Computer Science Logo Style
Beyond Programming

Brian Harvey

Computer Science Logo Style

SECOND EDITION

Volume 3

Beyond Programming

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.

Copyright credits for quoted material on page 349.

This book was typeset in the Baskerville typeface.

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

Preface

The phrase "computer science" is still, in some circles, battling for acceptance. Some people, not necessarily antagonistic to computers, consider it an illegitimate merger of two disconnected ideas (much as I feel myself about the phrase "computer literacy"). They don't see where the *science* comes in; what is taught in computer science departments is mostly how-to tricks of the trade, comparable to medical or legal training. Such training is valuable to the individual and to society, but the trainees are not learning to be scientists.

My own feeling is that there is some truth on both sides. There *is* some science in computer science. Abelson and Sussman, in *Structure and Interpretation of Computer Programs,* use the words "complexity" and "process" to explain what it is that computer scientists study. A process need not take place inside a computer, but it happens that computer processes are particularly amenable to formal study and they shed light on the idea of process in general. On the other hand, Abelson and Sussman are exceptional. A great deal of what is called computer science *is* much more a matter of programming techniques; many computer science students are first offered courses in several different programming languages and then taught specific techniques for particular problem domains, like graphics or data base systems or compiler construction. And many students who find themselves in computer science departments because they love programming computers are impatient with theory and weak in mathematical sophistication. Such students are perfectly satisfied with the how-to approach. (I don't mean this as an insult. I have in mind some excellent students I've taught who are brilliant programmers, very intelligent people, but happen not to have a theoretical bent.)

My goal in this book is to provide a bridge over which a lover of practical programming can cross into the world of theory. I envision someone who got bored or confused early in the high school math curriculum and was left with a distaste for formal thinking, but who is nevertheless a closet formalist when programming, getting the same joy from representing an intellectual problem in executable form that a traditional mathematician gets from representing a similar problem in an axiom system. I've tried to discuss the concerns of more abstract computer science using the language of concrete Logo programs that embody those concerns. This is an ambitious goal and I'm not sure how well I've succeeded.

For example, in automata theory there is an elementary result called Kleene's Theorem that establishes the equivalence of two different representations for a certain class of problems. One representation, the finite-state machine, is a *procedural* one, a sequence of steps, like a Pascal program. The other, the regular expression, is *declarative,* describing the desired result rather than the sequence of steps needed to get there, like a Prolog program. The representations are equivalent in the sense that any problem that can be described as a regular expression can be solved by a finite-state machine, and vice versa. (Not all problems are in this category.) Automata theory texts offer a formal proof of Kleene's Theorem using mathematical induction. What I offer is a Logo program that takes a regular expression as input and actually works out an equivalent finite-state machine for that particular expression. The program and the formal proof are similar in structure, embodying many of the same ideas. But the program is concrete and manipulable.*

Don't get the impression that the book contains nothing but formal mathematics, even in executable guise. The first two chapters (on automata theory and discrete mathematics) concern relatively abstract topics, although each has practical applications; the later chapters are more directly about the process of computer programming.

About This Series

The book you are reading is the final volume in a three-volume series, *Computer Science Logo Style.* In the introductions to the earlier volumes I have distinguished two approaches to computer science: the software engineering approach and the artificial intelligence approach. The former makes the idea of *algorithm* the central one. It thrives in a context of well-defined problems; a computer program proposed as a solution to such a problem

* Of course, computer programs are not concrete for everyone. Despite what I said above about students who like programming but don't like abstraction, anyone who has learned to see programs as concrete objects is well on the way to mastering mathematical abstraction too; that's what gives me hope about this enterprise. The fact that the Piagetian boundary between concrete objects and abstract ideas is not the same for everyone, and that what was formerly abstract can become concrete in a suitable learning environment (such as the one made possible by computers), was the insight that led Seymour Papert to espouse computer programming as an activity for children.

can be clearly judged correct or incorrect, and it can be more or less efficient than some other proposed solution. The artificial intelligence approach is harder to describe in one sentence. It embraces vaguely defined problems; it emphasizes an interactive process in which the programmer and the computer are participants.

