Computer Science Logo Style
Symbolic Computing

Brian Harvey

Computer Science Logo Style

SECOND EDITION

Volume 1
Symbolic Computing

The MIT Press
Cambridge, Massachusetts
London, England

For information on program diskettes for PC and Macintosh, please contact the Marketing Department, The MIT Press, 55 Hayward Street, Cambridge, Massachusetts, 02142.

Drawings on pages 53 and 169 by James Brzezinski. Photograph of U.C. Berkeley on page 234 by Dennis Galloway, courtesy of the Public Information Office, University of California. Photographs of Stanford University on page 235 courtesy of the News and Publications Service, Stanford University.

This book was typeset in the Baskerville typeface.

The cover art is an untitled mixed media acrylic monotype by San Francisco artist Jon Rife, copyright © 1996 by Jon Rife and reproduced by permission of the artist.

Library of Congress Cataloging-in-Publication Data

Harvey, Brian, 1949–
Computer Science Logo Style / Brian Harvey. — 2nd ed.
p. cm.
Includes indexes.
Contents: v. 1. Symbolic computing. — v. 2. Advanced techniques — v. 3. Beyond programming.
ISBN 0–262–58151–5 (set : pbk. : alk. paper). — ISBN 0–262–58148–5 (v. 1 : pbk. : alk. paper). — ISBN 0–262–58149–3 (v. 2 : pbk. : alk. paper). — ISBN 0–262–58150–7 (v. 3 : pbk. : alk. paper)
1. Electronic digital computers–Programming. 2. LOGO (Computer programming language) I. Title.
QA76.6.H385 1997
005.13′3—dc20 96–35371
CIP

Contents

Appendices

Preface

This book isn't for everyone.

Not everyone needs to program computers. There is a popular myth that if you aren't "computer literate," whatever that means, then you'll flunk out of college, you'll never get a job, and you'll be poor and miserable all your life. The myth is promoted by computer manufacturers, of course, and also by certain educators and writers.

The truth is that no matter how many people study computer programming in high school, there will still be only a certain number of programming jobs. When you read about "jobs in high-tech industry," they're talking mostly about manufacturing and sales jobs that are no better paid than any other manufacturing jobs. (Often, these days, those jobs are exported to someplace like Taiwan where they pay pennies a day.) It's quite true that many jobs in the future will involve *using* computers, but the computers will be disguised. When you use a microwave oven, drive a recently built car, or play a video game, you're using a computer, but you didn't have to take a "computer literacy" course to learn how. Even a computer that looks like a computer, as in a word processing system, can be mastered in an hour or two.

This book is for people who are interested in computer programming because it's fun.

The Intellectual Content of Computer Programming

When I wrote the first edition of this book in 1984, I said that the study of computer programming was intellectually rewarding for young children in elementary school, and for computer science majors in college, but that high school students and adults studying on their own generally had an intellectually barren diet, full of technical details of some particular computer brand.

At about the same time I wrote those words, the College Board was introducing an Advanced Placement exam in computer science. Since then, the AP course has become popular, and similar official or semi-official computer science curricula have been adopted in other countries as well. Meanwhile, the computers available to ordinary people have become large enough and powerful enough to run serious programming languages, breaking the monopoly of BASIC.

So, the good news is that intellectually serious computer science is within the reach of just about everyone. The bad news is that the curricula tend to be imitations of what is taught to beginning undergraduate computer science majors, and I think that's too rigid a starting point for independent learners, and especially for teenagers.

See, the wonderful thing about computer programming is that it *is* fun, perhaps not for everyone, but for very many people. There aren't many mathematical activities that appeal so spontaneously. Kids get caught up in the excitement of programming, in the same way that other kids (or maybe the same ones) get caught up in acting, in sports, in journalism (provided the paper isn't run by teachers), or in ham radio. If schools get too serious about computer science, that spontaneous excitement can be lost. I once heard a high school teacher say proudly that kids used to hang out in his computer lab at all hours, but since they introduced the computer science curriculum, the kids don't want to program so much because they've learned that programming is just a means to the end of understanding the curriculum. No! The ideas of computer science are a means to the end of getting computers to do what you want.

Computer Science Apprenticeship

My goal in this series of books is to make the goals and methods of a serious computer scientist accessible, at an introductory level, to people who are interested in computer programming but are not computer science majors. If you're an adult or teenaged hobbyist, or a teacher who wants to use the computer as an educational tool, you're definitely part of this audience. I've taught these ideas to teachers and to high school students. What I enjoy most is teaching high school freshmen who bring a love of programming into the class with them—the ones who are always tugging at my arm to tell me what they found in the latest *Byte*.

