GLOBAL EDITION

Concepts of Programming Languages

ELEVENTH EDITION

Robert W. Sebesta

ALWAYS LEARNING

PEARSON

digital resources for students

Your new textbook provides 12-month access to digital resources that may include VideoNotes (step-by-step video tutorials on programming concepts), source code, web chapters, quizzes, and more. Refer to the preface in the textbook for a detailed list of resources.

Follow the instructions below to register for the Companion Website for Robert Sebesta's *Concepts of Programming Languages,* Eleventh Edition, Global Edition.

1. Go to www.pearsonglobaleditions.com/Sebesta
2. Click Companion Website
3. Click Register and follow the on-screen instructions to create a login name and password

Use a coin to scratch off the coating and reveal your access code.
Do not use a sharp knife or other sharp object as it may damage the code.

Use the login name and password you created during registration to start using the digital resources that accompany your textbook.

IMPORTANT:

This access code can only be used once. This subscription is valid for 12 months upon activation and is not transferable. If the access code has already been revealed it may no longer be valid.

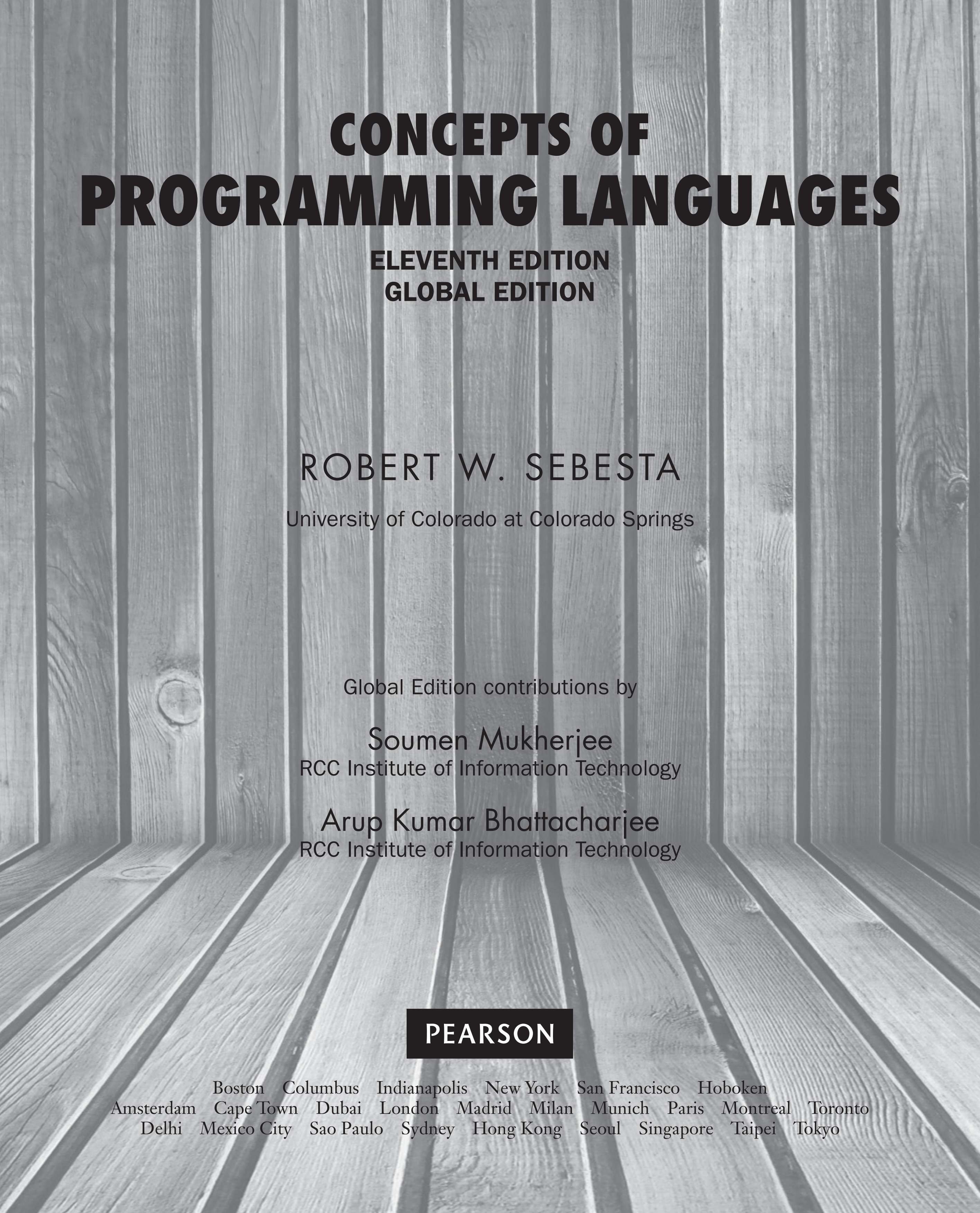
For technical support go to http://247pearsoned.custhelp.com

