Programming Languages

Principles and Paradigms

Allen Tucker
Robert Noonan

Programming Languages

Principles and Paradigms

Allen Tucker
Bowdoin College

Robert Noonan
College of William and Mary

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto

McGraw-Hill Higher Education

A Division of The ***McGraw-Hill*** *Companies*

PROGRAMMING LANGUAGES: PRINCIPLES AND PARADIGMS

Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020.

This book is printed on acid-free paper.

International 1 2 3 4 5 6 7 8 9 0 QPF/QPF 0 9 8 7 6 5 4 3 2 1
Domestic 2 3 4 5 6 7 8 9 0 QPF/QPF 0 9 8 7 6 5 4 3

ISBN 0–07–238111–6
ISBN 0–07–112280–X (ISE)

General manager: *Thomas E. Casson*
Publisher: *Elizabeth A. Jones*
Developmental editor: *Emily J. Lupash*
Executive marketing manager: *John Wannemacher*
Senior project manager: *Jayne Klein*
Production supervisor: *Kara Kudronowicz*
Coordinator of freelance design: *Michelle D. Whitaker*
Freelance cover/interior designer: *Rokusek Design*
Cover images: *Photographed by Allen Tucker*
Supplement producer: *Brenda A. Ernzen*
Media technology senior producer: *Phillip Meek*
Compositor: *Interactive Composition Corporation*
Typeface: *10/12 Times Roman*
Printer: *Quebecor World Fairfield, PA*

Library of Congress Cataloging-in-Publication Data

Tucker, Allen B.
Programming languages : principles and paradigms / Allen B. Tucker, Robert E. Noonan.—1st ed.
p. cm.
Includes index.
ISBN 0–07–238111–6 — ISBN 0–07–112280–X (ISE)
1. Programming languages (Electronic computers). I. Noonan, Robert E. II. Title.

QA76.7 .T83 2002
005.13—dc21 2001044137
CIP

INTERNATIONAL EDITION ISBN 0–07–112280–X

www.mhhe.com

To Anatoly Sachenko and our friends at
Ternopil Academy of National Economy
Ternopil, Ukraine.

Allen Tucker

To Debbie and Paul.

Robert Noonan

Preface

The study of programming languages at the advanced undergraduate or graduate level usually covers two main areas: principles of language design and two or three different programming paradigms. Texts for this study tend to fall into either of two categories: 1) concept-based surveys of a wide range of language design topics and paradigms; and 2) interpreter-based treatments of the design principles presented in a functional language.

APPROACH

This text attempts to unite the best features of these two approaches into a single and coherent framework. Like the interpreter-based texts, we include a rigorous, complete, and hands-on treatment of the principles using a formal grammar, type system, and denotational semantics, including an interpreter that implements the formal model. In contrast with these texts, we use Java as the language of illustration rather than a functional language. Like the concepts-based texts, this text presents and contrasts the major language design topics and programming paradigms. Unlike the concepts-based texts, we hope to cover this material in a more modern and coherent fashion.

Our approach is based on the belief that a formal treatment of syntax and semantics, a consistent use of the mathematical notations learned in discrete mathematics, and a hands-on treatment of the principles of language design are centrally important to the study of programming languages. This approach is advocated, for instance, in the design of the Programming Languages course in the *Liberal Arts Model Curriculum* [Walker 1996], and is also consistent with the recommendations of *Computing Curricula 2001* [CC 2001]. The concepts-based texts seem to have foregone such rigor in recent years in favor of surveying an increasingly wide variety of topics and languages. The topics that should be central to a student's understanding of language principles and paradigms, such as the formal treatment of semantics, are usually presented late in these texts, as one of many unrelated topics, and in a way that encourages instructors to skip them altogether. We think that a study of programming languages principles should integrate these topics in a more compelling way.

With regard to *Computing Curricula 2001* [CC2001], the material in this text covers all the topics (PL1 through PL11) in the Programming Languages section of the core body of knowledge. It also covers other topics in that core body of knowledge, such as event-driven and concurrent programming (PF6), memory management (OS5), and functional and logic programming (IS). However, this text generally treats these topics in greater depth than that suggested by *Computing Curricula 2001.*

Our treatment of syntax and semantics includes the use of BNF grammars and a formal denotational approach to type systems and semantics. This approach is fully illustrated, so that the theory can be explored by students with the aid of a Java interpreter that directly implements the formal semantics. This approach allows students to study all the dimensions of language design using the available formal tools: BNF grammars, abstract syntax, recursive descent parsing, and functional definitions of type systems and meaning. A small imperative language that we call "Jay" is used as a basis for illustrating the principles of language design and formal methods. Java is used throughout Chapters 2–5 as the implementation language for exercising the syntax, type-checking, and semantics functions of Jay.

