



SECOND EDITION

PROGRAMMING LANGUAGE PRAGMATICS

Michael L. Scott



Programming Language Pragmatics is a very well-written textbook that captures the interest and focus of the reader. Each of the topics is very well introduced, developed, illustrated, and integrated with the preceding and following topics. The author employs up-to-date information and illustrates each concept by using examples from various programming languages. The level of presentation is appropriate for students, and the pedagogical features help make the chapters very easy to follow and refer back to.

—Kamal Dahbur, DePaul University

Programming Language Pragmatics strikes a good balance between depth and breadth in its coverage on of both classic and updated languages.

—Jingke Li, Portland State University

Programming Language Pragmatics is the most comprehensive book to date on the theory and implementation of programming languages. Prof. Scott writes well, conveying both unifying fundamental principles and the differing design choices found in today's major languages. Several improvements give this new second edition a more user-friendly format.

—William Calhoun, Bloomsburg University

Prof. Scott has met his goal of improving Programming Language Pragmatics by bringing the text up-to-date and making the material more accessible for students. The addition of the chapter on scripting languages and the use of XML to illustrate the use of scripting languages is unique in programming languages texts and is an important addition.

—Eileen Head, Binghamton University

This new edition of Programming Language Pragmatics does an excellent job of balancing the three critical qualities needed in a textbook: breadth, depth, and clarity. Prof. Scott manages to cover the full gamut of programming languages, from the oldest to the newest with sufficient depth to give students a good understanding of the important features of each, but without getting bogged down in arcane and idiosyncratic details. The new chapter on scripting languages is a most valuable addition as this class of languages continues to emerge as a major mainstream technology. This book is sure to become the gold standard of the field.

—Christopher Vickery, Queens College of CUNY

Programming Language Pragmatics not only explains language concepts and implementation details with admirable clarity, but also shows how computer architecture and compilers influence language design and implementation... This book shows that programming languages are the true center of computer science—the bridges spanning the chasm between programmer and machine.

—From the Foreword by Jim Larus, Microsoft Research

Programming Language Pragmatics
SECOND EDITION

About the Author

Michael L. Scott is a professor and past chair of the Department of Computer Science at the University of Rochester. He received his Ph.D. in computer sciences in 1985 from the University of Wisconsin–Madison. His research interests lie at the intersection of programming languages, operating systems, and high-level computer architecture, with an emphasis on parallel and distributed computing. He is the designer of the Lynx distributed programming language and a codesigner of the Charlotte and Psyche parallel operating systems, the Bridge parallel file system, and the Cashmere and InterWeave shared memory systems. His MCS mutual exclusion lock, codesigned with John Mellor-Crummey, is used in a variety of commercial and academic systems. Several other algorithms, codesigned with Maged Michael and Bill Scherer, appear in the `java.util.concurrent` standard library.

Dr. Scott is a member of the Association for Computing Machinery, the Institute of Electrical and Electronics Engineers, the Union of Concerned Scientists, and Computer Professionals for Social Responsibility. He has served on a wide variety of program committees and grant review panels, and has been a principal or coinvestigator on grants from the NSF, ONR, DARPA, NASA, the Departments of Energy and Defense, the Ford Foundation, Digital Equipment Corporation (now HP), Sun Microsystems, Intel, and IBM. He has contributed to the GRE advanced exam in computer science, and is the author of some 95 refereed publications. In 2003 he chaired the ACM Symposium on Operating Systems Principles. He received a Bell Labs Doctoral Scholarship in 1983 and an IBM Faculty Development Award in 1986. In 2001 he received the University of Rochester’s Robert and Pamela Goergen Award for Distinguished Achievement and Artistry in Undergraduate Teaching.

Programming Language Pragmatics

SECOND EDITION

Michael L. Scott

*Department of Computer Science
University of Rochester*



AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an imprint of Elsevier



Publishing Director: Michael Forster
Publisher: Denise Penrose
Publishing Services Manager: Andre Cuello
Assistant Publishing Services Manager
Project Manager: Carl M. Soares
Developmental Editor: Nate McFadden
Editorial Assistant: Valerie Witte
Cover Design: Ross Carron Designs
Cover Image: © Brand X Pictures/Corbin Images
Text Design: Julio Esperas
Composition: VTEX
Technical Illustration: Dartmouth Publishing Inc.
Copyeditor: Debbie Prato
Proofreader: Phyllis Coyne et al. Proofreading Service
Indexer: Ferreira Indexing Inc.
Interior printer: Maple-Vail
Cover printer: Phoenix Color

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

© 2006 by Elsevier Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, scanning, or otherwise—with prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail: permissions@elsevier.com. You may also complete your request on-line via the Elsevier homepage (<http://elsevier.com>) by selecting "Customer Support" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data
Application Submitted

ISBN 13: 978-0-12-633951-2
ISBN10: 0-12-633951-1

For information on all Morgan Kaufmann publications,
visit our Web site at www.mkp.com or www.books.elsevier.com

Printed in the United States of America
05 06 07 08 09 5 4 3 2 1

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER BOOK AID
International Sabre Foundation

To the roses now in full bloom.

Foreword

Computer science excels at layering abstraction on abstraction. Our field’s facility for hiding details behind a simplified interface is both a virtue and a necessity. Operating systems, databases, and compilers are very complex programs shaped by forty years of theory and development. For the most part, programmers need little or no understanding of the internal logic or structure of a piece of software to use it productively. Most of the time, ignorance is bliss.

Opaque abstraction, however, can become a brick wall, preventing forward progress, instead of a sound foundation for new artifacts. Consider the subject of this book, programs and programming languages. What happens when a program runs too slowly, and profiling cannot identify any obvious bottleneck or the bottleneck does not have an algorithmic explanation? Some potential problems are the translation of language constructs into machine instructions or how the generated code interacts with a processor’s architecture. Correcting these problems requires an understanding that bridges levels of abstraction.

Abstraction can also stand in the path of learning. Simple questions—how programs written in a small, stilted subset of English can control machines that speak binary or why programming languages, despite their ever growing variety and quantity, all seem fairly similar—cannot be answered except by diving into the details and understanding computers, compilers, and languages.

A computer science education, taken as a whole, can answer these questions. Most undergraduate programs offer courses about computer architecture, operating systems, programming language design, and compilers. These are all fascinating courses that are well worth taking—but difficult to fit into most study plans along with the many other rich offerings of an undergraduate computer science curriculum. Moreover, courses are often taught as self-contained subjects and do not explain a subject’s connections to other disciplines.

This book also answers these questions, by looking beyond the abstractions that divide these subjects. Michael Scott is a talented researcher who has made major contributions in language implementation, run-time systems, and computer architecture. He is exceptionally well qualified to draw on all of these fields

to provide a coherent understanding of modern programming languages. This book not only explains language concepts and implementation details with admirable clarity, but also shows how computer architecture and compilers influence language design and implementation. Moreover, it neatly illustrates how different languages are actually used, with realistic examples to clearly show how problem domains shape languages as well.

In interest of full disclosure, I must confess this book worried me when I first read it. At the time, I thought Michael's approach de-emphasized programming languages and compilers in the curriculum and would leave students with a superficial understanding of the field. But now, having reread the book, I have come to realize that in fact the opposite is true. By presenting them in their proper context, this book shows that programming languages are the true center of computer science—the bridges spanning the chasm between programmer and machine.

James Larus, Microsoft Research

Contents

Foreword	ix
Preface	xxiii

FOUNDATIONS

I Introduction	3
1.1 The Art of Language Design	5
1.2 The Programming Language Spectrum	8
1.3 Why Study Programming Languages?	11
1.4 Compilation and Interpretation	13
1.5 Programming Environments	21
1.6 An Overview of Compilation	22
1.6.1 Lexical and Syntax Analysis	23
1.6.2 Semantic Analysis and Intermediate Code Generation	25
1.6.3 Target Code Generation	28
1.6.4 Code Improvement	30
1.7 Summary and Concluding Remarks	31
1.8 Exercises	32
1.9 Explorations	33
1.10 Bibliographic Notes	35

2 Programming Language Syntax	37
2.1 Specifying Syntax	38
2.1.1 Tokens and Regular Expressions	39
2.1.2 Context-Free Grammars	42
2.1.3 Derivations and Parse Trees	43
2.2 Scanning	46
2.2.1 Generating a Finite Automaton	49
2.2.2 Scanner Code	54
2.2.3 Table-Driven Scanning	58
2.2.4 Lexical Errors	58
2.2.5 Pragmas	60
2.3 Parsing	61
2.3.1 Recursive Descent	64
2.3.2 Table-Driven Top-Down Parsing	70
2.3.3 Bottom-Up Parsing	80
2.3.4 Syntax Errors	CD I · 93
2.4 Theoretical Foundations	CD I3 · 94
2.4.1 Finite Automata	CD I3
2.4.2 Push-Down Automata	CD I6
2.4.3 Grammar and Language Classes	CD I7
2.5 Summary and Concluding Remarks	95
2.6 Exercises	96
2.7 Explorations	101
2.8 Bibliographic Notes	101
3 Names, Scopes, and Bindings	103
3.1 The Notion of Binding Time	104
3.2 Object Lifetime and Storage Management	106
3.2.1 Static Allocation	107
3.2.2 Stack-Based Allocation	109
3.2.3 Heap-Based Allocation	111
3.2.4 Garbage Collection	113
3.3 Scope Rules	114
3.3.1 Static Scope	115
3.3.2 Nested Subroutines	117
3.3.3 Declaration Order	119
3.3.4 Modules	124

3.3.5 Module Types and Classes	128
3.3.6 Dynamic Scope	131
3.4 Implementing Scope	CD 23 • 135
3.4.1 Symbol Tables	CD 23
3.4.2 Association Lists and Central Reference Tables	CD 27
3.5 The Binding of Referencing Environments	136
3.5.1 Subroutine Closures	138
3.5.2 First- and Second-Class Subroutines	140
3.6 Binding Within a Scope	142
3.6.1 Aliases	142
3.6.2 Overloading	143
3.6.3 Polymorphism and Related Concepts	145
3.7 Separate Compilation	CD 30 • 149
3.7.1 Separate Compilation in C	CD 30
3.7.2 Packages and Automatic Header Inference	CD 33
3.7.3 Module Hierarchies	CD 35
3.8 Summary and Concluding Remarks	149
3.9 Exercises	151
3.10 Explorations	157
3.11 Bibliographic Notes	158
4 Semantic Analysis	161
4.1 The Role of the Semantic Analyzer	162
4.2 Attribute Grammars	166
4.3 Evaluating Attributes	168
4.4 Action Routines	179
4.5 Space Management for Attributes	CD 39 • 181
4.5.1 Bottom-Up Evaluation	CD 39
4.5.2 Top-Down Evaluation	CD 44
4.6 Decorating a Syntax Tree	182
4.7 Summary and Concluding Remarks	187
4.8 Exercises	189
4.9 Explorations	193
4.10 Bibliographic Notes	194