Of course the terms in which I describe the two approaches are not value-free. A software engineer would use different language. Nor would all experts necessarily accept my dichotomy in the first place. Alan Perlis says, in his foreword to *Structure and Interpretation,* "After all, the critical programming concerns of software engineering and artificial intelligence tend to coalesce as the systems under investigation become larger. This explains why there is such growing interest in Lisp outside of artificial intelligence."

I have also described the sequence of stages through which I think an apprentice programmer travels. Volume 1 of this series, *Symbolic Computing,* teaches the rules of the game. It is addressed to a reader who has probably done some computer programming before, but is just starting to get serious about it. It differs from most introductory Logo programming books in that the latter focus attention on a particular programming goal, usually turtle graphics, and try to make the language itself as transparent as possible. I prefer to make the rules of the language an explicit object of study, since the design of any language embodies the designer's ideas about the structure of computer science.

Volume 2, *Advanced Techniques,* combines additional tutorial chapters about advanced Logo features with a collection of more or less practical programming projects. As in any field, apprentices learn by doing more than they learn by reading books; yet they require the attention of a master to see that they are learning a good style of work and not practicing bad habits. To speak of apprenticeship in this century sounds quaint, and in fact most of our master programmers do not take on apprentices. That's a pity, I think; too many would-be apprentices don't find suitable guidance. (They sometimes try to fill the gap by learning how to break into some company's computer, and then they get in trouble.) No book can be as good as living contact with a master programmer, but Volume 2 is my attempt.

This volume, *Beyond Programming,* is for the reader with a few substantial programming projects, or more than a few, behind him or her, who is starting to feel bored with programming for its own sake but isn't sure what to do instead. Some people never do experience that sense of constriction, and that's okay, as I said earlier. But I love mathematics myself, and I confess that I'm always a little disappointed if one of my best students doesn't come to share that love. In this volume my goal is to tempt them.

How to Read This Book

Slowly.

Each chapter introduces some rather sophisticated ideas. You may find that reading the chapter once leaves you with only a vague understanding. There are two things you can do about that: Read it again and experiment with the programs in the chapter. Test the limits of the programs; see what problems you can solve that are similar to the ones I solve, and what problems don't yield to the techniques I use. Think about how you could extend the techniques. You should understand how each program works, but don't fall into the trap of thinking that the program is the most important thing in the chapter. You should also understand how the program fits into a broader theoretical framework—how it embodies the *ideas* of the chapter.

The programs in each chapter are available on diskette and through the Internet along with Berkeley Logo, a free Logo interpreter that runs on PC, Macintosh, and Unix systems. Please note, though, that in a few places I show alternate versions of a program in the text. In those situations the diskette contains the final version, but it's worth your while to work through the development of the program by typing in the earlier versions yourself. (I don't do this with enormous programs.)

Most of the chapters concentrate on a single idea selected from a broad topic. Earlier I mentioned Kleene's Theorem as an example; that theorem is a very small piece of automata theory, but it takes up the bulk of the chapter. Only in the final pages do I hint at some of the other topics within automata theory. I think it's better to teach you one idea in depth than to give a handwavy picture of an entire area of study. You should expect to have to explore each area further by reading books about that topic; this book ends with a bibliography to help you.

The specific ideas I present are generally not at the cutting edge of current research in computer science; instead, they are older, more fundamental ideas. In part this is inherent in the introductory nature of the book. In part it reflects the limitations of my own knowledge. And in part it reflects the limitations of the home computers I expect my readers to have available to them. Then, sometimes the older ideas are easier to present in a complete, coherent, concrete form. For example, in the chapter on artificial intelligence I present an implementation of a program from 1964. In that program, the method used in the understanding of English sentences is closely connected to the particular problem (solving algebra word problems) that that program handles. A more modern English sentence parsing program is not only slower and more complicated but also is hard to demonstrate unless it's attached to an equally slow and complicated expert system or other purposeful program. I hint about the newer techniques, but I don't demonstrate them.

Chapters make occasional references to earlier chapters, so it's best if you read the book in order, although the references are rarely so substantial that you can't survive skipping a chapter. (But definitely read Chapter 4 before Chapter 5.)

