Instructor's Resource Manual

to

Concepts of Programming Languages

Fourth Edition

R.W. Sebesta

Preface

The goals, overall structure, and approach of this fourth edition of *Concepts of Programming Languages* remain the same as those of the three earlier editions. The principal goal is to provide the reader with the tools necessary for the critical evaluation of existing and future programming languages and constructs. An additional goal is to prepare the reader for the study of compiler design and construction.

The book should also answer a myriad of questions that may have occurred to the reader who may know only one high-level programming language. For example, why are there so many different programming languages? How and why were they developed? In what ways are they similar? What are their differences? What kinds of programming languages may be developed and used in the future? Why wouldn't we simply continue to use what we have now?

There are two ways in which a book on the concepts of programming languages can be organized: a horizontal approach and a vertical approach. With the horizontal approach, each language selected is presented in some depth. With the vertical approach, the general concepts and constructs of programming languages are described in some particular sequence. For each construct, design issues are explored and examples from a variety of languages are presented. Both methods have merit. In order to accurately describe individual language concepts it is important to focus on the concepts and consider their impact on programming and the evolution of languages. However, a chronological analysis of language developments necessitates the study of specific languages and their origins and development. Furthermore, the design of a specific facility of a particular language is often influenced by other characteristics of the language. Because of these considerations, this book uses the vertical approach for the majority of the material, but the horizontal approach when it is advantageous.

In this book I describe the fundamental concepts of programming languages by defining 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 the design alternatives.

Taking this approach requires studying a collection of closely related topics. To discuss languages and language constructs, descriptive tools are vital. I discuss in detail the most effective and widely used methods of syntax description. I also introduce the most common methods for describing the semantics of programming languages. To understand some of the reasons why the particular design choices for existing languages were made, I describe the historical context and specific needs that spawned them. Because difficulty of implementation is often a significant influence on language design, discussions of implementation methods and issues are integrated throughout the book.

The following paragraphs outline the contents of the third edition.

Chapter 1 begins with a rationale for studying programming languages. It then discusses the criteria for evaluating programming languages. I recognize that defining these criteria is risky; however, evaluation principles are essential to any serious study of the design of programming languages. The primary influences on language design,

common design trade-offs, and the basic approaches to implementation are also examined in the chapter.

Chapter 2 uses the horizontal approach to chart the chronological 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 the further study of language design and evaluation. However, since none of the remainder of the book depends on Chapter 2, it can be skipped in its entirety.

Chapter 3 describes the primary formal methods for describing the syntax of programming languages: EBNF and syntax graphs. This is followed by a description of attribute grammars, which play a prominent role in compiler design. The difficult task of semantic description is then explored, including brief introductions to the three most common methods: operational, axiomatic, and denotational semantics.

Chapters 4–13 use the vertical approach to 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, the many characteristics of variables are covered in Chapter 4; more complicated data types in Chapter 5; expressions and assignment statements in Chapter 6; control statements in Chapter 7; subprograms and their implementation in Chapters 8 and 9; data abstraction facilities in Chapter 10; language features that support object-oriented programming (inheritance and dynamic method binding) in Chapter 11; concurrent program units in Chapter 12; exception handling in Chapter 13. I use the vertical approach because it is inappropriate to describe and evaluate the details of a particular construct in several different parts of the book, as the horizontal approach would require for these topics. Discussing in a single chapter the various methods for providing concurrency, for example, allows for a concise comparison and evaluation of those methods.

The last two chapters (14 and 15) describe two of the most important alternative programming paradigms: functional programming and logic programming. Each is discussed as a programming methodology, and then exemplified through a brief introduction to a specific language.

Specifically, Chapter 14 begins by discussing simple mathematical functions, functional forms, and functional programming languages. It then presents an introduction to Scheme, including descriptions of some of its primitive functions, special forms, functional forms, and 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 languages. The chapter concludes with a comparison of functional and imperative languages.

The topic of Chapter 15 is logic programming and logic programming languages. I begin by introducing predicate calculus and explaining how it is used to prove theorems. This is followed by an overview of logic programming. The bulk of the chapter is an introduction to Prolog, including descriptions of resolution and unification, and some example programs and descriptions of their behavior.

Changes for the Fourth Edition

The fourth edition of this book is a significant revision of the third edition. Most of the changes result from the growing dominance of the object-oriented programming paradigm. The following paragraphs list the most important of these changes.

In a clear break from the earlier editions, and with most other books on programming languages, the fourth edition does *not* include a chapter on object-oriented languages. It *does* include a chapter on language support for object-oriented programming, specifically inheritance and dynamic method binding. This chapter, which is a greatly revised version of Chapter 15 in the third edition, has been moved to its more logical position as Chapter 11, immediately following the chapter on data abstraction. It has been significantly expanded to include an extensive discussion of a collection of design issues, which provides a framework for the descriptions and evaluations of the various language designs for inheritance and dynamic method binding.

There are two reasons for this new approach: First, the great majority of object-oriented software that is now written is written in languages that are similar to the imperative languages of the past four decades. The expressions, assignment statements, data structures, and control structures of these languages are very similar to those of C and Pascal. Therefore there is no reason to treat these features of these languages separately. By our definition of an imperative language, C++, Ada 95, and Java are imperative languages. I regard their support for object-oriented programming as being the next stage of development of the imperative languages. While the object-oriented software development paradigm is very different from the procedural paradigm, the languages in which these two approaches are used are not all that different. The difference between a language that supports data-oriented programming, such as Ada 83, and one that supports object-oriented programming is even less significant. The second reason for the change in attitude about object-oriented languages is that object-oriented programming is no longer the new and experimental paradigm it was not too many years ago. It is now the dominant approach to software development and the languages used for it are the most widely used languages around today. Therefore, a book such as this should not push the discussions of language features for it off into a single late chapter, such as we still do with logic programming languages. It clearly should be integrated into most of the chapters of the book, which is what we have done in the fourth edition.

Other changes include the following: The appearance of Java and its rapid rise in popularity has caused me to add coverage of several of its interesting features. Specifically, its support for object-oriented programming has been added to Chapter 11, its concurrency to Chapter 12, and its exception handling to Chapter 13. In addition, some of its other features appear in earlier chapters.

Because Miranda is proprietary and Haskell is in the public domain, we have replaced the discussion or Miranda in Chapter 14 with one on Haskell.

The sections in Chapter 3 on axiomatic and denotational semantics have been again strengthened in the fourth edition.

Besides adding coverage of new languages and new features of older languages, we have deleted some discussion of older languages. For

example, Modula-2 coroutines and its support for abstract data types have been dropped.

Numerous smaller changes ensure that the fourth edition correctly reflects the current state of programming language evolution.

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. Chapter 2 requires little lecture time because of its lack of hard technical content. Students find it interesting and beneficial reading, however. Because no material in subsequent chapters depends on Chapter 2, it can, as noted earlier, be skipped entirely.

Chapters 4–8 and 10 should be relatively easy for students with extensive programming experience in Pascal, C, C++, or Ada. Chapters 9, 11, 12, and 13 are more challenging and require more detailed lectures.

Chapters 14 and 15 are entirely new to most students at the junior level. Ideally, language processors for Scheme and Prolog should be available for Chapters 14 and 15. Sufficient material is included in these chapters 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.

Supplements

Two important and useful supplements are available for this book. A disk-based solutions manual (ISBN ???) that includes answers to many of the problems in the chapter problem sets can be obtained upon request from an Addison-Wesley Publishing sales representative. A set of lecture notes slides is also available. These slides are in the form of Microsoft Powerpoint source files, one for each of the first 13 chapters of the book. I developed them over the past few years in teaching a course based on the book. The Powerpoint files are available through an anonymous ftp account on aw.com (?) in directory /cseng/authors/sebesta (?). Please check the README or .message files (?)at this site for further details and information on this and other supplements.

