ADVANCED PROGRAMMING LANGUAGE DESIGN

RAPHAEL A. FINKEL

ADVANCED PROGRAMMING LANGUAGE DESIGN

Raphael A. Finkel

UNIVERSITY OF KENTUCKY

Addison-Wesley Publishing Company

Menlo Park, California • Reading, Massachusetts
New York • Don Mills, Ontario • Harlow, U.K. • Amsterdam
Bonn • Paris • Milan • Madrid • Sydney • Singapore • Tokyo
Seoul • Taipei • Mexico City • San Juan, Puerto Rico

Acquisitions Editor: J. Carter Shanklin
Editorial Assistant: Christine Kulke
Senior Production Editor: Teri Holden
Copy Editor: Nick Murray
Manufacturing Coordinator: Janet Weaver
Printer: The Maple-Vail Book Manufacturing Group
Composition and Film Coordinator: Vivian McDougal
Proofreader: Holly McLean-Aldis
Text Designer: Peter Vacek, Eigentype
Film Preparation: Lazer Touch, Inc.
Cover Designer: Yvo Riezebos

Library of Congress Cataloging-in-Publication Data

```
    Finkel, Raphael A.
        Advanced programming languages / Raphael A. Finkel.
        p.   cm.
        Includes index.
        ISBN 0-8053-1191-2
        1.  Programming languages (Electronic computers)   I. Title.
 QA76.7.F56  1995                                                 95-36693
 005.13--dc20                                                          CIP
```

1 2 3 4 5 6 7 8 9—MA—99 98 97 96 95

Addison-Wesley Publishing Company
2725 Sand Hill Road
Menlo Park, CA 94025

Preface

This book stems in part from courses taught at the University of Kentucky and at the University of Wisconsin–Madison on programming language design. There are many good books that deal with the subject at an undergraduate level, but there are few that are suitable for a one-semester graduate-level course. This book is my attempt to fill that gap.

The goal of this course, and hence of this book, is to expose first-year graduate students to a wide range of programming language paradigms and issues, so that they can understand the literature on programming languages and even conduct research in this field. It should improve the students' appreciation of the art of designing programming languages and, to a limited degree, their skill in programming.

This book does not focus on any one language, or even on a few languages; it mentions, at least in passing, over seventy languages, including well-known ones (Algol, Pascal, C, C++, LISP, Ada, FORTRAN), important but less known ones (ML, SR, Modula-3, SNOBOL), significant research languages (CLU, Alphard, Linda), and little-known languages with important concepts (Io, Gödel). Several languages are discussed in some depth, primarily to reinforce particular programming paradigms. ML and LISP demonstrate functional programming, Smalltalk and C++ demonstrate object-oriented programming, and Prolog demonstrates logic programming.

Students are expected to have taken an undergraduate course in programming languages before using this book. The first chapter includes a review of much of the material on imperative programming languages that would be covered in such a course. This review makes the book self-contained, and also makes it accessible to advanced undergraduate students.

Most textbooks on programming languages cover the well-trodden areas of the field. In contrast, this book tries to go beyond the standard territory, making brief forays into regions that are under current research or that have been proposed and even rejected in the past. There are many fascinating constructs that appear in very few, if any, production programming languages. Some (like power loops) should most likely not be included in a programming language. Others (like Io continuations) are so strange that it is not clear how to program with them. Some (APL arrays) show alternative ways to structure languages. These unusual ideas are important even though they do not pass the test of current usage, because they elucidate important aspects of programming language design, and they allow students to evaluate novel concepts.

Certain themes flow through the entire book. One is the interplay between what can be done at compile time and what must be deferred to runtime. Actions performed at compile time make for more efficient and less error-prone execution. Decisions deferred until runtime lead to greater flexibility. Another theme is how patterns and pattern matching play a large role in many ways in programming languages. Pattern matching is immediately important for string manipulation, but it is also critical in steering logic programming, helpful for extracting data from structures in ML, and for associating caller and callee in CSP. A third theme is the quest for uniformity. It is very much like the mathematical urge to generalize. It can be seen in polymorphism, which generalizes the concept of type, and in overloading, which begins by unifying operators and functions and then unifies disparate functions under one roof. It can be seen in the homoiconic forms of LISP, in which program and data are both presented in the same uniform way.

Two organizing principles suggest themselves for a book on programming languages. The first is to deal separately with such issues as syntax, types, encapsulation, parallelism, object-oriented programming, pattern matching, dataflow, and so forth. Each section would introduce examples from all relevant languages. The other potential organizing principle is to present individual languages, more or less in full, and then to derive principles from them.

