

Instructor's Solutions Manual

to

Concepts of Programming Languages

Sixth Edition

R.W. Sebesta

Preface

Changes to the Sixth Edition

The goals, overall structure, and approach of this sixth edition of *Concepts of Programming Languages* remain the same as those of the five earlier editions. The principal goal is 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. There were several sources of our motivations for the changes in the sixth edition. First, to maintain the currency of the material, much of the discussion of older programming languages has been removed. In its place is material on newer languages. Especially interesting historical information on older programming languages has been retained but placed in historical side boxes. Second, the material has been updated to reflect the fact that most students now come to this course with a basic understanding of object-oriented programming. We shortened the discussion of basics and expanded the discussion of advanced topics.

Third, reviewer comments have prompted several changes. For example, the material on functional programming languages has been reorganized and strengthened. Also, we have added a programming exercises section at the end of most chapters to give students experience with the concepts described in the book and to make the concepts more realistic and appealing. The book now has a new supplement: a companion Web site with a few small language manuals, interactive quizzes for students, and additional programming projects. Finally, interviews with the designers of recent languages that have achieved widespread use appear in several places in the book. These show the human side of language development.

Four specific changes distinguish the sixth edition text from its predecessor. First, the material on implementing subprograms has been condensed, largely because the virtual disappearance of Pascal and Modula-2, as well as the shrinking usage of Ada, has made the implementation of nested subprograms with static scoping less important. All of the relevant Pascal examples were rewritten in Ada. Second, Chapter 14 has been expanded to cover both exception handling and event handling. This change was motivated by the great increase in interest and importance of event handling that has come with the wide use of interactive Web documents. Third, the introduction to Smalltalk has been eliminated because we believe the syntactic details of Smalltalk are no longer relevant to the material of the book. Fourth, there are numerous significant changes motivated by the aging of existing programming languages and the emergence of new programming languages. There is now little mention of Modula-2, Pascal, and the ALGOLs. Also, the coverage of Ada and Fortran has been whittled down to the more interesting of their features that do not appear in other popular languages. New material on JavaScript, PHP, and C# has been added where appropriate. Finally, most chapters now include a new section, Programming Exercises.

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 sixth 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 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, axiomatic, and denotational 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, 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 and 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 COMMON LISP, 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 last two chapters in detail. Graduate courses, however, by skipping over parts of the early chapters on imperative languages, will be able to completely discuss the nonimperative languages.

Supplemental Materials

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

- ☞☞ A set of lecture notes slides. These slides are in the form of Microsoft PowerPoint source files, one for each of the chapters of the book.
- ☞☞ PowerPoint slides of all the figures in the book, should you wish to create your own lecture notes.
- ☞☞ A companion web site. With the sixth edition we are introducing a brand-new supplements package for students. 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, the edition will be accompanied by a comprehensive web site with the following content:
 1. Mini manuals (approximately 100-page tutorials) on a handful of languages. These will 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 manuals are planned in C++, C, Java, and C#.
 2. Lab projects. A series of lab projects will be defined for each concept taught in the book. The solutions will be available exclusively to those teaching a course.
 3. Self-grading review exercises. Using the Addison-Wesley software engine, students can complete a series of multiple-choice and fill-in-the-blank exercises to check their understanding of the chapter just read.

Solutions to many of the problem sets are available only to qualified instructors. Please contact your local Addison-Wesley sales representative, or send e-mail to aw.cse@aw.com, for information about how to access them.

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#	http://microsoft.com
Java	http://java.sun.com
Haskell	http://haskell.org
Scheme	http://www.cs.rice.edu/CS/PLT/packages/drscheme/
Perl	http://www.perl.com

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:

Charles Dana, California Polytechnic State University, San Luis Obispo
Eric Joanis, University of Toronto
Donald H. Kraft, Louisiana State University

Dennis L. Mumaugh, DePaul University
Sibylle Schupp, Rensselaer, Polytechnic Institute
Neelam Soundarajan, Ohio State University
Ryan Stansifer, Florida Institute of Technology
Steve Stevenson, Clemson University
Virginia Teller, Hunter College CUNY
Salih Yurttas, Texas A&M University
Yang Wang, Southwest Missouri 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, John Crenshaw, Barbara Ann Griem, Mary Lou Haag, Eileen Head, Ralph C. Hilzer, Hikyoo Koh, Jiang B. Liu, Meiliu Lu, Jon Mauney, Bruce R. Maxim, Robert McCoard, Michael G. Murphy, Andrew Oldroyd, Rebecca Parsons, Jeffery Popyack, Steven Rapkin, Hamilton Richard, Tom Sager, Joseph Schell, Mary Louise Soffa and John M. Weiss.

Maite Suarez-Rivas, Editor, Katherine Harutunian, Project Editor, and Juliet Silveri, Production Supervisor at Addison-Wesley, and Daniel Rausch at Argosy, all deserve my gratitude for their efforts to produce the sixth edition quickly, as well as help make it significantly more complete than the fifth.

Finally, I thank my children, Jake and Darcie, for their patience in enduring my absence from them throughout the endless hours of effort I invested in writing the six editions of this book.

About the Author

Robert Sebesta is an Associate Professor 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 his M.S. and Ph.D. degrees in Computer Science from the Pennsylvania State University. He has taught computer science for over 30 years. His professional interests are the design and evaluation of programming languages, compiler design, and software testing methods and tools. He is a member of the ACM and the IEEE Computer Society.

TABLE of CONTENTS

Chapter 1	Preliminaries	1
1.1	Reasons for Studying Concepts of Programming Languages	2
1.2	Programming Domains	5
1.2.1	Scientific Applications	5
1.2.2	Business Applications	5
1.2.3	Artificial Intelligence	6
1.2.4	Systems Programming	6
1.2.5	Scripting Languages	7
1.2.6	Special Purpose Languages	7
1.3	Language Evaluation Criteria	8
1.3.1	Readability	8
1.3.1.1	Overall Simplicity	9
1.3.1.2	Orthogonality	10
1.3.1.3	Control Statements	12
1.3.1.4	Data Types and Structures	13
1.3.1.5	Syntax Considerations	14
1.3.2	Writability	15
1.3.2.1	Simplicity and Orthogonality	15
1.3.2.2	Support for Abstraction	16
1.3.2.3	Expressivity	16
1.3.3	Reliability	17
1.3.3.1	Type Checking	17
1.3.3.2	Exception Handling	17
1.3.3.3	Aliasing	18
1.3.3.4	Readability and Writability	18
1.3.4	Cost	18
1.4	Influences on Language Design	20
1.4.1	Computer Architecture	20
1.4.2	Programming Methodologies	22
1.5	Language Categories	23
1.6	Language Design Trade-Offs	24
1.7	Implementation Methods	25
1.7.1	Compilation	27

1.7.2	Pure Interpretation	30
1.7.3	Hybrid Interpretation Systems	31
1.8	Programming Environments	32
	Summary	32
	Review Questions	33
	Problem Set	34

Chapter 2 Evolution of the Major Imperative Programming Languages 37

2.1	Zuse's Plankalkül	38
2.1.1	Historical Background	38
2.1.2	Language Overview	40
2.2	Minimal Hardware Programming: Pseudocodes	41
2.2.1	Short Code	42
2.2.2	Speedcoding	42
2.2.3	The UNIVAC "Compiling" System	43
2.2.4	Related Work	43
2.3	The IBM 704 and Fortran	43
2.3.1	Historical Background	44
2.3.2	Design Process	44
2.3.3	Fortran I Overview	45
2.3.4	Fortran II	46
2.3.5	Fortran IV, Fortran 77, 90, and 95	46
2.3.6	Evaluation	48
2.4	Functional Programming: LISP	49
2.4.1	The Beginnings of Artificial Intelligence and List Processing	49
2.4.2	LISP Design Process	50
2.4.3	Language Overview	51
2.4.3.1	Data Structures	51
2.4.3.2	Processes in Functional Programming	52
2.4.3.3	The Syntax of LISP	52
2.4.4	Evaluation	53
2.4.5	Two Descendants of LISP	53
2.4.5.1	Scheme	54
2.4.5.2	COMMON LISP	54
2.4.6	Related Languages	54
2.5	The First Step Toward Sophistication: ALGOL 60	55
2.5.1	Historical Background	55
2.5.2	Early Design Process	56
2.5.3	ALGOL 58 Overview	56
2.5.4	Reception of the ALGOL 58 Report	57
2.5.5	ALGOL 60 Design Process	58
2.5.6	ALGOL 60 Overview	58
2.5.7	ALGOL 60 Evaluation	59
2.6	Computerizing Business Records: COBOL	61
2.6.1	Historical Background	61
2.6.2	FLOW-MATIC	62
2.6.3	COBOL Design Process	62
2.6.4	Evaluation	63
2.7	The Beginnings of Timesharing: BASIC	66
2.7.1	Design Process	66
2.7.2	Language Overview	67

2.7.3	Evaluation		67
2.8	Everything for Everybody: PL/I	70	
2.8.1	Historical Background		70
2.8.2	Design Process		72
2.8.3	Language Overview		72
2.8.4	Evaluation		73
2.9	Two Early Dynamic Languages: APL and SNOBOL		74
2.9.1	Origins and Characteristics of APL		75
2.9.2	Origins and Characteristics of SNOBOL		75
2.10	The Beginnings of Data Abstraction: SIMULA 67		76
2.10.1	Design Process		76
2.10.2	Language Overview		76
2.11	Orthogonal Design: ALGOL 68		77
2.11.1	Design Process		77
2.11.2	Language Overview		77
2.11.3	Evaluation		78
2.12	Some Important Descendants of the ALGOLs	79	
2.12.1	Simplicity by Design: Pascal	79	
2.12.1.1	Historical Background		79
2.12.1.2	Evaluation		80
2.12.2	A Portable Systems Language: C		81
2.12.2.1	Historical Background		82
2.12.2.2	Evaluation		82
2.12.3	Other ALGOL Descendants		83
2.12.3.1	Modula-2		83
2.12.3.2	Oberon		85
2.13	Programming Based on Logic: Prolog	85	
2.13.1	Design Process		85
2.13.2	Language Overview		86
2.13.3	Evaluation		86
2.14	History's Largest Design Effort: Ada		87
2.14.1	Historical Background		87
2.14.2	Design Process		87
2.14.3	Language Overview		89
2.14.4	Evaluation		89
2.14.5	Ada 95		91
2.15	Object-Oriented Programming: Smalltalk		92
2.15.1	Design Process		92
2.15.2	Language Overview		93
2.15.3	Evaluation		94
2.16	Combining Imperative and Object-Oriented Features: C++		96
2.16.1	Design Process		96
2.16.2	Language Overview		97
2.16.3	Evaluation		97
2.16.4	A Related Language: Eiffel		98
2.16.5	Another Related Language: Delphi		98
2.17	An Imperative-Based Object-Oriented Language: Java	99	
2.17.1	Design Process		99
2.17.2	Language Overview		100
2.17.3	Evaluation		100
2.18	Scripting Languages for the Web: JavaScript and PHP	103	
2.18.1	Origins and Characteristics of JavaScript		103
2.18.2	Origins and Characteristics of PHP		105
2.19	A New Language for the New Millenium: C#	106	