I said earlier that I think that for most people programming as job training is nonsense. But if you happen to be interested in programming, studying it in some depth can be valuable for the same reasons that other people benefit from acting, music, or being a news reporter: it's a kind of intellectual apprenticeship. You're learning the discipline of serious thinking and of taking pride in your work. In the case of computer

programming, in particular, what you're learning is *mathematical* thinking, or *formal* thinking. (If you like programming, but you hate mathematics, don't panic. In that case it's not really mathematics you hate, it's school. The programming you enjoy is much more like real mathematics than the stuff you get in most high school math classes.) In these books I try to encourage this sort of formal thinking by discussing programming in terms of general rules rather than as a bag of tricks.

When I wrote the first edition of this book, in 1984, it was controversial to suggest that not everyone has to learn to program. I was accused of elitism, of wanting to keep computers as a tool for the rich, while condemning poorer students to dead-end jobs. Today it's more common that I have to fight the opposite battle, trying to convince people why *anyone* should learn about computer programming. After all, there is all that great software out there; instead of wasting time on programming, I'm told, kids should learn to use Microsoft Word or Adobe Illustrator or Macromind Director. At the same time, kids who've grown up with intricate and beautifully illustrated video games are frustrated by the relatively primitive results of their own first efforts at programming. A decade ago it was thrilling to be able to draw a square on a computer screen; today you can do that with two clicks of a mouse.

There are two reasons why you might still want to learn to program. One is that more and more application programs have programming languages built in; you can customize the program's behavior if you learn to speak its "extension" language. (One well-known example is the Hypertalk extension language for the Hypercard program; the one that has everyone excited as I'm writing this is the inclusion of the Java programming language as the extension language for the Netscape World Wide Web browser.) But I think a more important reason is that programming—learning how to express a method for solving a problem in a formal language—can still be very empowering. It's not the same kind of fast-paced fun as playing a video game; it feels more like solving a crossword puzzle.

I've tried to make these books usable either with a teacher or on your own. But since the ideas in these books are rather different from those of most computer science curricula, the odds are that you're reading this on your own. (When I published the first edition, one exception was that this first volume was used fairly commonly in teacher training classes, for elementary school teachers who'd be using Logo in their work.)

About the Second Edition

Three things have happened since the first edition of these books to warrant a revision. The first is that I know more about computer science than I did then! In this volume,

the topics of recursion and functional programming are explained better than they were the first time; there is a new chapter on higher order functions early in the book. There are similar improvements in the later volumes, too.

Second, I've learned from both my own and other people's experiences teaching these ideas. I originally envisioned a style of work in which high school students would take a programming course in their first year, then spend several years working on independent projects, and perhaps take a more advanced computer science class senior year. That's why I put all the programming language instruction in the first volume and all the project ideas in the second one. In real life, most students don't spread out their programming experience in that way, and so the projects in the second volume didn't get a chance to inspire most readers. In the second edition, I've mixed projects with language teaching. This first volume teaches the core Logo capabilities that every programming student should know, along with sample projects illustrating both the technical details and the range of possibilities for your own projects. The second volume, *Advanced Techniques,* teaches more advanced language features, along with larger and more intricate projects.

Volume three, *Beyond Programming,* is still a kind of sampler of a university computer science curriculum. Each chapter is an introduction to a topic that you might study in more depth during a semester at college, if you go on to study computer science. Some of the topics, like artificial intelligence, are about programming methods for particular applications. Others, like automata theory, aren't how-to topics at all but provide a mathematical approach to understanding what programming is all about. I haven't changed the table of contents, but most of the chapters have been drastically rewritten to improve both the technical content and the style of presentation.

The third reason for a second edition of these books is that the specific implementations of Logo that I used in 1984 are all obsolete. (One of them, IBM Logo, is still available if you try very hard, but it's ridiculously expensive and most IBM sales offices seem to deny that it exists.) The commercial Logo developers have moved toward products in which Logo is embedded in some point-and-click graphical application program, with more emphasis on shapes and colors, and less emphasis on programming itself. That's probably a good decision for their purposes, but not for mine. That's why this new edition is based on Berkeley Logo, a free implementation that I developed along with some of my students. Berkeley Logo is available for Unix systems, DOS machines, and Macintosh, and the language is exactly the same on all platforms. That means I don't have to clutter the text with footnotes like "If you're using this kind of computer, type that instead."

Why Logo?

