

课后答案网 您最真诚的朋友



www.hackshp.cn网团队竭诚为学生服务，免费提供各门课后答案，不用积分，甚至不用注册，旨在为广大学生提供自主学习的平台！

课后答案网：www.hackshp.cn

视频教程网：www.efanjy.com

PPT课件网：www.ppthouse.com

课后答案网
www.hackshp.cn

Instructor's Solutions Manual

to

Concepts of Programming Languages

Seventh Edition

R.W. Sebesta

Preface

Changes to the Seventh Edition

The goals, overall structure, and approach of this seventh edition of *Concepts of Programming Languages* remain the same as those of the six 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 seventh edition evolved from the sixth through several 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 To maintain the currency of the material, some of the discussion of older programming languages has been replaced by material on newer languages.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, sections were added to both Chapters 1 and 2 that discuss markup/programming hybrid languages, using XSLT and JSP as examples. A section on program proofs using axiomatic semantics was added, including a new proof. The section on recursive descent parsing in Chapter 4 was strengthened by adding a new parsing routine and including a trace of a complete parse using the recursive descent algorithm. Material has been added in several places to introduce the most interesting features of Java 5.0, including its new iterative statement, its enumeration class, and its generics. Finally, minor changes, primarily to improve clarity, were made to a large number of sections of 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 seventh 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, 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 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, 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 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.
- 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. The web site contains:
 1. 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.
 2. 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#	microsoft.com
Java	java.sun.com

Haskell haskell.org
Scheme www.cs.rice.edu/CS/PLT/packages/drscheme/
Perl 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:

Liang Cheng, Lehigh University
Amer Diwan, University of Colorado
Nigel Gwee, Louisiana State University
John V. Harrison, University of Nevada, Las Vegas
Leon Jololian, New Jersey City University
K.N. King, Georgia State University
Donald Kraft, Louisiana State University
Simon H. Lin, California State University, Northridge
Meilu Lu, California State University, Sacramento
Amar Raheja, California Polytechnic University
Hossein Saiedian, University of Kansas
Raghvinder Sangwan, Penn State University
Young Park, Bradley University
Steve J. Phelps, California State University, Fullerton
Yang Wang, Southwest Missouri State University
Franck Xia, University of Missouri-Rolla
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, John Crenshaw, Charles Dana, Barbara Ann Griem, Mary Lou Haag, Eileen Head, Ralph C. Hilzer, Eric Joanis, Hikyoo Koh, Donald Kraft, Jiang B. Liu, Meiliu Lu, Jon Mauney, Bruce R. Maxim, Robert McCoard, Dennis L Mumaugh, Michael G. Murphy, Andrew Oldroyd, Rebecca Parsons, Jeffery Popyack, 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, and Salih Yurttas.

Matt Goldstein, Editor, Katherine Harutunian, Project Editor, and Pat Mahtani, Production Supervisor at Addison-Wesley, and Daniel Rausch and Edalin Michael at Argosy, all deserve my gratitude for their efforts to produce the seventh edition both quickly and carefully.

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 seven 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 M.S. and Ph.D. degrees in Computer Science from the Pennsylvania State University. He has taught computer science for more than 34 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 23
1.6	Language Design Trade-offs 24
1.7	Implementation Methods 25
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 45
2.4	Functional Programming: LISP 52
2.5	The First Step Toward Sophistication: ALGOL 60 57
2.6	Computerizing Business Records: COBOL 63
2.7	The Beginnings of Timesharing: BASIC 68
Interview: Alan Cooper—User Design and Language Design 72	
2.8	Everything for Everybody: PL/I 74
2.9	Two Early Dynamic Languages: APL and SNOBOL 78
2.10	The Beginnings of Data Abstraction: SIMULA 67 79
2.11	Orthogonal Design: ALGOL 68 80
2.12	Some Early Descendants of the ALGOLs 82
2.13	Programming Based on Logic: Prolog 90
2.14	History's Largest Design Effort: Ada 92
2.15	Object-Oriented Programming: Smalltalk 97
2.16	Combining Imperative and Object-Oriented Features: C++ 101
2.17	An Imperative-Based Object-Oriented Language: Java 104
2.18	Scripting Languages: JavaScript, PHP, and Python 108
2.19	A C-Based Language for the New Millennium: C# 112
2.20	Markup/Programming Hybrid Languages 115
Summary • Bibliographic Notes • Review Questions • Problem Set 117	
Chapter 3	Describing Syntax and Semantics 123
3.1	Introduction 124
3.2	The General Problem of Describing Syntax 125
3.3	Formal Methods of Describing Syntax 127
3.4	Attribute Grammars 141
History Note 142	
3.5	Describing the Meanings of Programs: Dynamic Semantics 148
History Note 164	
Summary • Bibliographic Notes • Review Questions • Problem Set •	
Programming Exercises 169	
Chapter 4	Lexical and Syntax Analysis 175
4.1	Introduction 176
4.2	Lexical Analysis 177
4.3	The Parsing Problem 181

