PROGRAMMING LANGUAGE PRAGMATICS

Michael L. Scott

Programming Language Pragmatics

Michael L. Scott
University of Rochester

Morgan Kaufmann Publishers
San Francisco, California

Senior Editor: Denise E. M. Penrose
Director of Production & Manufacturing: Yonie Overton
Production Editor: Edward Wade
Editorial Coordinator: Meghan Keeffe
Cover Design: Ross Carron Design
Cover Photograph: © Ann Cecil/Photo 20–20/PNI
Text Design: Rebecca Evans and Associates
Composition: Ed Sznyter, Babel Press
Technical Illustration: Cherie Plumlee
Copyeditor: Donna King, Progressive Publishing Alternatives
Proofreader: Christine Sabooni
Indexer: Ty Koontz
Printer: Courier Corporation

Morgan Kaufmann Publishers
Editorial and Sales Office
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205
USA

Telephone	415/392-2665
Facsimile	415/982-2665
Email	*mkp@mkp.com*
WWW	*http://www.mkp.com*
Order toll free	800/745-7323

Advice, Praise, and Errors: Any correspondence related to this publication or intended for the author should be addressed to the Editorial and Sales Office of Morgan Kaufmann Publishers, Dept. PLP APE or sent electronically to *plp@mkp.com*. Information regarding error sightings is encouraged; electronic mail can be sent to *plpbugs@mkp.com*. Please check the errata page at *www.mkp.com/plp* to see if the bug has already been reported.

Printed in the United States of America

04 03 02 01 00 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data

Scott, Michael Lee
Programming language pragmatics / Michael L. Scott.
p. cm.
Includes bibliographical references and index.
ISBN 1-55860-442-1 (hardback) – ISBN 1-55860-578-9 (paperback)
1. Programming languages (Electronic computers) I. Title.
QA76.7.S38 2000
005.13–dc21 99-047125
CIP

Programming Language Pragmatics

About the Author

Michael L. Scott is a professor of computer science at the University of Rochester, where he chaired the Computer Science Department from 1996 to 1999. He received his Ph.D. in computer sciences in 1985 from the University of Wisconsin-Madison, where he was a member of the Crystal and Charlotte research groups. His research interests lie in parallel and distributed computing, including operating systems, languages, architecture, and tools. He is the designer of the Lynx distributed programming language and a co-designer of the Charlotte and Psyche parallel operating systems, the Bridge parallel file system, and the Cashmere distributed shared memory system. His MCS mutual exclusion lock, co-designed with John Mellor-Crummey of Rice University, appears in a wide variety of commercial and academic systems.

Dr. Scott is a member of the Institute of Electrical and Electronics Engineers, the Association for Computing Machinery, the Union of Concerned Scientists, and Computer Professionals for Social Responsibility. He has served on a wide variety of program committees and grant review panels, and has been a principal or co-investigator on grants from the NSF, ONR, DARPA, NASA, the Department of Defense, the Ford Foundation, and Digital Equipment Corporation (now Compaq). He has contributed to the GRE advanced exam in computer science, and is the author of more than fifty refereed publications. He received a Bell Labs Doctoral Scholarship in 1983 and an IBM Faculty Development Award in 1986.

To my three Irish roses

Preface

A course in computer programming provides the typical student's first exposure to the field of computer science. Most of the students in such a course will have had previous exposure to computers, in the form of games and other personal computer applications, but it is not until they write their own programs that they begin to appreciate how these applications work. After gaining a certain level of facility as programmers (presumably with the help of a good course in data structures and algorithms), the natural next step is to wonder how programming languages work. This book provides an explanation.

In the conventional "systems" curriculum, the material beyond data structures (and possibly computer organization) is compartmentalized by subarea, with courses in programming languages, compilers, computer architecture, operating systems, database management systems, and possibly software engineering, graphics, or user interface systems. One problem with this approach is that many of the most interesting things in computer science occur at the *boundaries* between these subareas. The RISC revolution, for example, has forged an intimate alliance between computer architecture and compiler construction. The advent of microkernels has blurred the boundary between the operating system kernel and the language run-time library. The spread of Java-based systems has similarly blurred the boundary between the compiler and the run-time library. Aggressive memory systems for supercomputers are redefining the relative roles of the operating system, the compiler, and the hardware. And programming language design has always been heavily influenced by implementation issues. Increasingly both educators and researchers are recognizing the need to focus on these interactions.