5 Target Machine Architecture	195
5.1 The Memory Hierarchy	196
5.2 Data Representation	199
5.2.1 Computer Arithmetic	CD 54 • 199
5.3 Instruction Set Architecture	201
5.3.1 Addressing Modes	201
5.3.2 Conditions and Branches	202
5.4 Architecture and Implementation	204
5.4.1 Microprogramming	205
5.4.2 Microprocessors	206
5.4.3 RISC	207
5.4.4 Two Example Architectures: The x86 and MIPS	CD 59 • 208
5.4.5 Pseudo-Assembly Notation	209
5.5 Compiling for Modern Processors	210
5.5.1 Keeping the Pipeline Full	211
5.5.2 Register Allocation	216
5.6 Summary and Concluding Remarks	221
5.7 Exercises	223
5.8 Explorations	226
5.9 Bibliographic Notes	227
II CORE ISSUES IN LANGUAGE DESIGN	231
6 Control Flow	233
6.1 Expression Evaluation	234
6.1.1 Precedence and Associativity	236
6.1.2 Assignments	238
6.1.3 Initialization	246
6.1.4 Ordering Within Expressions	249
6.1.5 Short-Circuit Evaluation	252
6.2 Structured and Unstructured Flow	254
6.2.1 Structured Alternatives to <code>goto</code>	255
6.2.2 Continuations	259

6.3 Sequencing	260
6.4 Selection	261
6.4.1 Short-Circuited Conditions	262
6.4.2 Case/Switch Statements	265
6.5 Iteration	270
6.5.1 Enumeration-Controlled Loops	271
6.5.2 Combination Loops	277
6.5.3 Iterators	278
6.5.4 Generators in Icon	CD 69 • 284
6.5.5 Logically Controlled Loops	284
6.6 Recursion	287
6.6.1 Iteration and Recursion	287
6.6.2 Applicative- and Normal-Order Evaluation	291
6.7 Nondeterminacy	CD 72 • 295
6.8 Summary and Concluding Remarks	296
6.9 Exercises	298
6.10 Explorations	304
6.11 Bibliographic Notes	305
7 Data Types	307
7.1 Type Systems	308
7.1.1 Type Checking	309
7.1.2 Polymorphism	309
7.1.3 The Definition of Types	311
7.1.4 The Classification of Types	312
7.1.5 Orthogonality	319
7.2 Type Checking	321
7.2.1 Type Equivalence	321
7.2.2 Type Compatibility	327
7.2.3 Type Inference	332
7.2.4 The ML Type System	CD 81 • 335
7.3 Records (Structures) and Variants (Unions)	336
7.3.1 Syntax and Operations	337
7.3.2 Memory Layout and Its Impact	338
7.3.3 With Statements	CD 90 • 341
7.3.4 Variant Records	341

7.4 Arrays	349
7.4.1 Syntax and Operations	349
7.4.2 Dimensions, Bounds, and Allocation	353
7.4.3 Memory Layout	358
7.5 Strings	366
7.6 Sets	367
7.7 Pointers and Recursive Types	369
7.7.1 Syntax and Operations	370
7.7.2 Dangling References	379
7.7.3 Garbage Collection	383
7.8 Lists	389
7.9 Files and Input/Output	CD 93 • 392
7.9.1 Interactive I/O	CD 93
7.9.2 File-Based I/O	CD 94
7.9.3 Text I/O	CD 96
7.10 Equality Testing and Assignment	393
7.11 Summary and Concluding Remarks	395
7.12 Exercises	398
7.13 Explorations	404
7.14 Bibliographic Notes	405
8 Subroutines and Control Abstraction	407
8.1 Review of Stack Layout	408
8.2 Calling Sequences	410
8.2.1 Displays	CD 107 • 413
8.2.2 Case Studies: C on the MIPS; Pascal on the x86	CD 111 • 414
8.2.3 Register Windows	CD 119 • 414
8.2.4 In-Line Expansion	415
8.3 Parameter Passing	417
8.3.1 Parameter Modes	418
8.3.2 Call by Name	CD 122 • 426
8.3.3 Special Purpose Parameters	427
8.3.4 Function Returns	432
8.4 Generic Subroutines and Modules	434
8.4.1 Implementation Options	435
8.4.2 Generic Parameter Constraints	437

8.4.3 Implicit Instantiation	440
8.4.4 Generics in C++, Java, and C#	CD 125 · 440
8.5 Exception Handling	441
8.5.1 Defining Exceptions	443
8.5.2 Exception Propagation	445
8.5.3 Example: Phrase-Level Recovery in a Recursive Descent Parser	448
8.5.4 Implementation of Exceptions	449
8.6 Coroutines	453
8.6.1 Stack Allocation	455
8.6.2 Transfer	457
8.6.3 Implementation of Iterators	CD 135 · 458
8.6.4 Discrete Event Simulation	CD 139 · 458
8.7 Summary and Concluding Remarks	459
8.8 Exercises	460
8.9 Explorations	466
8.10 Bibliographic Notes	467
9 Data Abstraction and Object Orientation	469
9.1 Object-Oriented Programming	471
9.2 Encapsulation and Inheritance	481
9.2.1 Modules	481
9.2.2 Classes	484
9.2.3 Type Extensions	486
9.3 Initialization and Finalization	489
9.3.1 Choosing a Constructor	490
9.3.2 References and Values	491
9.3.3 Execution Order	495
9.3.4 Garbage Collection	496
9.4 Dynamic Method Binding	497
9.4.1 Virtual and Nonvirtual Methods	500
9.4.2 Abstract Classes	501
9.4.3 Member Lookup	502
9.4.4 Polymorphism	505
9.4.5 Closures	508
9.5 Multiple Inheritance	CD 146 · 511
9.5.1 Semantic Ambiguities	CD 148

9.5.2 Replicated Inheritance	CD 151
9.5.3 Shared Inheritance	CD 152
9.5.4 Mix-In Inheritance	CD 154
9.6 Object-Oriented Programming Revisited	512
9.6.1 The Object Model of Smalltalk	CD 158 • 513
9.7 Summary and Concluding Remarks	513
9.8 Exercises	515
9.9 Explorations	517
9.10 Bibliographic Notes	518

|||| ALTERNATIVE PROGRAMMING MODELS 521

10 Functional Languages	523
10.1 Historical Origins	524
10.2 Functional Programming Concepts	526
10.3 A Review/Overview of Scheme	528
10.3.1 Bindings	530
10.3.2 Lists and Numbers	531
10.3.3 Equality Testing and Searching	532
10.3.4 Control Flow and Assignment	533
10.3.5 Programs as Lists	535
10.3.6 Extended Example: DFA Simulation	537
10.4 Evaluation Order Revisited	539
10.4.1 Strictness and Lazy Evaluation	541
10.4.2 I/O: Streams and Monads	542
10.5 Higher-Order Functions	545
10.6 Theoretical Foundations	CD 166 • 549
10.6.1 Lambda Calculus	CD 168
10.6.2 Control Flow	CD 171
10.6.3 Structures	CD 173
10.7 Functional Programming in Perspective	549
10.8 Summary and Concluding Remarks	552

10.9 Exercises	552
10.10 Explorations	557
10.11 Bibliographic Notes	558
11 Logic Languages	559
11.1 Logic Programming Concepts	560
11.2 Prolog	561
11.2.1 Resolution and Unification	563
11.2.2 Lists	564
11.2.3 Arithmetic	565
11.2.4 Search/Execution Order	566
11.2.5 Extended Example: Tic-Tac-Toe	569
11.2.6 Imperative Control Flow	571
11.2.7 Database Manipulation	574
11.3 Theoretical Foundations	CD 180 · 579
11.3.1 Clausal Form	CD 181
11.3.2 Limitations	CD 182
11.3.3 Skolemization	CD 183
11.4 Logic Programming in Perspective	579
11.4.1 Parts of Logic Not Covered	580
11.4.2 Execution Order	580
11.4.3 Negation and the “Closed World” Assumption	581
11.5 Summary and Concluding Remarks	583
11.6 Exercises	584
11.7 Explorations	586
11.8 Bibliographic Notes	587
12 Concurrency	589
12.1 Background and Motivation	590
12.1.1 A Little History	590
12.1.2 The Case for Multithreaded Programs	593
12.1.3 Multiprocessor Architecture	597
12.2 Concurrent Programming Fundamentals	601
12.2.1 Communication and Synchronization	601
12.2.2 Languages and Libraries	603
12.2.3 Thread Creation Syntax	604

12.2.4 Implementation of Threads	613
12.3 Shared Memory	619
12.3.1 Busy-Wait Synchronization	620
12.3.2 Scheduler Implementation	623
12.3.3 Semaphores	627
12.3.4 Monitors	629
12.3.5 Conditional Critical Regions	634
12.3.6 Implicit Synchronization	638
12.4 Message Passing	642
12.4.1 Naming Communication Partners	642
12.4.2 Sending	646
12.4.3 Receiving	651
12.4.4 Remote Procedure Call	656
12.5 Summary and Concluding Remarks	660
12.6 Exercises	662
12.7 Explorations	668
12.8 Bibliographic Notes	669
13 Scripting Languages	671
13.1 What Is a Scripting Language?	672
13.1.1 Common Characteristics	674
13.2 Problem Domains	677
13.2.1 Shell (Command) Languages	677
13.2.2 Text Processing and Report Generation	684
13.2.3 Mathematics and Statistics	689
13.2.4 "Glue" Languages and General Purpose Scripting	690
13.2.5 Extension Languages	698
13.3 Scripting the World Wide Web	701
13.3.1 CGI Scripts	702
13.3.2 Embedded Server-Side Scripts	703
13.3.3 Client-Side Scripts	708
13.3.4 Java Applets	708
13.3.5 XSLT	712
13.4 Innovative Features	722
13.4.1 Names and Scopes	723
13.4.2 String and Pattern Manipulation	728
13.4.3 Data Types	736

13.4.4 Object Orientation	741
13.5 Summary and Concluding Remarks	748
13.6 Exercises	750
13.7 Explorations	755
13.8 Bibliographic Notes	756
IV A CLOSER LOOK AT IMPLEMENTATION	759
14 Building a Runnable Program	761
14.1 Back-End Compiler Structure	761
14.1.1 A Plausible Set of Phases	762
14.1.2 Phases and Passes	766
14.2 Intermediate Forms	CD 189 • 766
14.2.1 Diana	CD 189
14.2.2 GNU RTL	CD 192
14.3 Code Generation	769
14.3.1 An Attribute Grammar Example	769
14.3.2 Register Allocation	772
14.4 Address Space Organization	775
14.5 Assembly	776
14.5.1 Emitting Instructions	778
14.5.2 Assigning Addresses to Names	780
14.6 Linking	781
14.6.1 Relocation and Name Resolution	782
14.6.2 Type Checking	783
14.7 Dynamic Linking	CD 195 • 784
14.7.1 Position-Independent Code	CD 195
14.7.2 Fully Dynamic (Lazy) Linking	CD 196
14.8 Summary and Concluding Remarks	786
14.9 Exercises	787
14.10 Explorations	789
14.11 Bibliographic Notes	790

Preface

A course in computer programming provides the typical student's first exposure to the field of computer science. Most students in such a course will have used computers all their lives, for e-mail, games, web browsing, word processing, instant messaging, and a host of other tasks, but it is not until they write their first programs that they begin to appreciate how applications work. After gaining a certain level of facility as programmers (presumably with the help of a good course in data structures and algorithms), the natural next step is to wonder how programming languages work. This book provides an explanation. It aims, quite simply, to be the most comprehensive and accurate languages text available, in a style that is engaging and accessible to the typical undergraduate. This aim reflects my conviction that students will understand more, and enjoy the material more, if we explain what is really going on.