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:

Java `http://java.sun.com`

Haskell	`http://haskell.org`
Scheme	`http://www-swiss.ai.mit/ftpdir/scheme-7.4/`

Acknowledgments

The quality of this book was significantly improved as a result of the extensive suggestions, corrections, and comments provided by its reviewers. The first three editions were reviewed by Vicki Allan, Henry Bauer, Peter Brouwer, Paosheng Chang, John Crenshaw, Barbara Ann Griem, Mary Lou Haag, Jon Mauney, Robert McCoard, Michael G. Murphy, Andrew Oldroyd, Jeffery Popyack, Steven Rapkin, Hamilton Richard, Tom Sager, Joseph Schell, and Mary Louise Soffa. The fourth edition was reviewed by:

- Mary Lou Haag, University of Colorado at Colorado Springs
- Hikyoo Koh, Lamar University
- Bruce Maxim, University of Michigan at Dearborn
- L. Andrew Oldroyd, Washington University
- Rebecca Parsons, University of Central Florida

Maite Swarez-Rivas, editor, Molly Taylor, assistant editor, and Pat Unubun, production supervisor, all deserve my gratitude for their efforts to produce the fourth edition quickly, as well as helping it be significantly better than the third.

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 four editions of this book.

About the Author

Robert Sebesta is an Associate Professor and Chairman of 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 been teaching computer science for over 25 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 Reasons for Studying Concepts of Programming Languages
1.2 Programming Domains
1.2.1 Scientific Applications
1.2.2 Business Applications
1.2.3 Artificial Intelligence
1.2.4 Systems Programming
1.2.5 Scripting Languages
1.2.6 Special Purpose Languages
1.3 Language Evaluation Criteria
1.3.1 Readability
1.3.1.1 Overall Simplicity
1.3.1.2 Orthogonality
1.3.1.3 Control Statements
1.3.1.4 Data Types and Structures
1.3.1.5 Syntax Considerations
1.3.2 Writability
1.3.2.1 Simplicity and Orthogonality
1.3.2.2 Support for Abstraction
1.3.2.3 Expressivity
1.3.3 Reliability
1.3.3.1 Type Checking
1.3.3.2 Exception Handling
1.3.3.3 Aliasing
1.3.3.4 Readability and Writability
1.3.4 Cost
1.4 Influences on Language Design
1.4.1 Computer Architecture
1.4.2 Programming Methodologies
1.5 Language Categories
1.6 Language Design Trade-Offs
1.7 Implementation Methods
1.7.1 Compilation
1.7.2 Pure Interpretation
1.7.3 Hybrid Interpretation Systems
1.8 Programming Environments
Summary
Review Questions
Problem Set

Chapter 2 Evolution of the Major Imperative Programming Languages

2.1 Zuse's Plankalkül
2.1.1 Historical Background
2.1.2 Language Overview
2.2 Minimal Hardware Programming: Pseudocodes

2.2.1 Short Code
2.2.2 Speedcoding
2.2.3 The UNIVAC "Compiling" System
2.2.4 Related Work
2.3 The IBM 704 and FORTRAN
2.3.1 Historical Background
2.3.2 Design Process
2.3.3 FORTRAN I Overview
2.3.4 FORTRAN II Overview
2.3.5 FORTRAN IV, FORTRAN 77, and FORTRAN 90
2.3.6 Evaluation
2.4 Functional Programming: LISP
2.4.1 The Beginnings of Artificial Intelligence and List Processing
2.4.2 LISP Design Process
2.4.3 Language Overview
2.4.3.1 Data Structures
2.4.3.2 Processes in Functional Programming
2.4.3.3 The Syntax of LISP
2.4.4 Evaluation
2.4.5 Two Descendants of LISP
2.4.5.1 Scheme
2.4.5.2 COMMON LISP
2.4.6 Related Languages
2.5 The First Step Toward Sophistication: ALGOL 60
2.5.1 Historical Background
2.5.2 Early Design Process
2.5.3 ALGOL 58 Overview
2.5.4 Reception of the ALGOL 58 Report
2.5.5 ALGOL 60 Design Process
2.5.6 ALGOL 60 Overview
2.5.7 ALGOL 60 Evaluation
2.6 Computerizing Business Records: COBOL
2.6.1 Historical Background
2.6.2 FLOW-MATIC
2.6.3 COBOL Design Process
2.6.4 Evaluation
2.7 The Beginnings of Timesharing: BASIC
2.7.1 Design Process
2.7.2 Language Overview
2.7.3 Evaluation
2.8 Everything for Everybody: PL/I
2.8.1 Historical Background
2.8.2 Design Process
2.8.3 Language Overview
2.8.4 Evaluation
2.9 Two Early Dynamic Languages: APL and SNOBOL
2.9.1 Origins and Characteristics of APL
2.9.2 Origins and Characteristics of SNOBOL
2.10 The Beginnings of Data Abstraction: SIMULA 67
2.10.1 Design Process
2.10.2 Language Overview
2.11 Orthogonal Design: ALGOL 68
2.11.1 Design Process
2.11.2 Language Overview

2.11.3 Evaluation
2.12 Some Important Descendants of the ALGOLs
2.12.1 Simplicity by Design: Pascal
2.12.1.1 Historical Background
2.12.1.2 Evaluation
2.12.2 A Portable Systems Language: C
2.12.2.1 Historical Background
2.12.2.2 Evaluation
2.12.3 Other ALGOL Descendants
2.12.3.1 Modula-2
2.12.3.2 Modula-3
2.12.3.3 Oberon
2.12.3.4 Delphi
2.13 Programming Based on Logic: Prolog
2.13.1 Design Process
2.13.2 Language Overview
2.13.3 Evaluation
2.14 History's Largest Design Effort: Ada
2.14.1 Historical Background
2.14.2 Design Process
2.14.3 Language Overview
2.14.4 Evaluation
2.14.5 Ada 95
2.15 Object-Oriented Programming: Smalltalk
2.15.1 Design Process
2.15.2 Language Overview
2.15.3 Evaluation
2.16 Combining Imperative and Object-Oriented Features: C++
2.16.1 Design Process
2.16.2 Language Overview
2.16.3 Evaluation
2.16.4 A Related Language: Eiffel
2.17 Programming the World-Wide Web: Java
2.17.1 Design Process
2.17.2 Language Overview
2.17.3 Evaluation
Summary
Bibliographic Notes
Review Questions
Problem Set

Chapter 3 Describing Syntax and Semantics

3.1 Introduction
3.2 The General Problem of Describing Syntax
3.2.1 Language Recognizers
3.2.2 Language Generators
3.3 Formal Methods of Describing Syntax
3.3.1 Backus-Naur Form and Context-Free Grammars
3.3.1.1 Context-Free Grammars
3.3.1.2 Origins of Backus-Naur Form
3.3.1.3 Fundamentals
3.3.1.4 Describing Lists

3.3.1.5 Grammars and Derivations
3.3.1.6 Parse Trees
3.3.1.7 Ambiguity
3.3.1.8 Operator Precedence
3.3.1.9 Associativity of Operators
3.3.1.10 An Unambiguous Grammar for `if-then-else`
3.3.2 Extended BNF
3.3.3 Syntax Graphs
3.3.4 Grammars and Recognizers
3.4 Recursive Descent Parsing
3.5 Attribute Grammars
3.5.1 Static Semantics
3.5.2 Basic Concepts
3.5.3 Attribute Grammars Defined
3.5.4 Intrinsic Attributes
3.5.5 Example Attribute Grammars
3.5.6 Computing Attribute Values
3.5.7 Evaluation
3.6 Describing the Meanings of Programs: Dynamic Semantics
3.6.1 Operational Semantics
3.6.1.1 The Basic Process
3.6.1.2 Evaluation
3.6.2 Axiomatic Semantics
3.6.2.1 Assertions
3.6.2.2 Weakest Preconditions
3.6.2.3 Assignment Statements
3.6.2.4 Sequences
3.6.2.5 Selection
3.6.2.6 Logical Pretest Loops
3.6.2.7 Evaluation
3.6.3 Denotational Semantics
3.6.3.1 A Simple Example
3.6.3.2 The State of a Program
3.6.3.3 Expressions
3.6.3.4 Assignment Statements
3.6.3.5 Logical Pretest Loops
3.6.3.6 Evaluation
Summary
Bibliographic Notes
Review Questions
Problem Set

Chapter 4 Names, Bindings, Type Checking, and Scopes

4.1 Introduction
4.2 Names
4.2.1 Design Issues
4.2.2 Name Forms
4.2.3 Special Words
4.3 Variables
4.3.1 Name
4.3.2 Address
4.3.2.1 Aliases