This book steers a middle course. I have divided it into chapters, each of which deals primarily with one of the subjects mentioned above. Most chapters include an extended example from a particular language to set the stage. This section may introduce language-specific features not directly relevant to the subject of the chapter. The chapter then introduces related features from other languages.

Because this book covers both central and unusual topics, the instructor of a course using the book should pick and choose whatever topics are of personal interest. In general, the latter parts of chapters delve into stranger and more novel variants of material presented earlier. The book is intended for a one-semester course, but it is about 30 percent too long to cover fully in one semester. It is not necessary to cover every chapter, nor to cover every section of a chapter. Only Chapter 1 and the first seven sections of Chapter 3 are critical for understanding the other chapters. Some instructors will want to cover Chapter 4 before the discussion of ML in Chapter 3. Many instructors will decide to omit dataflow (Chapter 6). Others will wish to omit denotational semantics (in Chapter 10).

I have not described complete languages, and I may have failed to mention your favorite language. I have selected representative programming languages that display particular programming paradigms or language features clearly. These languages are not all generally available or even widely known. The appendix lists all the languages I have mentioned and gives you some pointers to the literature and to implementations and documentation available on the Internet through anonymous ftp (file-transfer protocol).

The exercises at the end of each chapter serve two purposes. First, they allow students to test their understanding of the subjects presented in the text by working exercises directly related to the material. More importantly, they push students beyond the confines of the material presented to consider new situations and to evaluate new proposals. Subjects that are only hinted at in the text are developed more thoroughly in this latter type of exercise.

In order to create an appearance of uniformity, I have chosen to modify the syntax of presented languages (in cases where the syntax is not the crucial issue), so that language-specific syntax does not obscure the other points that I am trying to make. For examples that do not depend on any particular language, I have invented what I hope will be clear notation. It is derived largely from Ada and some of its predecessors. This notation allows me to standardize the syntactic form of language, so that the syntax does not obscure the subject at hand. It is largely irrelevant whether a particular language uses **`begin`** and **`end`** or `{` and `}`. On the other hand, in those cases where I delve deeply into a language in current use (like ML, LISP, Prolog, Smalltalk, and C++), I have preserved the actual language. Where reserved words appear, I have placed them in **`bold monospace`**. Other program excerpts are in `monospace` font. I have also numbered examples so that instructors can refer to parts of them by line number. Each technical term that is introduced in the text is printed in **boldface** the first time it appears. All boldface entries are collected and defined in the glossary. I have tried to use a consistent nomenclature throughout the book.

In order to relieve the formality common in textbooks, I have chosen to write this book as a conversation between me, in the first singular person, and you, in the second person. When I say *we*, I mean you and me together. I hope you don't mind.

Several supplemental items are available to assist the instructor in using this text. Answers to the exercises are available from the publisher (ISBN: 0-201-49835-9) in a disk-based format. The figures from the text (in Adobe Acrobat format), an Adobe Acrobat reader, and the entire text of this book are available from the following site:

```
ftp://aw.com/cseng/authors/finkel
```

Please check the readme file for updates and changes. The complete text of this book is intended for on-screen viewing free of charge; use of this material in any other format is subject to a fee.

There are other good books on programming language design. I can particularly recommend the text by Pratt [Pratt 96] for elementary material and the text by Louden [Louden 93] for advanced material. Other good books include those by Sebesta [Sebesta 93] and Sethi [Sethi 89].

I owe a debt of gratitude to the many people who helped me write this book. Much of the underlying text is modified from course notes written by Charles N. Fischer of the University of Wisconsin–Madison. Students in my classes have submitted papers which I have used in preparing examples and text; these include the following:

Subject	Student	Year
C++	Feng Luo	1992
	Mike Rogers	1992
Dataflow	Chinya Ravishankar	1981
Gödel	James Gary	1992
Lynx	Michael Scott	1985
Mathematics languages	Mary Sue Powers	1994
Miranda	Manish Gupta	1992
Post	Chinya Ravishankar	1981
	Rao Surapaneni	1992
CLP	William Ralenkotter	1994
Russell	Rick Simkin	1981
	K. Lakshman	1992
	Manish Gupta	1992
Smalltalk/C++	Jonathan Edwards	1992