In the conventional "systems" curriculum, the material beyond data structures (and possibly computer organization) tends to be compartmentalized into a host of separate subjects, including programming languages, compiler construction, computer architecture, operating systems, networks, parallel and distributed computing, database management systems, and possibly software engineering, object-oriented design, graphics, or user interface systems. One problem with this compartmentalization is that the list of subjects keeps growing, but the number of semesters in a bachelor's program does not. More important, perhaps, many of the most interesting discoveries in computer science occur at the boundaries *between* subjects. The RISC revolution, for example, forged an alliance between computer architecture and compiler construction that has endured for 20 years. More recently, renewed interest in virtual machines has blurred the boundary between the operating system kernel and the language run-time system. The spread of Java and .NET has similarly blurred the boundary between the compiler and the run-time system. Programs are now routinely embedded in web pages, spreadsheets, and user interfaces.

Increasingly, both educators and practitioners are recognizing the need to emphasize these sorts of interactions. Within higher education in particular there is

a growing trend toward integration in the core curriculum. Rather than give the typical student an in-depth look at two or three narrow subjects, leaving holes in all the others, many schools have revised the programming languages and operating systems courses to cover a wider range of topics, with follow-on electives in various specializations. This trend is very much in keeping with the findings of the ACM/IEEE-CS *Computing Curricula 2001* task force, which emphasize the growth of the field, the increasing need for breadth, the importance of flexibility in curricular design, and the overriding goal of graduating students who “have a system-level perspective, appreciate the interplay between theory and practice, are familiar with common themes, and can adapt over time as the field evolves” [CR01, Sec. 11.1, adapted].

The first edition of *Programming Language Pragmatics* (PLP-1e) had the good fortune of riding this curricular trend. The second edition continues and strengthens the emphasis on integrated learning while retaining a central focus on programming language design.

At its core, PLP is a book about *how programming languages work*. Rather than enumerate the details of many different languages, it focuses on concepts that underlie all the languages the student is likely to encounter, illustrating those concepts with a variety of concrete examples, and exploring the tradeoffs that explain *why* different languages were designed in different ways. Similarly, rather than explain how to build a compiler or interpreter (a task few programmers will undertake in its entirety), PLP focuses on what a compiler does to an input program, and why. Language design and implementation are thus explored together, with an emphasis on the ways in which they interact.

Changes in the Second Edition

There were four main goals for the second edition:

1. Introduce new material, most notably scripting languages.
2. Bring the book up to date with respect to everything else that has happened in the last six years.
3. Resist the pressure toward rising textbook prices.
4. Strengthen the book from a pedagogical point of view, to make it more useful and accessible.

Item (1) is the most significant change in content. With the explosion of the World Wide Web, languages like Perl, PHP, Tcl/Tk, Python, Ruby, JavaScript, and XSLT have seen an enormous upsurge not only in commercial significance, but also in design innovation. Many of today’s graduates will spend more of their time working with scripting languages than with C++, Java, or C#. The new chapter on scripting languages (Chapter 13) is organized first by application domain (shell languages, text processing and report generation, mathematics and statistics, “glue” languages and general purpose scripting, extension languages, script-

ing the World Wide Web) and then by innovative features (names and scopes, string and pattern manipulation, high level data types, object orientation). References to scripting languages have also been added wherever appropriate throughout the rest of the text.

Item (2) reflects such key developments as the finalized C99 standard and the appearance of Java 5 and C# (version 2.0). Chapter 6 (Control Flow) now covers boxing, unboxing, and the latest iterator constructs. Chapter 8 (Subroutines) covers Java and C# generics. Chapter 12 (Concurrency) covers the Java 5 concurrency library (JSR 166). References to C# have been added where appropriate throughout. In keeping with changes in the microprocessor market, the ubiquitous Intel/AMD x86 has replaced the Motorola 68000 in the case studies of Chapters 5 (Architecture) and 8 (Subroutines). The MIPS case study in Chapter 8 has been updated to 64-bit mode. References to technological constants and trends have also been updated. In several places I have rewritten examples to use languages with which students are more likely to be familiar; this process will undoubtedly continue in future editions.

Many sections have been heavily rewritten to make them clearer or more accurate. These include coverage of finite automaton creation (2.2.1); declaration order (3.3.3); modules (3.3.4); aliases and overloading (3.6.1 and 3.6.2); polymorphism and generics (3.6.3, 7.1.2, 8.4, and 9.4.4); separate compilation (3.7); continuations, exceptions, and multilevel returns (6.2.1, 6.2.2, and 8.5); calling sequences (8.2); and most of Chapter 5.

Item (3) reflects Morgan Kaufmann's commitment to making definitive texts available at student-friendly prices. PLP-1e was larger and more comprehensive than competing texts, but sold for less. This second edition keeps a handle on price (and also reduces bulk) with high-quality paperback construction.

Finally, item (4) encompasses a large number of presentational changes. Some of these are relatively small. There are more frequent section headings, for example, and more historical anecdotes. More significantly, the book has been organized into four major parts:

Part I covers foundational material: (1) Introduction to Language Design and Implementation; (2) Programming Language Syntax; (3) Names, Scopes, and Bindings; (4) Semantic Analysis; and (5) Target Machine Architecture. The second and fifth of these have a fairly heavy focus on implementation issues. The first and fourth are mixed. The third introduces core issues in language design.

Part II continues the coverage of core issues: (6) Control Flow; (7) Data Types; (8) Subroutines and Control Abstraction; and (9) Data Abstraction and Object Orientation. The last of these has moved forward from its position in PLP-1e, reflecting the centrality of object-oriented programming to much of modern computing.

Part III turns to alternative programming models: (10) Functional Languages; (11) Logic Languages; (12) Concurrency; and (13) Scripting Languages. Functional and logic languages shared a single chapter in PLP-1e.

Part IV returns to language implementation: (14) Building a Runnable Program (code generation, assembly, and linking); and (15) Code Improvement (optimization).

The PLP CD

To minimize the physical size of the text, make way for new material, and allow students to focus on the fundamentals when browsing, approximately 250 pages of more advanced or peripheral material has been moved to a companion CD. For the most part (though not exclusively), this material comprises the sections that were identified as advanced or optional in PLP-1e.

The most significant single move is the entire chapter on code improvement (15). The rest of the moved material consists of scattered, shorter sections. Each such section is represented in the text by a brief introduction to the subject and an “In More Depth” paragraph that summarizes the elided material.

Note that the placement of material on the CD does *not* constitute a judgment about its technical importance. It simply reflects the fact that there is more material worth covering than will fit in a single volume or a single course. My intent is to retain in the printed text the material that is likely to be covered in the largest number of courses.

Design & Implementation Sidebars

PLP-1e placed a heavy emphasis on the ways in which language design constrains implementation options, and the ways in which anticipated implementations have influenced language design. PLP-2e uses more than 120 sidebars to make these connections more explicit. A more detailed introduction to these sidebars appears on page 7 (Chapter 1). A numbered list appears in Appendix B.

Numbered and Titled Examples

Examples in PLP-2e are intimately woven into the flow of the presentation. To make it easier to find specific examples, to remember their content, and to refer to them in other contexts, a number and a title for each is now displayed in a marginal note. There are nearly 900 such examples across the main text and the CD. A detailed list appears in Appendix C.

Exercise Plan

PLP-1e contained a total of 385 review questions and 312 exercises, located at the ends of chapters. Review questions in the second edition have been moved to the

ends of sections, closer to the material they cover, to make it easier to tell when one has grasped the central concepts. The total number of such questions has nearly doubled.

The problems remaining at the ends of chapters have now been divided into *Exercises* and *Explorations*. The former are intended to be more or less straightforward, though more challenging than the per-section review questions; they should be suitable for homework or brief projects. The exploration questions are more open-ended, requiring web or library research, substantial time commitment, or the development of subjective opinion. The total number of questions has increased from a little over 300 in PLP-1e to over 500 in the current edition. Solutions to the exercises (but not the explorations) are available to registered instructors from a password-protected web site: visit www.mkp.com/companions/0126339511/.

How to Use the Book

Programming Language Pragmatics covers almost all of the material in the PL “knowledge units” of the *Computing Curricula 2001* report [CR01]. The book is an ideal fit for the CS 341 model course (Programming Language Design), and can also be used for CS 340 (Compiler Construction) or CS 343 (Programming Paradigms). It contains a significant fraction of the content of CS 344 (Functional Programming) and CS 346 (Scripting Languages). Figure 1 illustrates several possible paths through the text.

For self-study, or for a full-year course (track F in Figure 1), I recommend working through the book from start to finish, turning to the PLP CD as each “In More Depth” section is encountered. The one-semester course at the University of Rochester (track R), for which the text was originally developed, also covers most of the book but leaves out most of the CD sections, as well as bottom-up parsing (2.3.3), message passing (12.4), web scripting (13.3), and most of Chapter 14 (Building a Runnable Program).

Some chapters (2, 4, 5, 14, 15) have a heavier emphasis than others on implementation issues. These can be reordered to a certain extent with respect to the more design-oriented chapters, but it is important that Chapter 5 or its equivalent be covered before Chapters 6 through 9. Many students will already be familiar with some of the material in Chapter 5, most likely from a course on computer organization. In this case the chapter may simply be skimmed for review. Some students may also be familiar with some of the material in Chapter 2, perhaps from a course on automata theory. Much of this chapter can then be read quickly as well, pausing perhaps to dwell on such practical issues as recovery from syntax errors, or the ways in which a scanner differs from a classical finite automaton.

A traditional programming languages course (track P in Figure 1) might leave out all of scanning and parsing, plus all of Chapters 4 and 5. It would also deemphasize the more implementation-oriented material throughout. In place

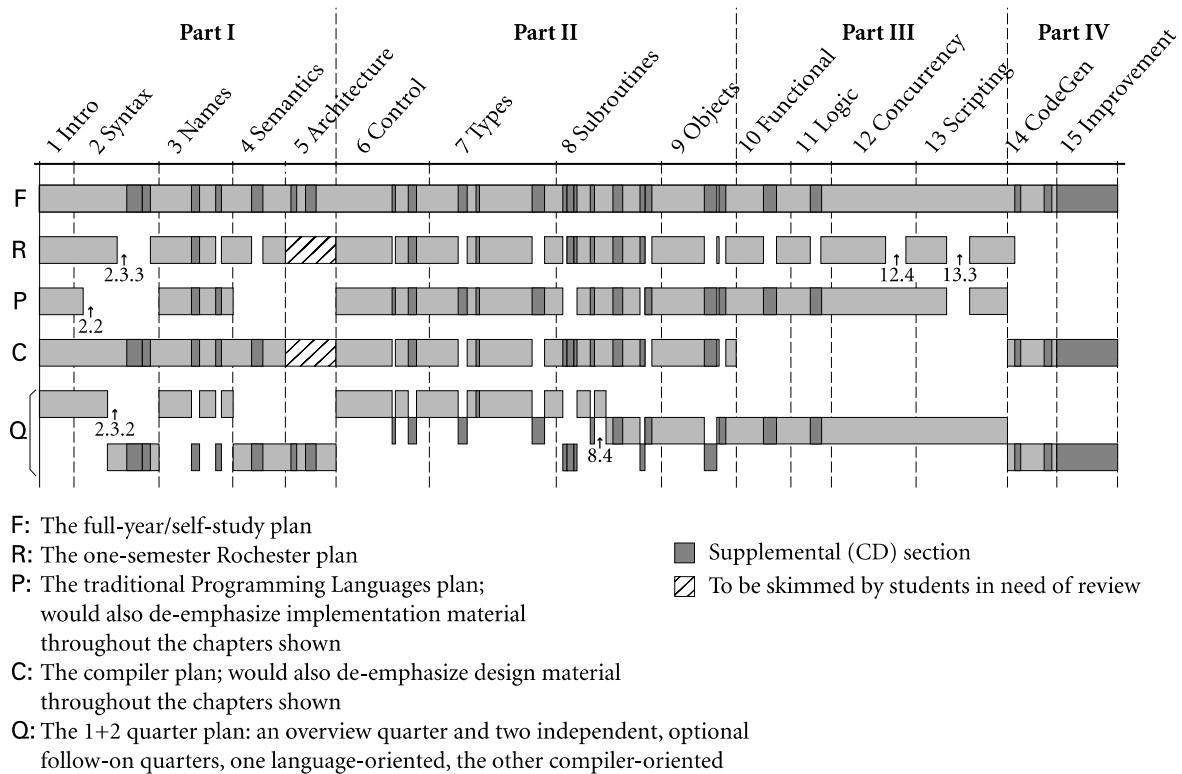


Figure 1 Paths through the text. Darker shaded regions indicate supplemental “In More Depth” sections on the PLP CD. Section numbers are shown for breaks that do not correspond to supplemental material.

of these it could add such design-oriented CD sections as the ML type system (7.2.4), multiple inheritance (9.5), Smalltalk (9.6.1), lambda calculus (10.6), and predicate calculus (11.3).

