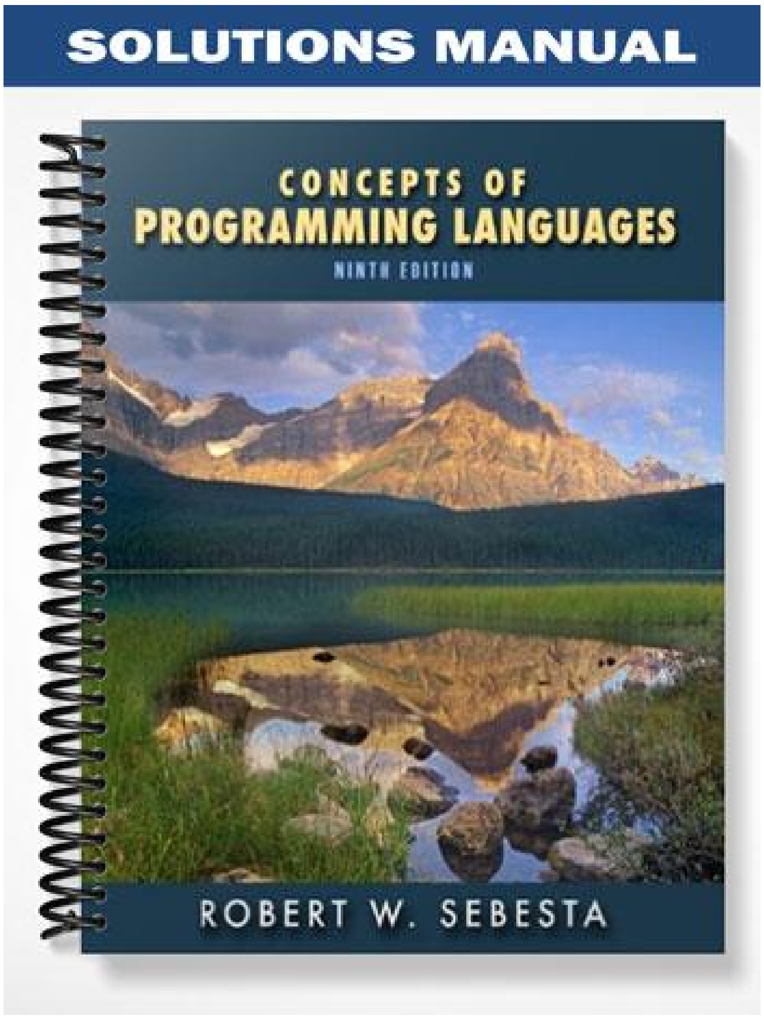
SOLUTIONS MANUAL

CONCEPTS OF PROGRAMMING LANGUAGES

NINTH EDITION

ROBERT W. SEBESTA

Instructor's Solutions Manual

to

Concepts of Programming Languages

Ninth Edition

R.W. Sebesta

Preface

Changes for the Ninth Edition

The goals, overall structure, and approach of this ninth edition of **Concepts of Programming Languages** remain the same as those of the eight earlier editions. The principal goals are to introduce the main constructs of contemporary programming languages and to provide the reader with the tools necessary for the critical evaluation of existing and future programming languages. An additional 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 ninth edition evolved from the eighth through several different kinds of changes. To maintain the currency of the material, some of the discussion of older programming languages has been replaced by material on newer languages. For example, discussions of Lua's table types, Python's tuples and list comprehensions, and Ruby's slices were added to Chapter 6. Some older material was removed from Chapter 7, shortening it by three pages. The section on JavaScript's object model was removed from Chapter 11. In some cases, material has been moved. For example, three sections of Chapter 5 on typing were moved to Chapter 6, whose topic is data types. In some sections, material has been expanded. For example, in Chapter 4, the lexical analyzer code was expanded to a complete program and actual output from it has been included. The section on scoping in Chapter 5 was expanded by adding subsections on global scope and declaration order. The discussion of Ruby's abstract data types in Chapter 11 was revised and enlarged. In Chapter 15, a section on tail recursion in Scheme was added. In Chapter 3, the introduction to semantics was revised and the subsections were reordered. In addition, numerous minor changes were made to a large number of sections of the book, primarily to improve clarity. Finally, over 50 problems and programming exercises, as well as over 100 review questions, were added to the book.

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 ninth 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 trade-offs, and the basic approaches to implementation are also examined.

Chapter 2 outlines the evolution of most of the important languages discussed in this book. Although no language is described 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. In addition, 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 colleges that no longer require a compiler design course in their curricula. Like Chapter 2, this chapter stands alone and can be read independently of the rest of the book.

Chapters 5 through 14 describe in detail the design issues for the primary constructs of the imperative 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.

The last two chapters (15 and 16) describe two of the most important alternative programming paradigms: functional programming and logic programming. 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 and Haskell are given to illustrate some different kinds of functional language. 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 parts of the early chapters on imperative languages.

Supplemental Materials

The following supplements are available to all readers of this book at www.aw.com/cssupport.

- A set of lecture note slides. PowerPoint® slides are available for each chapter in the book.
- PowerPoint® slides containing all the figures in the book.