Another point of departure from the concepts-based texts is that we have tried to focus on a single language per paradigm. We believe that a deeper understanding of each paradigm is more important than a survey of the many languages that support it. Often the same problem is solved in each paradigm so that an instructor can better illustrate the differences between the paradigms.

Java is ideally suited to supporting most of the topics in this text. It is a widely popular language, designed in a more principled way and containing a richer collection of features than most of its predecessors. This versatility allows Java to be used in most of the lab exercises and illustrations that accompany this text. So we use Java for the imperative, object-oriented, event-driven, and concurrent programming paradigms. We use Scheme and Haskell for the functional paradigm, and Prolog for the logic programming paradigm. Why two languages for functional programming? Scheme represents a more traditional, widely used Lisp-like functional style which cannot be summarily overlooked. However, Haskell contains several contemporary features—lazy evaluation, list continuations, and a strong and versatile type system—that set it apart from the traditional functional programming style, and thus merit its separate inclusion. As a practical matter, we recommend that instructors cover only one of these two languages in the functional programming part of the course.

We have extended the discussion of formal semantics into some of the paradigm chapters as well. That is, the run-time semantics of Jay are reimplemented in the object-oriented, functional, and logic programming chapters. This strategy provides more coherence for the book overall by working out a single substantial example in each of several different programming styles.

Another *thread* that can be followed through the book is formal correctness of programs. Axiomatic correctness of imperative programs is treated in Chapter 3. Both the chapter on object-oriented programming and the one on functional programming contain a section on formal correctness.

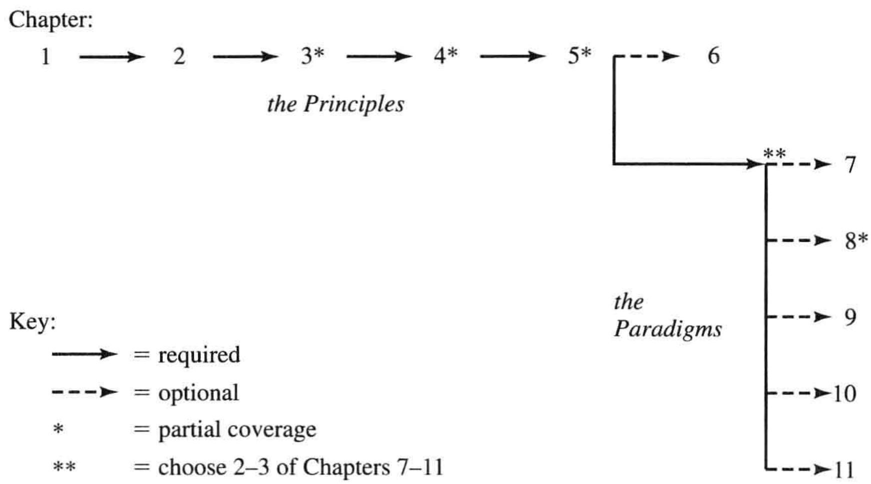
Beyond the "standard" paradigms—imperative, object-oriented, functional, and logic programming—this book identifies two other paradigms: *event-driven programming* and *concurrent programming.*

Event-driven programming characterizes programs that respond to events arriving in an unpredictable sequence, rather than controlling *a priori* the sequence in which these events occur. The most dramatic current examples of the event-driven paradigm are those programs written for Web-based interactions—online registration and electronic commerce applications, for example. But event-driven programming is a more mature paradigm than these recent applications suggest. It also appears in programs that are embedded in vehicles, operating systems, networks, and home alarm systems. We are convinced that this paradigm is sufficiently mature and distinctive from the others that it can no longer be ignored in any study of programming languages that presumes to cover the major paradigms.

Concurrent programming is a sixth paradigm treated in this text. Parallelism is central in modern computing, and is seeing increasing visibility at the programming language and application level, especially in scientific computing. Combining this emergence with traditional applications at the operating system and hardware levels, concurrent programming now requires first-class attention in a programming languages text. Thus, the book has a chapter on concurrent programming, including the related concepts of parallelism and nondeterminism that occur at the systems and applications levels.