Another problem with the compartmentalized curriculum is that it offers more courses than the typical undergraduate can afford to take. A student who wants to gain a solid background in theory, artificial intelligence, numerical methods, or various allied fields cannot afford to take five upper-level courses in systems. Rather than give the student an in-depth look at two or three relatively narrow subareas, I believe it makes sense to provide an integrated look at the most fundamental material *across* subareas.

At its core, *Programming Language Pragmatics* is a book about how programming languages work. It is in some sense a mixture of traditional texts in programming languages and compilers, with just enough assembly-level architecture to accommodate the student who has not yet had a course in computer organization. It is not a language survey text: rather than enumerate the details of many different languages, it focuses on concepts that underlie all of the languages the student is likely to encounter, illustrating those concepts with examples from various languages. It is also not a compiler construction text: rather than explain how to build a compiler (a task few programmers will ever need to tackle in its entirety, though they may use front-end techniques in other tools), it explains how a compiler works, what it does to a source program, and why. Language design and implementation are thus explored together, with an emphasis on the ways in which they interact. When discussing iteration (Section 6.5.1), we can see how semantic issues (what is the scope of an index variable? what happens if the body of a loop tries to modify the index or loop bounds?) have interacted with pragmatic issues (how many branch instructions must we execute in each iteration of the loop? how do we avoid arithmetic overflow when updating the index?) to shape the evolution of loop constructs. When discussing object-oriented programming, we can see how the tension between semantic elegance and implementation speed has shaped the design of languages such as Smalltalk, Eiffel, C++, and Java.

In the typical undergraduate curriculum, this book is intended for the programming languages course. It has a bit less survey-style detail than certain other texts, but it covers the same breadth of languages and concepts, and includes much more information on implementation issues. Students with a strong interest in language design should be encouraged to take additional courses in such areas as formal semantics, type theory, or object-oriented design. Similarly, students with a strong interest in language implementation should take a subsequent course in compiler construction. With this book as background, the compiler course will be able to devote much more time than is usually possible to code generation and optimization, where most of the interesting work these days is taking place.

At the University of Rochester, the material in this book has been used for over a decade to teach a course entitled "Software Systems." The course draws a mixture of mid- to upper-level undergraduates and first-year graduate students. The book should also be of value to professional programmers and other practitioners who simply wish to gain a better understanding of what's going on "under the hood" in their favorite programming language. By integrating the discussion of syntactic, semantic, and pragmatic (implementation) issues, the book attempts to provide a more complete and balanced treatment of language design than is possible in most texts. The hope is that students will come to understand why language features were designed the way they were, and that as programmers they will be able to choose an appropriate language for a given application, learn new languages easily, and make clear and efficient use of any given language.

In most chapters the concluding section returns to the theme of design and

implementation, highlighting interactions between the two that appeared in preceding sections. In addition, Appendix B contains a summary list of interactions, with references to the sections in which they are discussed. These interactions are grouped into several categories, including language features that most designers now believe were mistakes, at least in part because of implementation difficulties; potentially useful features omitted from some languages because of concern that they might be too difficult or slow to implement; and language features introduced at least in part to facilitate efficient or elegant implementations.

Some chapters (2, 4, 5, 9, and 13) have a heavier emphasis than others on implementation issues. These can be reordered to a certain extent with respect to the more design-oriented chapters, but it is important that Chapter 5 or its equivalent be covered before Chapters 6, 7, or 8. Many readers will already be familiar with some or all of the material in Chapter 5, most likely from a course on computer organization. In this case the chapter can easily be skipped. Be warned, however, that later chapters assume an understanding of the assembly-level architecture of modern (i.e., RISC) microprocessors. Some readers may also be familiar with some of the material in Chapter 2, perhaps from a course on automata theory. Much of this chapter can then be read quickly, pausing perhaps to dwell on such practical issues as recovery from syntax errors.

For self-study, or for a full-year course, I recommend working through the book from start to finish. In the one-semester course at Rochester, we also cover most of the book, but at a somewhat shallower level. The lectures focus on the instructor's choice of material from the following chapters and sections: 1, 2.1 through 2.2.3, 3, 4, 6, 7, 8, 9.1 through 9.3, and 10 through 12. Students are asked to read all of this material except for those sections marked with an asterisk. They are also asked to skim Chapter 5; most have already taken a course in computer organization.

