Logo
AND MODELS OF COMPUTATION

An Introduction to Computer Science

Michael E. Burke

L. Roland Genise

Logo
AND MODELS OF COMPUTATION

An Introduction to Computer Science

Michael E. Burke

San Jose State University

L. Roland Genise

San Jose Unified School District

Addison Wesley Publishing Company

Menlo Park, California • Reading, Massachusetts
Don Mills, Ontario • Wokingham, England
Amsterdam • Sydney • Singapore • Tokyo
Madrid • Bogotá • Santiago • San J

We dedicate this book to

Deanna, Katie, and Kevin

Millie, Livia, Valerie, Roland, Ronnie, and Lisa

M.E.B.
L.R.G.

This book is published by the Addison-Wesley INNOVATIVE DIVISION,

Design and Production by TechArt,
San Francisco, California.

ISBN 0-201-20791-5

ABCDEFGHIJKL-AL-8909876

Foreword

About the Text

The goals of *Logo and Models of Computation* are to provide an introduction to computer science, to illustrate the relationship of computation to mathematics, to aid in the development of a useful problem solving discipline, and to provide an intellectual framework in which new concepts in computer science can easily be assimilated. The Logo programming language is considered by us to be the ideal vehicle to reach these goals.

The major themes of the text include understanding the difference between functional programming and procedural programming, appreciating and making use of models of computation, thinking about problem solving at various levels of abstraction, and using symbolic computation.

The functional view of programming is characterized by defining operations that produce a result without modifying anything in the computational environment. This style of programming parallels the mathematical concept of function. Furthermore, the mathematical model of simplification and substitution provides a simple way to describe how a computer carries out a computation. This in turn provides an appreciation of the tools of mathematics. The tools of functional programming consist of recursion, conditional branching, and function application. Coupled with the flexibility of lists in the representation of data, the student discovers a conceptually simple form of computation that is very powerful. The concept of recursion is closely related to mathematical induction and Logo provides a very nice environment to discuss mathematical induction.

The procedural view of programming is characterized by defining procedures that execute sequences of statements designed to change the state of one or more objects in the programming environment. The assignment statement is the primary modifier of the world. The substitution and simplification model of computation does not hold up in this setting, and we must turn to another model that views computation as a sequence of states.

Problem solving involves thinking about a problem at several levels of abstraction. Programming can be especially useful in learning this

art, but there are two important other ingredients that must be present: The language must support programming at various levels, and the person guiding the exploration of a problem must be able to point out that this is indeed what is happening.

As an example, suppose that the problem concerns creating a game. One aspect is that the roll of a pair of dice will determine a player's options. The ladder of problems associated with the main task is as follows:

1. How is the roll of two dice going to be used?
2. How do you obtain and represent the roll of two dice?
3. How do you obtain and represent the roll of one die?
4. How do you generate a random number in a particular range (an integer between one and six in this case)?
5. How do you generate a random integer in Logo?

The problems are listed in a top-down fashion. Each can be attacked independently of the others. As long as the ladder of subproblems is understood, the order in which they are solved does not matter. In fact each task can be handled by a different person. Person 1 will say to person 2, "Give me two numbers representing the roll of two dice and I'll handle the rest." Person 2 tells person 3, "Give me a way to get the roll of one die, and I'll handle the rest." Person 3 tells person 4, "Give me a way to generate a random number in a particular interval, and I'll handle the rest." Finally, person 4 tells person 5, "Give me a way to generate a random integer in Logo, and I'll handle the rest."

The problem just described will not be solved until all subproblems have been solved, but it does not have to be solved in a top-down fashion or a bottom-up fashion. A middle-out, or "do the even numbered subproblems first," works just as well. In fact, the problem solver is most likely to start with the subproblem he or she understands best.

Logo supports this concept of problem solving better than either FORTRAN or Pascal. BASIC does not support it at all. We have more to say about this later in this preamble.

In the body of the text, we sometimes attack problems at the very bottom. This means we are learning about the level of support provided by Logo. At other times, we attack a problem in the middle. This means that it is only after we solve the problem at hand that we discover the real problem. This process repeated itself several times during the writing of

the book. There are a couple of cases where we knew what we were doing from the start. In this case the problem was carefully outlined and implemented in a top-down fashion.

The emphasis of the applications and programs is on writing programs that are symbolic in nature—for example, the manipulation of mathematical objects such as sets, rational numbers, and polygons. In the case of polygons, we do not mean drawing polygons but manipulating them as one would manipulate the turtle. For example, we want to be able to move them around and change their sizes. The properties of polygons are separated from their pictures. They are called *graphical objects*.

