Part VII
Conclusion: Computer Science

This is the end of the technical material in this book. We've explored the big ideas of composition of functions, functions as data, recursion, abstraction, and sequential programming. As a review, you might enjoy rereading Chapter 1 to see how the formerly mysterious examples now make sense.

This book is intended both as a standalone programming course for people whose main interest is in something else and as preparation for those who intend to continue studying computer science. The final chapter is directed mainly at the latter group, to tell them what to expect. We confess, though, to the hope that some of the former may have enjoyed this experience enough to be lured into further study. If you're in that category, you'll be particularly interested in a preview of coming attractions.

Structure and Interpretation of Computer Programs

Second Edition

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

26 What's Next?

We've concluded this introduction to computer science with two examples of "real world" programming—spreadsheets and databases. What may have seemed like a pointless game near the beginning of the book allowed us to write these serious programs.

But we've only begun to explore the ideas that make up computer science. If you're interested in learning more, where do you go from here?

The Best Computer Science Book

The next step is to read *Structure and Interpretation of Computer Programs* by Harold Abelson and Gerald Jay Sussman with Julie Sussman (MIT Press/McGraw-Hill, Second Edition, 1996). Our book was inspired by theirs, and our main goal in writing this book has been to prepare you for that one. If you're a student and your school offers a course based on *SICP,* take it! If not, read the book on your own.

The big organizing idea of *SICP* is abstraction. We've used this idea in several ways. We've talked about *data abstraction* such as inventing the sentence and tree data types. We've also invented more specialized data types, such as positions in the tic-tac-toe program and cells in the spreadsheet program. We've discussed how higher-order procedures abstract a pattern of computation by using an extra argument to represent the function that's abstracted out.

What we've done in this book covers most of the main ideas in about the first hundred pages of *SICP.* But don't skip over those pages. In the footnotes alone you'll find ideas about numerical analysis, cryptography, epistemology (the study of what it means to know something), number theory, programming language design, and the analysis of algorithms.

Because there *is* some overlap between what we teach and what they teach, you may think at first that you can breeze through a course based on *SICP.* Don't fall into that trap; even from the beginning there will be some ideas that are new to you, and after about four weeks it will *all* be new to you.

The core ideas of this book are in the first chapter of *SICP:* functions, recursion, the substitution model, and higher-order functions. The second chapter is about lists and data abstraction, extending the ladder of data abstractions in both directions. (That is, in our book, lists are the fundamental data type used to construct the *sentence* and *tree* abstract data types. In *SICP* you first learn that lists themselves are built of an even more fundamental type, the *pairs* that we mentioned in a pitfall in Chapter 17. Then you learn about several more abstract data types that can be built out of lists.) The idea of data abstraction extends beyond inventing new types. For example, *SICP* uses data structures not only to contain passive information but also to control the algorithms used in a computation. You got a taste of this in the a-list of functions in the `functions` program.

The third chapter of *SICP* is largely about non-functional programming, a topic we only begin in this book. Specifically, *SICP* introduces object-oriented programming, a very popular technique in which the dichotomy between "smart" procedures and passive data is broken down. Instead of a single program controlling the computation, there are many *objects* that contain both algorithms and data; each object acts independently of the others. (This is a metaphor; there is still one computer implementing all this activity. The point is that the programmer is able to think in terms of the metaphor, and the mechanism that simulates the independent objects is hidden behind the object abstraction.)

You may have forgotten by now that *SICP* stands for *Structure and Interpretation of Computer Programs.* We've been talking about the "structure" part: how a program is organized to reflect the algorithm it implements. The "interpretation" half of the book explains how a programming language like Scheme really works—how a computer understands the program and carries out the activities that the program requests. The centerpiece of the interpretation part of the book is the Scheme interpreter in the fourth chapter, a program that takes any other Scheme program as its data and does what that program specifies. (The `compute` procedure that evaluates arithmetic expression trees in Chapter 18 and the `ss-eval` procedure in our spreadsheet program give a hint of what the *SICP* Scheme evaluator is like.) The book goes on to implement different programming languages and illustrate a variety of implementation techniques. Finally, the fifth chapter introduces the study of machine organization: how computer hardware carries out a program.

Beyond *SICP*

Computer science can be broadly divided into three areas: software, hardware, and theory. (Of course not everyone will agree with our division in detail.) This book is about software. *SICP* is also mostly about software, although it introduces some ideas from the other two areas. More advanced computer science courses will concentrate on one particular area.

Software includes programming languages, operating systems, and application programs. You know what a programming language is; there are courses on language design questions (why one language is different from another) and implementation (for example, how to write a Scheme interpreter).* An operating system is a program that maintains disk file structures and allows different programs to run on a computer without interfering with each other. Application programming techniques studied in computer science courses include database management, graphics programming, user interfaces, and artificial intelligence.