To reinforce learning in the classroom, to assist with the hands-on lab component of this course, and/or to facilitate students in a distance learning situation, access the Companion Website at www.aw.com/sebesta. This site contains:

- Mini-manuals (approximately 100-page tutorials) on a handful of languages. These proceed on the assumption 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.
- Self-Assessment Quizzes. Students can complete a series of multiple-choice and fill-in-the-blank exercises to check their understanding of each chapter.

Solutions to many of the problem sets are available to qualified instructors on our Instructor Resource Center at www.aw-bc.com/irc. Please contact your local Addison-Wesley sales representative or send an email to **computing@aw.com** for more information.

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#	**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/en/**

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.

Acknowledgements

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

I-ping Chu,	**DePaul University**
Amer Diwan,	**University of Colorado**
Stephen Edwards,	**Virginia Tech**
Nigel Gwee,	**Southern University–Baton Rouge**
K. N. King,	**Georgia State University**
Donald Kraft,	**Louisiana State University**
Simon H. Lin,	**California State University–Northridge**
Mark Llewellyn,	**University of Central Florida**
Bruce R. Maxim,	**University of Michigan–Dearborn**
Gloria Melara,	**California State University–Northridge**
Frank J. Mitropoulos,	**Nova Southeastern University**
Euripides Montagne,	**University of Central Florida**
Bob Neufeld,	**Wichita State University**
Amar Raheja,	**California State Polytechnic University–Pomona**
Hossein Saiedian,	**University of Kansas**
Neelam Soundarajan,	**Ohio State University**
Paul Tymann,	**Rochester Institute of Technology**
Cristian Videira Lopes,	**University of California–Irvine**
Salih Yurttas,	Texas A&M 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, Raghvinder Sangwan, Steven Rapkin, Hamilton Richard, Tom Sager, 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, Sarah Milmore, editorial assistant, and Meredith Gertz, senior production supervisor of Addison-Wesley, and Dennis Free of Aptara, all deserve my gratitude for their efforts to produce the ninth edition both quickly and carefully.

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 B.S. in applied mathematics from the University of Colorado in Boulder and M.S. and Ph.D. degrees in Computer Science from the Pennsylvania State University. He has taught computer science for more than 38 years. His professional interests are the design and evaluation of programming languages, compiler design, and software testing methods and tools.

Contents

Answers to Selected Problems

Chapter 1

Problem Set:

3. Some arguments for having a single language for all programming domains are: It would dramatically cut the costs of programming training and compiler purchase and maintenance; it would simplify programmer recruiting and justify the development of numerous language dependent software development aids.

4. Some arguments against having a single language for all programming domains are: The language would necessarily be huge and complex; compilers would be expensive and costly to maintain; the language would probably not be very good for any programming domain, either in compiler efficiency or in the efficiency of the code it generated. More importantly, it would not be easy to use, because regardless of the application area, the language would include many unnecessary and confusing features and constructs (those meant for other application areas). Different users would learn different subsets, making maintenance difficult.

5. One possibility is wordiness. In some languages, a great deal of text is required for even simple complete programs. For example, COBOL is a very wordy language. In Ada, programs require a lot of duplication of declarations. Wordiness is usually considered a disadvantage, because it slows program creation, takes more file space for the source programs, and can cause programs to be more difficult to read.

7. The argument for using the right brace to close all compounds is simplicity—a right brace always terminates a compound. The argument against it is that when you see a right brace in a program, the location of its matching left brace is not always obvious, in part because all multiple-statement control constructs end with a right brace.

8. The reasons why a language would distinguish between uppercase and lowercase in its identifiers are: (1) So that variable identifiers may look different than identifiers that are names for constants, such as the convention of using uppercase for constant names and using lowercase for variable names in C, and (2) so that catenated words as names can have their first letter distinguished, as in `TotalWords`. (Some think it is better to include a connector, such as underscore.) The primary reason why a language would not distinguish between uppercase and lowercase in identifiers is it makes programs less readable, because words that look very similar are actually completely different, such as `SUM` and `Sum`.

10. One of the main arguments is that regardless of the cost of hardware, it is not free. Why write a program that executes slower than is necessary. Furthermore, the difference between a well-written efficient program and one that is poorly written can be a factor of two or three. In many other fields of endeavor, the difference between a good job and a poor job may be 10 or 20 percent. In programming, the difference is much greater.

15. The use of type declaration statements for simple scalar variables may have very little effect on the readability of programs. If a language has no type declarations at all, it may be an aid to readability, because regardless of where a variable is seen in the program text, its type can be determined without looking elsewhere. Unfortunately, most languages that allow implicitly declared variables also include explicit declarations. In a program in such a language, the

declaration of a variable must be found before the reader can determine the type of that variable when it is used in the program.

18. The main disadvantage of using paired delimiters for comments is that it results in diminished reliability. It is easy to inadvertently leave off the final delimiter, which extends the comment to the end of the next comment, effectively removing code from the program. The advantage of paired delimiters is that you **can** comment out areas of a program. The disadvantage of using only beginning delimiters is that they must be repeated on every line of a block of comments. This can be tedious and therefore error-prone. The advantage is that you cannot make the mistake of forgetting the closing delimiter.