4.3.3 Type
4.3.4 Value
4.4 The Concept of Binding
4.4.1 Binding of Attributes to Variables
4.4.2 Type Bindings
4.4.2.1 Variable Declarations
4.4.2.2 Dynamic Type Binding
4.4.2.3 Type Inference
4.4.3 Storage Bindings and Lifetime
4.4.3.1 Static Variables
4.4.3.2 Stack-dynamic Variables
4.4.3.3 Explicit Heap-Dynamic Variables
4.4.3.4 Implicit Dynamic Variables
4.5 Type Checking
4.6 Strong Typing
4.7 Type Compatibility
4.8 Scope
4.8.1 Static Scope
4.8.2 Blocks
4.8.3 Evaluation of Static Scoping
4.8.4 Dynamic Scope
4.8.5 Evaluation of Dynamic Scoping
4.9 Scope and Lifetime
4.10 Referencing Environments
4.11 Named Constants
4.12 Variable Initialization
Summary
Review Questions
Problem Set

Chapter 5 Data Types

5.1 Introduction
5.2 Primitive Data Types
5.2.1 Numeric Types
5.2.1.1 Integer
5.2.1.2 Floating-Point
5.2.1.3 Decimal
5.2.2 Boolean Types
5.2.3 Character Types
5.3 Character String Types
5.3.1 Design Issues
5.3.2 Strings and Their Operations
5.3.3 String Length Options
5.3.4 Evaluation
5.3.5 Implementation of Character String Types
5.4 User-Defined Ordinal Types
5.4.1 Enumeration Types
5.4.1.1 Designs
5.4.1.2 Evaluation
5.4.2 Subrange Types
5.4.2.1 Designs
5.4.2.2 Evaluation

5.4.3 Implementation of User-Defined Ordinal Types
5.5 Array Types
5.5.1 Design Issues
5.5.2 Arrays and Indices
5.5.3 Subscript Bindings and Array Categories
5.5.4 The Number of Subscripts in Arrays
5.5.5 Array Initialization
5.5.6 Array Operations
5.5.7 Slices
5.5.8 Evaluation
5.5.9 Implementation of Array Types
5.6 Associative Arrays
5.6.1 Structure and Operations
5.6.2 Implementing Associative Arrays
5.7 Record Types
5.7.1 Definitions of Records
5.7.2 References to Record Fields
5.7.3 Operations on Records
5.7.4 Evaluation
5.7.5 Implementation of Record Types
5.8 Union Types
5.8.1 Design Issues
5.8.2 Free Unions
5.8.3 The Discriminated Unions of ALGOL 68
5.8.4 Pascal Union Types
5.8.5 Ada Union Types
5.8.6 Evaluation
5.8.7 Implementation of Union Types
5.9 Set Types
5.9.1 Sets in Pascal and Modula-2
5.9.2 Evaluation
5.9.3 Implementation of Set Types
5.10 Pointer Types
5.10.1 Design Issues
5.10.2 Pointer Operations
5.10.3 Pointer Problems
5.10.3.1 Dangling Pointers
5.10.3.2 Lost Heap-Dynamic Variables
5.10.4 Pointers in Pascal
5.10.5 Pointers in Ada
5.10.6 Pointers in C and C++
5.10.7 Pointers in FORTRAN 90
5.10.8 Reference Types
5.10.9 Evaluation
5.10.10 Implementation of Pointer and Reference Types
5.10.10.1 Representations of Pointers and References
5.10.10.2 Solutions to the Dangling Pointer Problem
5.10.10.3 Heap Management
Summary
Bibliographic Notes
Review Questions
Problem Set

Chapter 6 Expressions and the Assignment Statement

6.1 Introduction
6.2 Arithmetic Expressions
6.2.1 Operator Evaluation Order
6.2.1.1 Precedence
6.2.1.2 Associativity
6.2.1.3 Parentheses
6.2.1.4 Conditional Expressions
6.2.2 Operand Evaluation Order
6.2.2.1 Side Effects
6.3 Overloaded Operators
6.4 Type Conversions
6.4.1 Coercion in Expressions
6.4.2 Explicit Type Conversions
6.4.3 Errors in Expressions
6.5 Relational and Boolean Expressions
6.5.1 Relational Expressions
6.5.2 Boolean Expressions
6.6 Short-Circuit Evaluation
6.7 Assignment Statements
6.7.1 Simple Assignments
6.7.2 Multiple Targets
6.7.3 Conditional Targets
6.7.4 Compound Assignment Operators
6.7.5 Unary Operator Assignments
6.7.6 Assignment as an Expression
6.8 Mixed-Mode Assignment
Summary
Review Questions
Problem Set

Chapter 7 Statement-Level Control Structures

7.1 Introduction
7.2 Compound Statements
7.3 Selection Statements
7.3.1 Two-Way Selection Statements
7.3.1.1 Design Issues
7.3.1.2 Examples of Two-Way Selectors
7.3.1.3 Nesting Selectors
7.3.1.4 Special Words and Selection Closure
7.3.2 Multiple Selection Constructs
7.3.2.1 Design Issues
7.3.2.2 Early Multiple Selectors
7.3.2.3 Modern Multiple Selectors
7.4 Iterative Statements
7.4.1 Counter-Controlled Loops
7.4.1.1 Design Issues
7.4.1.2 The DO Statement of FORTRAN 77 and FORTRAN 90
7.4.1.3 The ALGOL 60 `for` Statement
7.4.1.4 The Pascal `for` Statement

7.4.1.5 The Ada `for` Statement
7.4.1.6 The `for` Statement of C, C++, and Java
7.4.2 Logically Controlled Loops
7.4.2.1 Design Issues
7.4.2.2 Examples
7.4.3 User-Located Loop Control Mechanisms
7.4.4 Iteration Based on Data Structures
7.5 Unconditional Branching
7.5.1 Problems with Unconditional Branching
7.5.2 Label Forms
7.5.3 Restrictions on Branches
7.6 Guarded Commands
7.7 Conclusions
Summary
Review Questions
Problem Set

Chapter 8 Subprograms

8.1 Introduction
8.2 Fundamentals of Subprograms
8.2.1 General Subprogram Characteristics
8.2.2 Basic Definitions
8.2.3 Parameters
8.2.4 Procedures and Functions
8.3 Design Issues for Subprograms
8.4 Local Referencing Environments
8.5 Parameter-Passing Methods
8.5.1 Semantics Models of Parameter Passing
8.5.2 Implementation Models of Parameter Passing
8.5.2.1 Pass-by-Value
8.5.2.2 Pass-by-Result
8.5.2.3 Pass-by-Value-Result
8.5.2.4 Pass-by-Reference
8.5.2.5 Pass-by-Name
8.5.3 Parameter-Passing Methods of the Major Languages
8.5.4 Type-Checking Parameters
8.5.5 Implementing Parameter-Passing Methods
8.5.6 Multidimensional Arrays as Parameters
8.5.7 Design Considerations
8.5.8 Examples of Parameter Passing
8.6 Parameters That Are Subprogram Names
8.7 Overloaded Subprograms
8.8 Generic Subprograms
8.8.1 Generic Subprograms in Ada
8.8.2 Generic Functions in C++
8.9 Separate and Independent Compilation
8.10 Design Issues for Functions
8.10.1 Functional Side Effects
8.10.2 Types of Return Values
8.11 Accessing Nonlocal Environments
8.11.1 FORTRAN COMMON Blocks

8.11.2 External Declarations and Modules
8.12 User-Defined Overloaded Operators
8.13 Coroutines
Summary
Review Questions
Problem Set

Chapter 9 Implementing Subprograms

9.1 The General Semantics of Calls and Returns
9.2 Implementing FORTRAN 77 Subprograms
9.3 Implementing Subprograms in ALGOL-like Languages
9.3.1 More Complex Activation Records
9.3.2 An Example without Recursion and Nonlocal References
9.3.3 Recursion
9.3.4 Mechanisms for Implementing Nonlocal References
9.3.4.1 Static Chains
9.3.4.2 Displays
9.4 Blocks
9.5 Implementing Dynamic Scoping
9.5.1 Deep Access
9.5.2 Shallow Access
9.6 Implementing Parameters That Are Subprogram Names
9.6.1 Static Scoping
9.6.2 Displays
9.6.3 Referencing Environment Confusion Revisited
Summary
Bibliographic Notes
Review Questions
Problem Set