The study of computer hardware ranges from physical principles, through the workings of electronic components such as transistors, to large-scale circuits such as memories and adders, to the design of actual computers. This is a range of levels of abstraction, just as there are levels of abstraction in programming. At the higher levels of abstraction, you can think of an electronic circuit as a perfect representation of some function that it implements; that is, the signals coming out the output wires are functions of the signals coming in at the input wires. At a lower level of abstraction, you have to think about the limitations of physical devices. (For example, a device may fail if its output is attached to the inputs of too many other devices. It's as if you could use the return value from a function only a certain number of times.)

Theoretical computer science is a kind of applied mathematics. It includes analysis of algorithms, which is the study of how fast one program runs compared to another. (We touched on this topic when we mentioned that the simple selection sort is slower than the more complicated mergesort. Analysis of algorithms provides the tools to show exactly how much slower.) Theoreticians also study problems that can't be solved by any computer program at all; the most famous example is the *halting problem* of determining

* Many beginners think that studying computer science means learning a lot of different programming languages. Perhaps you start with BASIC, then you learn Pascal, then C, and so on. That's not true. The crucial ideas in computer science can be expressed in any modern language; it's unusual for a computer science student to be taught more than two languages throughout college.

in finite time whether some other program would run forever. *Automata theory* is the study of simplified pseudo-computers to prove theorems about the capabilities of computers in general.

A few topics don't fit readily into one of our three groups, such as *numerical analysis,* the study of how computers do arithmetic to ensure that the algorithms used really give correct results. This field involves aspects of programming, hardware design, and mathematics. *Robotics* involves artificial intelligence programming techniques along with electrical and mechanical engineering.

Standard Scheme

As we've mentioned, this book uses a lot of "primitive" procedures that aren't part of standard Scheme. These are procedures we wrote in Scheme and included in the file `simply.scm`, which is listed in Appendix C.

When you use Scheme in some other context, it probably won't come with these procedures included. There are three things you can do about this:

- Learn to love standard Scheme. That means not using the word and sentence functions, not using our implementation of trees, not using our higher-order procedures (`map` and `for-each` are the only standard ones), and not using `read-line`, `read-string`, `show`, `show-line`, `align`, or `close-all-ports`. Also, you may find it necessary to use data types we haven't fully explained: characters, strings, and pairs.
- Load our `simply.scm` and continue to work in the style we've used in this book. The advantage is that (in our humble opinion) we've provided some very convenient facilities. The disadvantage is that other Scheme programmers won't understand your programs unless they've read our book.
- Develop your own collection of tools to match the kind of programming you're doing in the new situations that come up. You can start with some of ours and add new procedures that you invent.

Actually, all three of these are good ideas. If you are going to continue working with Scheme, you'll certainly need to know what's in the standard and what's an extension to the language. After a while you're bound to develop a collection of tools that fit with your own preferred programming style. And sometimes `butfirst` will be just what you need for some project.

You may be curious about the *implementation* of our tool procedures. You've already seen some of them, such as the higher-order procedures whose implementation is described in Chapter 19. The input/output procedures (`show` and its friends) are straightforward once you've learned about the character data type. But you'll find that the implementation of words is quite complicated. Not only did we have to write the obvious `word`, `first`, and so on, but we also had to rewrite all of the arithmetic procedures so they work on words like `"007"` as well as ordinary numbers.

There are two documents that specify exactly what makes up standard Scheme. The first is updated fairly frequently to reflect ongoing experimentation in the language design. As we go to press, the most recent edition is called the *Revised*^{5} *Report on the Algorithmic Language Scheme.* The second document is meant to provide a more stable basis for people who depend on a language that's guaranteed not to become obsolete; it's IEEE Standard 1178-1990, *IEEE Standard for the Scheme Programming Language,* published by the Institute of Electrical and Electronic Engineers in 1991. Appendix A tells you how to find both documents.

Last Words

It's hard to wrap up something like this without sounding preachy. Perhaps you'll forgive us this one section since we've been so cool all through the rest of the book.

We thought of two general points that we want to leave you with. First, in our teaching experience at Berkeley we've seen many students learn the ideas of functional programming in Scheme, then seem to forget all the ideas when they use another programming language, such as C. Part of the skill of a computer scientist is to see past surface differences in notation and understand, for example, that if the best way to solve some problem in Scheme is with a recursive procedure, then it's probably the best way in C, too.

The second point is that it's very easy to get a narrow technical education, learn lots of great ideas about computer science, and still have a hard time dealing with the rest of reality. The utilitarian way to put it is that when you work as a computer programmer it's rare that you can just sit in your corner and write programs. Instead, you have to cooperate with other people on a team project; you have to write documentation both for other programmers and for the people who will eventually use your program; you have to talk with customers and find out what they really want the program to do, before you write it. For all these reasons you have to work at developing communication skills just as much as you work at your programming skills. But the utilitarian argument is just our sneaky way of convincing you; the truth is that we want you to know about things that have nothing to do with your technical work. Matt majored in music along with computer science; Brian has a degree in clinical psychology. After you read Abelson and Sussman, go on to read Freud and Marx.