PLP has also been used at some schools for an introductory compiler course (track C in Figure 1). The typical syllabus leaves out most of Part III (Chapters 10 through 13), and deemphasizes the more design-oriented material throughout. In place of these it includes all of scanning and parsing, Chapters 14 and 15, and a slightly different mix of other CD sections.

For a school on the quarter system, an appealing option is to offer an introductory one-quarter course and two optional follow-on courses (track Q in Figure 1). The introductory quarter might cover the main (non-CD) sections of Chapters 1, 3, 6, and 7, plus the first halves of Chapters 2 and 8. A language-oriented follow-on quarter might cover the rest of Chapter 8, all of Part III, CD sections from Chapters 6 through 8, and possibly supplemental material on formal semantics, type systems, or other related topics. A compiler-oriented follow-on quarter might cover the rest of Chapter 2; Chapters 4–5 and 14–15, CD sec-

tions from Chapters 3 and 8–9, and possibly supplemental material on automatic code generation, aggressive code improvement, programming tools, and so on.

Whatever the path through the text, I assume that the typical reader has already acquired significant experience with at least one imperative language. Exactly which language it is shouldn't matter. Examples are drawn from a wide variety of languages, but always with enough comments and other discussion that readers without prior experience should be able to understand easily. Single-paragraph introductions to some 50 different languages appear in Appendix A. Algorithms, when needed, are presented in an informal pseudocode that should be self-explanatory. Real programming language code is set in "typewriter" font. Pseudocode is set in a sans-serif font.

Supplemental Materials

In addition to supplemental sections of the text, the PLP CD contains a variety of other resources:

- Links to language reference manuals and tutorials on the Web
- Links to Open Source compilers and interpreters
- Complete source code for all nontrivial examples in the book (more than 300 source files)
- Search engine for both the main text and the CD-only content

Additional resources are available at www.mkp.com/companions/0126339511/ (you may wish to check back from time to time). For instructors who have adopted the text, a password-protected page provides access to

- Editable PDF source for all the figures in the book
- Editable PowerPoint slides
- Solutions to most of the exercises
- Suggestions for larger projects

Acknowledgments for the Second Edition

In preparing the second edition I have been blessed with the generous assistance of a very large number of people. Many provided errata or other feedback on the first edition, among them Manuel E. Bermudez, John Boyland, Brian Cumming, Stephen A. Edward, Michael J. Eulenstein, Tayssir John Gabbour, Tommaso Galleri, Eileen Head, David Hoffman, Paul Ilardi, Lucian Ilie, Rahul Jain, Eric Joanis, Alan Kaplan, Les Lander, Jim Larus, Hui Li, Jingke Li, Evangelos Milios, Eduardo Pinheiro, Barbara Ryder, Nick Stuifbergen, Raymond Toal, Andrew Tolmach, Jens Troeger, and Robbert van Renesse. Zongyan Qiu prepared the Chinese translation, and found several bugs in the process. Simon Fillat maintained

the Morgan Kaufmann web site. I also remain indebted to the many other people, acknowledged in the first edition, who helped in that earlier endeavor, and to the reviewers, adopters, and readers who made it a success. Their contributions continue to be reflected in the current edition.

Work on the second edition began in earnest with a “focus group” at SIGCSE ’02; my thanks to Denise Penrose, Emilia Thiuri, and the rest of the team at Morgan Kaufmann for organizing that event, to the approximately two dozen attendees who shared their thoughts on content and pedagogy, and to the many other individuals who reviewed two subsequent revision plans.

A draft of the second edition was class tested in the fall of 2004 at eight different universities. I am grateful to Gerald Baumgartner (Louisiana State University), William Calhoun (Bloomsburg University), Betty Cheng (Michigan State University), Jingke Li (Portland State University), Beverly Sanders (University of Florida), Darko Stefanovic (University of New Mexico), Raymond Toal (Loyola Marymount University), Robert van Engelen (Florida State University), and all their students for a mountain of suggestions, reactions, bug fixes, and other feedback. Professor van Engelen provided several excellent end-of-chapter exercises.

External reviewers for the second edition also provided a wealth of useful suggestions. My thanks to Richard J. Botting (California State University, San Bernardino), Kamal Dahbur (DePaul University), Stephen A. Edwards (Columbia University), Eileen Head (Binghamton University), Li Liao (University of Delaware), Christopher Vickery (Queens College, City University of New York), Garrett Wollman (MIT), Neng-Fa Zhou (Brooklyn College, City University of New York), and Cynthia Brown Zickos (University of Mississippi). Garrett Wollman’s technical review of Chapter 13 was particularly helpful, as were his earlier comments on a variety of topics in the first edition. Sadly, time has not permitted me to do justice to everyone’s suggestions. I have incorporated as much as I could, and have carefully saved the rest for guidance on the third edition. Problems that remain in the current edition are entirely my own.

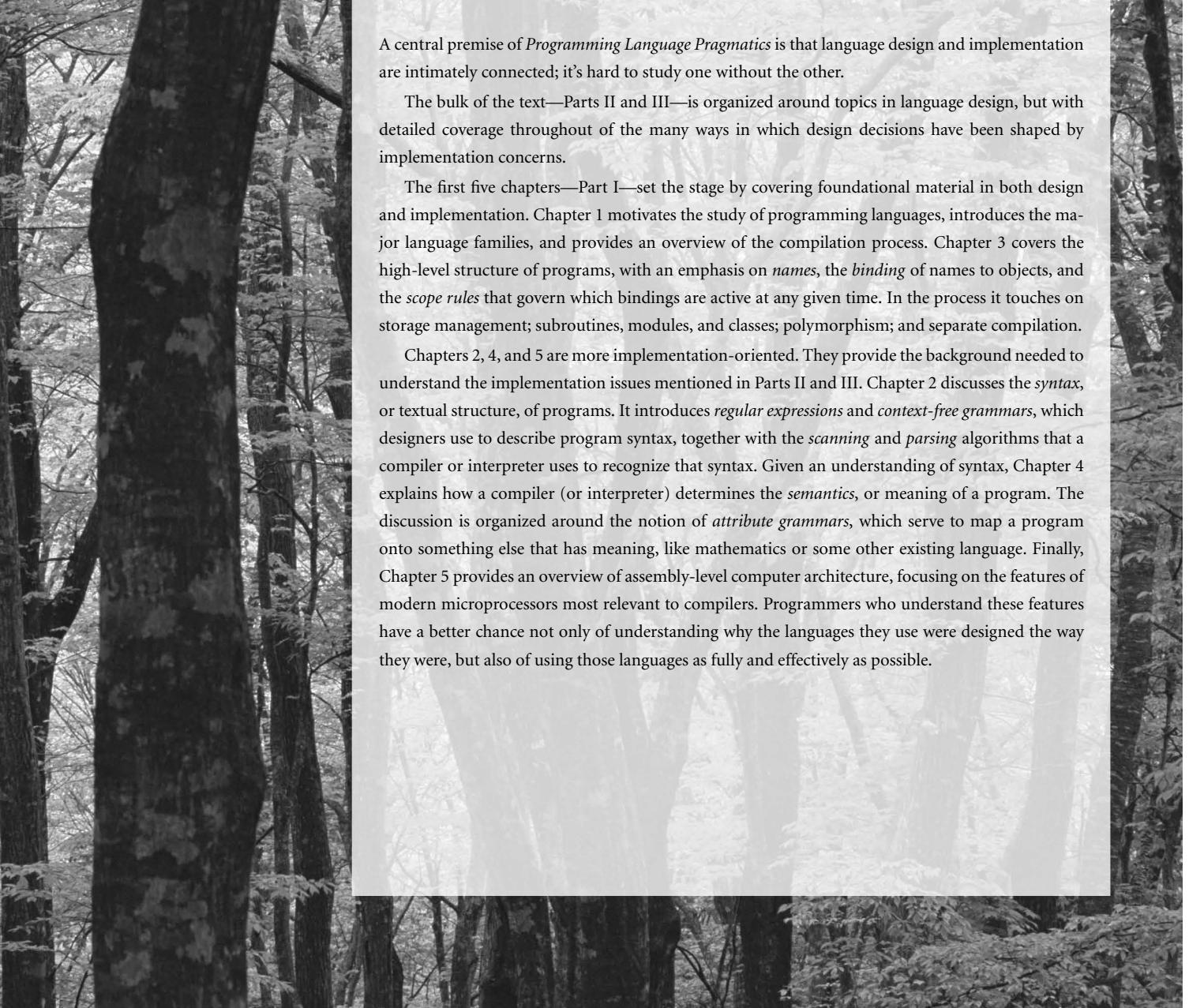
PLP-2e was also class tested at the University of Rochester in the fall of 2004. I am grateful to all my students, and to John Heidkamp, David Lu, and Dan Mullaney in particular, for their enthusiasm and suggestions. Mike Spear provided several helpful pointers on web technology for Chapter 13. Over the previous several years, my colleagues Chen Ding and Sandhya Dwarkadas taught from the first edition several times and had many helpful suggestions. Chen’s feedback on Chapter 15 (assisted by Yutao Zhong) was particularly valuable. My thanks as well to the rest of my colleagues, to department chair Mitsunori Ogihara, and to the department’s administrative, secretarial, and technical staff for providing such a supportive and productive work environment.

As they were on the first edition, the staff at Morgan Kaufmann have been a genuine pleasure to work with, on both a professional and a personal level. My thanks in particular to Denise Penrose, publisher; Nate McFadden, editor; Carl Soares, production editor; Peter Ashenden, CD designer; Brian Grimm, marketing manager; and Valerie Witte, editorial assistant.

Most important, I am indebted to my wife, Kelly, and our daughters, Erin and Shannon, for their patience and support through endless months of writing and revising. Computing is a fine profession, but family is what really matters.

Michael L. Scott
Rochester, NY
April 2005





Foundations

A central premise of *Programming Language Pragmatics* is that language design and implementation are intimately connected; it's hard to study one without the other.

The bulk of the text—Parts II and III—is organized around topics in language design, but with detailed coverage throughout of the many ways in which design decisions have been shaped by implementation concerns.