Chapter 10 Data Abstraction

10.1 The Concept of Abstraction
10.2 Encapsulation
10.3 Introduction to Data Abstraction
10.3.1 Floating-Point as an Abstract Data Type
10.3.2 User-Defined Abstract Data Types
10.3.3 An Example
10.4 Design Issues
10.5 Language Examples
10.5.1 SIMULA 67 Classes
10.5.1.1 Encapsulation
10.5.1.2 Information Hiding
10.5.1.3 Evaluation
10.5.2 Abstract Data Types in Ada
10.5.2.1 Encapsulation
10.5.2.2 Information Hiding
10.5.2.3 An Example
10.5.2.4 A Related Language: Modula-2
10.5.3 Abstract Data Types in C++
10.5.3.1 Encapsulation

10.5.3.2 Information Hiding
10.5.3.3 An Example
10.5.3.4 Evaluation
10.5.3.5 A Related Language: Java
10.6 Parameterized Abstract Data Types
10.6.1 Ada
10.6.2 C++
Summary
Review Questions
Problem Set

Chapter 11 Support for Object-Oriented Programming

11.1 Introduction
11.2 Object-Oriented Programming
11.2.1 Introduction
11.2.2 Inheritance
11.2.3 Polymorphism and Dynamic Binding
11.2.4 Computing with an Object-Oriented Language
11.3 Design Issues for Object-Oriented Language
11.3.1 The Exclusivity of Objects
11.3.2 Are Subclasses Subtypes?
11.3.3 Implementation and Interface Inheritance
11.3.4 Type Checking and Polymorphism
11.3.5 Single and Multiple Inheritance
11.3.6 Allocation and Deallocation of Objects
11.3.7 Dynamic and Static Binding
11.4 Overview of Smalltalk
11.4.1 General Characteristics
11.4.2 The Smalltalk Environment
11.5 The Smalltalk Language
11.5.1 Expressions
11.5.1.1 Literals
11.5.1.2 Variables
11.5.1.3 Message Expressions
11.5.2 Methods
11.5.3 Assignment Statements
11.5.4 Blocks and Control Structures
11.5.4.1 Blocks
11.5.4.2 Iteration
11.5.4.3 Selection
11.5.5 Classes
11.5.6 More about Methods
11.6 Smalltalk Example Programs
11.6.1 A Simple Table Handler
11.6.2 LOGO-Style Graphics
11.7 Large-Scale Features of Smalltalk
11.7.1 Type Checking and Polymorphism
11.7.2 Inheritance
11.8 Evaluation
11.9 Support for Object-Oriented Programming in C++
11.9.1 General Characteristics
11.9.2 Inheritance

11.9.3 Dynamic Binding
11.9.4 Evaluation
11.10 Support for Object-Oriented Programming in Java
11.10.1 General Characteristics
11.10.2 Inheritance
11.10.3 Dynamic Binding
11.10.4 Encapsulation
11.10.5 Evaluation
11.11 Support for Object-Oriented Programming in Ada 95
11.11.1 General Characteristics
11.11.2 Inheritance
11.11.3 Dynamic Binding
11.11.4 Evaluation
11.12 Support for Object-Oriented Programming in Eiffel
11.12.1 General Characteristics
11.12.2 Inheritance
11.12.3 Dynamic Binding
11.12.4 Evaluation
11.13 Implementation of Object-Oriented Constructs
11.13.1 Instance Data Storage
11.13.2 Dynamic Binding of Messages to Methods
Summary
Review Questions
Problem Set

Chapter 12 Concurrent Subprograms

12.1 Introduction
12.1.1 Multiprocessor Architectures
12.1.2 Classes of Concurrency
12.1.3 Motivation for Studying Concurrency
12.2 Introduction to Subprogram-Level Concurrency
12.2.1 Fundamental Concepts
12.2.2 Language Design for Concurrency
12.2.2.1 Design Issues
12.3 Semaphores
12.3.1 Introduction
12.3.2 Cooperation Synchronization
12.3.3 Competition Synchronization
12.3.4 Evaluation
12.4 Monitors
12.4.1 Introduction
12.4.2 Competition Synchronization
12.4.3 Cooperation Synchronization
12.4.4 Evaluation
12.5 Message Passing
12.5.1 Introduction
12.5.2 The Concept of Synchronous Message Passing
12.5.3 The Ada 83 Message-Passing Model
12.5.4 Cooperation Synchronization
12.5.5 Competition Synchronization
12.5.6 Task Termination
12.5.7 Priorities
12.5.8 Binary Semaphores

12.5.9 Evaluation
12.6 Concurrency in Ada 95
12.6.1 Protected Objects
12.6.2 Asynchronous Messages
12.7 Java Threads
12.7.1 The `Thread` Class
12.7.2 Priorities
12.7.3 Competition Synchronization
12.7.4 Cooperation Synchronization
12.7.5 Evalation
12.8 Statement-Level Concurrency
12.8.1 High-Performance FORTRAN
Summary
Bibliographic Notes
Review Questions
Problem Set

Chapter 13 Exception Handling

13.1 Introduction to Exception Handling
13.1.1 Basic Concepts
13.1.2 Design Issues
13.1.3 History
13.2 Exception Handling in PL/I
13.2.1 Exception Handlers
13.2.2 Binding Exceptions to Handlers
13.2.3 Continuation
13.2.4 Other Design Choices
13.2.5 An Example
13.2.6 Evaluation
13.3 Exception Handling in Ada
13.3.1 Exception Handlers
13.3.2 Bindings of Exceptions to Handlers
13.3.3 Continuation
13.3.4 Other Design Choices
13.3.5 An Example
13.3.6 Evaluation
13.4 Exception Handling in C++
13.4.1 Exception Handlers
13.4.2 Binding Exceptions to Handlers
13.4.3 Continuation
13.4.4 Other Design Choices
13.4.5 An Example
13.4.6 Evaluation
13.5 Exception Handling in Java
13.5.1 Classes of Exceptions
13.5.2 Exception Handlers
13.5.3 Binding Exceptions to Handlers
13.5.4 Continuation
13.5.5 Other Design Choices
13.5.6 An Example
13.5.7 The **`finally`** Clause

13.5.8 Evaluation
Summary
Bibliographic Notes
Review Questions
Problem Set

Chapter 14 Functional Programming Languages

14.1 Introduction
14.2 Mathematical Functions
14.2.1 Simple Functions
14.2.2 Functional Forms
14.3 Fundamentals of Functional Programming Languages
14.4 The First Functional Programming Language: LISP
14.4.1 Data Types and Structures
14.4.2 The First LISP Interpreter
14.5 An Introduction to Scheme
14.5.1 Origins of Scheme
14.5.2 Primitive Functions
14.5.3 Functions for Constructing Functions
14.5.4 Control Flow
14.5.5 Example Scheme Functions
14.5.6 Functional Forms
14.5.6.1 Functional Composition
14.5.6.2 An Apply-to-All Functional Form
14.5.7 Functions That Build Code
14.5.8 Imperative Features of Scheme
14.6 COMMON LISP
14.7 ML
14.8 Haskell
14.9 Applications of Functional Languages
14.10 A Comparison of Functional and Imperative Languages
Summary
Bibliographic Notes
Review Questions
Problem Set

Chapter 15 Logic Programming Languages

15.1 Introduction
15.2 A Brief Introduction to Predicate Calculus
15.2.1 Propositions
15.2.2 Clausal Form
15.3 Predicate Calculus and Proving Theorems
15.4 An Overview of Logic Programming
15.5 The Origins of Prolog
15.6 The Basic Elements of Prolog
15.6.1 Terms
15.6.2 Fact Statements
15.6.3 Rule Statements
15.6.4 Goal Statements

15.6.5 The Inferencing Process in Prolog
15.6.6 Simple Arithmetic
15.6.7 List Structures
15.7 Deficiencies of Prolog
15.7.1 Resolution Order Control
15.7.2 The Closed World Assumption
15.7.3 The Negation Problem
15.7.4 Intrinsic Limitations
15.8 Applications of Logic Programming
15.8.1 Relational Database Management Systems
15.8.2 Expert Systems
15.8.3 Natural Language Processing
15.8.4 Education
15.9 Conclusions
Summary
Bibliographic Notes
Review Questions
Problem Set

Answers to Selected Problems

Chapter 1

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. One alternative for the syntax to terminate a control statement is to use a single special word, such as the **`END`** of Modula-2, to terminate all control statements. Another is to use different special words to terminate different control statements, such as the **`end if`** and **`end loop`** of Ada. The advantage of the Modula-2 approach is simplicity--with only one way to end a control statement, writing programs is simplified; also, the language itself is simpler by having fewer special words. The disadvantage of this approach is that it results in less readable programs. This is because when **`END`** is read, the reader must search backwards through the program text to determine which kind of control statement is being terminated. When a special terminate for each control statement is used, the reader immediately knows the kind of control statement being terminated.