This page intentionally left blank

CONCEPTS OF PROGRAMMING LANGUAGES

ELEVENTH EDITION
GLOBAL EDITION

This page intentionally left blank



CONCEPTS OF PROGRAMMING LANGUAGES

ELEVENTH EDITION
GLOBAL EDITION

ROBERT W. SEBESTA

University of Colorado at Colorado Springs

Global Edition contributions by

Soumen Mukherjee
RCC Institute of Information Technology

Arup Kumar Bhattacharjee
RCC Institute of Information Technology

PEARSON

Boston Columbus Indianapolis New York San Francisco Hoboken
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editorial Director: *Marcia Horton*
Executive Editor: *Matt Goldstein*
Editorial Assistant: *Kelsey Loanes*
VP of Marketing: *Christy Lesko*
Director of Field Marketing: *Tim Galligan*
Product Marketing Manager: *Bram van Kempen*
Field Marketing Manager: *Demetrius Hall*
Marketing Assistant: *Jon Bryant*
Director of Product Management: *Erin Gregg*
Team Lead Product Management: *Scott Disanno*
Program Manager: *Carole Snyder*
Production Project Manager: *Pavithra Jayapaul, Jouve India*
Procurement Manager: *Mary Fischer*
Senior Specialist, Program Planning and Support: *Maura Zaldivar-Garcia*
Assistant Acquisitions Editor, Global Edition: *Aditee Agarwal*
Project Editor, Global Edition: *Amrita Naskar*
Manager, Media Production, Global Edition: *Vikram Kumar*
Senior Manufacturing Controller, Production, Global Edition: *Trudy Kimber*
Cover Designer: *Lumina Datamatics Ltd.*
Manager, Rights Management: *Rachel Youdelman*
Senior Project Manager, Rights Management: *Timothy Nicholls*
Cover Art: *Viachaslau Kraskouski/Shutterstock*
Full-Service Project Management: *Mahalatchoumy Saravanan, Jouve India*

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com

ISBN 10: 1-292-10055-9
ISBN 13: 978-1-292-10055-5

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1

Typeset in Janson Text LT Std 10/12 by Aptara

Printed and bound by Vivar in Malaysia

Changes for the Eleventh Edition of Concepts of Programming Languages

- **Chapter 6**: Deleted the discussions of Ada's subrange types, array initialization, records, union types, pointers, and strong typing
- **Chapter 7**: Deleted the discussions of Ada operator associativity and mixed-mode expressions
- **Chapter 8**: Expanded the paragraph on F# selection statements in Section 8.2.1.5

  Deleted the discussion of the Ada **`for`** statement
- **Chapter 9**: Added three design issues for subprograms in Section 9.3

  Deleted the discussions of Ada and Fortran multidimensional parameters
- **Chapter 10**: Replaced example program **`Main_2`**, written in Ada, with an equivalent program written in JavaScript in Section 10.4.2

  Changed Figure 10.9 to reflect this new JavaScript example
- **Chapter 11**: Deleted the discussions of Ada abstract data types, generic procedures, and packages

  Added a new paragraph to Section 11.4.3 (Abstract Data Types in Java)
- **Chapter 12**: Added a paragraph to Section 12.2.2 (Inheritance) that discusses access control

  Expanded the discussion of class variables in Section 12.2.2
  Added a paragraph to Section 12.4.4 that discusses final classes in Objective-C
  Reorganized Sections 12.5 to 12.9 into a single section
  Added Table 12.1 on language design choices to Section 12.4.6.4
  Added a new section, Section 6 (Reflection), including example programs in Java and C#
- **Chapter 13**: Deleted the discussions of Ada task termination and task priorities
- **Chapter 14**: Deleted exception handling in Ada

  Added a new section, 14.4 (Exception Handling in Python and Ruby)

This page intentionally left blank

Preface

Changes for the Eleventh Edition

The goals, overall structure, and approach of this eleventh edition of *Concepts of Programming Languages* remain the same as those of the ten earlier editions. The principal goals are to introduce the fundamental constructs of contemporary programming languages and to provide the reader with the tools necessary for the critical evaluation of existing and future programming languages. A secondary goal is to prepare the reader for the study of compiler design, by providing an in-depth discussion of programming language structures, presenting a formal method of describing syntax, and introducing approaches to lexical and syntactic analysis.