4.4	Recursive-Descent Parsing	185	
4.5	Bottom-Up Parsing	193	
	Summary • Review Questions • Problem Set • Programming Exercises		200
Chapter 5	Names, Bindings, Type Checking, and Scopes		205
5.1	Introduction	206	
5.2	Names	207	
	History Note		207
	History Note		208
5.3	Variables	209	
	History Note		210
5.4	The Concept of Binding	212	
	Interview: Rasmus Lerdorf—Scripting Languages and Other Examples of Slick Solutions		216
5.5	Type Checking	222	
5.6	Strong Typing	223	
5.7	Type Compatibility	225	
	History Note		225
5.8	Scope	228	
5.9	Scope and Lifetime	237	
5.10	Referencing Environments	237	
5.11	Named Constants	239	
	Summary • Review Questions • Problem Set • Programming Exercises	242
Chapter 6	Data Types	251	
6.1	Introduction	252	
6.2	Primitive Data Types	253	
6.3	Character String Types	256	
	History Note		258
6.4	User-Defined Ordinal Types	261	
6.5	Array Types	266	
	History Note		267
	History Note		269
6.6	Associative Arrays	279	
	Interview: Rasmus Lerdorf—The Open Source Movement and Work Life		280
6.7	Record Types	284	
6.8	Union Types	288	
6.9	Pointer and Reference Types	292	
	History Note		296
	Summary • Bibliographic Notes • Review Questions • Problem Set • Programming Exercises		305
Chapter 7	Expressions and Assignment Statements	311	
7.1	Introduction	312	
7.2	Arithmetic Expressions	313	
	History Note		320
7.3	Overloaded Operators	321	
7.4	Type Conversions	323	
	History Note		324
7.5	Relational and Boolean Expressions	326	
	History Note		326
7.6	Short-Circuit Evaluation	329	

7.7	Assignment Statements	330	
History Note		333	
7.8	Mixed-mode Assignment	334	
Summary • Review Questions • Problem Set • Programming Exercises		335
Chapter 8	Statement-Level Control Structures		341
8.1	Introduction	342	
8.2	Selection Statements	343	
History Note		344	
History Note		346	
8.3	Iterative Statements	352	
History Note		354	
Interview: Larry Wall—Part 1: Linguistics and the Birth of Perl		362	
8.4	Unconditional Branching	366	
History Note		366	
8.5	Guarded Commands	367	
8.6	Conclusions	371	
Summary • Review Questions • Problem Set • Programming Exercises			372
Chapter 9	Subprograms	377	
9.1	Introduction	378	
9.2	Fundamentals of Subprograms	378	
9.3	Design Issues for Subprograms	385	
9.4	Local Referencing Environments	385	
9.5	Parameter-Passing Methods	387	
Interview: Larry Wall—Part 2: Scripting Languages in General and Perl in Particular			388
History Note		396	
History Note		397	
History Note		401	
9.6	Parameters That Are Subprogram Names	408	
History Note		409	
9.7	Overloaded Subprograms	410	
9.8	Generic Subprograms	411	
9.9	Design Issues for Functions	417	
9.10	User-Defined Overloaded Operators	418	
9.11	Coroutines	419	
History Note		419	
Summary • Review Questions • Problem Set • Programming Exercises			421
Chapter 10	Implementing Subprograms		427
10.1	The General Semantics of Calls and Returns	428	
10.2	Implementing “Simple” Subprograms	429	
10.3	Implementing Subprograms with Stack-Dynamic Local Variables		431
10.4	Nested Subprograms	439	
Interview: Niklaus Wirth—Keeping It Simple		440	
10.5	Blocks	447	
10.6	Implementing Dynamic Scoping	449	
Summary • Review Questions • Problem Set			453

Chapter 11	Abstract Data Types and Encapsulation Constructs	
459		
11.1	The Concept of Abstraction	460
11.2	Introduction to Data Abstraction	461
11.3	Design Issues for Abstract Data Types	463
11.4	Language Examples	464
Interview: Bjarne Stroustrup—C++: Its Birth, Its Ubiquitousness, and Common Criticisms		466
11.5	Parameterized Abstract Data Types	478
11.6	Encapsulation Constructs	481
11.7	Naming Encapsulations	484
Summary • Review Questions • Problem Set • Programming Exercises		488
Chapter 12	Support for Object-Oriented Programming	493
12.1	Introduction	494
12.2	Object-Oriented Programming	494
12.3	Design Issues for Object-Oriented Languages	497
12.4	Support for Object-Oriented Programming in Smalltalk	502
12.5	Support for Object-Oriented Programming in C++	504
Interview: Bjarne Stroustrup—On Paradigms and Better Programming		506
12.6	Support for Object-Oriented Programming in Java	514
12.7	Support for Object-Oriented Programming in C#	517
12.8	Support for Object-Oriented Programming in Ada 95	519
12.9	The Object Model of JavaScript	524
12.10	Implementation of Object-Oriented Constructs	527
Summary • Review Questions • Problem Set • Programming Exercises		530
Chapter 13	Concurrency	535
13.1	Introduction	536
13.2	Introduction to Subprogram-Level Concurrency	539
13.3	Semaphores	543
History Note		543
13.4	Monitors	548
13.5	Message Passing	550
13.6	Ada Support for Concurrency	551
13.7	Java Threads	562
13.8	C# Threads	570
13.9	Statement-Level Concurrency	572
Summary • Bibliographic Notes • Review Questions • Problem Set • Programming Exercises		574
Chapter 14	Exception Handling and Event Handling	579
14.1	Introduction to Exception Handling	580
History Note		584
14.2	Exception Handling in Ada	586
14.3	Exception Handling in C++	593
14.4	Exception Handling in Java	597
Interview: James Gosling—The Birth of Java		600
14.5	Introduction to Event Handling	607
14.6	Event Handling with Java	608

