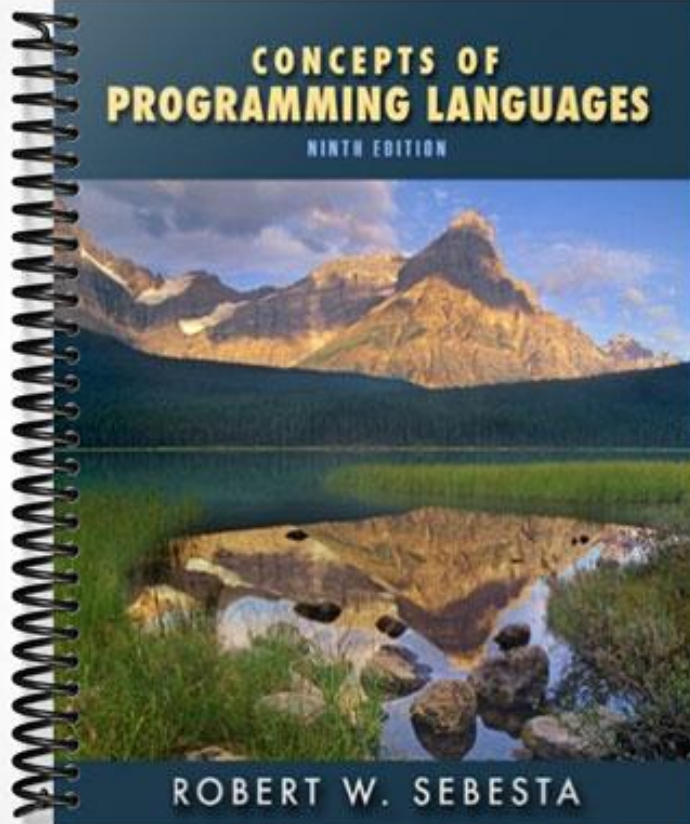


SOLUTIONS MANUAL

**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 To maintain the currency of the material, some of the discussion of older programming languages has been replaced by material on newer languages.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

- Chapter 1 Preliminaries 1**
 - 1.1 Reasons for Studying Concepts of Programming Languages 2
 - 1.2 Programming Domains 5
 - 1.3 Language Evaluation Criteria 7
 - 1.4 Influences on Language Design 20
 - 1.5 Language Categories 24
 - 1.6 Language Design Trade-offs 25
 - 1.7 Implementation Methods 26
 - 1.8 Programming Environments 33
 - Summary • Review Questions • Problem Set 34

- Chapter 2 Evolution of the Major Programming Languages 39**
 - 2.1 Zuse’s Plankalkül 40
 - 2.2 Minimal Hardware Programming: Pseudocodes 43
 - 2.3 The IBM 704 and Fortran 46
 - 2.4 Functional Programming: LISP 51
 - 2.5 The First Step Toward Sophistication: ALGOL 60 56
 - 2.6 Computerizing Business Records: COBOL 62
 - 2.7 The Beginnings of Timesharing: BASIC 67
 - Interview: **ALAN COOPER**—User Design and Language Design 70
 - 2.8 Everything for Everybody: PL/I 72
 - 2.9 Two Early Dynamic Languages: APL and SNOBOL 75
 - 2.10 The Beginnings of Data Abstraction: SIMULA 67 76
 - 2.11 Orthogonal Design: ALGOL 68 77
 - 2.12 Some Early Descendants of the ALGOLs 79
 - 2.13 Programming Based on Logic: Prolog 85
 - 2.14 History’s Largest Design Effort: Ada 87
 - 2.15 Object-Oriented Programming: Smalltalk 91
 - 2.16 Combining Imperative and Object-Oriented Features: C++ 94
 - 2.17 An Imperative-Based Object-Oriented Language: Java 97

2.18	Scripting Languages	101
2.19	A C-Based Language for the New Millennium: C#	105
2.20	Markup/Programming Hybrid Languages	108
Summary • Bibliographic Notes • Review Questions • Problem Set		110

Chapter 3 Describing Syntax and Semantics 115

3.1	Introduction	116
3.2	The General Problem of Describing Syntax	117
3.3	Formal Methods of Describing Syntax	119
3.4	Attribute Grammars	134
History Note		135
3.5	Describing the Meanings of Programs: Dynamic Semantics	140
History Note		156
Summary • Bibliographic Notes • Review Questions • Problem Set • Programming Exercises		161

Chapter 4 Lexical and Syntax Analysis 167

4.1	Introduction	168
4.2	Lexical Analysis	169
4.3	The Parsing Problem	175
4.4	Recursive-Descent Parsing	179
4.5	Bottom-Up Parsing	188
Summary • Review Questions • Problem Set • Programming Exercises		195

Chapter 5 Names, Bindings, and Scopes 201

5.1	Introduction	202
5.2	Names	203
History Note		203
History Note		204
5.3	Variables	205
History Note		206
5.4	The Concept of Binding	207
Interview: RASMUS LERDORF—		
	Scripting Languages and Other Examples of Slick Solutions	212
5.5	Scope	225

5.6Scope and Lifetime	234
5.7Referencing Environments	234
5.8Named Constants	236
Summary • Review Questions • Problem Set • Programming Exercises		...239

Chapter 6 Data Types **247**

6.1Introduction	248
6.2 Primitive Data Types	249
6.3 Character String Types	253
History Note	255
6.4User-Defined Ordinal Types	258
6.5 Array Types	263
History Note	264
History Note	267
Interview: RASMUS LERDORF —The Open-Source Movement and Work Life		276
6.6 Associative Arrays	280
6.7 Record Types	282
6.8 Union Types	286
6.9Pointer and Reference Types	291
History Note	295
6.10Type Checking	286
6.11 Strong Typing	286
6.12 Type Equivalence	286
6.13 Theory and Data Types	286
Summary • Bibliographic Notes • Review Questions • Problem Set • Programming Exercises		304