Jonathan Edwards read an early draft of the text carefully and made many helpful suggestions. Michael Scott assisted me in improving Chapter 7 on concurrency. Arcot Rajasekar provided important feedback on Chapter 8 on logic programming. My editor, J. Carter Shanklin, and the reviewers he selected, made a world of difference in the presentation and coverage of the book. These reviewers were David Stotts (University of North Carolina at Chapel Hill), Spiro Michaylov (Ohio State University), Michael G. Murphy (Southern College of Technology), Barbara Ann Greim (University of North Carolina at Wilmington), Charles Elkan (University of California, San Diego), Henry Ruston (Polytechnic University), and L. David Umbaugh (University of Texas at Arlington). The University of Kentucky provided sabbatical funding to allow me to pursue this project, and Metropolitan College in Kuala Lumpur, Malaysia, provided computer facilities that allowed me to work on it. This book was prepared on the Linux version of the Unix operating system. Linux is the result of work by Linus Torvalds and countless others, primarily at the Free Software Foundation, who have provided an immense suite of programs I have used, including text editors, document formatters and previewers, spelling checkers, and revision control packages. I would have been lost without them. Finally, I would like to thank my wife, Beth L. Goldstein, for her support and patience, and my daughter, Penina, and son, Asher, for being wonderful.

Raphael A. Finkel
University of Kentucky

Contents

Chapter 1

Introduction

The purpose of this book is to study the principles and innovations found in modern programming languages. We will consider a wide variety of languages. The goal is not to become proficient in any of these languages, but to learn what contributions each has made to the "state of the art" in language design.

I will discuss various programming paradigms in this book. Some languages (such as Ada, Pascal, Modula-2) are **imperative**; they use variables, assignments, and iteration. For imperative languages, I will dwell on such issues as flow of control (Chapter 2) and data types (Chapter 3). Other languages (for example, LISP and FP) are **functional**; they have no variables, assignments, or iteration, but model program execution as expression evaluation. I discuss functional languages in Chapter 4. Other languages (for example, Smalltalk and C++), represent the **object-oriented** paradigm, in which data types are generalized to collections of data and associated routines (Chapter 5). **Dataflow languages** (Val, Sisal, and Post, Chapter 6) attempt to gain speed by simultaneous execution of independent computations; they require special computer architectures. A more common way to gain speed is by **concurrent** programming (typified by languages such as SR and Lynx, discussed in Chapter 7). Another major paradigm constitutes the **declarative** languages such as Prolog and Gödel (Chapter 8); they view programming as stating what is wanted and not necessarily how to compute it. **Aggregate languages** (Chapter 9) form a a final loosely knit paradigm that includes languages with special-purpose data formats, such as strings (SNOBOL and Icon), arrays (APL), databases (dBASE and SQL), and mathematical formulas (Mathematica and Maple).

In addition to studying actual programming language constructs, I will present formal semantic models in Chapter 10. These models allow a precise specification of what a program means, and provide the basis for reasoning about the correctness of a program.

1 ◆ PROGRAMMING LANGUAGES AS SOFTWARE TOOLS

Programming languages fit into a larger subject that might be termed **software tools**. This subject includes such fields as interactive editors (text, picture, spreadsheet, bitmap, and so forth), data transformers (compilers, assemblers, stream editors, macro processors, text formatters), operating systems, database management systems, and tools for program creation, testing, and maintenance (script files, source-code management tools, debuggers).

In general, software tools can be studied as interfaces between clients, which are usually humans or their programs, and lower-level facilities, such as files or operating systems.