The eleventh edition evolved from the tenth through several different kinds of changes. To maintain the currency of the material, much of the discussion of older programming languages, particularly Ada and Fortran, has been removed. For example, the descriptions of Ada's records, union types, and pointers were removed from Chapter 6. Likewise, the description of Ada's **`for`** statement was removed from Chapter 8 and the discussion of Ada's abstract data types was removed from Chapter 11.

On the other hand, a section on reflection that includes two complete program examples was added to Chapter 12, a section on exception handling in Python and Ruby was added to Chapter 14, and a table of the design choices of a few common languages for support for object-oriented programming was added to Chapter 12.

In some cases, material has been moved. For example, Section 9.10 was moved backward to become the new Section 9.8.

In one case, example program `MAIN_2` in Chapter 10 was rewritten in JavaScript, previously having been written in Ada.

Chapter 12 was substantially revised, with several new paragraphs, two new sections, and numerous other changes to improve clarity.

The Vision

This book describes the fundamental concepts of programming languages by discussing the design issues of the various language constructs, examining the design choices for these constructs in some of the most common languages, and critically comparing design alternatives.

Any serious study of programming languages requires an examination of some related topics, among which are formal methods of describing the syntax and semantics of programming languages, which are covered in Chapter 3. Also, implementation techniques for various language constructs must be considered: Lexical and syntax analysis are discussed in Chapter 4, and implementation of subprogram linkage is covered in Chapter 10. Implementation of some other language constructs is discussed in various other parts of the book.

The following paragraphs outline the contents of the eleventh edition.

Chapter Outlines

Chapter 1 begins with a rationale for studying programming languages. It then discusses the criteria used for evaluating programming languages and language constructs. The primary influences on language design, common design tradeoffs, and the basic approaches to implementation are also examined.

Chapter 2 outlines the evolution of the languages that are discussed in this book. Although no attempt is made to describe any language completely, the origins, purposes, and contributions of each are discussed. This historical overview is valuable, because it provides the background necessary to understanding the practical and theoretical basis for contemporary language design. It also motivates further study of language design and evaluation. Because none of the remainder of the book depends on Chapter 2, it can be read on its own, independent of the other chapters.

Chapter 3 describes the primary formal method for describing the syntax of programming language—BNF. This is followed by a description of attribute grammars, which describe both the syntax and static semantics of languages. The difficult task of semantic description is then explored, including brief introductions to the three most common methods: operational, denotational, and axiomatic semantics.

Chapter 4 introduces lexical and syntax analysis. This chapter is targeted to those Computer Science departments that no longer require a compiler design course in their curricula. Similar to Chapter 2, this chapter stands alone and can be studied independently of the rest of the book, except for Chapter 3, on which it depends.

Chapters 5 through 14 describe in detail the design issues for the primary constructs of programming languages. In each case, the design choices for several example languages are presented and evaluated. Specifically, Chapter 5 covers the many characteristics of variables, Chapter 6 covers data types, and Chapter 7 explains expressions and assignment statements. Chapter 8 describes control statements, and Chapters 9 and 10 discuss subprograms and their implementation. Chapter 11 examines data abstraction facilities. Chapter 12 provides an in-depth discussion of language features that support object-oriented programming (inheritance and dynamic method binding), Chapter 13 discusses concurrent program units, and Chapter 14 is about exception handling, along with a brief discussion of event handling.

Chapters 15 and 16 describe two of the most important alternative programming paradigms: functional programming and logic programming. However, some of the data structures and control constructs of functional programming languages are discussed in Chapters 6 and 8. Chapter 15 presents an introduction to Scheme, including descriptions of some of its primitive functions, special forms, and functional forms, as well as some examples of simple functions written in Scheme. Brief introductions to ML, Haskell, and F# are given to illustrate some different directions in functional language design. Chapter 16 introduces logic programming and the logic programming language, Prolog.