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 or 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 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.

Chapter 2

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.

11. 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.

12. 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.

15. 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.

19. 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

2a. <proc_head> → **procedure** <id> [(<formal_paramter> {; <formal_parameter>})]

<formal_parameter> → <var_param> | <value_param> | <proc_param>

| <function_param>

<var_param> → **var** <id_list> : <type_id>

<value_param> → <id_list> : <type_id>

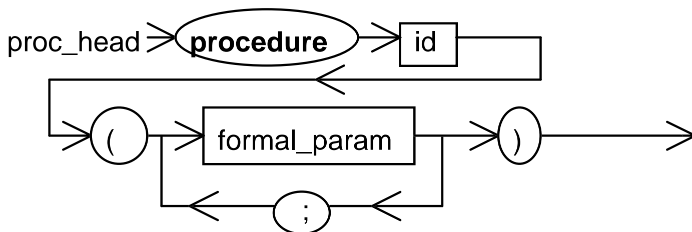
<id_list> → <id> { , <id>}

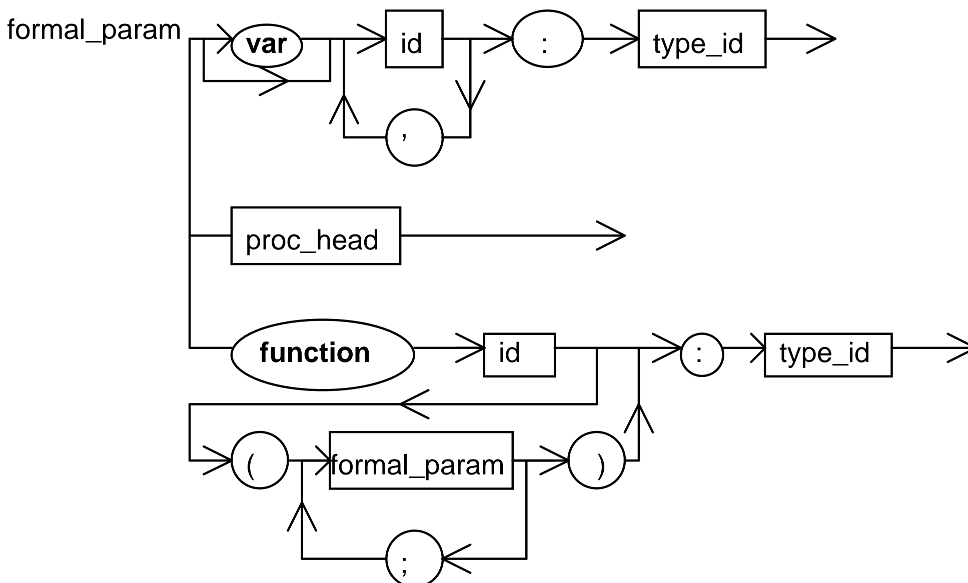
<proc_param> → <proc_head>

<function_param> → <function_head>

<function_head> → **function** <id> [(<formal_parameter> {; <formal_parameter<})]

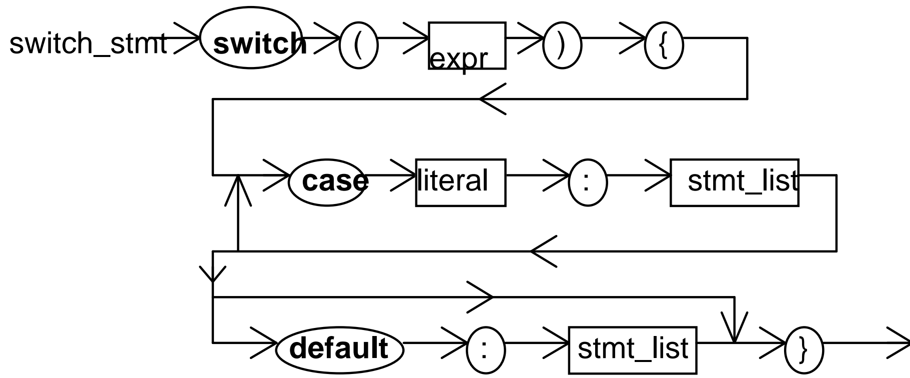
: <type_id>





2c. <switch_stmt> → **switch** (<expr>) { **case** <literal> : <stmt_list>

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



3.

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

=> A := <expr>

=> A := <id> * <expr>

=> A := A * <expr>

=> A := A * (<expr>)

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

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

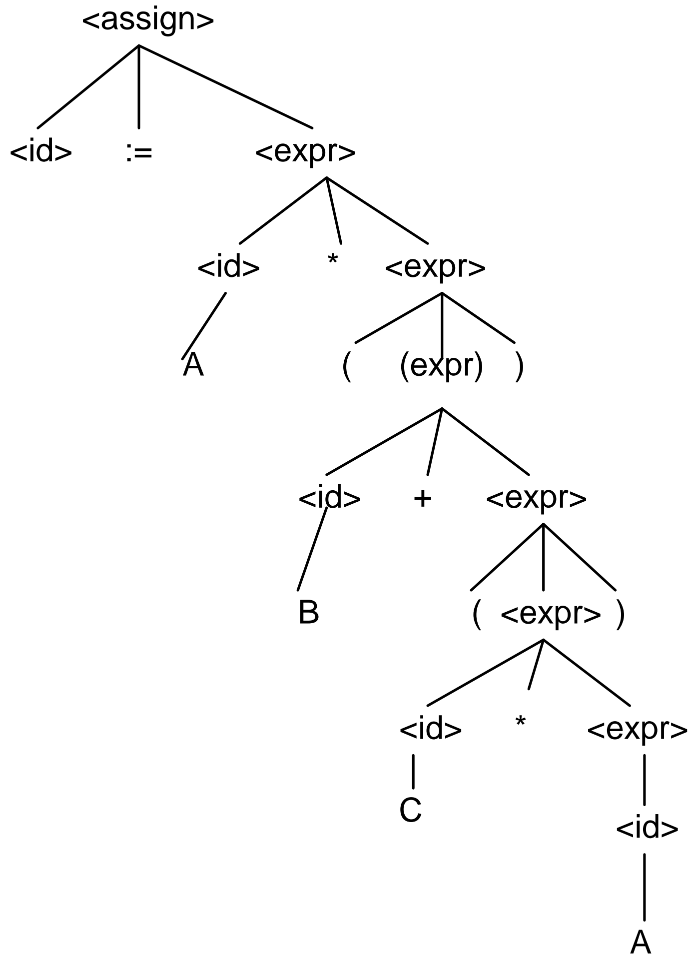
=> A := A * (B + (<expr>))

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

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

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

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



4.