Logo has been the victim of its own success in the elementary schools. It has acquired a reputation as a trivial language for babies. Why, then, do I use it as the basis for a series of books about serious computer science? Why not Pascal or C++ instead?

The truth is that Logo is one of the most powerful programming language available for home computers. (In 1984 I said "by far the most powerful," but now home computers have become larger and Logo finally has some competition.) It is a dialect of Lisp, the language used in the most advanced research projects in computer science, and especially in artificial intelligence. Until recently, all of the *books* about Logo have been pretty trivial, and they tend to underscore the point by strewing cute pictures of turtles around. But the cute pictures aren't the whole picture.

What does it mean for a language to be powerful? It *doesn't* mean that you can write programs in a particular language that do things you can't do in some other language. (In that sense, all languages are the same; if you can write a program in Logo, you can write it in Pascal or BASIC too, one way or another. And vice versa.) Instead, the power of a language is a way of measuring how much the language helps you concentrate on the actual problem you wanted to solve in the first place, rather than having to worry about the constraints of the language.

For example, in C, Pascal, Java, and all of the other languages derived originally from Fortran, the programmer has to be very explicit about what goes where in the computer's memory. If you want to group 20 numbers together as a unit, you must "declare an array," saying in advance that there will be exactly 20 numbers in it. If you change your mind later and want 21 numbers, too bad. You also have to say in advance that this array will contain 20 integers, or perhaps 20 numbers with fractions allowed, or perhaps 20 characters of text—but not some of each. In Logo the entire process of storage allocation is *automatic;* if your program produces a list of 20 numbers, the space for that list is provided with no effort by you. If, later, you want to add a 21st number, that's automatic also.

Another example is the *syntax* of a language, the rules for constructing legal instructions. All the Fortran-derived languages have a dozen or so types of instructions, each with its own peculiar syntax. For example, the BASIC `PRINT` statement requires a list of expressions you want printed. If you separate expressions with commas, it means to print them one way; if you separate them with semicolons, that means something else. But you aren't allowed to use semicolons in other kinds of statements that also require lists of expressions. In Logo there is only one syntax, the one that invokes a procedure.

It's not an accident that Logo is more powerful than Pascal or C++; nor is it just that Logo's designers were smarter. Fortran was invented before the mathematical basis of computer programming was well understood, so its design mostly reflects the capabilities (and the deficiencies) of the computers that happened to be available then. The Fortran-based languages still have the same fundamental design, although some of its worst details have been patched over in the more recent versions like Java and C++. More powerful languages are based on some particular mathematical model of computing and use that model in a consistent way. For example, APL is based on the idea of matrix manipulation; Prolog is based on predicate calculus, a form of mathematical logic. Logo, like Lisp, is based on the idea of composition of functions.

The trouble is that if you're just starting this book, you don't have the background yet to know what I'm talking about! So for now, please just take my word for it that I'm not insulting you by asking you to use a "baby" language. After you finish the book, come back and read this section again.

A big change since 1984 is that Logo is no longer the only member of the Lisp family available for home computers. Another dialect, Scheme, has become popular in education. Scheme has many virtues in its own right, but its popularity is also due in part to the fact that it's the language used in the best computer science book ever written: *Structure and Interpretation of Computer Programs,* by Harold Abelson and Gerald Jay Sussman with Julie Sussman (MIT Press/McGraw-Hill, 1985). I have a foot in both camps, since I am co-author, with Matthew Wright, of *Simply Scheme: Introducing Computer Science* (MIT Press, 1994), which is sort of a Scheme version of the philosophy of this book.

The main difference between Scheme and Logo is that Scheme is more consistent in its use of functional programming style. For example, in Scheme, every procedure is what Logo calls an operation—a procedure that returns a computed value for use by some other procedure. Instead of writing a program as a sequence of instructions, as in Logo, the Scheme programmer writes a single expression whose complexity takes the form of composition of functions.

The Scheme approach is definitely more powerful and cleaner for writing advanced projects. Its cost is that the Scheme learner must come to terms from the beginning with the difficult idea of function as object. Logo is more of a compromise with the traditional, sequential programming style. That traditional style is limiting, in the end, but people seem to find it more natural at first. My guess is that ultimately, Logo programmers who maintain their interest in computing will want to learn Scheme, but that there's still a place for Logo as a more informal starting point.

Hardware and Software Requirements

The programs in this series of books are written using Berkeley Logo, a free interpreter that is available on diskette from the MIT Press or on the Internet. (Details are in Appendix A.) Berkeley Logo runs on Unix systems, DOS machines, and Macintosh.