COURSE ORGANIZATION

This text contains more material than can be covered in a one-semester course. Our experience shows that there are at least two different paths through the text for a 14-week semester. There are definitely more, as the following diagram suggests:



The chapter sequence divides the course into two major parts: a study of the principles of programming languages (Chapters 1–6) and a study of two or three major paradigms (separate from the imperative paradigm). Which paradigms to cover, of course, will vary with the preferences of the instructor and the content of other courses in the curriculum. For instance, if the curriculum regularly offers a course in parallel computing, Chapter 11 might not be covered in this course.

Some of the chapters are marked for partial coverage, again depending on local conditions. Coverage of Chapter 3 (semantics) should include type systems and denotational semantics, but operational and/or axiomatic semantics may be skipped. Some sections of Chapters 4 and 5 may also be skipped, depending on the preferences of the instructor for depth in the imperative paradigm. We do recommend that, as a minimum, the complete syntax and semantics of Jay, along with the syntax and semantics of methods and parameters that appear in these chapters, not be excluded.

Moreover, coverage of Chapter 8 (functional programming) will normally include either Scheme or Haskell and omit the other. We have intentionally designed the "paradigms" chapters to be mutually independent. However, they do revisit topics in syntax

Table 1

Two feasible 1-semester course outlines

Bowdoin	William and Mary
Introduction 0.5	Introduction 0.5
Syntax 2	Object-oriented programming 2
Semantics 2	Syntax 1.5
Imperative Programming 2	Semantics 2
Memory Management 2	Imperative Programming 1.5
Object-oriented programming 1	Exception Handling 1
Functional Programming 2	Functional Programming 2
Logic Programming 2.5	Event-driven Programming 2
	Concurrent Programming 1
Total weeks = 14	Total weeks = 13.5

and semantics that are introduced in the first four chapters, so these four should normally be covered early in the course.

Table 1 shows two sample course outlines that we have used at Bowdoin and William and Mary while class-testing this material. The numbers beside the topics indicate the approximate number of weeks in a semester devoted to each topic.

We note that Chapter 6 on Exception Handling is somewhat problematic. Logically it belongs with Chapters 2–5 which cover the syntax and semantics of imperative languages. However, unlike these chapters, Chapter 6 is more conceptual in nature (like the paradigm Chapters 7–11) in that exceptions are not modeled formally. Also, because exceptions in Java are objects, some knowledge of the object-oriented paradigm is needed to present the material on exceptions. This option of covering Chapter 7 before Chapter 6 is reflected in the William and Mary course outline.

PREREQUISITES

Knowledge of Java is normally a prerequisite for using this text, since Java is the language of illustration in most chapters. However, at William and Mary we have used this material in a course where students had only C++ and imperative programming experience. In this case, we covered the Java Tutorial appendix and the object-oriented programming chapter immediately after Chapter 1, and Table 1 shows that it is a workable alternative. In any event, we recommend that students in this course have access to a good Java reference which will provide language-specific information beyond what appears in the Java Tutorial appendix.

On the other hand, we do expect that students in this course will bring some mathematical skills, as would be found in a discrete mathematics or discrete structures course. We assume that students are familiar with the basic notions of functions, sets, and logic, as well as some exposure to the basic ideas of recursion and proof. Such a course, along with a data structures course (i.e., familiarity with linked lists, stacks, flexible arrays, and hash tables), will normally be prerequisites for this type of programming languages course. Notions and notations for functions, sets, logic, and related mathematical topics are used throughout this text. A summary of these notations appears in Appendix A.

Familiarity with the Java realizations of these ideas is helpful, but not necessary since they are explained as they are used in the text and are also summarized in Appendix C. The software for this text can be used with any implementation of Java 1.1 or higher. We have implemented the Java software for this book using both Sun's JDK Java and Metrowerks' Codewarrior Java.

WEBSITE AND PEDAGOGICAL SUPPORT

We have also developed a considerable suite of software to accompany this text. You should use the website **www.mhhe.com/tucker** as a source for downloading that software and communicating with the authors as you teach the course. This website contains the following specific pedagogical support materials:

- A complete Java implementation of the formal syntax, type system, and semantics of Jay, as discussed in Chapters 1–4 and summarized in Appendix B. Specific references to this software are made in many examples and exercises throughout the text.
- A set of tools for "animating" various syntactic and semantic functions that are discussed in this text. Algorithm animation is an active area of research in computer science education, and we encourage instructors to experiment with these tools to help students visualize the semantic features of programming languages.
- A set of downloadable transparency masters for all figures and tables in the text.
- Answers to the exercises; available to instructors via a secure password.