The first five chapters—Part I—set the stage by covering foundational material in both design and implementation. Chapter 1 motivates the study of programming languages, introduces the major language families, and provides an overview of the compilation process. Chapter 3 covers the high-level structure of programs, with an emphasis on *names*, the *binding* of names to objects, and the *scope rules* that govern which bindings are active at any given time. In the process it touches on storage management; subroutines, modules, and classes; polymorphism; and separate compilation.

Chapters 2, 4, and 5 are more implementation-oriented. They provide the background needed to understand the implementation issues mentioned in Parts II and III. Chapter 2 discusses the *syntax*, or textual structure, of programs. It introduces *regular expressions* and *context-free grammars*, which designers use to describe program syntax, together with the *scanning* and *parsing* algorithms that a compiler or interpreter uses to recognize that syntax. Given an understanding of syntax, Chapter 4 explains how a compiler (or interpreter) determines the *semantics*, or meaning of a program. The discussion is organized around the notion of *attribute grammars*, which serve to map a program onto something else that has meaning, like mathematics or some other existing language. Finally, Chapter 5 provides an overview of assembly-level computer architecture, focusing on the features of modern microprocessors most relevant to compilers. Programmers who understand these features have a better chance not only of understanding why the languages they use were designed the way they were, but also of using those languages as fully and effectively as possible.

Introduction

EXAMPLE 1.1

GCD program in MIPS machine language

The first electronic computers were monstrous contraptions, filling several rooms, consuming as much electricity as a good-size factory, and costing millions of 1940s dollars (but with the computing power of a modern hand-held calculator). The programmers who used these machines believed that the computer's time was more valuable than theirs. They programmed in machine language. Machine language is the sequence of bits that directly controls a processor, causing it to add, compare, move data from one place to another, and so forth at appropriate times. Specifying programs at this level of detail is an enormously tedious task. The following program calculates the greatest common divisor (GCD) of two integers, using Euclid's algorithm. It is written in machine language, expressed here as hexadecimal (base 16) numbers, for the MIPS R4000 processor.

```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c  
00401825 10820008 0064082a 10200003 00000000 10000002 00832023  
00641823 1483ffff 0064082a 0c1002b2 00000000 8fbf0014 27bd0020  
03e00008 00001025
```

EXAMPLE 1.2

GCD program in MIPS assembler

```
addiu   sp,sp,-32          b      C  
sw      ra,20(sp)          subu   a0,a0,v1  
jal     getint            subu   v1,v1,a0  
nop  
jal     getint            bne    a0,v1,A  
sw      v0,28(sp)          slt    at,v1,a0  
lw      a0,28(sp)          jal    putint  
move   v1,v0              nop  
beq    a0,v0,D             lw     ra,20(sp)  
slt    at,v1,a0            addiu sp,sp,32  
A:   beq    at,zero,B  
      nop                jr     ra  
                          move   v0,zero
```

Assembly languages were originally designed with a one-to-one correspondence between mnemonics and machine language instructions, as shown in this example.¹ Translating from mnemonics to machine language became the job of a systems program known as an *assembler*. Assemblers were eventually augmented with elaborate “macro expansion” facilities to permit programmers to define parameterized abbreviations for common sequences of instructions. The correspondence between assembly language and machine language remained obvious and explicit, however. Programming continued to be a machine-centered enterprise: each different kind of computer had to be programmed in its own assembly language, and programmers thought in terms of the instructions that the machine would actually execute.

As computers evolved, and as competing designs developed, it became increasingly frustrating to have to rewrite programs for every new machine. It also became increasingly difficult for human beings to keep track of the wealth of detail in large assembly language programs. People began to wish for a machine-independent language, particularly one in which numerical computations (the most common type of program in those days) could be expressed in something more closely resembling mathematical formulae. These wishes led in the mid-1950s to the development of the original dialect of Fortran, the first arguably high-level programming language. Other high-level languages soon followed, notably Lisp and Algol.

Translating from a high-level language to assembly or machine language is the job of a systems program known as a *compiler*. Compilers are substantially more complicated than assemblers because the one-to-one correspondence between source and target operations no longer exists when the source is a high-level language. Fortran was slow to catch on at first, because human programmers, with some effort, could almost always write assembly language programs that would run faster than what a compiler could produce. Over time, however, the performance gap has narrowed and eventually reversed. Increases in hardware complexity (due to pipelining, multiple functional units, etc.) and continuing improvements in compiler technology have led to a situation in which a state-of-the-art compiler will usually generate better code than a human being will. Even in cases in which human beings can do better, increases in computer speed and program size have made it increasingly important to economize on programmer effort, not only in the original construction of programs, but in subsequent program *maintenance*—enhancement and correction. Labor costs now heavily outweigh the cost of computing hardware.

1 Each of the 23 lines of assembly code in the example is encoded in the corresponding 32 bits of the machine language. Note for example that the two `sw` (store word) instructions begin with the same 11 bits (`afa` or `afb`). Those bits encode the operation (`sw`) and the base register (`sp`).

The Art of Language Design

Today there are thousands of high-level programming languages, and new ones continue to emerge. Human beings use assembly language only for special purpose applications. In a typical undergraduate class, it is not uncommon to find users of scores of different languages. Why are there so many? There are several possible answers:

Evolution. Computer science is a young discipline; we're constantly finding better ways to do things. The late 1960s and early 1970s saw a revolution in "structured programming," in which the go to-based control flow of languages like Fortran, Cobol, and Basic² gave way to while loops, case statements, and similar higher-level constructs. In the late 1980s the nested block structure of languages like Algol, Pascal, and Ada began to give way to the object-oriented structure of Smalltalk, C++, Eiffel, and the like.

Special Purposes. Many languages were designed for a specific problem domain. The various Lisp dialects are good for manipulating symbolic data and complex data structures. Snobol and Icon are good for manipulating character strings. C is good for low-level systems programming. Prolog is good for reasoning about logical relationships among data. Each of these languages can be used successfully for a wider range of tasks, but the emphasis is clearly on the specialty.

Personal Preference. Different people like different things. Much of the parochialism of programming is simply a matter of taste. Some people love the terseness of C; some hate it. Some people find it natural to think recursively; others prefer iteration. Some people like to work with pointers; others prefer the implicit dereferencing of Lisp, Clu, Java, and ML. The strength and variety of personal preference make it unlikely that anyone will ever develop a universally acceptable programming language.

Of course, some languages are more successful than others. Of the many that have been designed, only a few dozen are widely used. What makes a language successful? Again there are several answers:

Expressive Power. One commonly hears arguments that one language is more "powerful" than another, though in a formal mathematical sense they are all Turing equivalent—each can be used, if awkwardly, to implement arbitrary algorithms. Still, language features clearly have a huge impact on the programmer's ability to write clear, concise, and maintainable code, especially for very

² The name of each of these languages is sometimes written entirely in uppercase letters and sometimes in mixed case. For consistency's sake, I adopt the convention in this book of using mixed case for languages whose names are pronounced as words (e.g., Fortran, Cobol, Basic) and uppercase for those pronounced as a series of letters (e.g., APL, PL/I, ML).

large systems. There is no comparison, for example, between early versions of Basic on the one hand and Common Lisp or Ada on the other. The factors that contribute to expressive power—abstraction facilities in particular—are a major focus of this book.

Ease of Use for the Novice. While it is easy to pick on Basic, one cannot deny its success. Part of that success is due to its very low “learning curve.” Logo is popular among elementary-level educators for a similar reason: even a 5-year-old can learn it. Pascal was taught for many years in introductory programming language courses because, at least in comparison to other “serious” languages, it is compact and easy to learn. In recent years Java has come to play a similar role. Though substantially more complex than Pascal, it is much simpler than, say, C++.

Ease of Implementation. In addition to its low learning curve, Basic is successful because it could be implemented easily on tiny machines, with limited resources. Forth has a small but dedicated following for similar reasons. Arguably the single most important factor in the success of Pascal was that its designer, Niklaus Wirth, developed a simple, portable implementation of the language, and shipped it free to universities all over the world (see Example 1.12).³ The Java designers have taken similar steps to make their language available for free to almost anyone who wants it.

Open Source. Most programming languages today have at least one open source compiler or interpreter, but some languages—C in particular—are much more closely associated than others with freely distributed, peer reviewed, community supported computing. C was originally developed in the early 1970s by Dennis Ritchie and Ken Thompson at Bell Labs,⁴ in conjunction with the design of the original Unix operating system. Over the years Unix evolved into the world’s most portable operating system—the OS of choice for academic computer science—and C was closely associated with it. With the standardization of C, the language has become available on an enormous variety of additional platforms. Linux, the leading open source operating system, is written in C. As of March 2005, C and its descendants account for 60% of the projects hosted at sourceforge.net.

Excellent Compilers. Fortran owes much of its success to extremely good compilers. In part this is a matter of historical accident. Fortran has been around longer than anything else, and companies have invested huge amounts of time

3 Niklaus Wirth (1934–), Professor Emeritus of Informatics at ETH in Zürich, Switzerland, is responsible for a long line of influential languages, including Euler, Algol-W, Pascal, Modula, Modula-2, and Oberon. Among other things, his languages introduced the notions of enumeration, subrange, and set types, and unified the concepts of records (structs) and variants (unions). He received the annual ACM Turing Award, computing’s highest honor, in 1984.

4 Ken Thompson (1943–) led the team that developed Unix. He also designed the B programming language, a child of BCPL and the parent of C. Dennis Ritchie (1941–) was the principal force behind the development of C itself. Thompson and Ritchie together formed the core of an incredibly productive and influential group. They shared the ACM Turing Award in 1983.

and money in making compilers that generate very fast code. It is also a matter of language design, however: Fortran dialects prior to Fortran 90 lack recursion and pointers, features that greatly complicate the task of generating fast code (at least for programs that can be written in a reasonable fashion without them!). In a similar vein, some languages (e.g., Common Lisp) are successful in part because they have compilers and supporting tools that do an unusually good job of helping the programmer manage very large projects.

Economics, Patronage, and Inertia. Finally, there are factors other than technical merit that greatly influence success. The backing of a powerful sponsor is one. Cobol and PL/I, at least to first approximation, owe their life to IBM. Ada owes its life to the United States Department of Defense: it contains a wealth of excellent features and ideas, but the sheer complexity of implementation would likely have killed it if not for the DoD backing. Similarly, C#, despite its technical merits, would probably not have received the attention it has without the backing of Microsoft. At the other end of the life cycle, some languages remain widely used long after “better” alternatives are available because of a huge base of installed software and programmer expertise, which would cost too much to replace.

DESIGN & IMPLEMENTATION

Introduction

Throughout the book, sidebars like this one will highlight the interplay of language design and language implementation. Among other things, we will consider the following.

- Cases (such as those mentioned in this section) in which ease or difficulty of implementation significantly affected the success of a language
- Language features that many designers now believe were mistakes, at least in part because of implementation difficulties
- Potentially useful features omitted from some languages because of concern that they might be too difficult or slow to implement
- Language limitations adopted at least in part out of concern for implementation complexity or cost
- Language features introduced at least in part to facilitate efficient or elegant implementations
- Cases in which a machine architecture makes reasonable features unreasonably expensive
- Various other tradeoffs in which implementation plays a significant role

A complete list of sidebars appears in Appendix B.

Clearly no one factor determines whether a language is “good.” As we study programming languages, we shall need to consider issues from several points of view. In particular, we shall need to consider the viewpoints of both the programmer and the language implementor. Sometimes these points of view will be in harmony, as in the desire for execution speed. Often, however, there will be conflicts and tradeoffs, as the conceptual appeal of a feature is balanced against the cost of its implementation. The tradeoff becomes particularly thorny when the implementation imposes costs not only on programs that use the feature, but also on programs that do not.