2.19.1	Design Process	106
2.19.2	Language Overview	106
2.19.3	Evaluation	107
	Summary	108
	Bibliographic Notes	108
	Review Questions	109
	Problem Set	110
Chapter 3	Describing Syntax and Semantics	113
3.1	Introduction	114
3.2	The General Problem of Describing Syntax	115
3.2.1	Language Recognizers	116
3.2.2	Language Generators	116
3.3	Formal Methods of Describing Syntax	117
3.3.1	Backus-Naur Form and Context-Free Grammars	117
3.3.1.1	Context-Free Grammars	117
3.3.1.2	Origins of Backus-Naur Form	117
3.3.1.3	Fundamentals	118
3.3.1.4	Describing Lists	119
3.3.1.5	Grammars and Derivations	119
3.3.1.6	Parse Trees	121
3.3.1.7	Ambiguity	122
3.3.1.8	Operator Precedence	123
3.3.1.9	Associativity of Operators	125
3.3.1.10	An Unambiguous Grammar for <code>if-then-else</code>	126
3.3.2	Extended BNF	128
3.3.3	Grammars and Recognizers	130
3.4	Attribute Grammars	130
3.4.1	Static Semantics	130
3.4.2	Basic Concepts	131
3.4.3	Attribute Grammars Defined	131
3.4.4	Intrinsic Attributes	132
3.4.5	Example Attribute Grammars	132
3.4.6	Computing Attribute Values	134
3.4.7	Evaluation	136
3.5	Describing the Meanings of Programs: Dynamic Semantics	136
3.5.1	Operational Semantics	137
3.5.1.1	The Basic Process	137
3.5.1.2	Evaluation	139
3.5.2	Axiomatic Semantics	139
3.5.2.1	Assertions	139
3.5.2.2	Weakest Preconditions	140
3.5.2.3	Assignment Statements	141
3.5.2.4	Sequences	143
3.5.2.5	Selection	144
3.5.2.6	Logical Pretest Loops	144
3.5.2.7	Evaluation	150
3.5.3	Denotational Semantics	150
3.5.3.1	Two Simple Examples	150
3.5.3.2	The State of a Program	152
3.5.3.3	Expressions	153
3.5.3.4	Assignment Statements	154

3.5.3.5	Logical Pretest Loops	154
3.5.3.6	Evaluation	154
	Summary	155
	Bibliographic Notes	155
	Review Questions	156
	Problem Set	157

Chapter 4 Lexical and Syntax Analysis 161

4.1	Introduction	162
4.2	Lexical Analysis	163
4.3	The Parsing Problem	167
4.3.1	Introduction to Parsing	167
4.3.2	Top-Down Parsers	168
4.3.3	Bottom-Up Parsers	169
4.3.4	The Complexity of Parsing	170
4.4	Recursive-Descent Parsing	170
4.4.1	The Recursive-Descent Parsing Process	170
4.4.2	The LL Grammar Class	173
4.5	Bottom-Up Parsing	176
4.5.1	The Parsing Problem for Bottom-Up Parsers	176
4.5.2	Shift-Reduce Algorithms	178
4.5.3	LR Parsers	178
	Summary	183
	Review Questions	185
	Problem Set	186
	Programming Exercises	186

Chapter 5 Names, Bindings, Type Checking, and Scopes 189

5.1	Introduction	190
5.2	Names	190
5.2.1	Design Issues	191
5.2.2	Name Forms	191
5.2.3	Special Words	192
5.3	Variables	193
5.3.1	Name	193
5.3.2	Address	194
5.3.2.1	Aliases	194
5.3.3	Type	195
5.3.4	Value	195
5.4	The Concept of Binding	195
5.4.1	Binding of Attributes to Variables	196
5.4.2	Type Bindings	197
5.4.2.1	Variable Declarations	197
5.4.2.2	Dynamic Type Binding	198
5.4.2.3	Type Inference	199
5.4.3	Storage Bindings and Lifetime	202
5.4.3.1	Static Variables	202
5.4.3.2	Stack-dynamic Variables	203
5.4.3.3	Explicit Heap-Dynamic Variables	204
5.4.3.4	Implicit Dynamic Variables	205

5.5	Type Checking	205	
5.6	Strong Typing		206
5.7	Type Compatibility		208
5.8	Scope		211
5.8.1	Static Scope		211
5.8.2	Blocks		213
5.8.3	Evaluation of Static Scoping	214	
5.8.4	Dynamic Scope		217
5.8.5	Evaluation of Dynamic Scoping		218
5.9	Scope and Lifetime		219
5.10	Referencing Environments		220
5.11	Named Constants		222
5.12	Variable Initialization		223
	Summary		224
	Review Questions		225
	Problem Set		226
	Programming Exercises	231	
Chapter 6	Data Types		233
6.1	Introduction		234
6.2	Primitive Data Types		235
6.2.1	Numeric Types		236
6.2.1.1	Integer		236
6.2.1.2	Floating-Point		236
6.2.1.3	Decimal		237
6.2.2	Boolean Types		238
6.2.3	Character Types		238
6.3	Character String Types	239	
6.3.1	Design Issues	239	
6.3.2	Strings and Their Operations	239	
6.3.3	String Length Options		241
6.3.4	Evaluation		241
6.3.5	Implementation of Character String Types		242
6.4	User-Defined Ordinal Types		243
6.4.1	Enumeration Types		243
6.4.1.1	Designs		244
6.4.1.2	Evaluation		245
6.4.2	Subrange Types		246
6.4.2.1	Designs		246
6.4.2.2	Evaluation		247
6.4.3	Implementation of User-Defined Ordinal Types		247
6.5	Array Types		247
6.5.1	Design Issues	248	
6.5.2	Arrays and Indices		248
6.5.3	Subscript Bindings and Array Categories		249
6.5.4	Array Initialization		251
6.5.5	Array Operations		252
6.5.6	Rectangular and Jagged Arrays		254
6.5.7	Slices		254
6.5.8	Evaluation		255
6.5.9	Implementation of Array Types		256
6.6	Associative Arrays		260

6.6.1	Structure and Operations		261
6.6.2	Implementing Associative Arrays		264
6.7	Record Types		264
6.7.1	Definitions of Records		265
6.7.2	References to Record Fields	266	
6.7.3	Operations on Records		266
6.7.4	Evaluation		267
6.7.5	Implementation of Record Types		268
6.8	Union Types		268
6.8.1	Design Issues	269	
6.8.2	Discriminated versus Free Unions		269
6.8.3	Ada Union Types		269
6.8.4	Evaluation		271
6.8.5	Implementation of Union Types		271
6.9	Pointer and Reference Types		271
6.9.1	Design Issues	272	
6.9.2	Pointer Operations		273
6.9.3	Pointer Problems		274
6.9.3.1	Dangling Pointers		274
6.9.3.2	Lost Heap-Dynamic Variables		275
6.9.4	Pointers in Ada		275
6.9.5	Pointers in C and C++		276
6.9.6	Pointers in Fortran 95		277
6.9.7	Reference Types		278
6.9.8	Evaluation		279
6.9.9	Implementation of Pointer and Reference Types		279
6.9.9.1	Representations of Pointers and References	280	
6.9.9.2	Solutions to the Dangling Pointer Problem	280	
6.9.9.3	Heap Management	281	
	Summary		285
	Bibliographic Notes		286
	Review Questions		286
	Problem Set		287
	Programming Exercises	288	
Chapter 7	Expressions and the Assignment Statement		291
7.1	Introduction		292
7.2	Arithmetic Expressions	293	
7.2.1	Operator Evaluation Order		293
7.2.1.1	Precedence		294
7.2.1.2	Associativity		295
7.2.1.3	Parentheses		298
7.2.1.4	Conditional Expressions		298
7.2.2	Operand Evaluation Order		299
7.2.2.1	Side Effects		299
7.3	Overloaded Operators		301
7.4	Type Conversions		303
7.4.1	Coercion in Expressions		303
7.4.2	Explicit Type Conversions		305
7.4.3	Errors in Expressions	305	
7.5	Relational and Boolean Expressions		306

7.5.1	Relational Expressions		306
7.5.2	Boolean Expressions	307	
7.6	Short-Circuit Evaluation		308
7.7	Assignment Statements	310	
7.7.1	Simple Assignments	310	
7.7.2	Conditional Targets		310
7.7.3	Compound Assignment Operators		311
7.7.4	Unary Operator Assignments	311	
7.7.5	Assignment as an Expression	312	
7.8	Mixed-Mode Assignment		313
	Summary		314
	Review Questions		314
	Problem Set		315
	Programming Exercises	317	

Chapter 8 Statement-Level Control Structures 319

8.1	Introduction		320
8.2	Selection Statements		321
8.2.1	Two-Way Selection Statements		321
8.2.1.1	Design Issues		322
8.2.1.2	The Control Expression		322
8.3.1.3	Clause Form		322
8.2.1.4	Nesting Selectors		322
8.2.2	Multiple Selection Constructs	324	
8.2.2.1	Design Issues		325
8.2.2.2	Examples of Multiple Selectors		325
8.2.2.3	Multiple Selection Using if	328	
8.3	Iterative Statements		329
8.3.1	Counter-Controlled Loops		330
8.3.1.1	Design Issues		330
8.3.1.2	The Do Statement of Fortran 95		331
8.3.1.3	The Ada <code>for</code> Statement		332
8.3.1.4	The <code>for</code> Statement of the C-Based Languages		333
8.3.2	Logically Controlled Loops		335
8.3.2.1	Design Issues		335
8.3.2.2	Examples		335
8.3.3	User-Located Loop Control Mechanisms		337
8.3.4	Iteration Based on Data Structures		340
8.4	Unconditional Branching		342
8.4.1	Problems with Unconditional Branching		342
8.5	Guarded Commands		343
8.6	Conclusions		347
	Summary		348
	Review Questions		348
	Problem Set		349
	Programming Exercises	350	