Summary • Bibliographic Notes • Review Questions • Problem Set	614
Chapter 15 Functional Programming Languages	619
15.1 Introduction	620
15.2 Mathematical Functions	621
15.3 Fundamentals of Functional Programming Languages	623
15.4 The First Functional Programming Language: LISP	624
15.5 An Introduction to Scheme	628
15.6 COMMON LISP	645
15.7 ML	645
15.8 Haskell	649
15.9 Applications of Functional Languages	653
15.10 A Comparison of Functional and Imperative Languages	653
Summary • Bibliographic Notes • Review Questions • Problem Set • Programming Exercises	654
Chapter 16 Logic Programming Languages	659
16.1 Introduction	660
16.2 A Brief Introduction to Predicate Calculus	660
16.3 Predicate Calculus and Proving Theorems	664
16.4 An Overview of Logic Programming	666
16.5 The Origins of Prolog	668
16.6 The Basic Elements of Prolog	669
16.7 The Deficiencies of Prolog	684
16.8 Applications of Logic Programming	689
Summary • Bibliographic Notes • Review Questions • Problem Set • Programming Exercises	691
Bibliography	695
Index	705

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.

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 pages 161-163, 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>`

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

$\mid \langle \text{factor} \rangle$

$\langle \text{factor} \rangle \rightarrow (\langle \text{expr} \rangle)$

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

6.

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

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

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

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

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

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

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

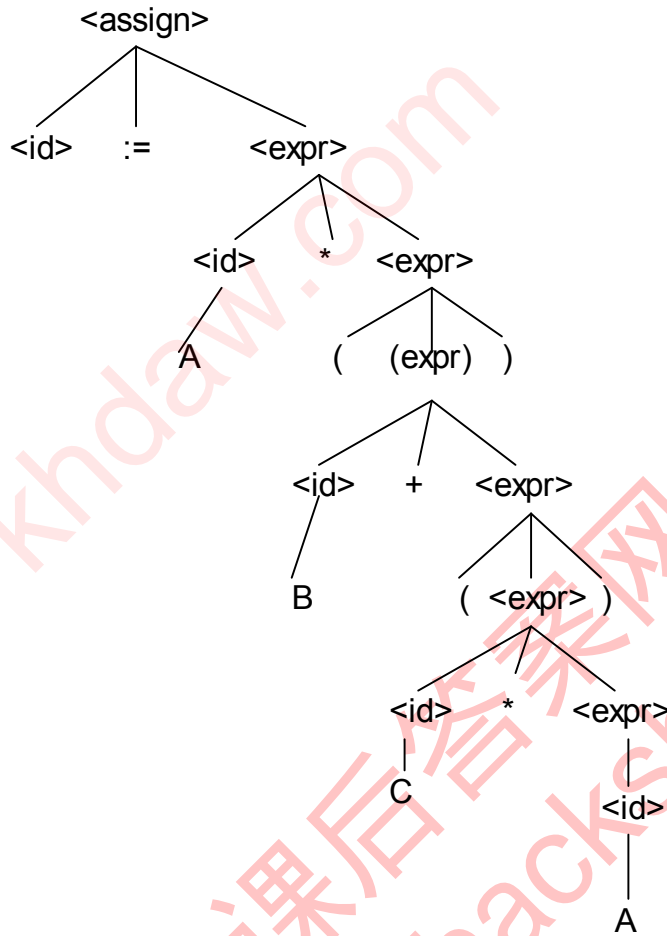
$\Rightarrow A = A * (B + (\langle \text{expr} \rangle))$

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

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

$\Rightarrow A = A * (B + (C * \langle \text{id} \rangle))$

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



7.