In the early days of computing the implementor’s viewpoint was predominant. Programming languages evolved as a means of telling a computer what to do. For programmers, however, a language is more aptly defined as a means of expressing algorithms. Just as natural languages constrain exposition and discourse, so programming languages constrain what can and cannot be expressed, and have both profound and subtle influence over what the programmer can *think*. Donald Knuth has suggested that programming be regarded as the art of telling another human being what one wants the computer to do [Knu84].⁵ This definition perhaps strikes the best sort of compromise. It acknowledges that both conceptual clarity and implementation efficiency are fundamental concerns. This book attempts to capture this spirit of compromise by simultaneously considering the conceptual and implementation aspects of each of the topics it covers.

1.2 The Programming Language Spectrum

EXAMPLE 1.3

Classification of
programming languages

The many existing languages can be classified into families based on their model of computation. Figure 1.1 shows a common set of families. The top-level division distinguishes between the *declarative* languages, in which the focus is on *what* the computer is to do, and the *imperative* languages, in which the focus is on *how* the computer should do it. ■

Declarative languages are in some sense “higher level”; they are more in tune with the programmer’s point of view, and less with the implementor’s point of view. Imperative languages predominate, however, mainly for performance reasons. There is a tension in the design of declarative languages between the desire to get away from “irrelevant” implementation details and the need to remain close enough to the details to at least control the outline of an algorithm. The design of efficient algorithms, after all, is what much of computer science is about.

5 Donald E. Knuth (1938–), Professor Emeritus at Stanford University and one of the foremost figures in the design and analysis of algorithms, is also widely known as the inventor of the TeX typesetting system (with which this book was produced) and of the *literate programming* methodology with which TeX was constructed. His multivolume *The Art of Computer Programming* has an honored place on the shelf of most professional computer scientists. He received the ACM Turing Award in 1974.

declarative		
functional	Lisp/Scheme, ML, Haskell	
dataflow	Id, Val	
logic, constraint-based	Prolog, spreadsheets	
template-based	XSLT	
imperative		
von Neumann	C, Ada, Fortran, ...	
scripting	Perl, Python, PHP, ...	
object-oriented	Smalltalk, Eiffel, C++, Java, ...	

Figure 1.1 Classification of programming languages. Note that the categories are fuzzy and open to debate. In particular, it is possible for a functional language to be object-oriented, and many authors do not consider functional programming to be declarative.

It is not yet clear to what extent, and in what problem domains, we can expect compilers to discover good algorithms for problems stated at a very high level. In any domain in which the compiler cannot find a good algorithm, the programmer needs to be able to specify one explicitly.

Within the declarative and imperative families, there are several important subclasses.

- *Functional* languages employ a computational model based on the recursive definition of functions. They take their inspiration from the *lambda calculus*, a formal computational model developed by Alonzo Church in the 1930s. In essence, a program is considered a function from inputs to outputs, defined in terms of simpler functions through a process of refinement. Languages in this category include Lisp, ML, and Haskell.
- *Dataflow* languages model computation as the flow of information (*tokens*) among primitive functional *nodes*. They provide an inherently parallel model: nodes are triggered by the arrival of input tokens, and can operate concurrently. Id and Val are examples of dataflow languages. Sisal, a descendant of Val, is more often described as a functional language.
- *Logic or constraint-based* languages take their inspiration from predicate logic. They model computation as an attempt to find values that satisfy certain specified relationships, using a goal-directed search through a list of logical rules. Prolog is the best-known logic language. The term can also be applied to the programmable aspects of spreadsheet systems such as Excel, VisiCalc, or Lotus 1-2-3.
- The *von Neumann* languages are the most familiar and successful. They include Fortran, Ada 83, C, and all of the others in which the basic means of computation is the modification of variables.⁶ Whereas functional languages

6 John von Neumann (1903–1957) was a mathematician and computer pioneer who helped to develop the concept of *stored program* computing, which underlies most computer hardware. In a stored program computer, both programs and data are represented as bits in memory, which the processor repeatedly fetches, interprets, and updates.

are based on expressions that have values, von Neumann languages are based on statements (assignments in particular) that influence subsequent computation via the *side effect* of changing the value of memory.

- *Scripting* languages are a subset of the von Neumann languages. They are distinguished by their emphasis on “gluing together” components that were originally developed as independent programs. Several scripting languages were originally developed for specific purposes: `csh` and `bash`, for example, are the input languages of job control (shell) programs; `Awk` was intended for text manipulation; `PHP` and `JavaScript` are primarily intended for the generation of web pages with dynamic content (with execution on the server and the client, respectively). Other languages, including `Perl`, `Python`, `Ruby`, and `Tcl`, are more deliberately general purpose. Most place an emphasis on rapid prototyping, with a bias toward ease of expression over speed of execution.
- *Object-oriented* languages are comparatively recent, though their roots can be traced to Simula 67. Most are closely related to the von Neumann languages but have a much more structured and distributed model of both memory and computation. Rather than picture computation as the operation of a monolithic processor on a monolithic memory, object-oriented languages picture it as interactions among semi-independent *objects*, each of which has both its own internal state and subroutines to manage that state. Smalltalk is the purest of the object-oriented languages; `C++` and `Java` are the most widely used. It is also possible to devise object-oriented functional languages (the best known of these is the CLOS [Kee89] extension to Common Lisp), but they tend to have a strong imperative flavor.

One might suspect that concurrent languages also form a separate class (and indeed this book devotes a chapter to the subject), but the distinction between concurrent and sequential execution is mostly orthogonal to the classifications above. Most concurrent programs are currently written using special library packages or compilers in conjunction with a sequential language such as Fortran or C. A few widely used languages, including Java, C#, Ada, and Modula-3, have explicitly concurrent features. Researchers are investigating concurrency in each of the language classes mentioned here.

It should be emphasized that the distinctions among language classes are not clear-cut. The division between the von Neumann and object-oriented languages, for example, is often very fuzzy, and most of the functional and logic languages include some imperative features. The preceding descriptions are meant to capture the general flavor of the classes, without providing formal definitions.

Imperative languages—von Neumann and object-oriented—receive the bulk of the attention in this book. Many issues cut across family lines, however, and the interested reader will discover much that is applicable to alternative computational models in most of the chapters of the book. Chapters 10 through 13 contain additional material on functional, logic, concurrent, and scripting languages.

15 Code Improvement	CD 202 • 791
15.1 Phases of Code Improvement	CD 204
15.2 Peephole Optimization	CD 206
15.3 Redundancy Elimination in Basic Blocks	CD 209
15.3.1 A Running Example	CD 210
15.3.2 Value Numbering	CD 211
15.4 Global Redundancy and Data Flow Analysis	CD 217
15.4.1 SSA Form and Global Value Numbering	CD 218
15.4.2 Global Common Subexpression Elimination	CD 220
15.5 Loop Improvement I	CD 227
15.5.1 Loop Invariants	CD 228
15.5.2 Induction Variables	CD 229
15.6 Instruction Scheduling	CD 232
15.7 Loop Improvement II	CD 236
15.7.1 Loop Unrolling and Software Pipelining	CD 237
15.7.2 Loop Reordering	CD 241
15.8 Register Allocation	CD 248
15.9 Summary and Concluding Remarks	CD 252
15.10 Exercises	CD 253
15.11 Explorations	CD 257
15.12 Bibliographic Notes	CD 258
A Programming Languages Mentioned	793
B Language Design and Language Implementation	803
C Numbered Examples	807
Bibliography	819
Index	837

1.3 Why Study Programming Languages?

Programming languages are central to computer science and to the typical computer science curriculum. Like most car owners, students who have become familiar with one or more high-level languages are generally curious to learn about other languages, and to know what is going on “under the hood.” Learning about languages is interesting. It’s also practical.

For one thing, a good understanding of language design and implementation can help one choose the most appropriate language for any given task. Most languages are better for some things than for others. No one would be likely to use APL for symbolic computing or string processing, but other choices are not nearly so clear-cut. Should one choose C, C++, or Modula-3 for systems programming? Fortran or Ada for scientific computations? Ada or Modula-2 for embedded systems? Visual Basic or Java for a graphical user interface? This book should help equip you to make such decisions.

Similarly, this book should make it easier to learn new languages. Many languages are closely related. Java and C# are easier to learn if you already know C++. Common Lisp is easier to learn if you already know Scheme. More important, there are basic concepts that underlie all programming languages. Most of these concepts are the subject of chapters in this book: types, control (iteration, selection, recursion, nondeterminacy, concurrency), abstraction, and naming. Thinking in terms of these concepts makes it easier to assimilate the syntax (form) and semantics (meaning) of new languages, compared to picking them up in a vacuum. The situation is analogous to what happens in natural languages: a good knowledge of grammatical forms makes it easier to learn a foreign language.

Whatever language you learn, understanding the decisions that went into its design and implementation will help you use it better. This book should help you

Understand obscure features. The typical C++ programmer rarely uses unions, multiple inheritance, variable numbers of arguments, or the `.*` operator. (If you don’t know what these are, don’t worry!) Just as it simplifies the assimilation of new languages, an understanding of basic concepts makes it easier to understand these features when you look up the details in the manual.

Choose among alternative ways to express things, based on a knowledge of implementation costs. In C++, for example, programmers may need to avoid unnecessary temporary variables, and use copy constructors whenever possible, to minimize the cost of initialization. In Java they may wish to use Executor objects rather than explicit thread creation. With certain (poor) compilers, they may need to adopt special programming idioms to get the fastest code: pointers for array traversal in C; `with` statements to factor out common address calculations in Pascal or Modula-3; `x**x` instead of `x**2` in Basic. In any

language, they need to be able to evaluate the tradeoffs among alternative implementations of abstractions—for example between computation and table lookup for functions like bit set cardinality, which can be implemented either way.

Make good use of debuggers, assemblers, linkers, and related tools. In general, the high-level language programmer should not need to bother with implementation details. There are times, however, when an understanding of those details proves extremely useful. The tenacious bug or unusual system-building problem is sometimes a lot easier to handle if one is willing to peek at the bits.

Simulate useful features in languages that lack them. Certain very useful features are missing in older languages but can be emulated by following a deliberate (if unenforced) programming style. In older dialects of Fortran, for example, programmers familiar with modern control constructs can use comments and self-discipline to write well-structured code. Similarly, in languages with poor abstraction facilities, comments and naming conventions can help imitate modular structure, and the extremely useful *iterators* of Clu, Icon, and C# (which we will study in Section 6.5.3) can be imitated with subroutines and static variables. In Fortran 77 and other languages that lack recursion, an iterative program can be derived via mechanical hand transformations, starting with recursive pseudocode. In languages without named constants or enumeration types, variables that are initialized once and never changed thereafter can make code much more readable and easy to maintain.

Make better use of language technology wherever it appears. Most programmers will never design or implement a conventional programming language, but most will need language technology for other programming tasks. The typical personal computer contains files in dozens of structured formats, encompassing web content, word processing, spreadsheets, presentations, raster and vector graphics, music, video, databases, and a wide variety of other application domains. Each of these structured formats has formal syntax and semantics, which tools must understand. Code to parse, analyze, generate, optimize, and otherwise manipulate structured data can thus be found in almost any sophisticated program, and all of this code is based on language technology. Programmers with a strong grasp of this technology will be in a better position to write well-structured, maintainable tools.