Chapter 9 Subprograms 353

9.1	Introduction		354
-----	--------------	--	-----

9.2	Fundamentals of Subprograms	354
9.2.1	General Subprogram Characteristics	354
9.2.2	Basic Definitions	355
9.2.3	Parameters	356
9.2.4	Procedures and Functions	359
9.3	Design Issues for Subprograms	360
9.4	Local Referencing Environments	361
9.5	Parameter-Passing Methods	363
9.5.1	Semantics Models of Parameter Passing	363
9.5.2	Implementation Models of Parameter Passing	364
9.5.2.1	Pass-by-Value	364
9.5.2.2	Pass-by-Result	364
9.5.2.3	Pass-by-Value-Result	365
9.5.2.4	Pass-by-Reference	365
9.5.2.5	Pass-by-Name	367
9.5.3	Parameter-Passing Methods of the Major Languages	368
9.5.4	Type-Checking Parameters	371
9.5.5	Implementing Parameter-Passing Methods	373
9.5.6	Multidimensional Arrays as Parameters	374
9.5.7	Design Considerations	378
9.5.8	Examples of Parameter Passing	378
9.6	Parameters That Are Subprogram Names	381
9.7	Overloaded Subprograms	383
9.8	Generic Subprograms	384
9.8.1	Generic Subprograms in Ada	385
9.8.2	Generic Functions in C++	387
9.8.3	Generic Subprograms in Other Languages	388
9.9	Design Issues for Functions	389
9.9.1	Functional Side Effects	389
9.9.2	Types of Return Values	389
9.10	User-Defined Overloaded Operators	389
9.11	Coroutines	390
	Summary	393
	Review Questions	393
	Problem Set	394
	Programming Exercises	395
Chapter 10	Implementing Subprograms	397
10.1	The General Semantics of Calls and Returns	398
10.2	Implementing “Simple” Subprograms	399
10.3	Implementing Subprograms with Stack-Dynamic Local Variables	401
10.3.1	More Complex Activation Records	401
10.3.2	An Example without Recursion	403
10.3.3	Recursion	406
10.4	Nested Subprograms	407
10.4.1	The Basics	409
10.4.2	Static Chains	409
10.4.2	Displays	416
10.5	Blocks	417
10.6	Implementing Dynamic Scoping	419
10.6.1	Deep Access	419
10.6.2	Shallow Access	420

Summary		422
Review Questions		423
Problem Set		423
Chapter 11	Data Abstraction	427
11.1	The Concept of Abstraction	428
11.2	Introduction to Data Abstraction	429
11.2.1	Floating-Point as an Abstract Data Type	429
11.2.2	User-Defined Abstract Data Types	430
11.2.3	An Example	431
11.4	Design Issues for Abstract Data Types	432
11.4	Language Examples	433
11.4.1	Abstract Data Types in Ada	433
11.4.1.1	Encapsulation	433
11.4.1.2	Information Hiding	433
11.4.1.3	An Example	437
11.4.2	Abstract Data Types in C++	439
11.4.2.1	Encapsulation	439
11.4.2.2	Information Hiding	439
11.4.2.3	An Example	440
11.4.2.4	Evaluation	442
11.4.3	Abstract Data Types in Java	442
11.4.4	Abstract Data Types in C#	443
11.5	Parameterized Abstract Data Types	445
11.5.1	Ada	445
11.5.2	C++	446
11.6	Encapsulation Construct	448
11.6.1	Introduction	448
11.6.2	Nested Subprograms	448
11.6.3	Encapsulation in C	448
11.6.4	Encapsulation in C++	449
11.6.5	Ada Packages	450
11.6.6	C# Assemblies	450
11.7	Naming Encapsulations	451
11.7.1	C++ Namespaces	451
11.7.2	Java Packages	452
11.7.3	Ada Packages	453
Summary		453
Review Questions		454
Problem Set		455
Programming Exercises		456
Chapter 12	Support for Object-Oriented Programming	457
12.1	Introduction	458
12.2	Object-Oriented Programming	458
12.2.1	Introduction	458
12.2.2	Inheritance	459
12.2.3	Polymorphism and Dynamic Binding	460
12.3	Design Issues for Object-Oriented Language	461
12.3.1	The Exclusivity of Objects	461

12.3.2	Are Subclasses Subtypes?	462
12.3.3	Type Checking and Polymorphism	463
12.3.4	Single and Multiple Inheritance	463
12.3.5	Allocation and Deallocation of Objects	464
12.3.6	Dynamic and Static Binding	465
12.4	Support for Object-Oriented Programming in Smalltalk	465
12.4.1	General Characteristics	465
12.4.2	Type Checking and Polymorphism	466
12.4.3	Inheritance	466
12.4.4	Evaluation of Smalltalk	467
12.5	Support for Object-Oriented Programming in C++	467
12.5.1	General Characteristics	469
12.5.2	Inheritance	470
12.5.3	Dynamic Binding	474
12.5.4	Evaluation	476
12.6	Support for Object-Oriented Programming in Java	477
12.6.1	General Characteristics	477
12.6.2	Inheritance	478
12.6.3	Dynamic Binding	479
12.6.4	Evaluation	479
12.7	Support for Object-Oriented Programming in C#	479
12.7.1	General Characteristics	479
12.7.2	Inheritance	480
12.7.3	Dynamic Binding	480
12.7.4	Evaluation	481
12.8	Support for Object-Oriented Programming in Ada 95	481
12.8.1	General Characteristics	481
12.8.2	Inheritance	482
12.8.3	Dynamic Binding	483
12.8.4	Evaluation	484
12.9	The Object Model of JavaScript	485
12.9.1	General Characteristics	485
12.9.2	JavaScript Objects	486
12.9.3	Object Creation and Modification	486
12.9.4	Evaluation	488
12.10	Implementation of Object-Oriented Constructs	488
12.10.1	Instance Data Storage	488
12.10.2	Dynamic Binding of Method Calls to Methods	488
	Summary	491
	Review Questions	492
	Problem Set	493
	Programming Exercises	494
Chapter 13	Concurrency	495
13.1	Introduction	496
13.1.1	Multiprocessor Architectures	497
13.1.2	Categories of Concurrency	498
13.1.3	Motivations for Studying Concurrency	499
13.2	Introduction to Subprogram-Level Concurrency	499
13.2.1	Fundamental Concepts	499
13.2.2	Language Design for Concurrency	503
13.2.3	Design Issues	503

13.3	Semaphores		503
13.3.1	Introduction		503
13.3.2	Cooperation Synchronization	504	
13.3.3	Competition Synchronization	506	
13.3.4	Evaluation		508
13.4	Monitors		508
13.4.1	Introduction		508
13.4.2	Competition Synchronization	510	
13.4.3	Cooperation Synchronization	510	
13.4.4	Evaluation		513
13.5	Message Passing		513
13.5.1	Introduction		513
13.5.2	The Concept of Synchronous Message Passing		513
13.5.3	The Ada Synchronous Message-Passing Model		514
13.5.4	Cooperation Synchronization	519	
13.5.5	Competition Synchronization	520	
13.5.6	Task Termination		521
13.5.7	Priorities		521
13.5.8	Binary Semaphores		522
13.5.9	Protected Objects		522
13.5.10	Asynchronous Message Passing		524
13.5.11	Evaluation		524
13.6	Java Threads		525
13.6.1	The <code>Thread</code> Class		525
13.6.2	Priorities		527
13.6.3	Competition Synchronization	528	
13.6.4	Cooperation Synchronization	528	
13.6.5	Evaluation		532
13.7	C# Threads		532
13.7.1	Basic Thread Operations		532
13.7.2	Synchronizing Threads		533
13.7.3	Evaluation		534
13.8	Statement-Level Concurrency		534
13.8.1	High-Performance Fortran		534
	Summary		536
	Bibliographic Notes		537
	Review Questions		538
	Problem Set		539
	Programming Exercises	540	
Chapter 14	Exception Handling and Event Handling		541
14.1	Introduction to Exception Handling		542
14.1.1	Basic Concepts		543
14.1.2	Design Issues	545	
14.2	Exception Handling in Ada		548
14.2.1	Exception Handlers		548
14.2.2	Binding Exceptions to Handlers		549
14.2.3	Continuation		550
14.2.4	Other Design Choices		551
14.2.5	An Example		552
14.2.6	Evaluation		554
14.3	Exception Handling in C++		554

14.3.1	Exception Handlers	554
14.3.2	Binding Exceptions to Handlers	555
14.3.3	Continuation	556
14.3.4	Other Design Choices	556
14.3.5	An Example	556
14.3.6	Evaluation	558
14.4	Exception Handling in Java	558
14.4.1	Classes of Exceptions	558
14.4.2	Exception Handlers	559
14.4.3	Binding Exceptions to Handlers	559
14.4.4	Other Design Choices	562
14.4.5	An Example	563
14.4.6	The <code>finally</code> Clause	565
14.4.7	Evaluation	566
14.5	Introduction to Event Handling	566
14.5.1	Basic Concepts of Event Handling	567
14.6	Event Handling with Java	567
14.6.1	Java Swing GUI Components	568
14.6.2	The Java Event Model	570
	Summary	574
	Bibliographic Notes	575
	Review Questions	575
	Problem Set	576
	Programming Exercises	578

Chapter 15 Functional Programming Languages 579

15.1	Introduction	580
15.2	Mathematical Functions	581
15.2.1	Simple Functions	581
15.2.2	Functional Forms	582
15.3	Fundamentals of Functional Programming Languages	583
15.4	The First Functional Programming Language: LISP	584
15.4.1	Data Types and Structures	584
15.4.2	The First LISP Interpreter	586
15.5	An Introduction to Scheme	587
15.5.1	Origins of Scheme	587
15.5.2	The Scheme Interpreter	588
15.5.3	Primitive Numeric Functions	588
15.5.4	Defining Functions	589
15.5.5	Output Functions	590
15.5.6	Numeric Predicate Functions	591
15.5.7	Control Flow	591
15.5.8	List Functions	593
15.5.9	Predicate Functions for Symbolic Atoms and Lists	596
15.5.10	Example Scheme Functions	597
15.5.11	Functional Forms	601
15.5.11.1	Functional Composition	601
15.5.11.2	An Apply-to-All Functional Form	602
15.5.12	Functions That Build Code	602
15.6	COMMON LISP	603
15.7	ML	604
15.8	Haskell	607

15.9	Applications of Functional Languages	611
15.10	A Comparison of Functional and Imperative Languages	612
	Summary	612
	Bibliographic Notes	613
	Review Questions	614
	Problem Set	614
	Programming Exercises	615
Chapter 16 Logic Programming Languages		617
16.1	Introduction	618
16.2	A Brief Introduction to Predicate Calculus	618
16.2.1	Propositions	619
16.2.2	Clausal Form	621
16.3	Predicate Calculus and Proving Theorems	621
16.4	An Overview of Logic Programming	624
16.5	The Origins of Prolog	625
16.6	The Basic Elements of Prolog	626
16.6.1	Terms	626
16.6.2	Fact Statements	627
16.6.3	Rule Statements	628
16.6.4	Goal Statements	629
16.6.5	The Inferencing Process in Prolog	629
16.6.6	Simple Arithmetic	632
16.6.7	List Structures	635
16.7	Deficiencies of Prolog	640
16.7.1	Resolution Order Control	640
16.7.2	The Closed-World Assumption	643
16.7.3	The Negation Problem	643
16.7.4	Intrinsic Limitations	645
16.8	Applications of Logic Programming	646
16.8.1	Relational Database Management Systems	646
16.8.2	Expert Systems	646
16.8.3	Natural Language Processing	647
	Summary	648
	Bibliographic Notes	648
	Review Questions	649
	Problem Set	649
	Programming Exercises	650

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.
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`. (I 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.