To the Instructor

In the junior-level programming language course at the University of Colorado at Colorado Springs, the book is used as follows: We typically cover Chapters 1 and 3 in detail, and though students find it interesting and beneficial reading, Chapter 2 receives little lecture time due to its lack of hard technical content. Because no material in subsequent chapters depends on Chapter 2, as noted earlier, it can be skipped entirely, and because we require a course in compiler design, Chapter 4 is not covered.

Chapters 5 through 9 should be relatively easy for students with extensive programming experience in C++, Java, or C#. Chapters 10 through 14 are more challenging and require more detailed lectures.

Chapters 15 and 16 are entirely new to most students at the junior level. Ideally, language processors for Scheme and Prolog should be available for students required to learn the material in these chapters. Sufficient material is included to allow students to dabble with some simple programs.

Undergraduate courses will probably not be able to cover all of the material in the last two chapters. Graduate courses, however, should be able to completely discuss the material in those chapters by skipping over some parts of the early chapters on imperative languages.

Supplemental Materials

The following supplements are available to all readers of this book at *www.pearsonglobaleditions.com/Sebesta*.

- A set of lecture note slides. PowerPoint slides are available for each chapter in the book.
- All of the figures from the book.

A companion Web site to the book is available at *www.pearsonglobaleditions.com/Sebesta*. This site contains mini-manuals (approximately 100-page tutorials) on a handful of languages. These assume that the student knows how to program in some other language, giving the student enough information to complete the chapter materials in each language. Currently the site includes manuals for C++, C, Java, and Smalltalk.

Solutions to many of the problem sets are available to qualified instructors in our Instructor Resource Center at *www.pearsonglobaleditions.com/Sebesta*.

Language Processor Availability

Processors for and information about some of the programming languages discussed in this book can be found at the following Web sites:

C, C++, Fortran, and Ada	*gcc.gnu.org*
C# and F#	*microsoft.com*
Java	*java.sun.com*
Haskell	*haskell.org*
Lua	*www.lua.org*
Scheme	*www.plt-scheme.org/software/drscheme*
Perl	*www.perl.com*
Python	*www.python.org*
Ruby	*www.ruby-lang.org*

JavaScript is included in virtually all browsers; PHP is included in virtually all Web servers.

All this information is also included on the companion Web site.

Acknowledgments

The suggestions from outstanding reviewers contributed greatly to this book's present form and contents. In alphabetical order, they are:

Aaron Rababaah	*University of Maryland at Eastern Shore*
Amar Raheja	*California State Polytechnic University–Pomona*
Amer Diwan	*University of Colorado*
Bob Neufeld	*Wichita State University*
Bruce R. Maxim	*University of Michigan–Dearborn*
Charles Nicholas	*University of Maryland–Baltimore County*
Cristian Videira Lopes	*University of California–Irvine*
Curtis Meadow	*University of Maine*
David E. Goldschmidt	
Donald Kraft	*Louisiana State University*
Duane J. Jarc	*University of Maryland, University College*
Euripides Montagne	*University of Central Florida*
Frank J. Mitropoulos	*Nova Southeastern University*
Gloria Melara	*California State University–Northridge*
Hossein Saiedian	*University of Kansas*
I-ping Chu	*DePaul University*
Ian Barland	*Radford University*
K. N. King	*Georgia State University*
Karina Assiter	*Wentworth Institute of Technology*
Mark Llewellyn	*University of Central Florida*
Matthew Michael Burke	
Michael Prentice	*SUNY Buffalo*
Nancy Tinkham	*Rowan University*
Neelam Soundarajan	*Ohio State University*
Nigel Gwee	*Southern University–Baton Rouge*
Pamela Cutter	*Kalamazoo College*
Paul M. Jackowitz	*University of Scranton*
Paul Tymann	*Rochester Institute of Technology*
Richard M. Osborne	*University of Colorado–Denver*
Richard Min	*University of Texas at Dallas*
Robert McCloskey	*University of Scranton*
Ryan Stansifer	*Florida Institute of Technology*
Salih Yurttas	*Texas A&M University*
Saverio Perugini	*University of Dayton*
Serita Nelesen	*Calvin College*
Simon H. Lin	*California State University–Northridge*