In a similar vein, most tools themselves can be customized, via start-up configuration files, command-line arguments, input commands, or built-in *extension languages* (considered in more detail in Chapter 13). My home directory holds more than 250 separate configuration (“preference”) files. My personal configuration files for the `emacs` text editor comprise more than 1200 lines of Lisp code. The user of almost any sophisticated program today will need to make good use of configuration or extension languages. The designers of such a program will need either to adopt (and adapt) some existing extension language, or to invent new notation of their own. Programmers with a strong grasp of language theory will be in a better position to design elegant,

well-structured notation that meets the needs of current users and facilitates future development.

Finally, this book should help prepare you for further study in language design or implementation, should you be so inclined. It will also equip you to understand the interactions of languages with operating systems and architectures, should those areas draw your interest.

CHECK YOUR UNDERSTANDING

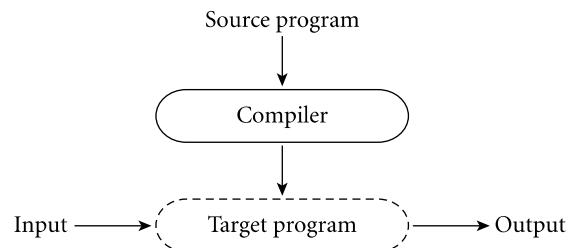
1. What is the difference between machine language and assembly language?
2. In what way(s) are high-level languages an improvement on assembly language? In what circumstances does it still make sense to program in assembler?
3. Why are there so many programming languages?
4. What makes a programming language successful?
5. Name three languages in each of the following categories: von Neumann, functional, object-oriented. Name two logic languages. Name two widely used concurrent languages.
6. What distinguishes declarative languages from imperative languages?
7. What organization spearheaded the development of Ada?
8. What is generally considered the first high-level programming language?
9. What was the first functional language?

1.4 Compilation and Interpretation

EXAMPLE 1.4

Pure compilation

At the highest level of abstraction, the compilation and execution of a program in a high-level language look something like this:

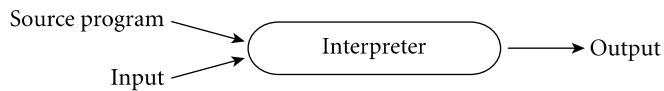


The compiler *translates* the high-level source program into an equivalent target program (typically in machine language) and then goes away. At some arbitrary later time, the user tells the operating system to run the target program. The compiler is the locus of control during compilation; the target program is the locus of control during its own execution. The compiler is itself a machine language program, presumably created by compiling some other high-level program. When written to a file in a format understood by the operating system, machine language is commonly known as *object code*. ■

EXAMPLE 1.5

Pure interpretation

An alternative style of implementation for high-level languages is known as *interpretation*.



Unlike a compiler, an interpreter stays around for the execution of the application. In fact, the interpreter is the locus of control during that execution. In effect, the interpreter implements a virtual machine whose “machine language” is the high-level programming language. The interpreter reads statements in that language more or less one at a time, executing them as it goes along. ■

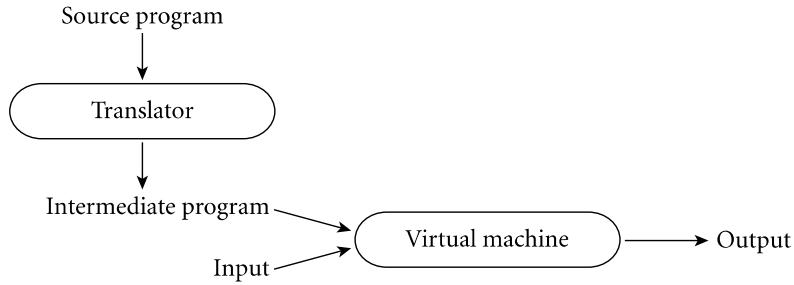
In general, interpretation leads to greater flexibility and better diagnostics (error messages) than does compilation. Because the source code is being executed directly, the interpreter can include an excellent source-level debugger. It can also cope with languages in which fundamental characteristics of the program, such as the sizes and types of variables, or even which names refer to which variables, can depend on the input data. Some language features are almost impossible to implement without interpretation: in Lisp and Prolog, for example, a program can write new pieces of itself and execute them on the fly. (Several scripting languages, including Perl, Tcl, Python, and Ruby, also provide this capability.) Delaying decisions about program implementation until run time is known as *late binding*; we will discuss it at greater length in Section 3.1.

Compilation, by contrast, generally leads to better performance. In general, a decision made at compile time is a decision that does not need to be made at run time. For example, if the compiler can guarantee that variable *x* will always lie at location 49378, it can generate machine language instructions that access this location whenever the source program refers to *x*. By contrast, an interpreter may need to look *x* up in a table every time it is accessed, in order to find its location. Since the (final version of a) program is compiled only once, but generally executed many times, the savings can be substantial, particularly if the interpreter is doing unnecessary work in every iteration of a loop.

EXAMPLE 1.6

Mixing compilation and interpretation

While the conceptual difference between compilation and interpretation is clear, most language implementations include a mixture of both. They typically look like this:



We generally say that a language is “interpreted” when the initial translator is simple. If the translator is complicated, we say that the language is “compiled.” The distinction can be confusing because “simple” and “complicated” are subjective terms, and because it is possible for a compiler (complicated translator) to produce code that is then executed by a complicated virtual machine (interpreter); this is in fact precisely what happens by default in Java. We still say that a language is compiled if the translator analyzes it thoroughly (rather than effecting some “mechanical” transformation) and if the intermediate program does not bear a strong resemblance to the source. These two characteristics—thorough analysis and nontrivial transformation—are the hallmarks of compilation. ■

In practice one sees a broad spectrum of implementation strategies. For example:

EXAMPLE 1.7

Preprocessing

- Most interpreted languages employ an initial translator (a *preprocessor*) that removes comments and white space, and groups characters together into *tokens*, such as keywords, identifiers, numbers, and symbols. The translator may also expand abbreviations in the style of a macro assembler. Finally, it may identify higher-level syntactic structures, such as loops and subroutines. The goal is to produce an intermediate form that mirrors the structure of the source but can be interpreted more efficiently. ■

DESIGN & IMPLEMENTATION

Compiled and interpreted languages

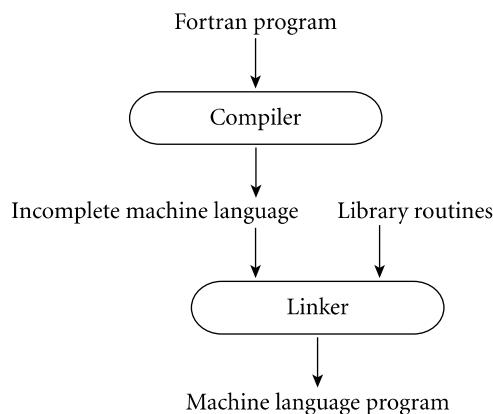
Certain languages (APL and Smalltalk, for example) are sometimes referred to as “interpreted languages” because most of their semantic error checking must be performed at run time. Certain other languages (Fortran and C, for example) are sometimes referred to as “compiled languages” because almost all of their semantic error checking can be performed statically. This terminology isn’t strictly correct: interpreters for C and Fortran can be built easily, and a compiler can generate code to perform even the most extensive dynamic semantic checks. That said, language design has a profound effect on “compilability.”

In some very early implementations of Basic, the manual actually suggested removing comments from a program in order to improve its performance. These implementations were pure interpreters; they would reread (and then ignore) the comments every time they executed a given part of the program. They had no initial translator.

EXAMPLE 1.8

Library routines and linking

- The typical Fortran implementation comes close to pure compilation. The compiler translates Fortran source into machine language. Usually, however, it counts on the existence of a *library* of subroutines that are not part of the original program. Examples include mathematical functions (`sin`, `cos`, `log`, etc.) and I/O. The compiler relies on a separate program, known as a *linker*, to merge the appropriate library routines into the final program:



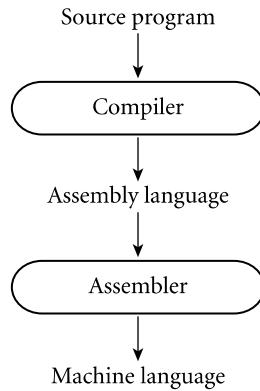
In some sense, one may think of the library routines as extensions to the hardware instruction set. The compiler can then be thought of as generating code for a virtual machine that includes the capabilities of both the hardware and the library.

In a more literal sense, one can find interpretation in the Fortran routines for formatted output. Fortran permits the use of `format` statements that control the alignment of output in columns, the number of significant digits and type of scientific notation for floating-point numbers, inclusion/suppression of leading zeros, and so on. Programs can compute their own formats on the fly. The output library routines include a `format` interpreter. A similar interpreter can be found in the `printf` routine of C and its descendants. ■

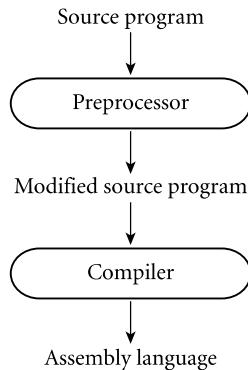
EXAMPLE 1.9

Post-compilation assembly

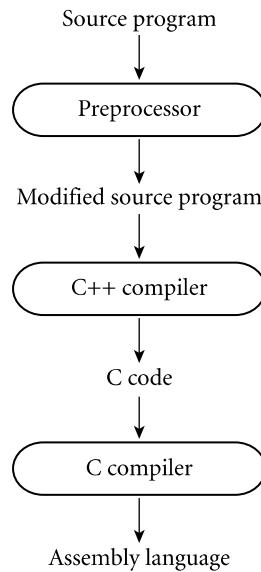
- Many compilers generate assembly language instead of machine language. This convention facilitates debugging, since assembly language is easier for people to read, and isolates the compiler from changes in the format of machine language files that may be mandated by new releases of the operating system (only the assembler must be changed, and it is shared by many compilers).

**EXAMPLE 1.10****The C preprocessor**

- Compilers for C (and for many other languages running under Unix) begin with a preprocessor that removes comments and expands macros. The preprocessor can also be instructed to delete portions of the code itself, providing a *conditional compilation* facility that allows several versions of a program to be built from the same source.

**EXAMPLE 1.11****Source-to-source translation (C++)**

- C++ implementations based on the early AT&T compiler actually generated an intermediate program in C, instead of in assembly language. This C++ compiler was indeed a true compiler: it performed a complete analysis of the syntax and semantics of the C++ source program, and with very few exceptions generated all of the error messages that a programmer would see prior to running the program. In fact, programmers were generally unaware that the C compiler was being used behind the scenes. The C++ compiler did not invoke the C compiler unless it had generated C code that would pass through the second round of compilation without producing any error messages.