For a more traditional programming languages course, one would leave out Section 2.2 and Chapters 4, 5, and 9, and deemphasize the implementation-oriented material in the remaining chapters, devoting the extra time to more careful examination of semantic issues and to alternative programming paradigms (e.g., the foundational material in Chapter 11). For a school on the quarter system, one appealing option is to offer an introductory one-quarter course and two optional follow-on courses. The introductory quarter might cover these chapters and sections: 1, 2.1 through 2.2.3, 3, 6, 7, and 8.1 through 8.4. A language-oriented follow-on quarter might cover Sections 8.5 through 8.6, Chapters 10 through 12, and possibly supplementary material on formal semantics, type systems, or other related topics. A compiler-oriented follow-on quarter might cover Sections 2.2.4 through 2.3, and Chapters 4, 5 (if necessary), 9, and 13, and possibly supplementary material on automatic code generation, aggressive code improvement, programming tools, and so on. One possible objection to this organization is that it leaves object orientation and functional and logic programming out of the introductory quarter. An alternative would be to start with a broader and more exclusively design-oriented view, moving Sections 1.4 through 1.6 and 2.2.1 through 2.2.3 into the compiler-oriented quarter,

deemphasizing the implementation-oriented material in Chapters 6 through 8, and adding Sections 10.1 through 10.4, 10.6, and the nonfoundational material in Chapter 11.

I assume that the typical reader already has significant experience with at least one high-level imperative programming language. Exactly which language it is shouldn't matter. Examples are drawn from a wide variety of languages, but always with enough comments and other discussion that readers not familiar with the language should be able to understand them easily. Algorithms, when needed, are presented in an informal pseudocode that should be self-explanatory. `Real programming language code is set in this font (Computer Modern).` Pseudocode is set in this font (UniversLight).

Each of the chapters ends with review questions and a set of more challenging exercises. Particularly valuable are those exercises that direct students toward languages or techniques that they are unlikely to have encountered elsewhere, or to encounter elsewhere soon. I recommend programming assignments in C++ or Java; Scheme, ML, or Haskell; and Prolog. An assignment in exception handling is also a good idea; it may be written in Ada, C++, Java, ML, or Modula-3. If concurrency is covered, an assignment should be given in SR, Java, Ada, or Modula-3, depending on local interest. Sources for language implementations are noted in Appendix A.

In addition to these smaller projects (or in place of them if desired), instructors may wish to have students work on a language implementation. Since building even the smallest compiler from scratch is a full-semester job, students at Rochester have been given the source for a working compiler and asked to make modifications. For many, this is their first experience reading, understanding, and modifying a large existing program—a valuable exercise in and of itself. The Rochester PL/0 compiler translates a simple language due to Wirth [Wir76, pp. 307–347] into MIPS I assembly language, widely considered the "friendliest" of the commercial RISC instruction sets. An excellent MIPS interpreter ("SPIM") is available from the Computer Science Department at the University of Wisconsin (*www.cs.wisc.edu/~larus/spim.html*). Source for the compiler itself is available from Rochester (*ftp://ftp.cs.rochester.edu/pub/packages/plzero/*). It is written in C++, with carefully separated phases and extensive documentation.

About the Cover

The triangular icon on the cover is meant to symbolize the syntactic, semantic, and pragmatic facets of programming language design, each of which finds meaning in the context of the other two. This book aims to pay equal attention to all three facets; it takes its name from the one that tends to receive the least attention in other language texts.

Acknowledgments

Many people have contributed to this book. My thanks to the many reviewers who contributed comments and suggestions, including Greg Andrews, John Boyland, Preston Briggs, Jim Larus, Steve Muchnick, David Notkin, Ron Olsson, Constantine Polychronopoulos, and others who remain anonymous. I have done my best to address their concerns; the errors that remain are entirely my own. My thanks also to the students of CSC 2/454 (formerly 2/452) who served as guinea pigs for early versions of the book, and whose feedback served to improve it greatly. Thanks in particular to Donna Byron and Brandon Sanders for their extensive comments, to Scott Ventura for the web browser example of Section 12.2.3, and to Angkul Kongmunvattana for helping me understand the difference between useful and distracting cross-references.

I am of course indebted to the hard-working staff of Morgan Kaufmann Publishers, and to Denise Penrose, Mike Morgan, Edward Wade, and Meghan Keeffe in particular. The University of Rochester and its Computer Science Department provided a stimulating and supportive academic home while I was working on the book. Sarada George and Dianne Reiman Cass spent days chasing bibliographic references. Finally, my thanks to my family for 5 long years of patience.

The original manuscript for this book was composed on an Apple Powerbook Duo computer using Marc Parmet's port of the `emacs` text editor and Andrew Trevorrow's OzTEX version of TEX. Diagrams were drawn with Adobe Illustrator.