Chapter 7 Expressions and Assignment Statements **311**

7.1Introduction	312
7.2Arithmetic Expressions	313
History Note	320
7.3Overloaded Operators	322
7.4Type Conversions	324
History Note	326

7.5Relational and Boolean Expressions	328
	History Note	328
7.6Short-Circuit Evaluation	331
7.7 Assignment Statements	332
	History Note	335
7.8Mixed-mode Assignment	337
	Summary • Review Questions • Problem Set • Programming Exercises.....	338

Chapter 8 Statement-Level Control Structures 343

8.1 Introduction	344
8.2 Selection Statements	346
	History Note	346
	History Note	348
8.3 Iterative Statements	356
	History Note	359
	Interview: LARRY WALL —Part 1: Linguistics and the Birth of Perl.....	364
8.4 Unconditional Branching	371
	History Note	372
8.5 Guarded Commands	372
8.6 Conclusions	377
	Summary • Review Questions • Problem Set • Programming Exercises.....	377

Chapter 9 Subprograms 383

9.1 Introduction	384
9.2 Fundamentals of Subprograms	384
9.3 Design Issues for Subprograms	394
9.4 Local Referencing Environments	395
9.5 Parameter-Passing Methods	397
 Interview: LARRY WALL —Part 2: Scripting Languages in General and Perl in Particular.....	398
	History Note	407
	History Note	407
	History Note	411
9.6 Parameters That Are Subprogram Names	418
	History Note	420

9.7	Overloaded Subprograms	420
9.8	Generic Subprograms	422
9.9	Design Issues for Functions	429
9.10	User-Defined Overloaded Operators	430
9.11	Coroutines	431
History Note		431
Summary • Review Questions • Problem Set • Programming Exercises		433
Chapter 10 Implementing Subprograms		439
10.1	The General Semantics of Calls and Returns	440
10.2	Implementing “Simple” Subprograms	441
10.3	Implementing Subprograms with Stack-Dynamic Local Variables	443
10.4	Nested Subprograms	451
Interview: NIKLAUS WIRTH —Keeping It Simple		452
10.5	Blocks	459
10.6	Implementing Dynamic Scoping	461
Summary • Review Questions • Problem Set		464
Chapter 11 Abstract Data Types and Encapsulation Constructs		469
11.1	The Concept of Abstraction	470
11.2	Introduction to Data Abstraction	471
11.3	Design Issues for Abstract Data Types	474
11.4	Language Examples	474
Interview: BJARNE STROUSTRUP —C++: Its Birth, Its Ubiquitousness, and Common Criticisms		478
11.5	Parameterized Abstract Data Types	490
11.6	Encapsulation Constructs	495
11.7	Naming Encapsulations	498
Summary • Review Questions • Problem Set • Programming Exercises		503
Chapter 12 Support for Object-Oriented Programming		507
12.1	Introduction	508
12.2	Object-Oriented Programming	508
12.3	Design Issues for Object-Oriented Languages	512
12.4	Support for Object-Oriented Programming in Smalltalk	516

12.5 Support for Object-Oriented Programming in C++	519
	Interview: BJARNE STROUSTRUP —On Paradigms and Better Programming.....	520
12.6 Support for Object-Oriented Programming in Java	530
12.7 Support for Object-Oriented Programming in C#	533
12.8 Support for Object-Oriented Programming in Ada	95 535
12.9 Support for Object-Oriented Programming in Ruby	540
12.10 Implementation of Object-Oriented Constructs	547
	Summary • Review Questions • Problem Set • Programming Exercises	550

Chapter 13 Concurrency **555**

13.1 Introduction	556
13.2 Introduction to Subprogram-Level Concurrency	559
13.3 Semaphores	563
	History Note	563
13.4 Monitors	568
13.5 Message Passing	570
13.6 Ada Support for Concurrency	571
13.7 Java Threads	583
13.8 C# Threads	590
13.9 Statement-Level Concurrency	592
	Summary • Bibliographic Notes • Review Questions • Problem Set • Programming Exercises	595

Chapter 14 Exception Handling and Event Handling **601**

14.1 Introduction to Exception Handling	602
	History Note	606
14.2 Exception Handling in Ada	608
14.3 Exception Handling in C++	615
14.4 Exception Handling in Java	620
	Interview: JAMES GOSLING —The Birth of Java	622
14.5 Introduction to Event Handling	630
14.6 Event Handling with Java	631
	Summary • Bibliographic Notes • Review Questions • Problem Set.....	636

Chapter 15 Functional Programming Languages **641**

15.1Introduction	642
15.2 Mathematical Functions	643
15.3 Fundamentals of Functional Programming Languages	645
15.4 The First Functional Programming Language: LISP	646
15.5 An Introduction to Scheme	650
15.6 COMMON LISP	666
15.7 ML	667
15.8 Haskell	670
15.9 Applications of Functional Languages	675
15.10 A Comparison of Functional and Imperative Languages	676
Summary • Bibliographic Notes • Review Questions • Problem Set • Programming Exercises		678

Chapter 16 Logic Programming Languages **683**

16.1Introduction	684
16.2A Brief Introduction to Predicate Calculus	684
16.3 Predicate Calculus and Proving Theorems	688
16.4An Overview of Logic Programming	690
16.5 The Origins of Prolog	692
16.6 The Basic Elements of Prolog	693
16.7 The Deficiencies of Prolog	707
16.8 Applications of Logic Programming	713
Summary • Bibliographic Notes • Review Questions • Problem Set • Programming Exercises		715

Bibliography **719**

Index **729**

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.