Since Berkeley Logo is free, I recommend using it with this book, even if you have another version of Logo that you use for other purposes. One of the frustrations I had in writing the first edition was dealing with all the trivial ways in which different Logo dialects differ. (For example, if you want to add 2 and 3, can you say `2+3`, or do you have to put spaces around the plus sign? Different dialects answer this question differently.) Nevertheless, the examples in this first volume should be workable in just about any Logo dialect with some effort in fixing syntactic differences. The later volumes in the series, though, depend on advanced features of Berkeley Logo that are missing from many other dialects.

The Berkeley Logo distribution includes the larger programs from these books. When a program is available in a file, the filename is shown at the beginning of the chapter. (There are only a few of these in the first volume, but more in later volumes.)

Words of Wisdom

The trick in learning to program, as in any intellectual skill, is to find a balance between theory and practice. This book provides the theory. One mistake would be to read through it without ever touching a computer. The other mistake would be to be so eager to get your hands on the keyboard that you just type in the examples and skip over the text.

There are no formal exercises at the ends of chapters. That's because (1) I hate a school-like atmosphere; (2) you're supposed to be interested enough already to explore on your own; and (3) I think it's better to encourage your creativity by letting you invent your own exercises. However, at appropriate points in the text you'll find questions like "What do you think would happen if you tried thus-and-such?" and suggestions for programs you can write. These questions and activities are indicated by this symbol: ☞ (the finger of fate). You'll get more out of the book if you take these questions seriously.

If you're not part of a formal class, consider working with a friend. Not only will you keep each other from handwaving too much but it's more fun.

Acknowledgments

The people who read and commented on early drafts of this book include Hal Abelson, Sharon Yoder, Michael Clancy, Jim Davis, Batya Friedman, Paul Goldenberg, Tessa Harvey, Phil Lewis, Margaret Minsky, and Cynthia Solomon. I am especially grateful to Paul Goldenberg and Cindy Carter for their professional, financial, and emotional support during the months I spent as a guest in their home while working on this project, keeping them from their own work and tying up Paul's computer equipment. This book wouldn't exist without them. Special mention also goes to Hal Abelson, without whose support this book wouldn't have found a publisher.

The main ideas in this book, and some of the specific examples, first surfaced in the form of self-paced curriculum units for a programming class at the Lincoln-Sudbury Regional High School, in Sudbury, Massachusetts. Alison Birch, Larry Davidson, and Phil Lewis were my colleagues there. (So, later, was Paul.) All of them helped debug the curriculum by finding mistakes and by pointing out the parts that were correct but incomprehensible. Larry, especially, was my mentor and untiring collaborator, helping me survive my first real teaching job, even though he had his own work and wasn't officially part of the computer department at all. I'm also grateful to the many students who served as guinea pigs for the curriculum, and to David Levington, then the district superintendent, who was generous with equipment and with administrative freedom in support of an untested idea.

My work at Lincoln-Sudbury would not have been possible without the strong support of computer scientists at the Massachusetts Institute of Technology, especially but not only the ones at the Logo Laboratory. Equipment grants from the Digital Equipment Corporation and from Atari, Inc., were also crucial to this work.

And thanks, also, to my faculty supervisors in the Graduate Group in Science and Mathematics Education, at the University of California at Berkeley, for their patience and understanding while I worked on this instead of my thesis.

Second Edition

In 1992 one of my then-undergraduate students, Matt Wright, suggested that we collaborate on a textbook for Berkeley's introductory programming course for non-majors. The book would use Scheme, the same language used in our first course for students in the computer science major, but would be based on the ideas in the first edition of this book. The result of that collaboration, *Simply Scheme,* was published in 1994.

In writing *Simply Scheme,* Matt and I reconsidered every detail of the presentation used in *Computer Science Logo Style.* We added a greater emphasis on higher order functions, and we completely reorganized the chapters on recursion. Large example programs were added to the text, along with suggestions for student projects.

Most of the changes in this second edition were inspired by the work that Matt and I did together for the Scheme book. In a few cases I have lifted entire paragraphs from it! Matt also read early drafts of some of the new chapters in this edition, and this text benefits from his comments.

Berkeley Logo, the interpreter used in this edition, is a collective effort of many people, both at Berkeley and across the Internet. My main debt in that project is to three former students: Dan van Blerkom, Michael Katz, and Doug Orleans. At the risk of missing someone, I also want to acknowledge substantial contributions by Freeman Deutsch, Khang Dao, Fred Gilham, Yehuda Katz, George Mills, Sanford Owings, and Randy Sargent.

Computer Science Logo Style
Symbolic Computing