Computer Science and Engineering Textbooks from Morgan Kaufmann Publishers

Artificial Intelligence

Genetic Programming: An Introduction
Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone

Essentials of Artificial Intelligence
Matt Ginsberg

Readings in Agents
Edited by Michael N. Huhns and Munindar P. Singh

Case-Based Reasoning
Janet Kolodner

Genetic Programming III: Darwinian Invention and Problem Solving
John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Keane

Elements of Machine Learning
Pat Langley

Readings in Intelligent User Interfaces
Mark T. Maybury and Wolfgang Wahlster

Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference
Judea Pearl

Artificial Intelligence: An Introduction
Nils J. Nilsson

Introduction to Knowledge Systems
Mark Stefik

Computer Architecture

The Student's Guide to VHDL
Peter J. Ashenden

Parallel Computer Architecture: A Hardware/Software Approach
David E. Culler and Jaswinder Pal Singh with Anoop Gupta

Computer Architecture: A Quantitative Approach 2ed.
John L. Hennessy and David A. Patterson

Readings in Computer Architecture
Edited by Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi

Introduction to Parallel Algorithms and Architectures: Arrays, Trees & Hypercubes
F. Thomson Leighton

Advanced Compiler Design & Implementation
Steven S. Muchnick

Parallel Programming with MPI
Peter S. Pacheco

Computer Organization & Design: The Hardware/Software Interface 2ed.
David A. Patterson and John L. Hennessy

Database

Distributed Algorithms
Nancy A. Lynch

Readings in Database Systems 3ed.
Edited by Michael Stonebraker and Joseph M. Hellerstein

Principles of Multimedia Database Systems
V. S. Subrahmanian

Principles of Database Query Processing for Advanced Applications
Clement T. Yu and Weiyi Meng

Advanced Database Systems
Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian, and Roberto Zicari

Human-Computer Interaction and Computer Graphics

Readings in Human-Computer Interaction: Toward the Year 2000 2ed.
Edited by Ronald M. Baecker, Jonathan Grudin, William Buxton, and Saul Greenberg

Readings in Information Visualization: Using Vision to Think
Edited by Stuart K. Card, Jock Mackinlay, and Ben Shneiderman

Multimedia Information & Systems

Readings in Information Retrieval
Edited by Karen Sparck Jones and Peter Willett

Networking

Understanding Networked Applications: A First Course
David G. Messerschmitt

Computer Networks: A Systems Approach 2ed.
Larry L. Peterson and Bruce S. Davie

Optical Networks: A Practical Perspective
Rajiv Ramaswami and Kumar N. Sivarajan

High-Performance Communication Networks 2ed.
Jean Walrand and Pravin Varaiya

Theory

Fundamentals of the Theory of Computation: Principles and Practice
Raymond Greenlaw and H. James Hoover

Forthcoming

Database: Principles, Programming, Performance 2ed.
Patrick E. O'Neil

Introduction to Data Compression 2ed.
Khalid Sayood

Interactive Programming in Java
Lynn Andrea Stein

Computers as Components: Principles of Embedded Computing System Design
Wayne Wolf

Contents

About the Author ii

Preface xvii

Chapter 1 **Introduction** 1

1.1 The Art of Language Design 3
1.2 The Programming Language Spectrum 5
1.3 Why Study Programming Languages? 7
1.4 Compilation and Interpretation 9
1.5 Programming Environments 14
1.6 An Overview of Compilation 15
1.6.1 Lexical and Syntax Analysis 16
1.6.2 Semantic Analysis and Intermediate Code Generation 18
1.6.3 Target Code Generation 22
1.6.4 Code Improvement 24
1.7 Summary and Concluding Remarks 24
1.8 Review Questions 25
1.9 Exercises 26
1.10 Bibliographic Notes 28

Chapter 2 **Programming Language Syntax** 31

2.1 Specifying Syntax: Regular Expressions and Context-Free Grammars 32
2.1.1 Tokens and Regular Expressions 33
2.1.2 Context-Free Grammars 34
2.1.3 Derivations and Parse Trees 36
2.2 Recognizing Syntax: Scanners and Parsers 39
2.2.1 Scanning 40
2.2.2 Top-Down and Bottom-Up Parsing 48
2.2.3 Recursive Descent 51

2.2.4* Syntax Errors 57
2.2.5 Table-Driven Top-Down Parsing 62
2.2.6 Bottom-Up Parsing 75
2.3* **Theoretical Foundations 87**
2.3.1 Finite Automata 88
2.3.2 Push-Down Automata 92
2.3.3 Grammar and Language Classes 93
2.4 **Summary and Concluding Remarks 94**
2.5 **Review Questions 97**
2.6 **Exercises 98**
2.7 **Bibliographic Notes 102**