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

=> A := <expr>

=> A := <term>

=> A := <factor> * <term>

=> A := (<expr>) * <term>

=> A := (<expr> + <term>) * <term>

=> A := (<term> + <term>) * <term>

=> A := (<factor> + <term>) * <term>

=> A := (<id> + <term>) * <term>

=> A := (A + <term>) * <term>

=> A := (A + <factor>) * <term>

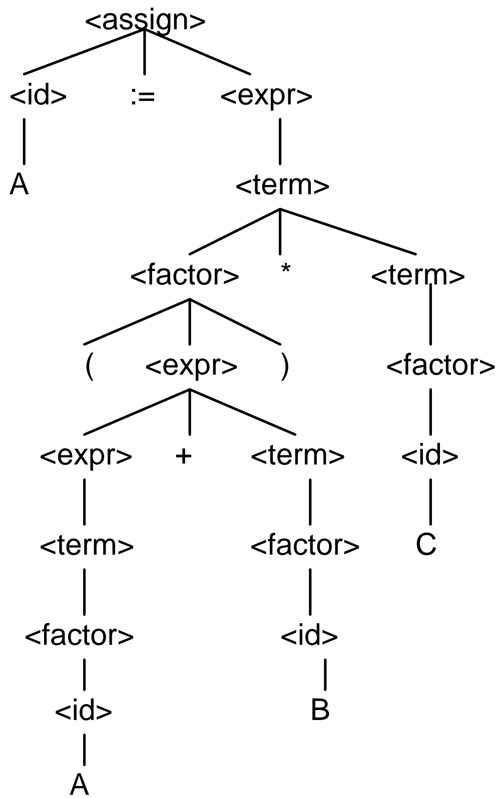
=> A := (A + <id>) * <term>

=> A := (A + B) * <term>

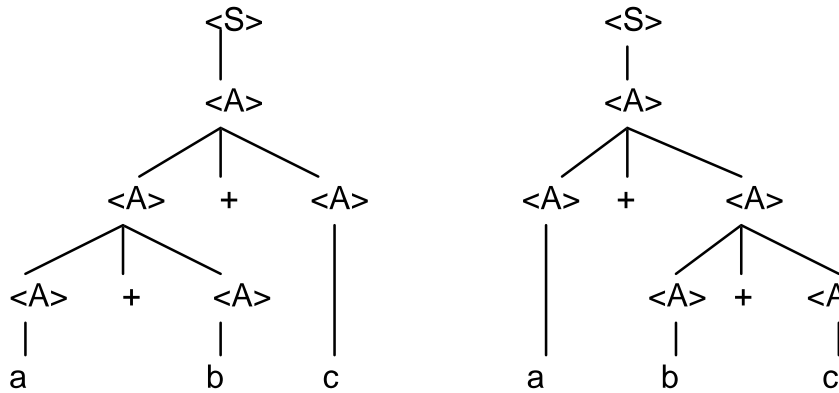
=> A := (A + B) * <factor>

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

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



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



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

<factor> → <id>

with

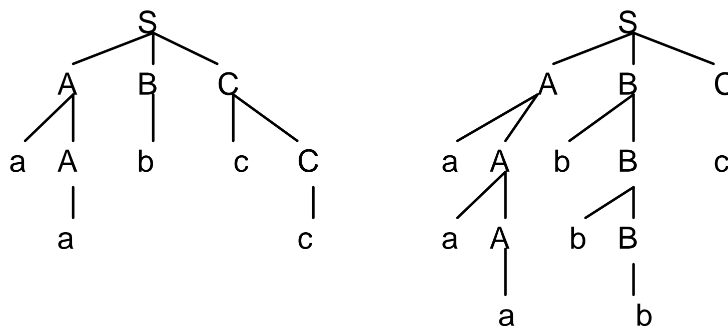
<factor> → + <id>

| - <id>

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

10. S → a S b | a b

11.



12.

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

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

(b) **for** I := first **downto** last **do**

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

(c) (FORTRAN 77 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:        ...
```

13.

(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

14.

(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

15a. M_{pf}(for var := init_expr to final_expr do L, s) $\triangleq$

if VARMAP(i, s) = **undef** for var or some i in init_expr or final_expr

then **error**

else if M_e(init_expr, s) > M_e(final_expr, s)

then s

else M_l(while init_expr - 1 <= final_expr do L, M_a(var := init_expr + 1, s))

15b. M_r(repeat L until B) $\triangleq$

if M_b(B, s) = **undef**

then **error**

else if M_{sl}(L, s) = **error**

then **error**

else if M_b(B, s) = **true**

then M_{sl}(L, s)

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

15c. M_b(B, s) $\triangleq$ if VARMAP(i, s) = **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)

15d. M_{cf}(for (expr1; expr2; expr3) L, s) $\triangleq$

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

then **error**

else if M_e (expr2, M_e (expr1, s)) = 0

then s

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

M_{help} (expr2, 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) = **error**

then s

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

16. The value of an intrisic 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.

17. Replace the second semantic rule with:

<var>[2].env ← <expr>.env

<var>[3].env ← <expr>.env

<expr>.actual_type ← <var>[2].actual_type

predicate: <var>[2].actual_type = <var>[3].actual_type

Chapter 4

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 wiht 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`:

Variable	Where Declared
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 5

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.

7. 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.

8. The only justification for the `->` operator in C and C++ is writability. It is slightly easier to write `p -> q` than `(*p).q`.

10. 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.

13. Require all references to enumeration constants to be qualified with the type name, as in `colors.blue`.

14. Let the subscript ranges of the three dimensions be named `min(1)`, `min(2)`, `min(3)`, `max(1)`, `max(2)`, and `max(3)`. Let the sizes of the subscript ranges be `size(1)`, `size(2)`, and `size(3)`. Assume the element size is 1.

Row Major Order:

```
location(a[i,j,k]) = (address of a[min(1),min(2),min(3)])

    +((i-min(1))*size(3) + (j-min(2)))*size(2) + (k-min(3))
```

Column Major Order:

```
location(a[i,j,k]) = (address of a[min(1),min(2),min(3)])

    +((k-min(3))*size(1) + (j-min(2)))*size(2) + (i-min(1))
```

15. 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.

22. 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 6

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 268.

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. The `BOOLEAN` variable `FOUND` is set to `TRUE` if `VALUE` is found in `LIST`. If found, `VALUE` is at `INDEX` of `LIST`.

```
INDEX := 1;

while (INDEX < LENGTH) and (LIST[INDEX] <> VALUE) do

  INDEX := INDEX + 1;

if (INDEX = LENGTH) and (LIST[LENGTH] <> VALUE) then

  FOUND := FALSE

else FOUND := TRUE;
```

10.

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

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

11.

(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 or ( c = ( d and e )`^{1} `)`^{2} `)`^{3} `)`^{4}

(e) `( a > ( xor ( c or ( d <= 17 )`^{1} `)`^{2} `)`^{3} `)`^{4}

(f) `( - ( a + b )`^{1} `)`^{2}

12. <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>

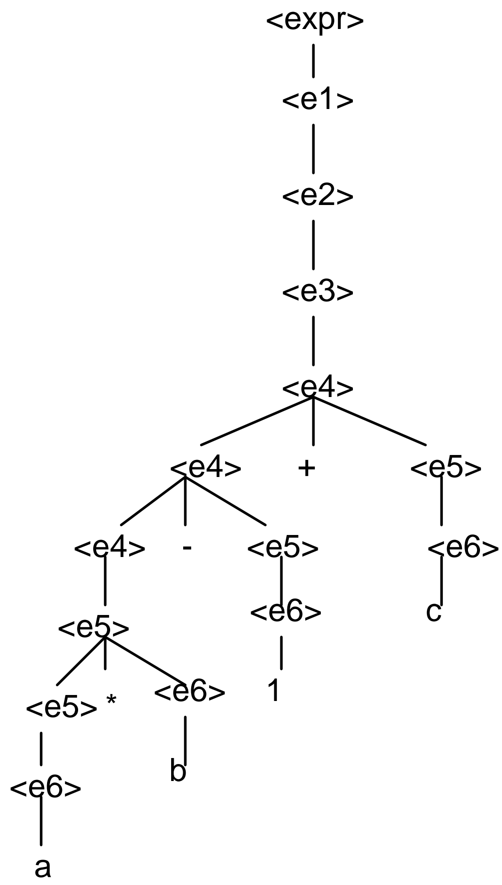
<e3> → <e4>

<e4> → <e4> + <e5> | <e4> - <e5> | <e4> & <e5> | <e4> mod <e5> | <e5>

<e5> → <e5> * <e6> | <e5> / <e6> | not <e5> | <e6>

<e6> → a | b | c | d | e | const | (<expr>)

13. (a)



13. (b)

```
                  <expr>
                    |
                   <e1>
                    |
                   <e2>
                    |
                   <e3>
                    |
                   <e4>
                    |
                   <e5>
              /     |      \
          <e5>     mod     <e6>
         /  |   \            |
     <e5>   /   <e6>         c
    /  |  \       |
<e5>   *  <e6>    c
  |       /  |    \
<e6>     (  <expr>  )
  |           |
  a          <e1>
              |
             <e2>
              |
             <e3>
              |
             <e4>
           /  |  \
       <e4>   -   <e5>
         |          |
       <e5>       <e6>
         |          |
       <e6>         1
         |
         b
```

13. (c)