What Isn't Included

When I first conceived of this series I described the third volume as "the first chapter of every graduate computer science course." As it turned out, the actual book does not pretend to cover anything like the entire field of computer science. It contains a selection of topics that I know about and find most worthwhile.

Some topics are omitted because I just don't care about them personally. For example, I don't have anything to say about numerical analysis. I'm glad there are people in the world who are concerned to ensure that when I use the square root primitive in Logo I get the right answer, but I am not such a person myself. That doesn't mean you have to share my taste, but if you want to know about numerical analysis you'll have to read someone else's book.

Other topics are omitted because I couldn't find any way to illustrate the topic through Logo programming on a microcomputer, which was one of the constraints I set for myself in planning this series. For example, one of the areas of computer science I find *most* interesting is operating systems. My high school students in Sudbury had access to a Unix timesharing system on a minicomputer and they did significant software development in that environment. But I don't know how to make that particular experience available on a single-user microcomputer in which the operating system is someone's trade secret.

Finally, some topics just didn't fit. I originally planned to have a chapter on graphics programming, but I decided that there are many books on the subject at both a popular and a professional level of expertise, and I had less to say about it than about some other areas.

The bibliography in Appendix A includes some pointers to information about some of the missing topics. In any case, the purpose of this book isn't to teach you everything there is to know about computers. It's to nurture in you a sense of what computer science is like, or at least one approach to computer science, in the hope that you'll be inspired to pursue the study in the "regular" college-level texts.

Computers and People

> Steve is a college sophomore, an engineering student who had never thought much about psychology. In the first month of an introductory computer-science course he saw how seemingly intelligent and autonomous systems could be programmed. This led him to the idea that there might be something illusory in his own subjective sense of autonomy and self-determination.
>
> Steve's classmate Paul had a very different reaction. He too came to ask whether free will was illusory. The programming course was his first brush with an idea that many other people encounter through philosophy, theology, or psychoanalysis: the idea that the conscious ego might not be a free agent. Having seen this possibility, he rejected it, with arguments about free will and the irreducibility of people's conscious sense of themselves. In his reaction to the computer, Paul made explicit a commitment to a concept of his own nature to which he had never before felt the need to pay any deliberate attention. For Paul, the programmed computer became the very antithesis of what it is to be human. The programmed computer became part of Paul's identity as not-computer.
>
> Paul and Steve disagree. But their disagreement is really not about computers. It is about determinism and free will. At different points in history this same debate has played on different stages. Traditionally a theological issue, in the first quarter of this century it was played out in debate about psychoanalysis. In the last quarter of this century it looks as though it is going to be played out in debate about machines.
>
> —*The Second Self: Computers and the Human Spirit,* Sherry Turkle, p. 23.

The psychology of computer programming, the sociology of the computer-intensive society, the economics of automation, the philosophy of mind, the ethics of computer use: these are the topics I find most interesting and most important in thinking about computers. That's why I'm a teacher instead of a computer programmer in industry.

In the original plan for this book there was to have been a chapter called "Computers and People" at the end of the book. I feel strongly that it's irresponsible to train people in the skills of computer technocracy without also encouraging their sensitivity to the human implications of their work. I ended up not writing that chapter, for several reasons. First, it would have to be very different in tone from the hands-on, experimental style of the rest of the book. I was afraid that, tacked on at the end, it would sound preachy, or worse, tokenistic and hypocritical. So instead there is this shorter discussion in the preface, where an author is allowed to sound preachy.

Second, I'm not sure that the relevant issues can be presented usefully to apprentices in the form of abstract reading. There is a lot of literature on this side of computing, some of which is listed in the bibliography. But the books, like all theoretical psychology or philosophy, are hard reading for people whose relevant practical experience is just beginning. Instead I think the best way to teach about the human side of computing is through sensitive adult attention to the actual experiences that take place in the computer center. (In general I've tried to write these books in a way that leaves open exactly who is reading them. I think this sort of approach to computer science can be useful to a wide range of people, kids and adults, in and out of formal educational settings. But I guess right now I am talking primarily to the teachers of high school students and undergraduates.)