Chapter 2

Problem Set:

6. Because of the simple syntax of LISP, few syntax errors occur in LISP programs. Unmatched parentheses is the most common mistake.

7. The main reason why imperative features were put in LISP was to increase its execution efficiency.

10. The main motivation for the development of PL/I was to provide a single tool for computer centers that must support both scientific and commercial applications. IBM believed that the needs of the two classes of applications were merging, at least to some degree. They felt that the simplest solution for a provider of systems, both hardware and software, was to furnish a single hardware system running a single programming language that served both scientific and commercial applications.

11. IBM was, for the most part, incorrect in its view of the future of the uses of computers, at least as far as languages are concerned. Commercial applications are nearly all done in languages that are specifically designed for them. Likewise for scientific applications. On the other hand, the IBM design of the 360 line of computers was a great success--it still dominates the area of computers between supercomputers and minicomputers. Furthermore, 360 series computers and their descendants have been widely used for both scientific and commercial applications. These applications have been done, in large part, in FORTRAN and COBOL.

14. The argument for typeless languages is their great flexibility for the programmer. Literally any storage location can be used to store any type value. This is useful for very low-level languages used for systems programming. The drawback is that type checking is impossible, so that it is entirely the programmer's responsibility to insure that expressions and assignments are correct.

18. A good deal of restraint must be used in revising programming languages. The greatest danger is that the revision process will continually add new features, so that the language grows more and more complex. Compounding the problem is the reluctance, because of existing software, to remove obsolete features.

Chapter 3

Instructor's Note:

In the program proof on page 149, there is a statement that may not be clear to all, specifically, $(n + 1) * \dots * n = 1$. The justification of this statement is as follows:

Consider the following expression:

$$(count + 1) * (count + 2) * \dots * n$$

The former expression states that when `count` is equal to `n`, the value of the later expression is 1. Multiply the later expression by the quotient:

$$(1 * 2 * \dots * count) / (1 * 2 * \dots * count)$$

whose value is 1, to get

$$\frac{(1 * 2 * \dots * count * (count + 1) * (count + 2) * \dots * n)}{(1 * 2 * \dots * count)}$$

The numerator of this expressions is `n!`. The denominator is `count!`. If `count` is equal to `n`, the value of the quotient is

$$n! / n!$$

or 1, which is what we were trying to show.

Problem Set:

2a. `<class_head> ? {<modifier>} class <id> [extends class_name]`

`[implements <interface_name> {, <interface_name>}]`

`<modifier> ? public | abstract | final`

2c. `<switch_stmt> ? switch (<expr>) {case <literal> : <stmt_list>`

`{case <literal> : <stmt_list> } [default : <stmt_list>] }`

3. `<assign> ? <id> = <expr>`

`<id> ? A | B | C`

`<expr> ? <expr> * <term>`

`| <term>`

`<term> ? <factor> + <term>`

`| <factor>`

`<factor> ? (<expr>)`

| <id>

6.

(a) <assign> => <id> = <expr>

=> A = <expr>

=> A = <id> * <expr>

=> A = A * <expr>

=> A = A * (<expr>)

=> A = A * (<id> + <expr>)

=> A = A * (B + <expr>)

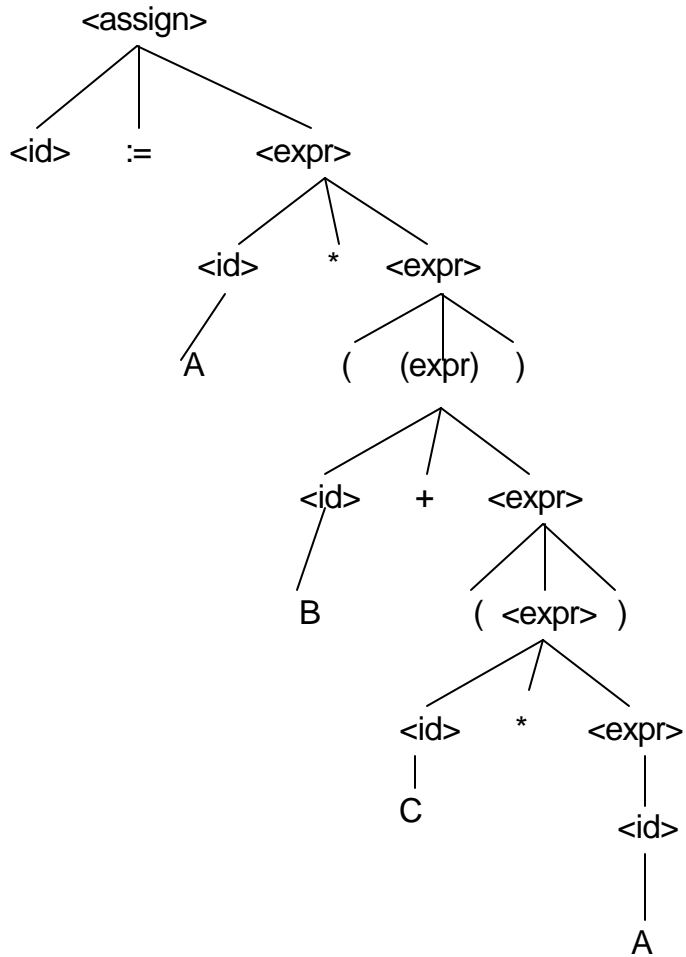
=> A = A * (B + (<expr>))

=> A = A * (B + (<id> * <expr>))

=> A = A * (B + (C * <expr>))

=> A = A * (B + (C * <id>))

=> A = A * (B + (C * A))



7.

(a) $\langle \text{assign} \rangle \Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\Rightarrow A = \langle \text{expr} \rangle$

$\Rightarrow A = \langle \text{term} \rangle$

$\Rightarrow A = \langle \text{factor} \rangle * \langle \text{term} \rangle$

$\Rightarrow A = (\langle \text{expr} \rangle) * \langle \text{term} \rangle$

$\Rightarrow A = (\langle \text{expr} \rangle + \langle \text{term} \rangle) * \langle \text{term} \rangle$

$\Rightarrow A = (\langle \text{term} \rangle + \langle \text{term} \rangle) * \langle \text{term} \rangle$

$\Rightarrow A = (\langle \text{factor} \rangle + \langle \text{term} \rangle) * \langle \text{term} \rangle$

$\Rightarrow A = (\langle id \rangle + \langle term \rangle) * \langle term \rangle$

$\Rightarrow A = (A + \langle term \rangle) * \langle term \rangle$

$\Rightarrow A = (A + \langle factor \rangle) * \langle term \rangle$

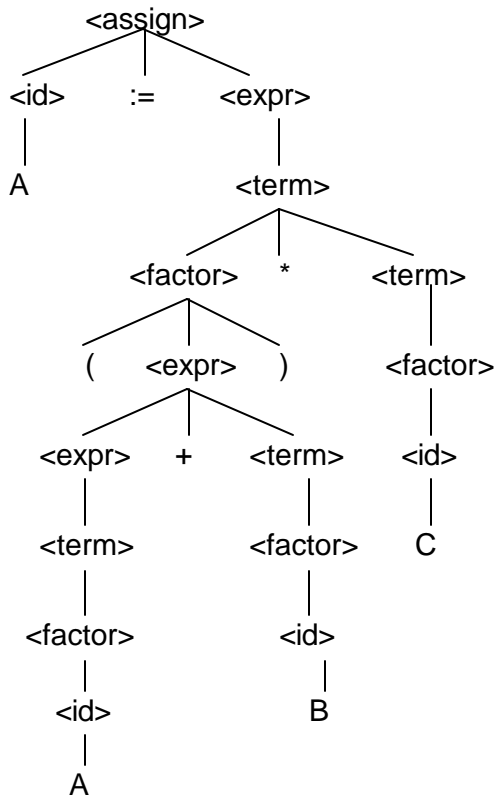
$\Rightarrow A = (A + \langle id \rangle) * \langle term \rangle$

$\Rightarrow A = (A + B) * \langle term \rangle$

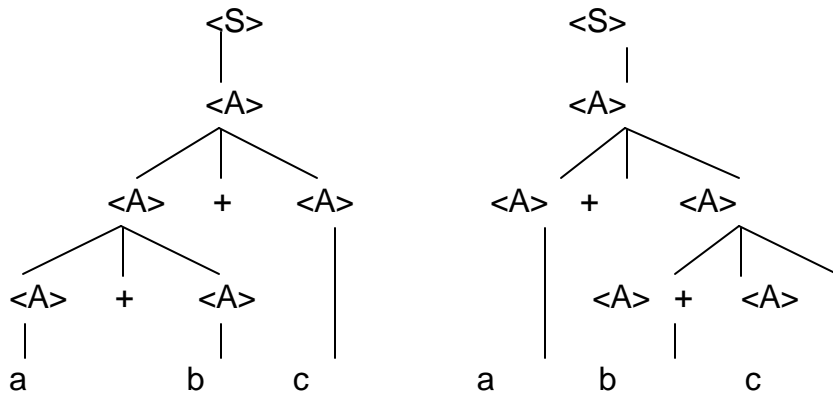
$\Rightarrow A = (A + B) * \langle factor \rangle$