We describe programs that deal with nonmathematical objects as well. We experiment with creating a multiple turtle world on top of the single Logo turtle. We solve the Towers of Hanoi problem by writing programs that manipulate disks and towers. We look at the missionaries and cannibals problem in symbolic terms as well.

The applications and programs have been chosen to promote abstract thinking, to broaden the view of computation, and to prepare for further study in LISP, mathematics, artificial intelligence, and computer science.

Who is this Book For?

The material in this book has been used successfully as an introduction to computer science in a course for all students (as well as computer science majors) at San Jose State University and as a text for in-service teacher-training institutes, high school computer courses, and some middle school computing courses.

For the university level course in computer science, the entire book can be covered in an eighteen-week semester. New thinking will need to be done if students have had previous exposure to BASIC and Pascal. Some do not see the light until the last week of the course. For those who choose to go on to a course in LISP and Artificial Intelligence, the rewards are great. The transition is smooth and natural. The usual initial reactions is, "This is just like Logo."

LISP and Artificial Intelligence are currently viewed as advanced courses in most college curriculums. By the time students take these courses, they have been so indoctrinated in the ways of BASIC,

FORTRAN and Pascal that they become lost in a forest of parentheses, and the transition to this rich and exciting field is a difficult one at best. The introductory course and this book were begun because of the near-overwhelming difficulties encountered in the LISP and AI courses. The transition into these courses is now much easier for both students and instructors.

The book has been used in three in-service teacher institutes. The institutes consisted of four weeks of coursework on Logo, mathematics, and problem solving and nine follow-up sessions during the school year. When used in this manner, it is necessary to move quickly (emphasizing the relationship between Logo and mathematics) through the first part of the book and get into the interesting problems in the latter part of the book.

The first thing to admit in this context is that you cannot produce advanced programmers in four weeks. The next thing to admit is that you do not need to. Teachers are in the enviable position of being able to learn while they teach. What they need are reasons to explore and experiment and a solid foundation to lean on. In four weeks they learn to program in Logo and experience a large amount of the material given in the introductory course. If they are motivated they will embark on teaching a variety of projects depending on their individual teaching situation and interests.

We believe the prospective teacher should also be exposed to this material. But this presents a different set of problems and needs another approach. The mathematical background of this group usually varies widely. The emphasis again should be on the relationship of computation and mathematics and problem solving, and the material should be presented in a deliberate fashion.

We do not believe it is important for every teacher (even every mathematics teacher) to be skilled in programming and problem solving, but we do think that every school should have such a teacher. We also feel that every teacher should be exposed to using the computer as a problem-solving tool.

Until we have teachers skilled in using the computer for problem solving, computational literacy will continue to mean: how to turn on a computer, how to insert a diskette in the disk drive, how to be drilled by a computer, how to run a program, and how to write a program. The situation parallels that of mathematics in the elementary school. Since

few (if any) elementary schools that have a mathematics teacher, it is not at all surprising that mathematics is viewed by most of the world as the mastery of arithmetic.

The material can be used at the middle school, and the emphasis is the same as at the university level. However, the material is covered at a much slower pace, and time is taken to explain and explore mathematical concepts that are new to these students. It is as much a course in mathematics as in computation.

The material is very appropriate for an introduction to computer science at the high school level. The only difference between high school and college use would be in the rate of coverage.

There are no mathematics prerequisites to the text, but there are two exercises that require trigonometry (they are footnoted), two projects (sections 9.7 and 9.8) that require some calculus, and the last two sections of Chapter 4 assume knowledge of graphing quadratic functions (algebra I). This material can be easily omitted. If a student is taking calculus simultaneously, sections 9.7 and 9.8 should definitely be covered. The presentation of the material is designed to solidify and expand the student's mathematical background, and is valuable in supplementing the mathematics curriculum.

About the Language

The most commonly used programming languages on microcomputers and hence in the public schools, BASIC, and Pascal, do not have the sophistication or the flexibility to develop the kinds of skills we have been talking about. More importantly, these languages encourage primitive levels of problem-solving techniques that tend to limit a person's ability to think on higher levels of abstraction.

The first programming language learned has a strong influence in shaping a person's thinking processes, and higher levels of thinking about a problem are difficult to assimilate if the current frame of reference is a primitive one. Thinking about problems at various levels of abstraction is a major theme of the book. BASIC provides the most primitive level of programming of any of the so-called high-level languages and teaches primitive levels of thinking. Programming in Pascal requires more thinking about writing syntactically correct programs than thinking about how to solve the problem at hand. The data structures of Pascal are also not flexible enough to allow easy representation of symbolic data.