ACKNOWLEDGMENTS

Many persons have helped guide us in the development of this text. James Lu was a key collaborator in the early conceptualization of this text. Colleagues Bill Bynum at the College of William and Mary and Laurie King at the College of the Holy Cross contributed to Chapters 4 and 8, respectively. The students at Bowdoin and William and Mary patiently worked through early versions of this material as we developed and class-tested it. Notably, Doug Vail developed solutions to some of the more challenging problems. We also appreciate the work of colleagues Eric Chown (Bowdoin) and Jack Wileden (University of Massachusetts, Amherst) for class-testing a complete draft of this text in Spring 2001, when they provided extensive and detailed suggestions. We thank all of our reviewers:

Manuel E. Bermudez	*University of Florida*
Sanjay Chawla	*University of Minnesota*
Charles Dana	*California Polytechnic University, San Luis Obispo*
Robert Van Engelen	*Florida State University*
Arthur Fleck	*University of Iowa*
Peter N. Gabrovsky	*California State University, Northridge*
Roger T. Hartley	*New Mexico State University*

Alan Kaplan	*Clemson University*
Srini Ramaswamy	*Tennessee Technological University*
Jack C. Wileden	*University of Massachusetts, Amherst*
Salih Yurttas	*Texas A&M University*

for their careful readings and constructive recommendations throughout the development of this text. Most of their recommendations have been incorporated in this final revision, and the text is greatly improved by their collective insight.

Finally, the cover design uses several photographs of Western Ukrainian cathedrals, which were taken in Spring 2001 by Allen Tucker. This design could convey the idea that overlaying a rigorous language design methodology (the quilt pattern) on top of many different languages (the cathedrals) can help dispel the idea that programming languages are no more than a modern Tower of Babel. If this seems like an over-interpretation, readers can simply enjoy the cover design for its sheer artistry!

Allen B. Tucker — *Bowdoin College*

Robert E. Noonan — *College of William and Mary*

Brief Contents

Contents

1

Overview

"... the tools we are trying to use and the language we are using to express or record our thoughts are the major determining factors determining what we can think or express at all!"

Edsger W. Dijkstra [1972]

CHAPTER OUTLINE

Programming languages, like our "natural" languages, are designed to facilitate the expression and communication of ideas between people. The ideas expressed in natural languages cover the whole spectrum of human expression, including prose and poetry, as well as a wide range of subject matter. However, programming languages differ from natural languages in two important ways. First, they have a narrower expressive domain, in that they facilitate only the communication of *algorithmic* ideas between people. Second, programming languages also enable the communication of algorithmic ideas between people and computing machines. Thus the design of a programming language must respond to different requirements than a natural language. We shall explore these requirements and design alternatives for programming languages in this text.

In this study, we will see that there are many similarities between the features of programming languages and the analogous features that characterize natural languages. We will also see that there are fundamental differences, brought on by the special computational environment in which a program must function. We explore these differences in a fairly rigorous way. This study includes a formal treatment of the principles of programming language design and a hands-on examination of the major programming paradigms that these languages support.

The programming languages of tomorrow's computers will be designed by those who understand not only the features, strengths, and weaknesses of the programming languages of today, but also the new application needs and programming potential that can be offered by the computers of the future.

1.1 PRINCIPLES OF LANGUAGE DESIGN

Language designers need to have a basic vocabulary about language structure, meaning, and other pragmatic features that aids in the understanding of how languages work with computers to help programmers express algorithmic ideas. This vocabulary most naturally expresses itself in the form of language design *principles,* many of which are borrowed from linguistics and mathematics. The principles that underlie the design of programming languages fall into the following categories:

- Syntax.
- Type systems and semantics.
- Memory management.
- Exception handling.

These areas are the principal topics of Chapters 2, 3, 5, and 6 respectively, and they are briefly summarized below.

Syntax This design category helps us understand what constitutes a correctly written program. That is, what is the grammar for writing programs in the language, and what is the basic vocabulary of words and symbols that programmers use to form syntactically correct programs. Programming language designers have borrowed strongly from the work of linguists in this area. We shall see that the syntactic structure of modern programming languages is defined using the linguistic formalism called a *context-free grammar.* This is done both for simplicity and clarity and to enable a more rigorous treatment of the concepts.

Type Systems and Semantics This area addresses the types of values that programs can manipulate and the meaning (semantics) of these programs. We shall see that type systems and semantics are also best understood using a formal approach. When we