Chapter 3 **Names, Scopes, and Bindings 105**

3.1 **The Notion of Binding Time 106**
3.2 **Object Lifetime and Storage Management 108**
3.2.1 Stack-Based Allocation 111
3.2.2 Heap-Based Allocation 113
3.2.3 Garbage Collection 114
3.3 **Scope Rules 115**
3.3.1 Static Scope 116
3.3.2 Dynamic Scope 129
3.3.3 Symbol Tables 132
3.3.4 Association Lists and Central Reference Tables 137
3.4 **The Binding of Referencing Environments 139**
3.4.1 Subroutine Closures 141
3.4.2 First- and Second-Class Subroutines 143
3.5 **Overloading and Related Concepts 144**
3.6 **Naming-Related Pitfalls in Language Design 149**
3.6.1 Scope Rules 149
3.6.2* Separate Compilation 151
3.7 **Summary and Concluding Remarks 155**
3.8 **Review Questions 157**
3.9 **Exercises 158**
3.10 **Bibliographic Notes 162**

Chapter 4 **Semantic Analysis 165**

4.1 **The Role of the Semantic Analyzer 166**
4.2 **Attribute Grammars 168**
4.3 **Attribute Flow 170**

4.4 **Action Routines 179**
4.5* **Space Management for Attributes 180**
4.5.1 Bottom-Up Evaluation 181
4.5.2 Top-Down Evaluation 186
4.6 **Annotating a Syntax Tree 191**
4.7 **Summary and Concluding Remarks 197**
4.8 **Review Questions 198**
4.9 **Exercises 199**
4.10 **Bibliographic Notes 202**

Chapter 5 **Assembly-Level Computer Architecture 203**

5.1 **Workstation Macro-Architecture 204**
5.2 **The Memory Hierarchy 207**
5.3 **Data Representation 209**
5.3.1 Integer Arithmetic 211
5.3.2 Floating-Point Arithmetic 212
5.4 **Instruction Set Architecture 214**
5.4.1 Addressing Modes 215
5.4.2 Conditional Branches 217
5.5 **The Evolution of Processor Architecture 218**
5.5.1 Two Example Architectures: The 680x0 and MIPS 220
5.5.2 Pseudoassembler Notation 225
5.6 **Compiling for Modern Processors 227**
5.6.1 Keeping the Pipeline Full 227
5.6.2 Register Allocation 234
5.7 **Summary and Concluding Remarks 242**
5.8 **Review Questions 243**
5.9 **Exercises 244**
5.10 **Bibliographic Notes 247**

Chapter 6 **Control Flow 249**

6.1 **Expression Evaluation 250**
6.1.1 Precedence and Associativity 251
6.1.2 Assignments 254
6.1.3 Ordering Within Expressions 262
6.1.4 Short-Circuit Evaluation 265
6.2 **Structured and Unstructured Flow 267**
6.3 **Sequencing 270**

6.4 **Selection 271**
6.4.1 Short-Circuited Conditions 272
6.4.2 Case/Switch Statements 275
6.5 **Iteration 280**
6.5.1 Enumeration-Controlled Loops 280
6.5.2 Combination Loops 286
6.5.3* Iterators 287
6.5.4 Logically Controlled Loops 294
6.6 **Recursion 297**
6.6.1 Iteration and Recursion 297
6.6.2 Applicative- and Normal-Order Evaluation 301
6.7* **Nondeterminacy 303**
6.8 **Summary and Concluding Remarks 308**
6.9 **Review Questions 310**
6.10 **Exercises 311**
6.11 **Bibliographic Notes 316**

Chapter 7 **Data Types 319**

7.1 **Type Systems 320**
7.1.1 The Definition of Types 322
7.1.2 The Classification of Types 323
7.2 **Type Checking 330**
7.2.1 Type Equivalence 330
7.2.2 Type Conversion and Casts 334
7.2.3 Type Compatibility and Coercion 337
7.2.4 Type Inference 341
7.2.5* The ML Type System 344
7.3 **Records (Structures) and Variants (Unions) 351**
7.3.1 Syntax and Operations 351
7.3.2 Memory Layout and Its Impact 353
7.3.3* `With` Statements 355
7.3.4 Variant Records 358
7.4 **Arrays 365**
7.4.1 Syntax and Operations 365
7.4.2 Dimensions, Bounds, and Allocation 369
7.4.3 Memory Layout 373
7.5 **Strings 379**
7.6 **Sets 381**