Stephen Edwards	*Virginia Tech*
Stuart C. Shapiro	*SUNY Buffalo*
Sumanth Yenduri	*University of Southern Mississippi*
Teresa Cole	*Boise State University*
Thomas Turner	*University of Central Oklahoma*
Tim R. Norton	*University of Colorado–Colorado Springs*
Timothy Henry	*University of Rhode Island*
Walter Pharr	*College of Charleston*
Xiangyan Zeng	*Fort Valley State University*

Numerous other people provided input for the previous editions of *Concepts of Programming Languages* at various stages of its development. All of their comments were useful and greatly appreciated. In alphabetical order, they are: Vicki Allan, Henry Bauer, Carter Bays, Manuel E. Bermudez, Peter Brouwer, Margaret Burnett, Paosheng Chang, Liang Cheng, John Crenshaw, Charles Dana, Barbara Ann Griem, Mary Lou Haag, John V. Harrison, Eileen Head, Ralph C. Hilzer, Eric Joanis, Leon Jololian, Hikyoo Koh, Jiang B. Liu, Meiliu Lu, Jon Mauney, Robert McCoard, Dennis L. Mumaugh, Michael G. Murphy, Andrew Oldroyd, Young Park, Rebecca Parsons, Steve J. Phelps, Jeffery Popyack, Steven Rapkin, Hamilton Richard, Tom Sager, Raghvinder Sangwan, Joseph Schell, Sibylle Schupp, Mary Louise Soffa, Neelam Soundarajan, Ryan Stansifer, Steve Stevenson, Virginia Teller, Yang Wang, John M. Weiss, Franck Xia, and Salih Yurnas.

Matt Goldstein, editor; Kelsey Loanes, editorial assistant; Team Lead Product Management, Scott Disanno; Pavithra Jayapaul, and Mahalatchoumy Saravanan, all deserve my gratitude for their efforts to produce the eleventh edition both quickly and carefully.

The publishers would like to thank the following for reviewing the Global Edition:

Sandeep B. L., *M. S. Ramaiah Institute of Technology*
Koushik S., *M. S. Ramaiah Institute of Technology*
Sheena V. M., *Don Bosco College*

About the Author

Robert Sebesta is an Associate Professor Emeritus in the Computer Science Department at the University of Colorado–Colorado Springs. Professor Sebesta received a BS in applied mathematics from the University of Colorado in Boulder and MS and PhD degrees in computer science from Pennsylvania State University. He has taught computer science for more than 40 years. His professional interests are the design and evaluation of programming languages and Web programming.

This page intentionally left blank

Contents

This page intentionally left blank

CONCEPTS OF PROGRAMMING LANGUAGES

ELEVENTH EDITION
GLOBAL EDITION

This page intentionally left blank

1 Preliminaries

Before we begin discussing the concepts of programming languages, we must consider a few preliminaries. First, we explain some reasons why computer science students and professional software developers should study general concepts of language design and evaluation. This discussion is especially valuable for those who believe that a working knowledge of one or two programming languages is sufficient for computer scientists. Then, we briefly describe the major programming domains. Next, because the book evaluates language constructs and features, we present a list of criteria that can serve as a basis for such judgments. Then, we discuss the two major influences on language design: machine architecture and program design methodologies. After that, we introduce the various categories of programming languages. Next, we describe a few of the major trade-offs that must be considered during language design.

Because this book is also about the implementation of programming languages, this chapter includes an overview of the most common general approaches to implementation. Finally, we briefly describe a few examples of programming environments and discuss their impact on software production.

1.1 Reasons for Studying Concepts of Programming Languages

It is natural for students to wonder how they will benefit from the study of programming language concepts. After all, many other topics in computer science are worthy of serious study. The following is what we believe to be a compelling list of potential benefits of studying concepts of programming languages:

- *Increased capacity to express ideas.* It is widely believed that the depth at which people can think is influenced by the expressive power of the language in which they communicate their thoughts. Those with only a weak understanding of natural language are limited in the complexity of their thoughts, particularly in depth of abstraction. In other words, it is difficult for people to conceptualize structures they cannot describe, verbally or in writing.

  Programmers, in the process of developing software, are similarly constrained. The language in which they develop software places limits on the kinds of control structures, data structures, and abstractions they can use; thus, the forms of algorithms they can construct are likewise limited. Awareness of a wider variety of programming language features can reduce such limitations in software development. Programmers can increase the range of their software development thought processes by learning new language constructs.