$\Rightarrow A = (A + B) * \langle id \rangle$

$\Rightarrow A = (A + B) * C$



8. The following two distinct parse tree for the same string prove that the grammar is ambiguous.



9. Assume that the unary operators can precede any operand. Replace the rule

$\langle \text{factor} \rangle \rightarrow \langle \text{id} \rangle$

with

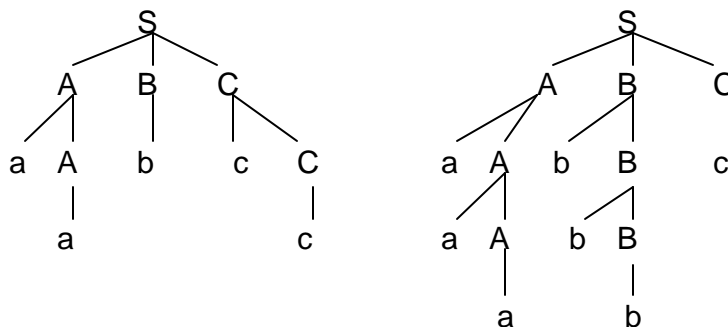
$\langle \text{factor} \rangle \rightarrow + \langle \text{id} \rangle$

$\quad \quad \quad | - \langle \text{id} \rangle$

10. One or more a's followed by one or more b's followed by one or more c's.

13. $S \rightarrow a S b \mid a b$

14.



16. $\langle \text{assign} \rangle ? \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\langle \text{id} \rangle ? A | B | C$

$\langle \text{expr} \rangle ? \langle \text{expr} \rangle (+ | -) \langle \text{expr} \rangle$

$| (\langle \text{expr} \rangle)$

$| \langle \text{id} \rangle$

18.

(a) (Pascal **repeat**) We assume that the logic expression is a single relational expression.

```
loop: ...  
      ...  
      if <relational_expression> goto out  
      goto loop  
out:  ...
```

(b) (Ada **for**) **for** I **in** first .. last **loop**

```
      I = first  
loop:  if I < last goto out  
      ...  
      I = I + 1  
      goto loop  
out:...
```

(c) (Fortran **Do**)

```
      K = start  
loop:  if K > end goto out
```

```
...
K = K + step
goto loop
out: ...
```

(e) (C **for**) **for** (expr1; expr2; expr3) ...

```
    evaluate(expr1)
loop: control = evaluate(expr2)
    if control == 0 goto out
    ...
    evaluate(expr3)
    goto loop
out: ...
```

19.

(a) $a = 2 * (b - 1) - 1 \{a > 0\}$

$$2 * (b - 1) - 1 > 0$$

$$2 * b - 2 - 1 > 0$$

$$2 * b > 3$$

$$b > 3 / 2$$

(b) $b = (c + 10) / 3 \{b > 6\}$

$$(c + 10) / 3 > 6$$

$$c + 10 > 18$$

$$c > 8$$

(c) $a = a + 2 * b - 1 \{a > 1\}$

$$a + 2 * b - 1 > 1$$

$$2 * b > 2 - a$$

$$b > 1 - a / 2$$

(d) $x = 2 * y + x - 1 \{x > 11\}$

$$2 * y + x - 1 > 11$$

$$2 * y + x > 12$$

20.

(a) $a = 2 * b + 1$

$$b = a - 3 \{b < 0\}$$

$$a - 3 < 0$$

$$a < 3$$

Now, we have:

$$a = 2 * b + 1 \{a < 3\}$$

$$2 * b + 1 < 3$$

$$2 * b + 1 < 3$$

$$2 * b < 2$$

$$b < 1$$

(b) $a = 3 * (2 * b + a);$

$$b = 2 * a - 1 \{b > 5\}$$

$$2 * a - 1 > 5$$

$$2 * a > 6$$

$$a > 3$$

Now we have:

$$a = 3 * (2 * b + a) \{a > 3\}$$

$$3 * (2 * b + a) > 3$$

$$6 * b + 3 * a > 3$$

$$2 * b + a > 1$$

$$n > (1 - a) / 2$$

21a. $M_{pf}(\text{for var in init_expr .. final_expr loop L end loop, s) \triangleq$

if $\text{VARMAP}(i, s) = \mathbf{undef}$ for var or some i in init_expr or final_expr

then **error**

else if $M_e(\text{init_expr}, s) > M_e(\text{final_expr}, s)$

then s

else $M_l(\text{while init_expr - 1} \leq \text{final_expr do L, } M_a(\text{var} := \text{init_expr} + 1, s))$

21b. $M_r(\text{repeat L until B}) \triangleq$

if $M_b(B, s) = \mathbf{undef}$

then **error**

else if $M_{sl}(L, s) = \mathbf{error}$

then **error**

else if $M_b(B, s) = \mathbf{true}$

then $M_{sl}(L, s)$

else $M_r(\text{repeat L until B}, M_{sl}(L, s))$

21c. $M_b(B, s) \triangleq$ if $\text{VARMAP}(i, s) = \mathbf{undef}$ for some i in B

then **error**
 else B', where B' is the result of
 evaluating B after setting each
 variable i in B to VARMAP(i, s)

21d. $M_{ct}(\text{for}(\text{expr1}; \text{expr2}; \text{expr3}) L, s) \triangleq$

if VARMAP (i, s) = **undef** for some i in expr1, expr2, expr3, or L

then **error**

else if $M_e(\text{expr2}, M_e(\text{expr1}, s)) = 0$

then s

else $M_{\text{help}}(\text{expr2}, \text{expr3}, L, s)$

$M_{\text{help}}(\text{expr2}, \text{expr3}, L, s) \triangleq$

if VARMAP (i, s) = **undef** for some i in expr2, expr3, or L

then **error**

else

if $M_{sl}(L, s) = \text{error}$

then s

else $M_{\text{help}}(\text{expr2}, \text{expr3}, L, M_{sl}(L, M_e(\text{expr3}, s)))$

22. The value of an intrinsic attribute is supplied from outside the attribute evaluation process, usually from the lexical analyzer. A value of a synthesized attribute is computed by an attribute evaluation function.

23. Replace the second semantic rule with:

$\langle \text{var} \rangle[2].\text{env} ? \langle \text{expr} \rangle.\text{env}$

$\langle \text{var} \rangle[3].\text{env} ? \langle \text{expr} \rangle.\text{env}$

$\langle \text{expr} \rangle.\text{actual_type} ? \langle \text{var} \rangle[2].\text{actual_type}$

predicate: $\langle \text{var} \rangle[2].\text{actual_type} = \langle \text{var} \rangle[3].\text{actual_type}$

Chapter 4

Problem Set:

1.

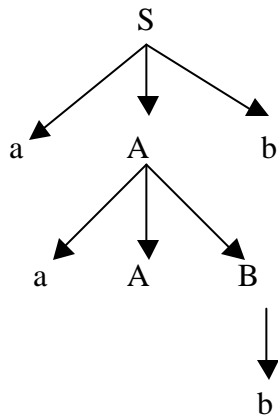
(a) $\text{FIRST}(aB) = \{a\}$, $\text{FIRST}(b) = \{b\}$, $\text{FIRST}(cBB) = \{c\}$, Passes the test

(b) $\text{FIRST}(aB) = \{a\}$, $\text{FIRST}(bA) = \{b\}$, $\text{FIRST}(aBb) = \{a\}$, Fails the test

(c) $\text{FIRST}(aaA) = \{a\}$, $\text{FIRST}(b) = \{b\}$, $\text{FIRST}(caB) = \{c\}$, Passes the test

3.

(a) aaAbb

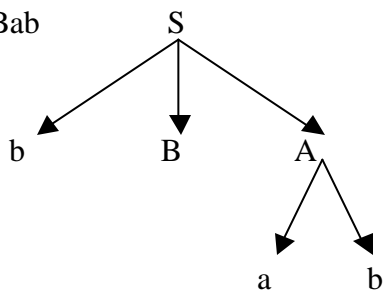


Phrases: aaAbb, aAb, b

Simple phrases: b

Handle: b

(b) bBab



Phrases: bBab, ab

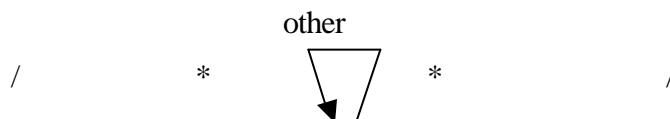
Simple phrases: ab

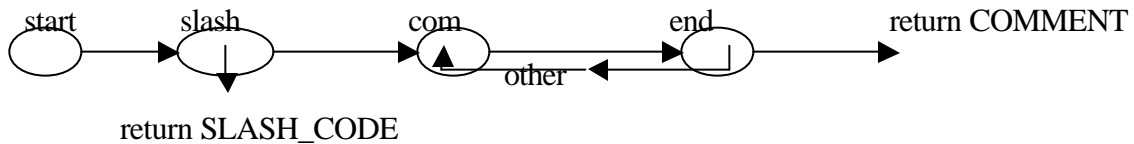
Handle: ab

5. Stack	Input	Action
0	id * (id + id) \$	Shift 5
0id5	* (id + id) \$	Reduce 6 (Use GOTO[0, F])
0F3	* (id + id) \$	Reduce 4 (Use GOTO[0, T])
0T2	* (id + id) \$	Reduce 2 (Use GOTO[0, E])
0T2*7	(id + id) \$	Shift 7
0T2*7(4	id + id) \$	Shift 4
0T2*7(4id5 + id) \$		Shift 5
0T2*7(4F3 + id) \$		Reduce 6 (Use GOTO[4, F])
0T2*7(4T2 + id) \$		Reduce 4 (Use GOTO[4, T])
0T2*7(4E8 + id) \$		Reduce 2 (Use GOTO[4, E])
0T2*7(4E8+6 id) \$		Shift 6
0T2*7(4E8+6id5) \$		Shift 5
0T2*7(4E8+6F3) \$		Reduce 6 (Use GOTO[6, F])
0T2*7(4E8+6T9) \$		Reduce 4 (Use GOTO[6, T])
0T2*7(4E8) \$		Reduce 1 (Use GOTO[4, E])
0T2*7(4E8)11 \$		Shift 11
0T2*7F10 \$		Reduce 5 (Use GOTO[7, F])
0T2 \$		Reduce 5 (Use GOTO[0, T])
0E1 \$		Reduce 2 (Use GOTO[0, E])
--ACCEPT--		

Programming Exercises:

1. Every arc in this graph is assumed to have `addChar` attached. Assume we get here only if `charClass` is `SLASH`.





```

3. int getComment() {
    getChar();

    /* The slash state */
    if (charClass != AST)
        return SLASH_CODE;
    else {
        /* The com state-end state loop */
        do {
            getChar();

            /* The com state loop */
            while (charClass != AST)
                getChar();
        } while (charClass != SLASH);
        return COMMENT;
    }
}

```

Chapter 5

Problem Set:

2. The advantage of a typeless language is flexibility; any variable can be used for any type values. The disadvantage is poor reliability due to the ease with which type errors can be made, coupled with the impossibility of type checking detecting them.