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

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

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

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

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

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

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

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

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

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

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

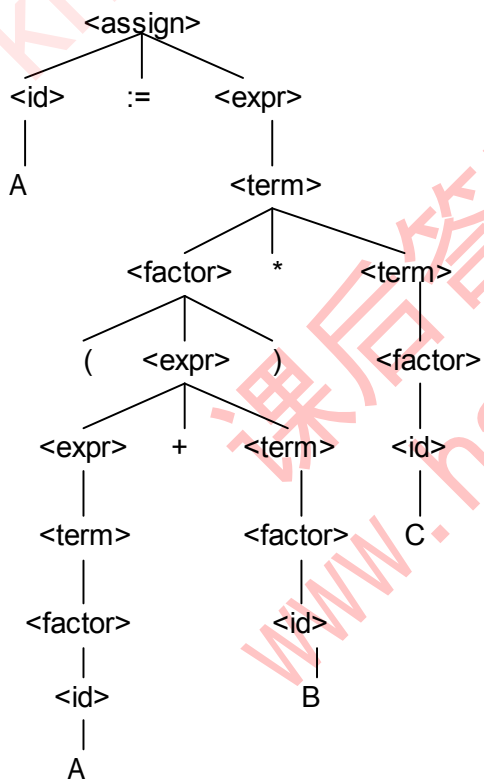
$\Rightarrow A = (A + \langle id \rangle) * \langle term \rangle$

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

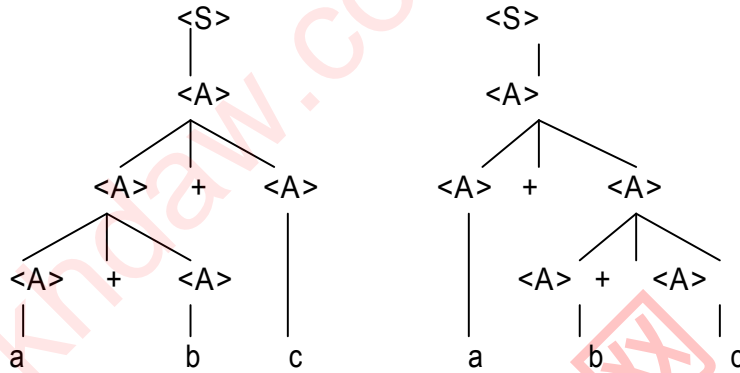
$\Rightarrow A = (A + B) * \langle factor \rangle$

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

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



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

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

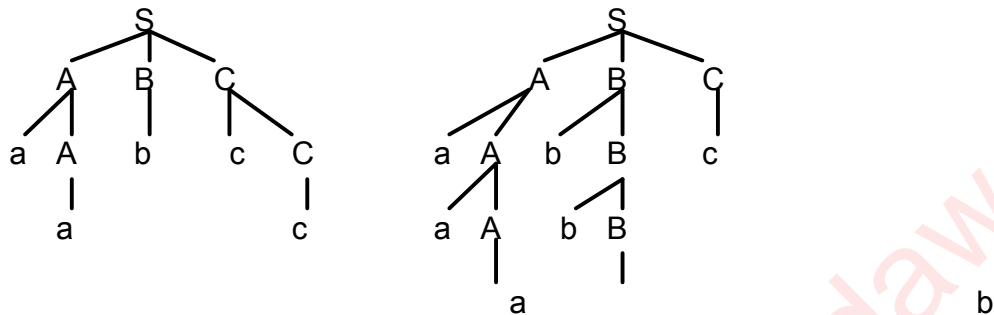
with

$\langle \text{factor} \rangle \rightarrow + \langle \text{id} \rangle$
 $\quad \quad \quad | - \langle \text{id} \rangle$

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

$\langle \text{id} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle (+ \mid -) \langle \text{expr} \rangle$

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

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

18.