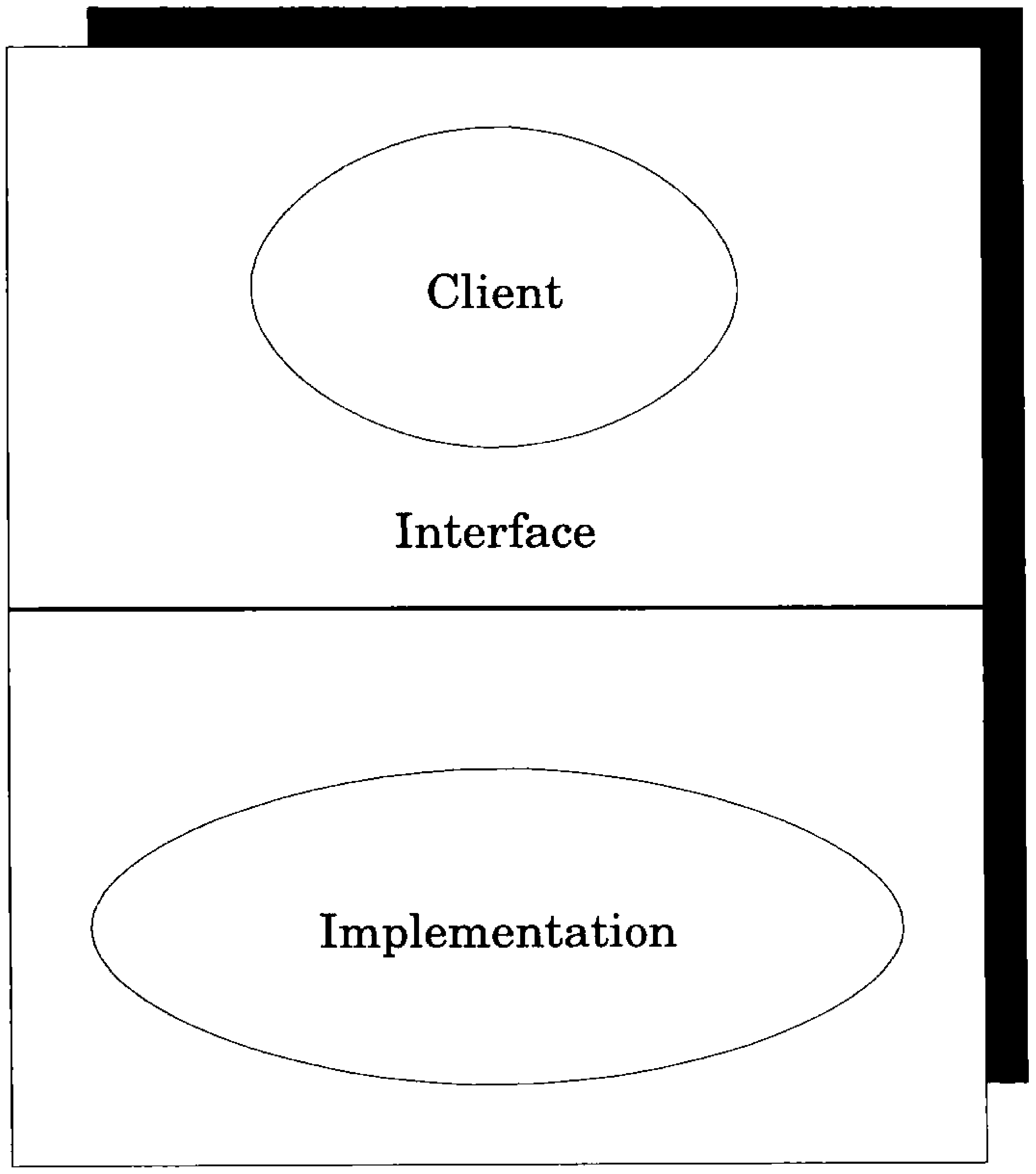

Figure 1.1 Software tools

Three questions arising from Figure 1.1 are worth discussing for any software tool:

1. What is the nature of the interface?
2. How can the interface be implemented by using the lower-level facilities?
3. How useful is the interface for humans or their agents?

When we deal with programming languages as software tools, these questions are transformed:

1. What is the structure (syntax) and meaning (semantics) of the programming language constructs? Usually, I will use informal methods to show what the constructs are and what they do. However, Chapter 10 presents formal methods for describing the semantics of programming languages.
2. How does the compiler writer deal with these constructs in order to translate them into assembler or machine language? The subject of compiler construction is large and fascinating, but is beyond the scope of this book. I will occasionally touch on this topic to assure you that the constructs can, in fact, be translated.
3. Is the programming language good for the programmer? More specifically, is it easy to use, expressive, readable? Does it protect the programmer from programming errors? Is it elegant? I spend a significant amount of effort trying to evaluate programming languages and their constructs in this way. This subject is both fascinating and difficult to be objective about. Many languages have their own fan clubs, and discussions often revolve about an ill-defined sense of elegance.

Programming languages have a profound effect on the ways programmers formulate solutions to problems. You will see that different paradigms impose very different programming styles, but even more important, they change the way the programmer looks at algorithms. I hope that this book will expand your horizons in much the same way that your first exposure to recursion opened up a new way of thinking. People have invented an amazing collection of elegant and expressive programming structures.

2 ◆ EVALUATING PROGRAMMING LANGUAGES

This book introduces you to some unusual languages and some unusual language features. As you read about them, you might wonder how to evaluate the quality of a feature or an entire language. Reasonable people disagree on what makes for a great language, which is why so many novel ideas abound in the arena of programming language design. At the risk of oversimplification, I would like to present a short list of