```
<expr>
└── <e1>
    └── <e2>
        └── <e3>
            └── <e4>
                ├── <e4>
                │   └── <e5>
                │       ├── <e5>
                │       │   └── <e6>
                │       │       ├── (
                │       │       ├── <expr>
                │       │       │   └── <e1>
                │       │       │       └── <e2>
                │       │       │           └── <e3>
                │       │       │               └── <e4>
                │       │       │                   ├── <e4>
                │       │       │                   │   └── <e5>
                │       │       │                   │       └── <e6>
                │       │       │                   │           └── a
                │       │       │                   ├── -
                │       │       │                   └── <e5>
                │       │       │                       └── <e6>
                │       │       │                           └── b
                │       │       └── )
                │       ├── /
                │       └── <e6>
                │           └── c
                ├── &
                └── <e5>
                    └── <e6>
                        ├── (
                        ├── <expr>
                        │   └── <e1>
                        │       └── <e2>
                        │           └── <e3>
                        │               └── <e4>
                        │                   ├── <e4>
                        │                   │   └── <e5>
                        │                   │       ├── <e5>
                        │                   │       │   ├── <e5>
                        │                   │       │   │   └── <e6>
                        │                   │       │   │       └── d
                        │                   │       │   ├── *
                        │                   │       │   └── <e6>
                        │                   │       │       └── e
                        │                   │       ├── /
                        │                   │       └── <e6>
                        │                   │           └── a
                        │                   ├── -
                        │                   └── <e5>
                        │                       └── <e6>
                        │                           └── 3
                        └── )
```

13. (d)

```
                 <expr>
              /     |     \
        <expr>     and     <e1>
          |                  |
        <e1>               <e2>
      /   |   \              |
  <e1>   or   <e2>         <e3>
    |        /  |  \         |
  <e2>   <e2>   =  <e3>    <e4>
    |      |         |       |
  <e3>   <e3>      <e4>    <e5>
  /  |     |         |       |
 -  <e4> <e4>      <e5>    <e6>
     |     |         |       |
   <e5>  <e5>      <e6>      e
     |     |         |
   <e6>  <e6>        d
     |     |
     a     c
```

13. (e)

<expr>

<expr> xor <e1>

<e1> <e1> or <e2>

<e2> <e2> <e2> <= <e3>

<e2> > <e3> <e3> <e3> <e4>

<e3> <e4> <e4> <e4> <e5>

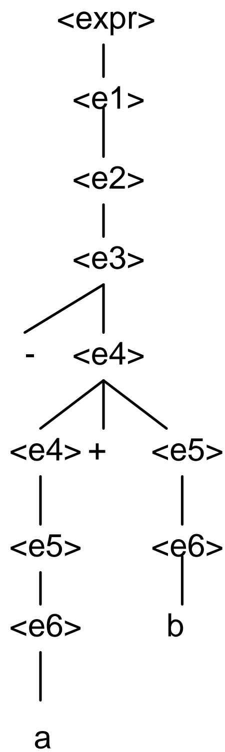
<e4> <e5> <e5> <e5> <e6>

<e5> <e6> <e6> <e6> 17

<e6> b c d

a

13. (f)



14. a. `SUM1 = 24` and `SUM2 = 26`

b. `SUM1 = 26` and `SUM2 = 24`

15. `sum1` is 48; `sum2` is `48`

The reason why both answers are the same is simple: The compiler evaluates the operand that is a function call first, regardless of whether it is the left or right operand in an expression.

Chapter 7

1. The justification for the three-way selection of FORTRAN I is that the primary goal of the language was efficiency. Because the IBM 704 had a single machine instruction to implement the three-way branch, there was simply no alternative that would not bring a much heavier runtime cost. Some sort of selection was obviously necessary, and the three-way branch was the fastest, by far, and thus the best alternative. The other factor to consider is that at the time no one had given a great deal of thought to the goodness or badness of branching and selection structures.

2. Label variables are a convenient way of providing exception handling. A procedure simply returns to the label variable parameter when it detects an error, without knowing anything about the program unit that called it. The caller can handle the error.

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

a. A list of values is to be added to a `SUM`, but the loop is to be exited if `SUM` exceeds some prescribed value.

b. 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. The FORTRAN computed `GOTO` is far less readable than the Pascal **`case`**, because **`case`** is encapsulated and therefore does not require the reader to look at parts of the program text other than the statements following the **`case`** down to and including the **`end`**. The FORTRAN computed `GOTO` is less reliable than the Pascal **`case`** for the same reason the general use of goto's is unreliable. Also, the explicit `GOTO` at the end of each selectable group of a computed `GOTO` can easily be forgotten, leading to erroneous control flow. This is not a problem with **`case`**, because the goto at the end of each selectable group is implicit.

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

10.

(a)
```
for k := (j + 13) div 27 to 10 do

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

(b)
```
  DO 10, K = (J + 13) / 27, 10

  I = 3 * (K + 1) - 1

10 CONTINUE
```

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

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

```
   end loop;
```

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

11.

(a)
```
k := (j + 13.0) / 27.0;
   while k <= 10.0 do
     begin
     i := 3.0 * (k + 1.2) - 1.0;
     k := k + 1.2
     end
```

(b)
```
    DO 10, K = (J + 13.0) / 27.0, 10.0, 1.2
     I = 3.0 * (K + 1.2) - 1.0
  10 CONTINUE
```

(c)
```
k := (j + 13.0) / 27.0;
    while (k <= 10.0) loop
      i := 3.0 * (k + 1.2) - 1.0;
      k := k + 1.2;
    end loop;
```

(d)
```
for (k = (j + 13.0) / 27.0; k <= 10.0;
        k = k + 1.2, i = 3.0 * k - 1)
```

12.

(a)
```
 case k of
     1, 2: j := 2 * k - 1;
     3, 5: j := 3 * k + 1;
     4: j := 4 * k - 1;
     6, 7, 8: j := k - 2;
     else writeln('Error in case, k = ', k)
```

```
   end
```

(b)
```
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
```

(c)
```
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;
```

(d)
```
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);
  }
```

13. (a) `i = 2`

    (b) `i = 2, 4`

    (c) `i = 2, 4, 6, 8, ...`

    (d) `i = 2`

16.
```
key := index - 1;
if (key = 2) or (key = 4) then
    even := even + 1;
  else if (key = 1) or (key = 3) then
    odd := odd + 1;
  else if (key = 0) then
    zero := zero + 1;
  else error := true;
```

17. `j = -3;`

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

18. (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;
```

(Pascal)

```
  i := 1;
  getout := false;
  while i <= n and getout = false do
    begin
    flag := true;
    for j := 1 to n do
      if x[i, j] <> 0 then
        flag := false;
    if flag = true then
      begin
      writeln('First all-zero row is: ', i);
      getout := true
```