Logo is a direct descendant of a family of languages beginning with LISP in 1959. These languages have been used in the implementation of our most sophisticated programming applications. As a result, Logo has inherited a good deal of the flexibility and sophistication of these languages.

Since many versions of Logo exist (the fundamental concepts are in all of them) and there is no agreed-upon common subset, we are forced to choose a particular version to describe in detail. We have chosen Apple Logo because it can be used with the 64K Apple II Plus computer. It is also appropriate to use the text with Apple Logo II (for Apple IIe and IIc users), IBM Logo (for IBM PC-compatible users), and Atari Logo. In fact, students having these systems at home will encounter no difficulty in adjusting. Appendixes C and D describe the differences encountered if Apple Logo II or IBM Logo is used with the text. Appendix E describes all primitives in Apple Logo, Apple Logo II, and IBM Logo. Experience has shown that students using other versions of Logo (MIT Logo from Terrapin and Krell, or Logo for the Macintosh from Micro-Soft) will face some adjustment of their programs, but without any major difficulty.

We are primarily interested in concepts of computation, and not in memory limitations or execution speed. Someday, Logo compilers will exist and the speed limitation will go away, and next year's hardware will solve today's memory limitations. We also are not overly concerned about the efficiency of our programs, although we do include a few examples illustrating more efficient alternatives. Usually, efficient programs hide the problem we are solving and, as a general rule, efficiency should not be considered until a working program exists. Frequently, inefficiencies and their corrections will jump out at you once a program has been written.

Some Notes on Presentation

1. The **MAKE** statement is not introduced until Chapter 6. We hesitated even to introduce it this early, but some programs would be unnecessarily complicated if we had not. Besides the fact that **MAKE** is widely misunderstood, getting students to fully understand and appreciate recursion is far easier if they do not have **MAKE** at their disposal. This is especially true of students who have been exposed to BASIC. If you allow BASIC programmers the use of **MAKE**, their programs will look just like BASIC programs every time. Furthermore,

simple and elegant models of computation are not possible once the assignment operation is introduced.

2. We use **SHOW** to display values because one never knows what kind of an object (a word or a list) is returned by an expression that is **PRINT**ed. It seems only to add to the confusion about what a particular expression has computed.

3. Confusion always reigns when **OUTPUT** is introduced. Many students have difficulty learning the difference between procedures that **SHOW** a result and functions that **OUTPUT** a result. Students need to be encouraged to define functions that **OUTPUT** a value rather than procedures that **SHOW** a result. If a value is displayed rather than returned, the program cannot be called by another program that would like to use the value in a further computation. The contrast is an effective way to make the concept of function a real one for the student.

4. The student needs to become comfortable with the Logo environment early in the course. Chapter 2 describes the entire Logo environment as a collection of modules, and is presented in one gigantic dump to separate it from the central ideas presented in the rest of the text. In the university-level course it is covered quickly to provide an overview of the environment and is used as a reference for details when needed in a program. It can also be covered in a leisurely manner in a slower-paced course, allowing the student to experiment with and obtain skills in using the editor, writing programs that obtain input from the keyboard, displaying text, and managing the workspace and files. Chapter 2 is also the chapter where the differences in the various versions of Logo are the greatest.

5. In describing the inputs to a procedure or function we use the following notation:

number indicates that the input value must be a number.
integer indicates that the input value must be an integer.
word indicates that the input value must be a word. This includes numbers.
list indicates that the input value must be a list.
object indicates that the input value must be a list or a word.
name indicates that the input value is a word that names something other than a number—for example, a procedure or a function.
filename indicates that the input value is a word that names a file.

6. The IF statement is not introduced until Chapter 4. The effect is to delay "interesting" programs until that time. This is a conscious decision because there is already a lot of information that needs to be covered and the concept of conditional control deserves full attention without distracting side issues. Also, the student will more fully appreciate its significance. The instructor may want to introduce it informally at an earlier time if pressed to do so by the students. The assignment statement (MAKE) should not be introduced early! In fact, one might want to cover parts of Chapters 7, 8, and 9 before doing Chapter 6.

7. Chapters 7 and 8 are independent of each other. If there is not enough time in the course to cover both chapters, choose which to eliminate based upon the emphasis of the course. Chapter 7 is more appropriate in a class that emphasizes the development of problem-solving skills; Chapter 8 is more appropriate for computer science majors.