(a) (Java **do-while**) We assume that the logic expression is a single relational expression.

```
loop: (do body)
      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. $M_b(B, s)$ 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. $M_{cf}(\text{for } (\text{expr1}; \text{expr2}; \text{expr3}) L, s) \triangleq$

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

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

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

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:

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

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

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

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

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. $a + b * c$

Call lex /* returns a */

Enter <expr>

Enter <term>

Enter <factor>

Call lex /* returns + */

Exit <factor>

Exit <term>

Call lex /* returns b */

Enter <term>

Enter <factor>

Call lex /* returns * */

Exit <factor>

Call lex /* returns c */

Enter <factor>

Call lex /* returns end-of-input */

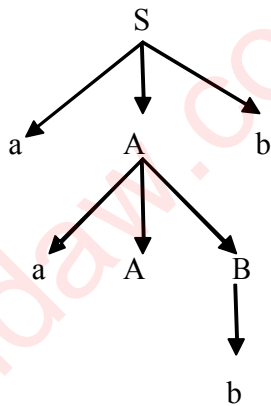
Exit <factor>

Exit <term>

Exit <expr>

5.

(a) aaAbb

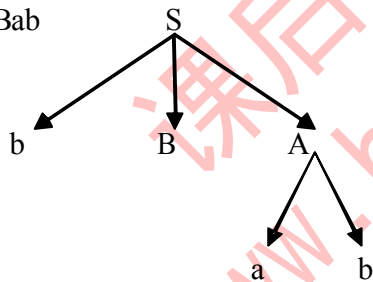


Phrases: aaAbb, aAb, b

Simple phrases: b

Handle: b

(b) bBab



Phrases: bBab, ab

Simple phrases: ab

Handle: ab

7. *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])

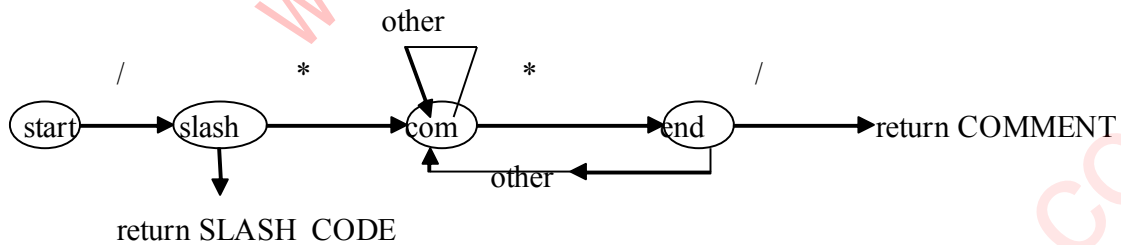
0T2*7

(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 is 5.

Dynamic scoping: x is 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 hands with a number of fingers that was a power of 2, these kinds of problems would not occur.

5. 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 `->` operator in C and C++ is writability. It is slightly easier to write `p -> q` than `(*p) . q`.

9. 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.