3. This is a good idea. It adds immensely to the readability of programs. Furthermore, aliasing can be minimized by enforcing programming standards that disallow access to the array in any executable statements. The alternative to this aliasing would be to pass many parameters, which is a highly inefficient process.

5. Implicit heap-dynamic variables acquire types only when assigned values, which must be at runtime. Therefore, these variables are always dynamically bound to types.

6. Suppose that a Fortran subroutine is used to implement a data structure as an abstraction. In this situation, it is essential that the structure persist between calls to the managing subroutine.

8.

(a) i. Sub1

ii. Sub1

iii. Main

(b) i. Sub1

ii. Sub1

iii. Sub1

9. Static scoping: $x = 5$.

Dynamic scoping: $x = 10$

10. Variable Where Declared

In Sub1:

A Sub1

Y Sub1

Z Sub1

X Main

In Sub2:

A Sub2

B Sub2

Z Sub2

Y Sub1

X Main

In Sub3:

A Sub3

X Sub3

W	Sub3
Y	Main
Z	Main

12. Point 1:

a	1
b	2
c	2
d	2

Point 2:

a	1
b	2
c	3
d	3
e	3

Point 3: same as Point 1

Point 4:

a	1
b	1
c	1

13. Variable Where Declared

(a)

d, e, f	fun3
c	fun2
b	fun1
a	main

(b)

d, e, f	fun3
b, c	fun1
a	main

(c)

b, c, d	fun1
e, f	fun3
a	main

(d)

b, c, d	fun1
e, f	fun3
a	main

(e)

c, d, e	fun2
f	fun3

	b	fun1
	a	main
(f)	b, c, d	fun1
	e	fun2
	f	fun3
	a	main

14. Variable Where Declared

(a)	A, X, W	Sub3
	B, Z	Sub2
	Y	Sub1
(b)	A, X, W	Sub3
	Y, Z	Sub1
(c)	A, Y, Z	Sub1
	X, W	Sub3
	B	Sub2
(d)	A, Y, Z	Sub1
	X, W	Sub3
(e)	A, B, Z	Sub2
	X, W	Sub3
	Y	Sub1
(f)	A, Y, Z	Sub1
	B	Sub2
	X, W	Sub3

Chapter 6

Problem Set:

1. Boolean variables stored as single bits are very space efficient, but on most computers access to them is slower than if they were stored as bytes.
2. Integer values stored in decimal waste storage in binary memory computers, simply as a result of the fact that it takes four binary bits to store a single decimal digit, but those four bits are capable of storing 16 different values. Therefore, the ability to store six out of every 16 possible values is wasted. Numeric values can be stored efficiently on binary memory

computers only in number bases that are multiples of 2. If humans had developed a number of fingers that was a power of 2, these kinds of problems would not occur.

6. When implicit dereferencing of pointers occurs only in certain contexts, it makes the language slightly less orthogonal. The context of the reference to the pointer determines its meaning. This detracts from the readability of the language and makes it slightly more difficult to learn.

7. The only justification for the \rightarrow operator in C and C++ is writability. It is slightly easier to write $p \rightarrow q$ than $(*p).q$.

8. The advantage of having a separate construct for unions is that it clearly shows that unions are different from records. The disadvantages are that it requires an additional reserved word and that unions are often separately defined but included in records, thereby complicating the program that uses them.

9. Let the subscript ranges of the three dimensions be named $\text{min}(1)$, $\text{min}(2)$, $\text{min}(3)$, $\text{max}(1)$, $\text{max}(2)$, and $\text{max}(3)$. Let the sizes of the subscript ranges be $\text{size}(1)$, $\text{size}(2)$, and $\text{size}(3)$. Assume the element size is 1.

Row Major Order:

$$\begin{aligned} \text{location}(a[i, j, k]) = & (\text{address of } a[\text{min}(1), \text{min}(2), \text{min}(3)]) \\ & + ((i - \text{min}(1)) * \text{size}(3) + (j - \text{min}(2))) * \text{size}(2) + (k - \text{min}(3)) \end{aligned}$$

Column Major Order:

$$\begin{aligned} \text{location}(a[i, j, k]) = & (\text{address of } a[\text{min}(1), \text{min}(2), \text{min}(3)]) \\ & + ((k - \text{min}(3)) * \text{size}(1) + (j - \text{min}(2))) * \text{size}(2) + (i - \text{min}(1)) \end{aligned}$$

10. The advantage of this scheme is that accesses that are done in order of the rows can be made very fast; once the pointer to a row is gotten, all of the elements of the row can be fetched very quickly. If, however, the elements of a matrix must be accessed in column order, these accesses will be much slower; every access requires the fetch of a row pointer and an address computation from there. Note that this access technique was devised to allow multidimensional array rows to be segments in a virtual storage management technique. Using this method, multidimensional arrays could be stored and manipulated that are much larger than the physical memory of the computer.

14. Implicit heap storage recovery eliminates the creation of dangling pointers through explicit deallocation operations, such as `delete`. The disadvantage of implicit heap storage recovery is the execution time cost of doing the recovery, often when it is not even necessary (there is no shortage of heap storage).

Chapter 7

Problem Set:

1. Suppose `Type1` is a subrange of `Integer`. It may be useful for the difference between `Type1` and `Integer` to be ignored by the compiler in an expression.

7. An expression such as `a + fun(b)`, as described on page 300.

8. Consider the integer expression `A + B + C`. Suppose the values of `A`, `B`, and `C` are 20,000, 25,000, and -20,000, respectively. Further suppose that the machine has a maximum integer value of 32,767. If the first addition is computed first, it will result in overflow. If the second addition is done first, the whole expression can be correctly computed.

9.

(a) $((a * b)^1 - 1)^2 + c)^3$

(b) $((a * (b - 1)^1)^2 / c)^3 \text{ mod } d)^4$

(c) $((a - b)^1 / c)^2 \& ((d * e)^3 / a)^4 - 3)^5)^6$

(d) $((-a)^1 \text{ or } (c = d)^2)^3 \text{ and } e)^4$

(e) $((a > b)^1 \text{ xor } (c \text{ or } (d \leq 17)^2)^3)^4$

(f) $(-(a + b)^1)^2$

10.

(a) $(a * (b - (1 + c)^1)^2)^3$

(b) $(a * ((b - 1)^2 / (c \text{ mod } d)^1)^3)^4$

(c) $((a - b)^5 / (c \& (d * (e / (a - 3)^1)^2)^3)^4)^6$

(d) $(-(a \text{ or } (c = (d \text{ and } e)^1)^2)^3)^4$

(e) $(a > (\text{ xor } (c \text{ or } (d \leq 17)^1)^2)^3)^4$

(f) $(-(a + b)^1)^2$

11. `<expr> ? <expr> or <e1> | <expr> xor <e1> | <e1>`

`<e1> ? <e1> and <e2> | <e2>`

`<e2> ? <e2> = <e3> | <e2> /= <e3> | <e2> < <e3>`

`| <e2> <= <e3> | <e2> > <e3> | <e2> >= <e3> | <e3>`

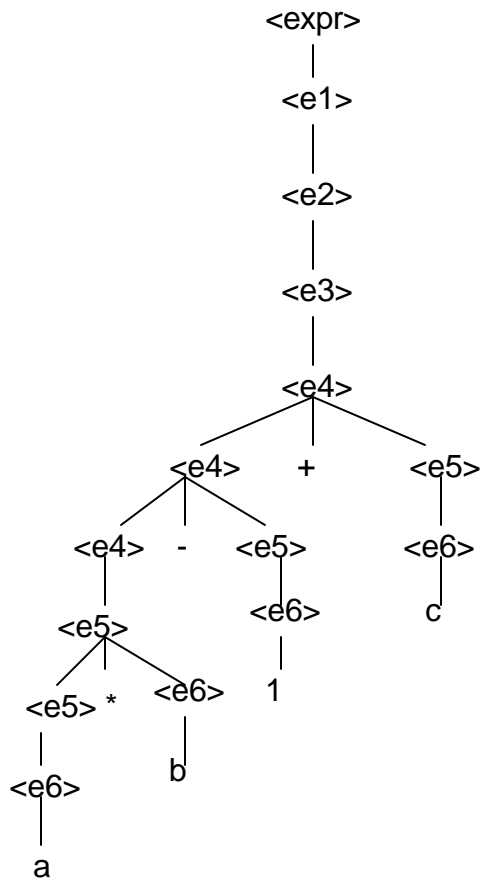
`<e3> ? <e4>`

`<e4> ? <e4> + <e5> | <e4> - <e5> | <e4> & <e5> | <e4> mod <e5> | <e5>`

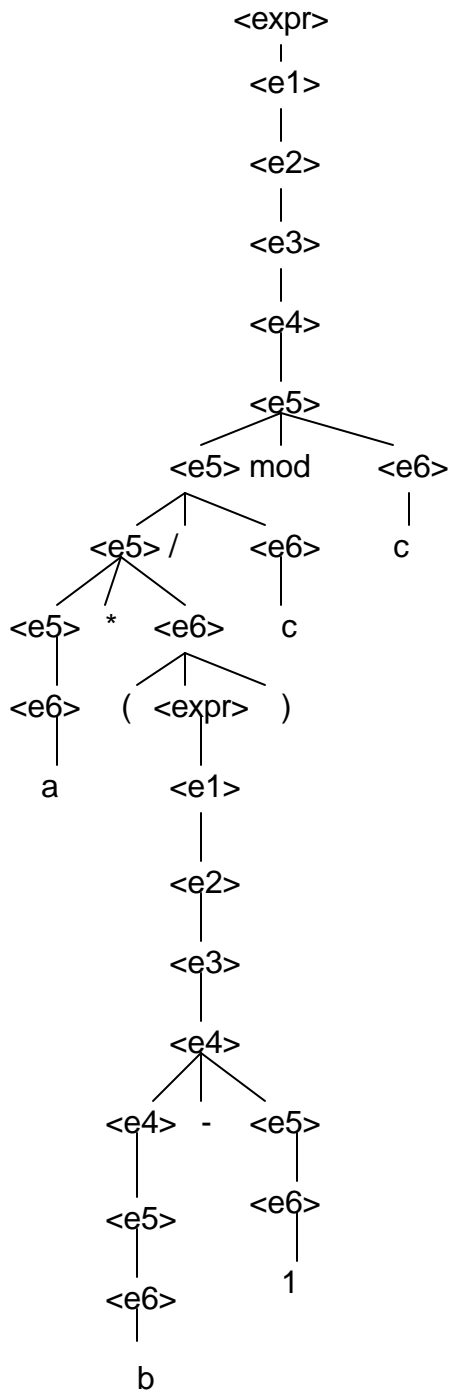
$\langle e5 \rangle ? \langle e5 \rangle * \langle e6 \rangle \mid \langle e5 \rangle / \langle e6 \rangle \mid \text{not } \langle e5 \rangle \mid \langle e6 \rangle$