  It might be argued that learning the capabilities of other languages does not help a programmer who is forced to use a language that lacks those capabilities. That argument does not hold up, however, because often, language constructs can be simulated in other languages that do not support those constructs directly. For example, a C programmer who had learned the structure and uses of associative arrays in Perl (Christianson et al., 2012) might design structures that simulate associative arrays in that language. In other words, the study of programming language concepts builds an appreciation for valuable language features and constructs and encourages programmers to use them, even when the language they are using does not directly support such features and constructs.

- *Improved background for choosing appropriate languages.* Some professional programmers have had little formal education in computer science; rather, they have developed their programming skills independently or through in-house training programs. Such training programs often limit instruction to one or two languages that are directly relevant to the current projects of the organization. Other programmers received their formal training years ago. The languages they learned then are no longer used, and many features now available in programming languages were not widely known at the time. The result is that many programmers, when given a choice of languages for a new project, use the language with which they are most familiar, even if it is poorly suited for the project at hand. If these programmers were familiar with a wider range of languages and language constructs, they would be better able to choose the language with the features that best address the problem.

  Some of the features of one language often can be simulated in another language. However, it is preferable to use a feature whose design has been integrated into a language than to use a simulation of that feature, which is often less elegant, more cumbersome, and less safe.

- *Increased ability to learn new languages.* Computer programming is still a relatively young discipline, and design methodologies, software development tools, and programming languages are still in a state of continuous evolution. This makes software development an exciting profession, but it also means that continuous learning is essential. The process of learning a new programming language can be lengthy and difficult, especially for someone who is comfortable with only one or two languages and has never examined programming language concepts in general. Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned. For example, programmers who understand the concepts of object-oriented programming will have a much easier time learning Ruby (Thomas et al., 2013) than those who have never used those concepts.

  The same phenomenon occurs in natural languages. The better you know the grammar of your native language, the easier it is to learn a second language. Furthermore, learning a second language has the benefit of teaching you more about your first language.

  The TIOBE Programming Community issues an index (`http://www.tiobe.com/index.php/content/paperinfo/tpci/index.htm`) that is an indicator of the relative popularity of programming languages. For example, according to the index, C, Java, and Objective-C were the three most popular languages in use in February 2014.[1] However, dozens of other

1. Note that this index is only one measure of the popularity of programming languages, and its accuracy is not universally accepted.

languages were widely used at the time. The index data also show that the distribution of usage of programming languages is always changing. The number of languages in use and the dynamic nature of the statistics imply that every software developer must be prepared to learn different languages.

Finally, it is essential that practicing programmers know the vocabulary and fundamental concepts of programming languages so they can read and understand programming language descriptions and evaluations, as well as promotional literature for languages and compilers. These are the sources of information needed in order to choose and learn a language.

- *Better understanding of the significance of implementation.* In learning the concepts of programming languages, it is both interesting and necessary to touch on the implementation issues that affect those concepts. In some cases, an understanding of implementation issues leads to an understanding of why languages are designed the way they are. In turn, this knowledge leads to the ability to use a language more intelligently, as it was designed to be used. We can become better programmers by understanding the choices among programming language constructs and the consequences of those choices.

  Certain kinds of program bugs can be found and fixed only by a programmer who knows some related implementation details. Another benefit of understanding implementation issues is that it allows us to visualize how a computer executes various language constructs. In some cases, some knowledge of implementation issues provides hints about the relative efficiency of alternative constructs that may be chosen for a program. For example, programmers who know little about the complexity of the implementation of subprogram calls often do not realize that a small subprogram that is frequently called can be a highly inefficient design choice.

  Because this book touches on only a few of the issues of implementation, the previous two paragraphs also serve well as rationale for studying compiler design.
- *Better use of languages that are already known.* Most contemporary programming languages are large and complex. Accordingly, it is uncommon for a programmer to be familiar with and use all of the features of a language he or she uses. By studying the concepts of programming languages, programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features.
- *Overall advancement of computing.* Finally, there is a global view of computing that can justify the study of programming language concepts. Although it is usually possible to determine why a particular programming language became popular, many believe, at least in retrospect, that the most popular languages are not always the best available. In some cases, it might be concluded that a language became widely used, at least in part, because those in positions to choose languages were not sufficiently familiar with programming language concepts.

  For example, many people believe it would have been better if ALGOL 60 (Backus et al., 1963) had displaced Fortran (McCracken, 1961) in the