10. 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:

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

11. 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.

15. 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) $((a * (b - 1)^1)^2 / c)^3 \bmod d)^4$

(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) $(a * ((b - 1)^2 / (c \bmod d)^1)^3)^4$

(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. $\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle \text{ or } \langle e1 \rangle \mid \langle \text{expr} \rangle \text{ xor } \langle e1 \rangle \mid \langle e1 \rangle$

$\langle e1 \rangle \rightarrow \langle e1 \rangle \text{ and } \langle e2 \rangle \mid \langle e2 \rangle$

$\langle e2 \rangle \rightarrow \langle e2 \rangle = \langle e3 \rangle \mid \langle e2 \rangle \neq \langle e3 \rangle \mid \langle e2 \rangle < \langle e3 \rangle$

$\mid \langle e2 \rangle \leq \langle e3 \rangle \mid \langle e2 \rangle > \langle e3 \rangle \mid \langle e2 \rangle \geq \langle e3 \rangle \mid \langle e3 \rangle$

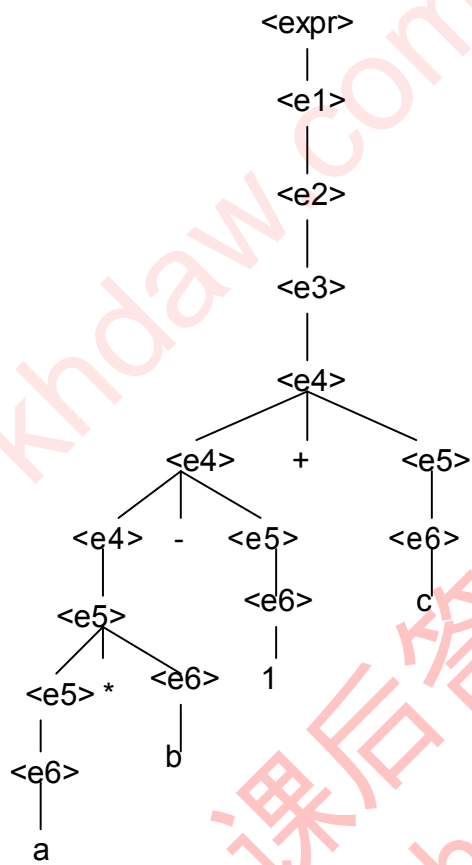
$\langle e3 \rangle \rightarrow \langle e4 \rangle$

$\langle e4 \rangle \rightarrow \langle e4 \rangle + \langle e5 \rangle \mid \langle e4 \rangle - \langle e5 \rangle \mid \langle e4 \rangle \& \langle e5 \rangle \mid \langle e4 \rangle \bmod \langle e5 \rangle \mid \langle e5 \rangle$

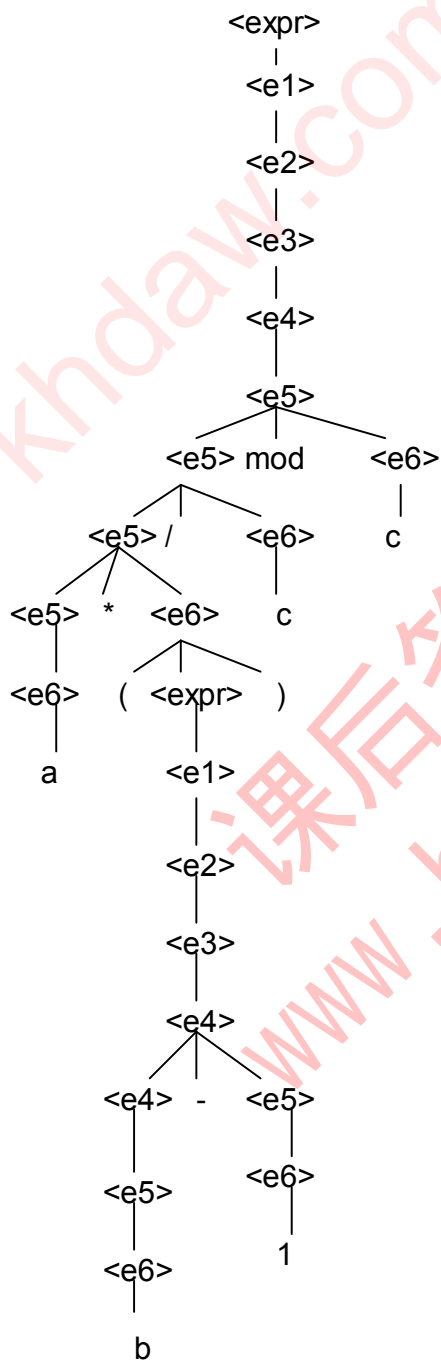
$\langle e5 \rangle \rightarrow \langle e5 \rangle * \langle e6 \rangle \mid \langle e5 \rangle / \langle e6 \rangle \mid \text{not } \langle e5 \rangle \mid \langle e6 \rangle$

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

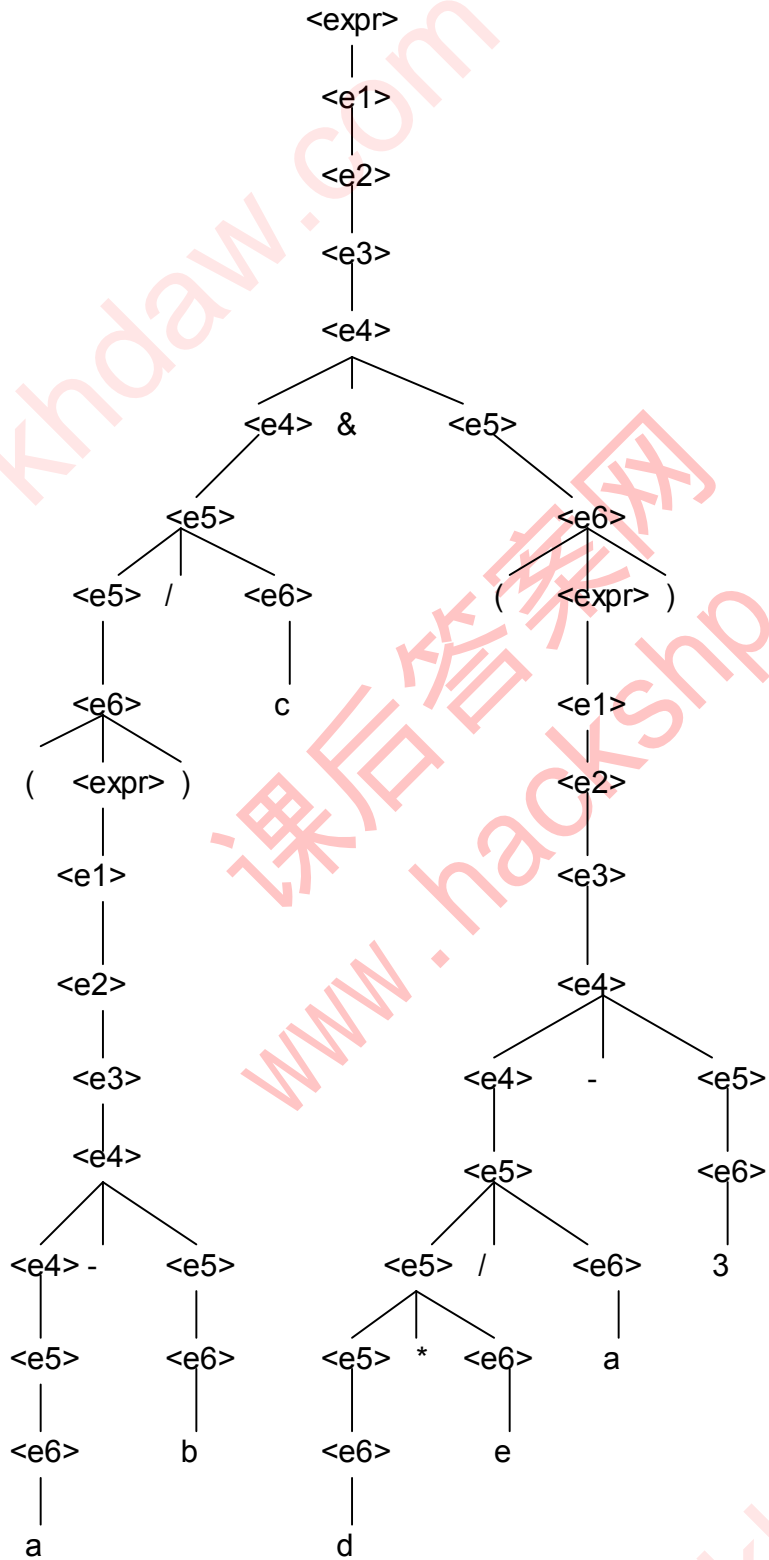
12. (a)



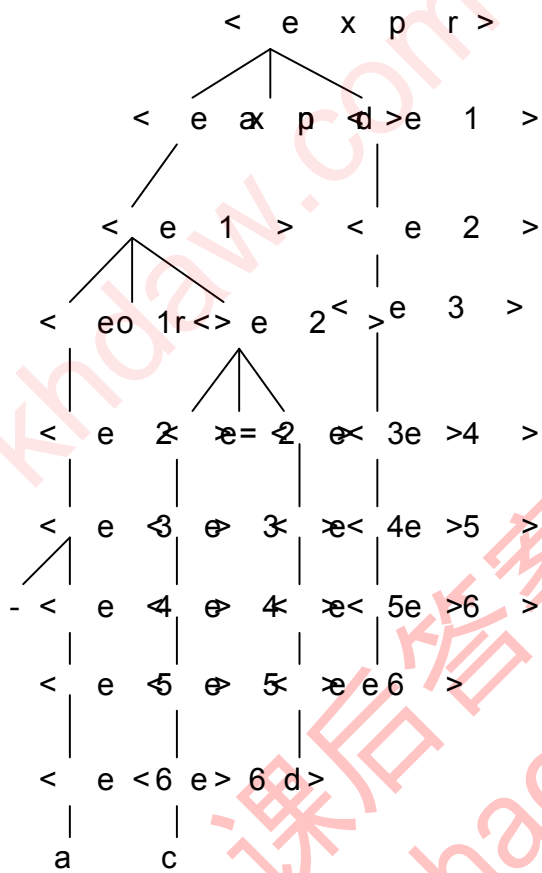
12. (b)



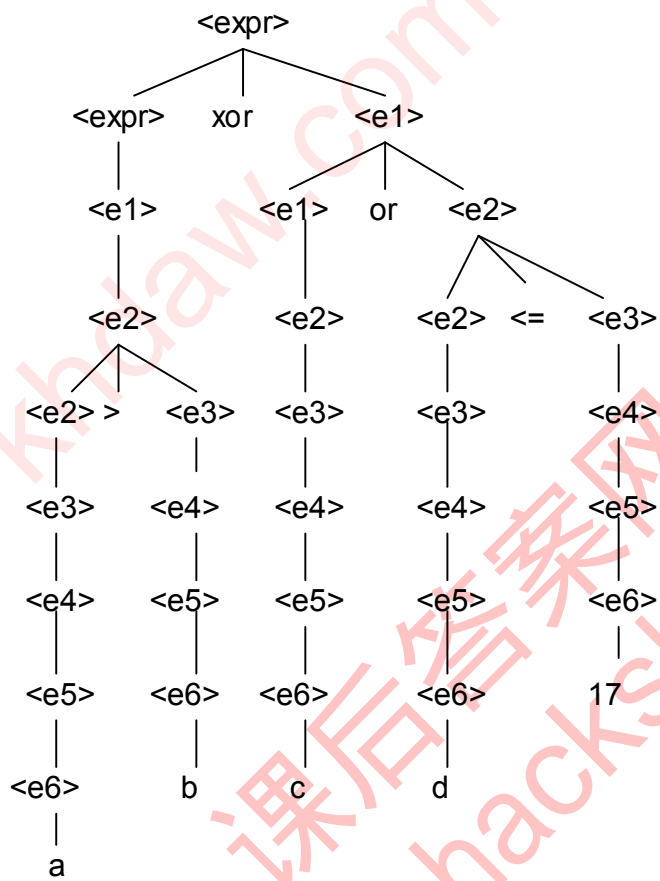
12. (c)



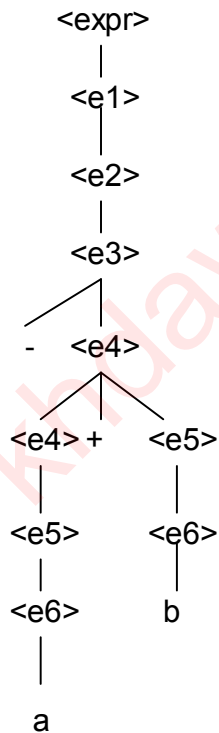
12. (d)



12. (e)



12. (f)



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

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