$\langle e6 \rangle ? a \mid b \mid c \mid d \mid e \mid \text{const} \mid (\langle \text{expr} \rangle)$

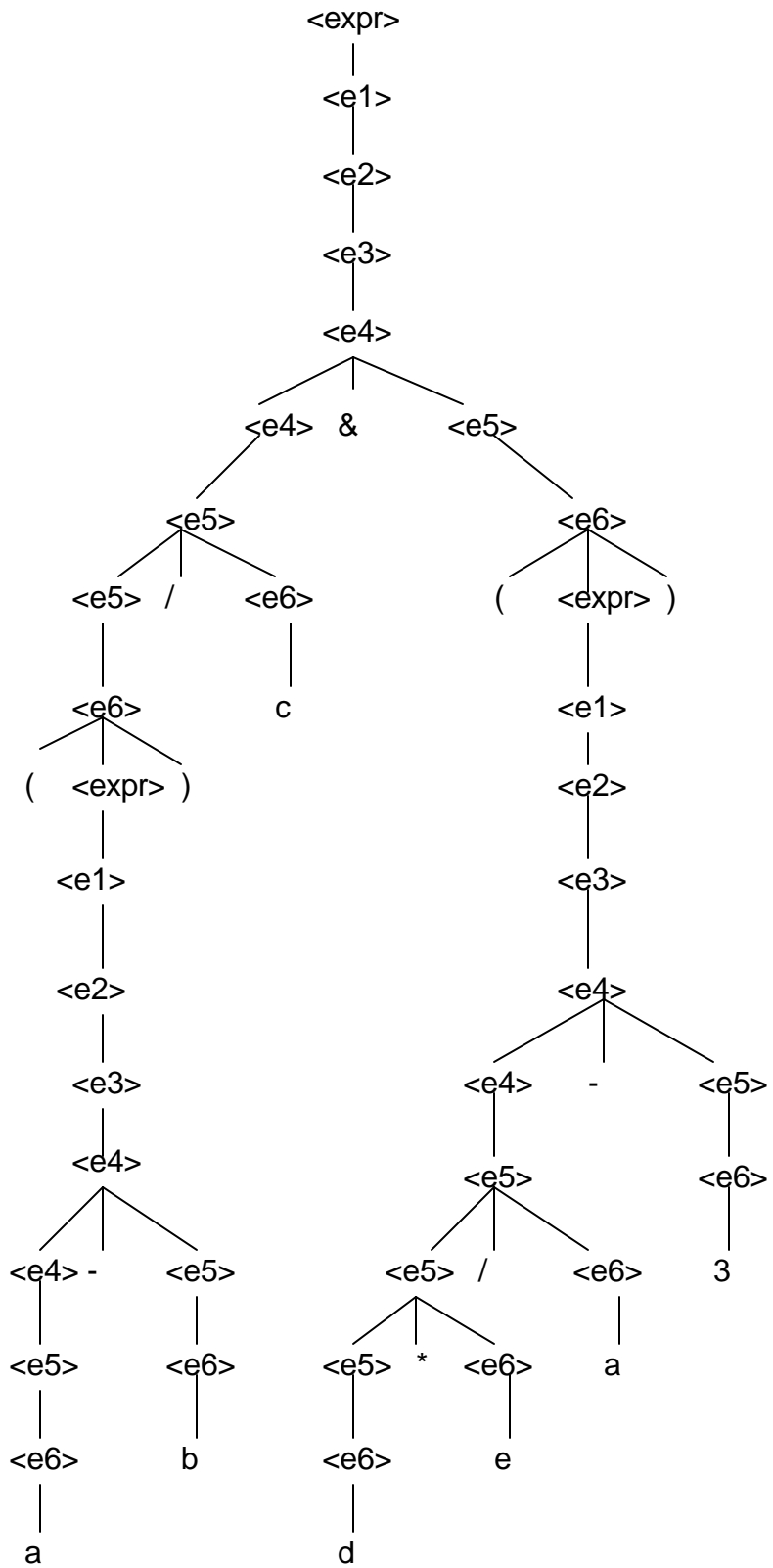
12. (a)



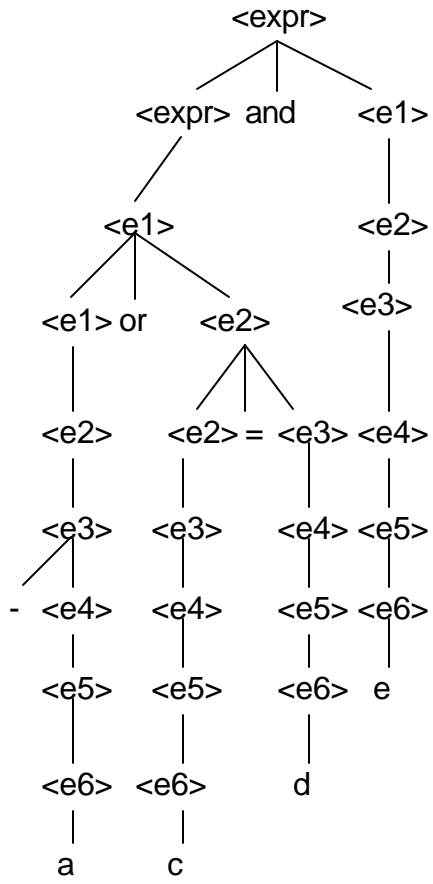
12. (b)



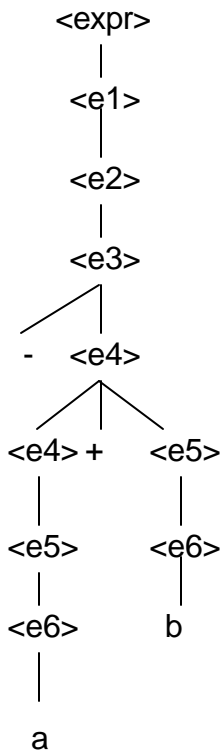
12. (c)



12. (d)



12. (f)



13. (a) (left -> right) sum1 is 46; sum2 is 48

(b) (right -> left) sum1 is 48; sum2 is 46

Chapter 8

Problem Set:

1. Three situations in which a combined counting and logical control loops are:

- A list of values is to be added to a SUM, but the loop is to be exited if SUM exceeds some prescribed value.
- A list of values is to be read into an array, where the reading is to terminate when either a prescribed number of values have been read or some special value is found in the list.
- The values stored in a linked list are to be moved to an array, where values are to be moved until the end of the linked list is found or the array is filled, whichever comes first.

4. Unique closing keywords on compound statements have the advantage of readability and the disadvantage of complicating the language by increasing the number of keywords.

8. The primary argument for using Boolean expressions exclusively as control expressions is the reliability that results from disallowing a wide range of types for this use. In C, for example, an expression of any type can appear as a control expression, so typing errors that result in references to variables of incorrect types are not detected by the compiler as errors.

Programming Exercises:

1.

(a) `Do K = (J + 13) / 27, 10`

`I = 3 * (K + 1) - 1`

`End Do`

(b) `for k in (j + 13) / 27 .. 10 loop`

`i := 3 * (k + 1) - 1;`

`end loop;`

(c) `for (k = (j + 13) / 27; k <= 10; i = 3 * (++k) - 1)`

2.

(a) `Do K = (J + 13.0) / 27.0, 10.0, 1.2`

`I = 3.0 * (K + 1.2) - 1.0`

`End Do`

(b) `while (k <= 10.0) loop`

`i := 3.0 * (k + 1.2) - 1.0;`

`k := k + 1.2;`

`end loop;`

(c) `for (k = (j + 13.0) / 27.0; k <= 10.0;`

`k = k + 1.2, i = 3.0 * k - 1)`

3.

(a) `Select Case (k)`

`Case (1, 2)`

`J = 2 * K - 1`

```

Case (3, 5)
    J = 3 * K + 1
Case (4)
    J = 4 * K - 1
Case (6, 7, 8)
    J = K - 2
Case Default
    Print *, 'Error in Select, K = ', K
End Select

```

(b) **case k is**

```

when 1 | 2 => j := 2 * k - 1;
when 3 | 5 => j := 3 * k + 1;
when 4 => j := 4 * k - 1;
when 6..8 => j := k - 2;
when others =>
    Put ("Error in case, k =');
    Put (k);
    New_Line;
end case;

```

(c) **switch** (k)

```

{
case 1: case 2:
    j = 2 * k - 1;
    break;
case 3: case 5:
    j = 3 * k + 1;
    break;

```



```

case 4:
    j = 4 * k - 1;
    break;
case 6: case 7: case 8:
    j = k - 2;
    break;
default:
    printf("Error in switch, k =%d\n", k);
}

```

4. j = -3;

```

key = j + 2;
for (i = 0; i < 10; i++){
    if ((key == 3) || (key == 2))
        j--;
    else if (key == 0)
        j += 2;
    else j = 0;
    if (j > 0)
        break;
    else j = 3 - i;
}

```

5. (C)

```

for (i = 1; i <= n; i++) {
    flag = 1;
    for (j = 1; j <= n; j++)
        if (x[i][j] <> 0) {
            flag = 0;
        }
}

```

```

        break;
    }
    if (flag == 1) {
        printf("First all-zero row is: %d\n", i);
        break;
    }
}

```

(Ada)

```

for I in 1..N loop
    Flag := true;
    for J in 1..N loop
        if X(I, J) /= 0 then
            Flag := false;
            exit;
        end if;
    end loop;
    if Flag = true then
        Put("First all-zero row is: ");
        Put(I);
        Skip_Line;
        exit;
    end if;
end loop;

```

Chapter 9

Problem Set:

2. The main advantage of this method is the fast accesses to formal parameters in subprograms. The disadvantages are that recursion is rarely useful when values cannot be passed, and also that a number of problems, such as aliasing, occur with the method.