At my high school computer center the kids liked to write video game programs. For a while some of the authors of these games included in the programs a list of which other kids were or weren't allowed to play the games. This practice let the game authors feel powerful and important, but of course it wasn't very helpful to the community spirit in the computer center. I didn't want to forbid the practice, making the issue one of rules and authority. Instead, in conversation with the students I tried to turn their attention away from ideas of intellectual property and entitlement—"It's my program and I have the right to decide who can use it"—and toward a sense of their own need for a strong community. Every time you write a program you're building on the work of last year's students who developed some of the techniques you use, on the work of people outside the school who designed the programming language, operating system, and so forth, and on the generosity of the adults in your community who paid for the equipment you use. In the end, I think the issue was settled not so much by my eloquence as by the example set by some other students who became important members of the community through their willingness to help others by teaching, encouraging, and sharing their own work. The kids all learned that it's possible to be respected, admired, and loved instead of respected but resented.

At many schools, when teachers express concern about the social issues in the computer center, the main focus of that concern is around the question of software piracy. Kids show up at school with a pirated version of the latest microcomputer game program, very proud of themselves for having it, and the teachers try to get the kids not to be proud of their theft of intangible property. But it seems to me that the other side of the issue, the spirit that's held up to kids as good computer citizenship, is marked by secrecy, distribution of programs in compiled form only, copy protection that works against networking, paranoia, and plain greed. I don't like *either* side of that dichotomy.

By contrast, in the university computer centers built around timesharing systems or networked workstations I see much more of a spirit of sharing, openness, community, programs provided with source code so that people can build on other people's work, trust, and an ideal of service to the community. That's another reason I chose to set up a Unix system in Sudbury. The ethical issues that arise in such a setting revolve around privacy of information. Kids find it a challenge to break into other people's accounts just as they do in the real-world computer systems that get into the newspapers, but at school the person who gets angry is another student rather than some faceless administrator. And students also experience the positive moral force of software sharing and collaboration.

I'm exaggerating the differences; I know that there is cooperation among microcomputer users and greed in the large computer world. Also, recent hardware developments are making the boundary less clear; home computers and workstations are built using the same processor chips. But the software is different and I think the style of human interaction is different as well. Still, the technical details of the facility are less important than the teacher's willingness to be a humane model and not just a fount of expertise. One of the virtues of that quaint idea of apprenticeship was that the apprentice was involved in the *entire* way of life of the master; there was no artificial separation between professional concerns and human concerns. What goes on among the people in a computer center is at least as important as what goes on between person and machine.

Acknowledgments

As for the previous two volumes, my greatest debts are to Hal Abelson and Paul Goldenberg. Both of them read the manuscript carefully through several drafts. Hal is great at noticing the large problems; he makes comments like "throw out this whole chapter" and "you are putting the cart before the horse here." Paul's comments were generally on a more detailed level, pointing out sections in which potentially valuable content was sabotaged by a presentation that nobody would understand. Together they have improved the book enormously.

Some of the examples in this book are ones that were posed to me by other people in other contexts. Horacio Reggini raised the issue of listing (not merely counting) the combinations of *r* elements of a list; Dick White asked me to investigate just how secure the Simplex lock is; Chris Anderson taught the probability class where the question about multinomial expansions arose. I'm grateful to Anita Harnadek, whom I've never met, for a logic problem I use to demonstrate inference systems. (She is, by the way, the author of a fantastic textbook called *Critical Thinking* that I recommend to teachers of almost any subject: math, English, or social studies.) Jim Davis's Logo interpreter in Logo (in the *LogoWorks* anthology I co-edited) was an inspiration for the Pascal compiler.

I'm grateful to Dan Bobrow, Sherry Turkle, and Terry Winograd for permission to quote from their work here. In particular, Bobrow's doctoral thesis forms the basis for my chapter on artificial intelligence, and I'm grateful for the program design as well as my extensive quotations from the thesis itself. He was also very patient in answering technical questions about details of a program he wrote over 20 years ago.

Mike Clancy taught me about generating functions and used them to find the closed form definition for the multinomial problem; Michael Somos, via the `sci.math` newsgroup, provided the closed form solution to the Simplex lock problem. Paul Hilfinger straightened me out about parser complexity.

Computer Science Logo Style
Beyond Programming