19. (a) 7

(b) 12

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.

- c. 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;

7.

I, J : Integer;

N : Integer := 100;

I = 0;

J = 17;

while I < N **loop**

Sum := Sum + I * J + 3;

I : I + 1;

J := J - 1;

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 writ ability.

7. (a) 1, 3

(b) 2, 6

(c) 2, 6

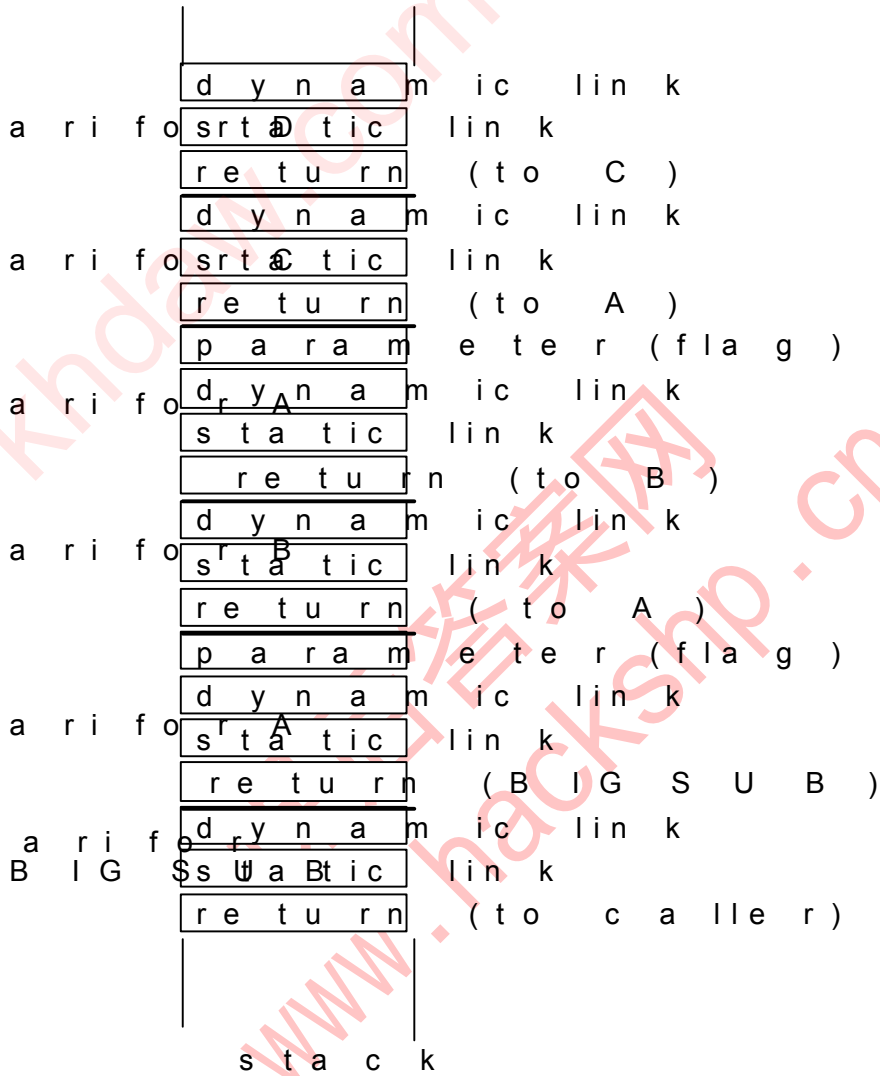
Chapter 10

Problem Set:

1.

	dynamic link
ari for B	static link
	return (to C)
	dynamic link
ari for C	static link
	return (to A)
	dynamic link
ari for A	static link
	return (to BIGSUB)
	dynamic link
ari for BIGSUB	static link
	return
	.
	.
stack	

3.



7. 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.

8. 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.

9. 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.

4. The problems created by making an abstract data type a pointer are: (a) There are many inherent problems in dealing with pointers (see Chapter 6), (b) comparisons between objects do not work as expected, because they are only pointer comparisons (rather than pointed-to value comparisons), and (c) the implementation of the abstract type cannot control allocation and deallocation of objects of the type (the user can create a pointer to an object with a variable declaration and use it without creating an object).

6. Implicit garbage collection removes the necessity of allowing users to explicitly deallocate objects, thereby eliminating the possibility of user-created dangling pointers.

8. There are several dangers inherent in C's approach to encapsulation. First, the user is allowed to simply paste the contents of the header file into the application file, which can lead to using subsequently updated implementation files without using the potentially updated header file. This can cause type conflicts between the implementation file and the header file. Another problem with pasting the header file into the implementation file is the loss of documentation of the dependence of the implementation file on the header file.