4. This can be done in both Java and C#, using a static (or class) data member for the page number.

5. Assume the calls are not accumulative; that is, they are always called with the initialized values of the variables, so their effects are not accumulative.

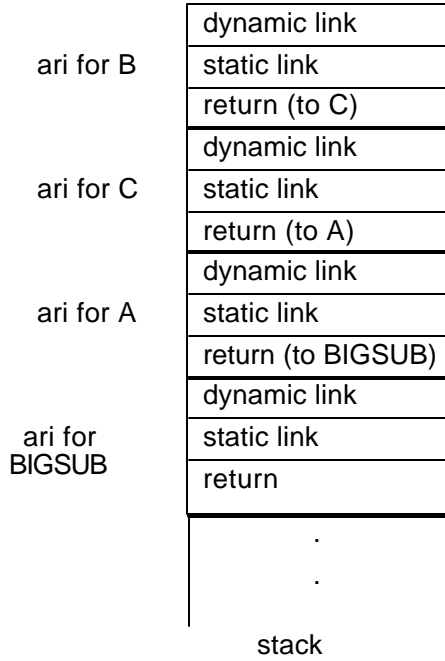
- | | | |
|---------------------|---------------------|--|
| a. 2, 1, 3, 5, 7, 9 | b. 1, 2, 3, 5, 7, 9 | c. 1, 2, 3, 5, 7, 9 |
| 2, 1, 3, 5, 7, 9 | 2, 3, 1, 5, 7, 9 | 2, 3, 1, 5, 7, 9 |
| 2, 1, 3, 5, 7, 9 | 5, 1, 3, 2, 7, 9 | 5, 1, 3, 2, 7, 9 (unless the addresses of the actual parameters are recomputed on return, in which case there will be an index range error.) |

6. It is rather weak, but one could argue that having both adds complexity to the language without sufficient increase in writability.

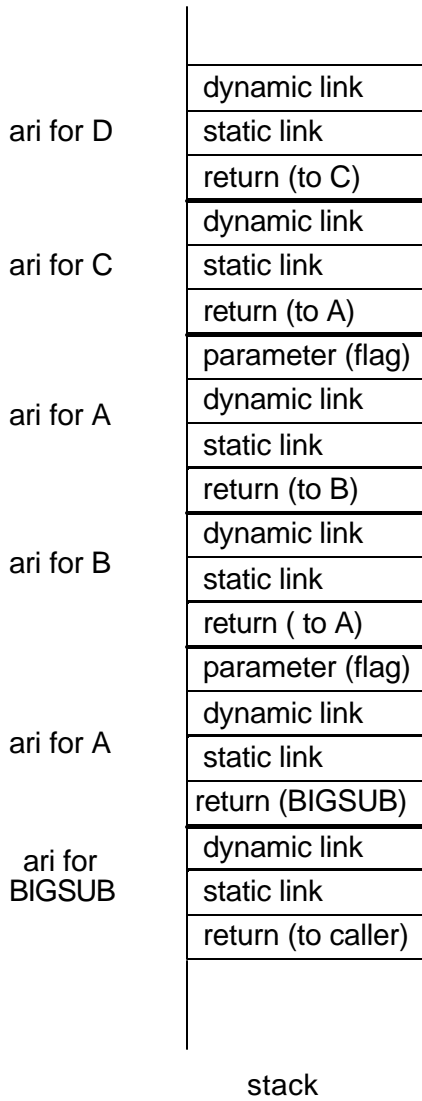
Chapter 10

Problem Set:

1.



2.



4. One very simple alternative is to assign integer values to all variable names used in the program. Then the integer values could be used in the activation records, and the comparisons would be between integer values, which are much faster than string comparisons.

5. Following the hint stated with the question, the target of every goto in a program could be represented as an address and a nesting_depth, where the nesting_depth is the difference between the nesting level of the procedure that contains the goto and that of the procedure containing the target. Then, when a goto is executed, the static chain is followed by the number of links indicated in the nesting_depth of the goto target. The stack top pointer is reset to the top of the activation record at the end of the chain.

6. Including two static links would reduce the access time to nonlocals that are defined in scopes two steps away to be equal to that for nonlocals that are one step away. Overall, because most nonlocal references are relatively close, this could significantly increase the execution efficiency of many programs.

Chapter 11

Problem Set:

2. The problem with this is that the user is given access to the stack through the returned value of the "top" function. For example, if `p` is a pointer to objects of the type stored in the stack, we could have:

```
p = top(stack1);  
  
*p = 42;
```

These statements access the stack directly, which violates the principle of a data abstraction.

Chapter 12

Problem Set:

1. In C++, a method can only be dynamically bound if all of its ancestors are marked `virtual`. By default, all method binding is static. In Java, method binding is dynamic by default. Static binding only occurs if the method is marked `final`, which means it cannot be overridden.
3. C++ has extensive access controls to its class entities. Individual entities can be marked `public`, `private`, or `protected`, and the derivation process itself can impose further access controls by being `private`. Ada, on the other hand, has no way to restrict inheritance of entities (other than through child libraries, which this book does not describe), and no access controls for the derivation process.

Chapter 13

Problem Set:

1. Competition synchronization is not necessary when no actual concurrency takes place simply because there can be no concurrent contention for shared resources. Two nonconcurrent processes cannot arrive at a resource at the same time.
2. When deadlock occurs, assuming that only two program units are causing the deadlock, one of the involved program units should be gracefully terminated, thereby allowing the other to continue.
3. The main problem with busy waiting is that machine cycles are wasted in the process.
4. Deadlock would occur if the `release(access)` were replaced by a `wait(access)` in the consumer process, because instead of relinquishing access control, the consumer would wait for control that it already had.
6. Sequence 1:
 - A fetches the value of `BUF_SIZE` (6)
 - A adds 2 to the value (8)
 - A puts 8 in `BUF_SIZE`
 - B fetches the value of `BUF_SIZE` (8)
 - B subtracts 1 (7)
 - B puts 7 in `BUF_SIZE`
 - `BUF_SIZE = 7`Sequence 2:
 - A fetches the value of `BUF_SIZE` (6)
 - B fetches the value of `BUF_SIZE` (6)
 - A adds 2 (8)
 - B subtracts 1 (5)
 - A puts 8 in `BUF_SIZE`

```

B puts 5 in BUF_SIZE
BUF_SIZE = 5
Sequence 3: A fetches the value of BUF_SIZE (6)
B fetches the value of BUF_SIZE (6)
A adds 2 (8)
B subtracts 1 (5)
B puts 5 in BUF_SIZE
A puts 8 in BUF_SIZE
BUF_SIZE = 8

```

Many other sequences are possible, but all produce the values 5, 7, or 8.

Chapter 14

Problem Set:

5. There are several advantages of a linguistic mechanism for handling exceptions, such as that found in Ada, over simply using a flag error parameter in all subprograms. One advantage is that the code to test the flag after every call is eliminated. Such testing makes programs longer and harder to read. Another advantage is that exceptions can be propagated farther than one level of control in a uniform and implicit way. Finally, there is the advantage that all programs use a uniform method for dealing with unusual circumstances, leading to enhanced readability.

6. There are several disadvantages of sending error handling subprograms to other subprograms. One is that it may be necessary to send several error handlers to some subprograms, greatly complicating both the writing and execution of calls. Another is that there is no method of propagating exceptions, meaning that they must all be handled locally. This complicates exception handling, because it requires more attention to handling in more places.

Chapter 15

Problem Set :

6. γ returns the given list with leading elements removed up to but not including the first occurrence of the first given parameter.

7. \times returns the number of non-NIL atoms in the given list.

Programming Exercises:

5. (DEFINE (deleteall atm lst)

```
(COND
  ((NULL? lst) '())
  ((EQ? atm (CAR lst)) (deleteall atm (CDR lst)))
  (ELSE (CONS (CAR lst) (deleteall atm (CDR lst))))
))
```

7. (DEFINE (deleteall atm lst)

```
(COND
  ((NULL? lst) '())
  ((NOT (LIST? (CAR lst)))
   (COND
     ((EQ? atm (CAR lst)) (deleteall atm (CDR lst)))
     (ELSE (CONS (CAR lst) (deleteall atm (CDR lst))))
   ))
  (ELSE (CONS (deleteall atm (CAR lst))
              (deleteall atm (CDR lst))))
))
```

9. (DEFINE (reverse lis)

```
(COND
  ((NULL? lis) '())
  (ELSE (APPEND (reverse (CDR lis)) (CONS (CAR lis) ())))
))
```

Chapter 16

Problem Set:

1. Ada variables are statically bound to types. Prolog variables are bound to types only when they are bound to values. These bindings take place during execution and are temporary.
2. On a single processor machine, the resolution process takes place on the rule base, one rule at a time, starting with the first rule, and progressing toward the last until a match is found. Because the process on each rule is independent of the process on the other rules, separate processors could concurrently operate on separate rules. When any of the processors finds a match, all resolution processing could terminate.
6. The list processing capabilities of Scheme and Prolog are similar in that they both treat lists as consisting of two parts, head and tail, and in that they use recursion to traverse and process lists.
7. The list processing capabilities of Scheme and Prolog are different in that Scheme relies on the primitive functions `CAR`, `CDR`, and `CONS` to build and dismantle lists, whereas with Prolog these functions are not necessary.

Programming Exercises:

2.

```
intersect([], X, []).

intersect([X | R], Y, [X | Z] :-
    member(X, Y),
    !,
    intersect(R, Y, Z).

intersect([X | R], Y, Z) :- intersect(R, Y, Z).
```

Note: this code assumes that the two lists, `x` and `y`, contain no duplicate elements.

3.

```
union([], X, X).

union([X | R], Y, Z) :- member(X, Y), !, union(R, Y, Z).

union([X | R], Y, [X | Z]) :- union(R, Y, Z).
```