has benefited from repeated class testing by Tim Colburn and Keith Pierce at the University of Minnesota at Duluth, and by Mike Hvidsten, Charley Sheaffer, and David Wolfe at Gustavus Adolphus College.

The ever-patient students at these two institutions also provided many valuable bug reports and suggestions. We'd particularly like to mention Rebekah Bloemker, Kristina Bovee, Gene Boyer, Jr., Brian Choc, Blaine Conklin, Scott Davis, Steve Davis, DeAnn DeLoach, John Engebretson, Lars Ericson, Melissa Evans, Bryan Kaehler, Andrew Kay, Tim Larson, Milo Martin, Jason Molesky, Oskar Norlander, Angela Peck, Ted Rice, Antony Sargent, Robert Shueey, Henrik Thorsell, Mark Tomforde, Dan Vanorny, and Cory Weinrich.

We received lots of valuable feedback from reviewers retained by the publishers. In addition to some who remained anonymous, these include Tim Colburn of the University of Minnesota at Duluth, Timothy Fossum of the University of Wisconsin at Parkside, Chris Haynes of Indiana University, Tim Hickey of Brandeis University, Rhys Price Jones of Oberlin College, Roger Kirchner of Carleton College, Stuart A. Kurtz of the University of Chicago, Keith Pierce of the University of Minnesota at Duluth, and John David Stone of Grinnell College.

Finally, a miscellaneous category of people made helpful suggestions without falling into any of the earlier categories: Hal Abelson and Bob Givan of MIT, Theodore Hailperin, and Carol Mohr.

We would like to extend a great big "Thanks!" to all the above people who contributed directly to the book, and also to the family members and colleagues who contributed indirectly through their support and encouragement.

Finally, we should mention some of the giants on whose shoulders we are standing. We all learned a great deal from Hal Abelson and Gerry Sussman of MIT—one of us as a student of theirs, all of us as students of their textbook [2]. Anyone familiar with their book will see plenty of echos in ours. The most significant new ingredient we added to their recipe is the interplay of proving with programming—which we learned from Bob Floyd and John McCarthy at Stanford.

Max Hailperin
Barbara Kaiser
Karl Knight