```
  end
end;
```

19. 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.

Chapter 8

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. The reason blank `COMMON` in FORTRAN cannot be statically initialized is that in early implementations of FORTRAN, blank `COMMON` space was overlayed on the compiler during compilation. This was required because of small memories of computers of the time and the desire to not write object code out to secondary memory during its creation.

5. This cannot be done in Pascal, but it can be done in FORTRAN by letting the page number be a local variable that is statically initialized in the subprogram. It can be done in C using a static local variable.

6. Multiple entries in a subprogram provides a method of reusing parts of the subprogram in certain calls. For example, if a subprogram consists of three parts, which we call `A`, `B`, and `C`, some calls could cause the execution of just `C`, and others could cause the execution of `B` and `C`, or even `A`, `B`, and `C`.

9. (a) `LIST` = 3, 1.

(b) `LIST` = 5, 1.

(c) `LIST` = 3, 5.

(d) `LIST` = 5, 1.

(We assume that the address of the actual parameter is computed at subprogram entry, and is not recomputed at subprogram termination.)

10. 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.	b.	c.	d.
2, 1, 3, 5, 7, 9	1, 2, 3, 5, 7, 9	1, 2, 3, 5, 7, 9	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, 3, 1, 5, 7, 9
2, 1, 3, 5, 7, 9	3, 1, 2, 5, 7, 9	3, 1, 3, 5, 2, 9	subscript range error
			5, 1, 3, 5, 7, 9

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

Chapter 9

1. Algorithm for display maintenance upon procedure call when parameters can be procedure names:

a. Save, in the new activation record instance, a copy of the entire display.

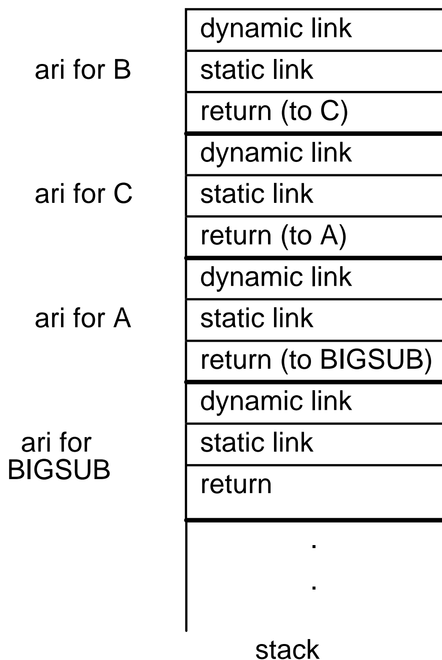
b. Look up the complete static ancestry of the called procedure, making a list of the procedure names in the order of largest scope first.

c. Find the most recent activation record instances of each of the static ancester procedures and construct a complete new display for the referencing environment of the called procedure, including a pointer to the new activation record.

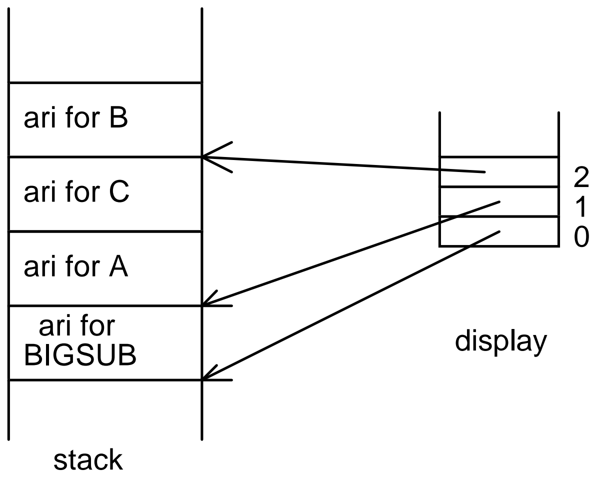
2. The algorithm for display maintenance upon procedure exit when parameters can be procedure names is simple:

> Replace the entire display with the saved display from the activation record of the procedure being exited.

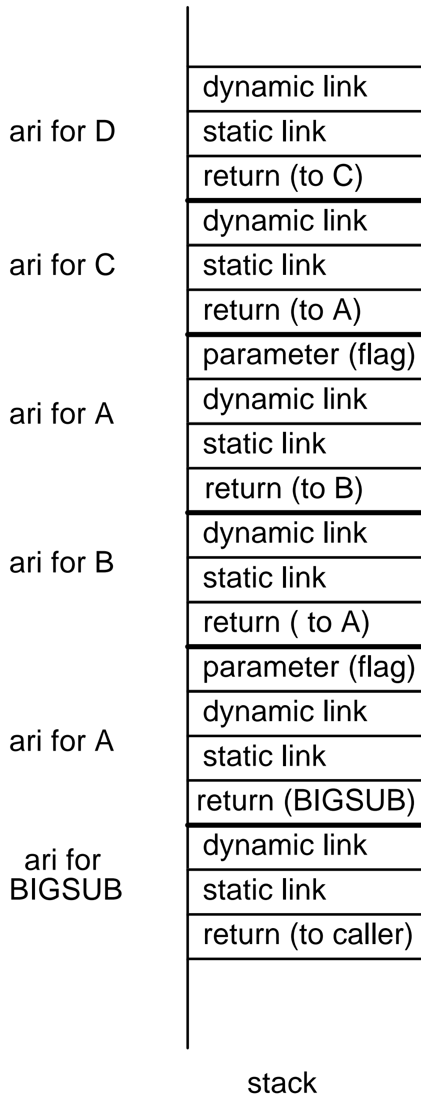
3.



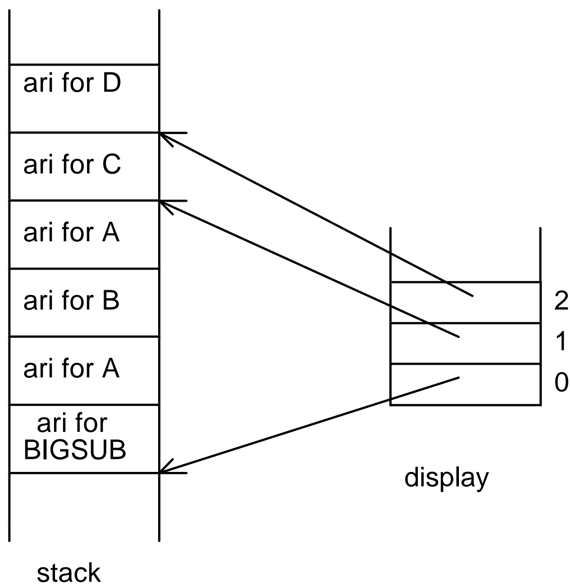
4.



5.



6.



7. Two consecutive calls to the same procedure would use the same stack location for a given local variable. If no other stack uses occur between the calls, the variable would have the last value of the first activation at the beginning of the second activation. Note that implementations rarely actually erase any information in the stack.

8. 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.

9. 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.

10. Exactly as in Problem 7, except that the nesting_depth is used to move the display top pointer, as well as the stack top pointer.

11. Simply assume, in the access mechanism, that the zeroeth entry in the display indicates a local (which it does), and not bother fetching the display pointer in those cases.

12. 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 10

2. The Pascal implementation would lack information hiding. Access could be made to the stack either through the provided mechanisms or directly.

4. The FORTRAN 77 version would be less reliable because of the lack of information hiding. Also, it would be far less flexible, because the Ada generic version can be easily instantiated for different type elements. The FORTRAN 77 verison would need to be modified with an editor to allow it to be used for elements of different types.

9. 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 11

1.
```
[count < 100]
  whileTrue:  [sum <- sum / (2 * count - 1).
               count <- count + 1]
```

2.
```
index <- 10.
[index > 0]
  whileTrue:  [sum <- sum + index.
               index <- index - 1]
```

3.
```
[count < 10]
    ifTrue: [answer <- 1]
    ifFalse: [answer <- 0.
              count <- 0]
```

4. Same as 1.

5. Same as 2.

6. Same as 3.

Chapter 12

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 allowed the other to continue.

5. The main problem with busy waiting is that machine cycles are wasted in the process.

6. 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.

9. 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.

Chapter 13

1. Pascal programs can detect and handle only end-of-file conditions.

6. 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.

7. 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.

15. `throw i` is handled by `catch(int)` in `fun2`

`throw f` is handled by `catch(float)` in `fun1`

`throw j` is handled by `catch(int)` in `fun2`

`throw g` is handled by `catch(float)` in `fun2`

Chapter 14

1.
```
(DEFINE (reverse lis)
  (COND
    ((NULL? lis) '())
    (ELSE (APPEND (reverse (CDR lis)) (CONS (CAR lis) () )))
))
```

2.
```
(DEFINE (eqstruc lis1 lis2)
  (COND
    ((NOT (LIST? lis1)) (NOT (LIST? lis2)))
    ((NOT (LIST? lis2)) '())
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) '())
    ((eqstruc (CAR lis1) (CAR lis2)
              (eqstruc (CDR lis1) (CDE lis2)))
    (ELSE '())
 ))
```

3.
```
(DEFINE (union lis1 lis2)
  (COND
    ((NULL? lis1) lis2)
    ((MEMBER (CAR lis1) lis2) (union (CDR lis1) lis2))
    (ELSE (union (CDR lis1) (CONS (CAR lis1) lis2)))
 ))
```

4.
```
(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))))
  ))
```

5.
```
(DEFINE (remove_second lis)
   (COND
     ((NULL? (CDR lis)) '())
     (ELSE (CONS (CAR lis) (CDDR lis)))
  ))
```

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

12. `x` returns the number of non-`NIL` atoms in the given list.

Chapter 15

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 tempoarary.

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.

5.
```
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.

6.
```
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).
```

8. 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.

9. 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.