9. C++ did not eliminate the problems described in Problem 8 because it uses the C linker on its programs.

11. The three ways a C++ client program can reference a name from a namespace defined in a header file are (assuming the namespace name is `MyStack` and the variable is named `topPtr`):

a. `MyStack::topPtr`

b. Including the statement:

```
using MyStack::topPtr;
```

in the program.

c. Including the statement:

```
using namespace MyStack;
```

in the program and referencing `topPtr` by its name.

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.
7. Two problems of abstract data types that are ameliorated by inheritance are: (a) Reuse of abstract data types is much easier, because the modifications of existing types need not be done on the legacy code, and (b) with abstract data types, all types are independent and at the same level, disallowing any logically hierarchical type dependencies.
9. One disadvantage of inheritance is that types cannot be defined to be independent.
11. If a subclass has an is-a relationship with its parent class, a variable of the subclass can appear anywhere a variable of the parent class is allowed to appear.
14. One reason why all Java objects have a common ancestor is so they can all inherit a few universally useful methods.
15. The **`finalize`** clause in Java allows a specific action to take place, regardless of whether a **`try`** clause succeeds or fails.

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 put 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.

10. The safety of cooperation shynchynchronization using semaphores is basically the same as using Ada's **when** clauses, although the **when** clauses are somewhat more readable than semaphores.

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

7. `x` 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).
```