8. Chapter 9 consists of projects of varying size and can be covered much earlier. Here is a guide to background needed prior to doing the projects:

   a. Sections 9.2, 9.3, and 9.4 can be covered after Chapter 4.

   b. Section 9.5 requires information from Chapters 6 and 8.

   c. Section 9.6 can be covered after Chapter 6.

   d. Sections 9.7 and 9.8 can be covered after Chapter 4 provided that the student has had some calculus (differentiation).

9. The material in Appendix A serves as an introduction to the theory of computation. It covers computability and proving the correctness of programs. The mathematical concepts introduced and used in this section are informal proofs, including proofs by induction.

10. The word *operation* is commonly associated with the arithmetic operations +, -, *, and /. We use the term *operation* synonomously with the word *function*.

Acknowledgments

We would like to thank the many people who have helped in the development of this book.

Dr. Barbara Pence and Dr. Lynne Gray provided lots of encouragement as well as a lot of constructive criticism of the text. As strong proponents of critical thinking and problem solving, they saw value in the text for mathematics education and insisted on using the text as part of the core for three one-year inservice programs for teachers of mathematics given at San Jose State University from 1984 through 1987. This provided us with the opportunity to use the material in another setting that proved to be extremely satisfying. They also introduced me to a large number of warm, dedicated, and enthusiastic K-14 teachers who were in the process of expanding their own critical thinking and problem-solving skills in mathematics.

Dr. Craig Smorynski wrote the appendix on theory and abstraction. This material explores the theoretical side of computer science using mathematics appropriate for high school students anticipating further study in mathematics or computer science. We regard this material very highly because it links computer science to mathematics, and we expect this bond to become stronger. Dr. Smorynski also aided greatly in making our use of the English language more effective.

Dr. John Mitchem carefully read the text and taught the introductory course in computer science at San Jose State using a preliminary draft. Many of his suggestions on pedagogy have been incorporated.

We are especially indebted to the teachers who have participated in the inservice programs and who keep coming back for more. They provided most of the motivation to finish the book. The students in our classes, in addition to being another testing ground, were a source of ideas as well. They had to cope with the frustrations of trying to make sense out of incomplete, often buggy, preliminary versions of the text. Their input was most valuable. Their questions led to new insights and their expressions told us what worked and what did not—we were all learning together. Specifically, we would like to mention Ronnie Genise who donated uncountable hours in writing, testing, and debugging Logo programs used in our courses as well as giving us his reactions to preliminary versions of the text. He always came up with the answers to

our technical problems. We also thank Donna Price for volunteering to put together the appendix of three versions of Logo primitives.

We thank the faculties and administrators of the Department of Mathematics and Computer Science at San Jose State and Steinbeck Middle School for giving us the opportunity and encouragement to test our ideas in the classroom.

Most of all, we appreciate the support and tolerance of our families throughout this writing project.

M.E.B.
L.R.G.

Contents

Chapter 3 Extending Logo

This chapter describes the graphics capabilities of Logo and introduces recursion. Procedural and functional programming are discussed. **OUTPUT** is contrasted with **SHOW**. The simplification and substitution model is expanded to include user-defined operations and the **REPEAT** procedure.

Chapter 4 The Ultimate in Computational Power

This chapter introduces the conditional form of control and turns our attention to writing recursive programs. The simplification and substitution model is used extensively to aid in the understanding of recursion. The last three sections describe a significant project involving list processing and graphics.

Chapter 5 Level Diagrams: A Model for Understanding Recursion

The level-diagram model of computation is introduced and used to describe a streamlined execution of Logo programs. It also adds a new medium in which to view recursion. Several new operations are defined to illustrate recursion. Many exercises are provided to give practice in writing recursive functions.

Chapter 6 Variables

Global and local variables are introduced with a full explanation of the Logo assignment statement. Guidelines for using local and global variables are given along with appropriate warnings. A programming application that maintains a telephone directory is described illustrating the appropriate use of global variables. Writing programs dealing with the mathematical concept of set are presented illustrating the use of local variables. Our models of computation can not deal with the assignment statement and the state diagram model is introduced to cope with the loss.

Chapter 7 Programming with Graphical Objects

In this chapter we create programming environments that deal with graphical objects such as multiple turtles and polygons. Property lists are presented as an alternative to global variables. The Towers of Hanoi problem is described in functional terms (returning a list of moves giving a solution) and in terms of programming with graphical objects.

Chapter 8 Building Your Own Computational Environment

This chapter describes the CATCH and THROW control mechanism of Logo, how it is used in error handling, and how it can be used in the creation of other computational environments. Algorithms for a printer, a scanner, a parser, and a simplifier are given for a rational arithmetic calculator.