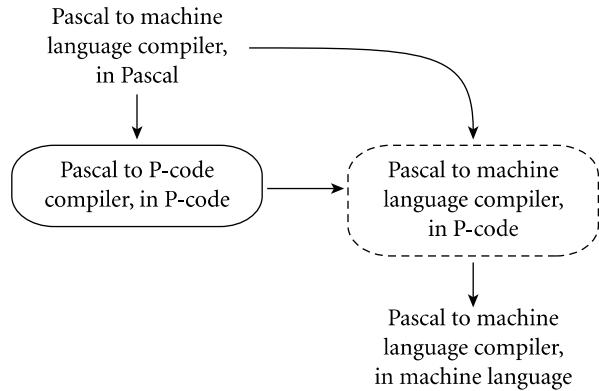


Occasionally one would hear the C++ compiler referred to as a preprocessor, presumably because it generated high-level output that was in turn compiled. I consider this a misuse of the term: compilers attempt to “understand” their source; preprocessors do not. Preprocessors perform transformations based on simple pattern matching, and may well produce output that will generate error messages when run through a subsequent stage of translation. ■

EXAMPLE 1.12
Bootstrapping

- Many early Pascal compilers were built around a set of tools distributed by Niklaus Wirth. These included the following.
 - A Pascal compiler, written in Pascal, that would generate output in *P-code*, a simple stack-based language
 - The same compiler, already translated into P-code
 - A P-code interpreter, written in Pascal

To get Pascal up and running on a local machine, the user of the tool set needed only to translate the P-code interpreter (by hand) into some locally available language. This translation was not a difficult task; the interpreter was small. By running the P-code version of the compiler on top of the P-code interpreter, one could then compile arbitrary Pascal programs into P-code, which could in turn be run on the interpreter. To get a faster implementation, one could modify the Pascal version of the Pascal compiler to generate a locally available variety of assembly or machine language, instead of generating P-code (a somewhat more difficult task). This compiler could then be “run through itself” in a process known as *bootstrapping*, a term derived from the intentionally ridiculous notion of lifting oneself off the ground by pulling on one’s bootstraps.



At this point, the P-code interpreter and the P-code version of the Pascal compiler could simply be thrown away. More often, however, programmers would choose to keep these tools around. The P-code version of a program tends to be significantly smaller than its machine language counterpart. On a circa 1970 machine, the savings in memory and disk requirements could really be important. Moreover, as noted near the beginning of this section, an interpreter will often provide better run-time diagnostics than will the output of a compiler. Finally, an interpreter allows a program to be rerun immediately after modification, without waiting for recompilation—a feature that can be particularly valuable during program development. Some of the best programming environments for imperative languages include both a compiler and an interpreter.

DESIGN & IMPLEMENTATION

The early success of Pascal

The P-code based implementation of Pascal is largely responsible for the language's remarkable success in academic circles in the 1970s. No single hardware platform or operating system of that era dominated the computer landscape the way the x86, Linux, and Windows do today.⁷ Wirth's toolkit made it possible to get an implementation of Pascal up and running on almost any platform in a week or so. It was one of the first great successes in system portability.

⁷ Throughout this book we will use the term “x86” to refer to the instruction set architecture of the Intel 8086 and its descendants, including the various Pentium processors. Intel calls this architecture the IA-32, but x86 is a more generic term that encompasses the offerings of competitors such as AMD as well.

EXAMPLE 1.13

Compiling interpreted languages

- One will sometimes find compilers for languages (e.g., Lisp, Prolog, Smalltalk, etc.) that permit a lot of late binding and are traditionally interpreted. These compilers must be prepared, in the general case, to generate code that performs much of the work of an interpreter, or that makes calls into a library that does that work instead. In important special cases, however, the compiler can generate code that makes reasonable assumptions about decisions that won't be finalized until run time. If these assumptions prove to be valid the code will run very fast. If the assumptions are not correct, a dynamic check will discover the inconsistency, and revert to the interpreter.

EXAMPLE 1.14

Dynamic and just-in-time compilation

- In some cases a programming system may deliberately delay compilation until the last possible moment. One example occurs in implementations of Lisp or Prolog that invoke the compiler on the fly, to translate newly created source into machine language, or to optimize the code for a particular input set. Another example occurs in implementations of Java. The Java language definition defines a machine-independent intermediate form known as *byte code*. Byte code is the standard format for distribution of Java programs; it allows programs to be transferred easily over the Internet and then run on any platform. The first Java implementations were based on byte-code interpreters, but more recent (faster) implementations employ a *just-in-time* compiler that translates byte code into machine language immediately before each execution of the program. C#, similarly, is intended for just-in-time translation. The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution. CIL is deliberately language independent, so it can be used for code produced by a variety of front-end compilers.

EXAMPLE 1.15

Microcode (firmware)

- On some machines (particularly those designed before the mid-1980s), the assembly-level instruction set is not actually implemented in hardware but in fact runs on an interpreter. The interpreter is written in low-level instructions called *microcode* (or *firmware*), which is stored in read-only memory and executed by the hardware. Microcode and microprogramming are considered further in Section 5.4.1.

As some of these examples make clear, a compiler does not necessarily translate from a high-level language into machine language. It is not uncommon for compilers, especially prototypes, to generate C as output. A little farther afield, text formatters like TeX and troff are actually compilers, translating high-level document descriptions into commands for a laser printer or phototypesetter. (Many laser printers themselves incorporate interpreters for the Postscript page-description language.) Query language processors for database systems are also compilers, translating languages like SQL into primitive operations on files. There are even compilers that translate logic-level circuit specifications into photographic masks for computer chips. Though the focus in this book is on imperative programming languages, the term "compilation" applies whenever we translate automatically from one nontrivial language to another, with full analysis of the meaning of the input.

1.5 Programming Environments

Compilers and interpreters do not exist in isolation. Programmers are assisted in their work by a host of other tools. Assemblers, debuggers, preprocessors, and linkers were mentioned earlier. Editors are familiar to every programmer. They may be assisted by cross-referencing facilities that allow the programmer to find the point at which an object is defined, given a point at which it is used. Pretty printers help enforce formatting conventions. Style checkers enforce syntactic or semantic conventions that may be tighter than those enforced by the compiler (see Exploration 1.11). Configuration management tools help keep track of dependences among the (many versions of) separately compiled modules in a large software system. Perusal tools exist not only for text but also for intermediate languages that may be stored in binary. Profilers and other performance analysis tools often work in conjunction with debuggers to help identify the pieces of a program that consume the bulk of its computation time.

In older programming environments, tools may be executed individually, at the explicit request of the user. If a running program terminates abnormally with a “bus error” (invalid address) message, for example, the user may choose to invoke a debugger to examine the “core” file dumped by the operating system. He or she may then attempt to identify the program bug by setting breakpoints, enabling tracing, and so on, and running the program again under the control of the debugger. Once the bug is found, the user will invoke the editor to make an appropriate change. He or she will then recompile the modified program, possibly with the help of a configuration manager.

More recent programming environments provide much more integrated tools. When an invalid address error occurs in an integrated environment, a new window is likely to appear on the user’s screen, with the line of source code at which the error occurred highlighted. Breakpoints and tracing can then be set in this window without explicitly invoking a debugger. Changes to the source can be made without explicitly invoking an editor. The editor may also incorporate knowledge of the language syntax, providing templates for all the standard control structures, and checking syntax as it is typed in. If the user asks to rerun the program after making changes, a new version may be built without explicitly invoking the compiler or configuration manager.

DESIGN & IMPLEMENTATION

Powerful development environments

Sophisticated development environments can be a two-edged sword. The quality of the Common Lisp environment has arguably contributed to its widespread acceptance. On the other hand, the particularity of the graphical environment for Smalltalk (with its insistence on specific fonts, window styles, etc.) has made it difficult to port the language to systems accessed through a textual interface, or to graphical systems with a different “look and feel.”

Integrated environments have been developed for a variety of languages and systems. They are fundamental to Smalltalk—it is nearly impossible to separate the language from its graphical environment—and are widely used with Common Lisp. They are common on personal computers; examples include the Visual Studio environment from Microsoft and the Project Builder environment from Apple. Several similar commercial and open source environments are available for Unix, and much of the appearance of integration can be achieved within sophisticated editors such as `emacs`.

CHECK YOUR UNDERSTANDING

10. Explain the distinction between interpretation and compilation. What are the comparative advantages and disadvantages of the two approaches?
 11. Is Java compiled or interpreted (or both)? How do you know?
 12. What is the difference between a compiler and a preprocessor?
 13. What was the intermediate form employed by the original AT&T C++ compiler?
 14. What is P-code?
 15. What is bootstrapping?
 16. What is a just-in-time compiler?
 17. Name two languages in which a program can write new pieces of itself “on-the-fly.”
 18. Briefly describe three “unconventional” compilers—compilers whose purpose is not to prepare a high-level program for execution on a microprocessor.
 19. Describe six kinds of tools that commonly support the work of a compiler within a larger programming environment.
-

1.6 An Overview of Compilation

EXAMPLE 1.16

Phases of compilation

Compilers are among the most well-studied types of computer programs. In a typical compiler, compilation proceeds through a series of well-defined *phases*, shown in Figure 1.2. Each phase discovers information of use to later phases, or transforms the program into a form that is more useful to the subsequent phase.

The first few phases (up through semantic analysis) serve to figure out the meaning of the source program. They are sometimes called the *front end* of the compiler. The last few phases serve to construct an equivalent target program. They are sometimes called the *back end* of the compiler. Many compiler phases

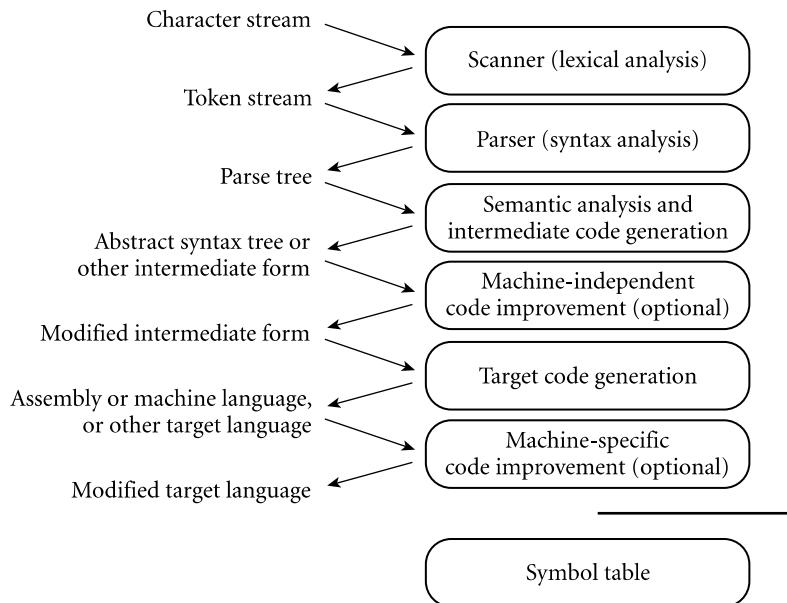


Figure 1.2 Phases of compilation. Phases are listed on the right and the forms in which information is passed between phases are listed on the left. The symbol table serves throughout compilation as a repository for information about identifiers.

can be created automatically from a formal description of the source and/or target languages.

One will sometimes hear compilation described as a series of *passes*. A pass is a phase or set of phases that is serialized with respect to the rest of compilation: it does not start until previous phases have completed, and it finishes before any subsequent phases start. If desired, a pass may be written as a separate program, reading its input from a file and writing its output to a file. Compilers are commonly divided into passes so that the front end may be shared by compilers for more than one machine (target language), and so that the back end may be shared by compilers for more than one source language. Prior to the dramatic increases in memory sizes of the mid- to late 1980s, compilers were also sometimes divided into passes to minimize memory usage: as each pass completed, the next could reuse its code space.

1.6.1 Lexical and Syntax Analysis

EXAMPLE 1.17

GCD program in Pascal

Consider the greatest common divisor (GCD) program introduced at the beginning of this chapter. Written in Pascal, the program might look like this:⁸

⁸ We use Pascal for this example because its lexical and syntactic structure is significantly simpler than that of most modern imperative languages.

```

program gcd(input, output);
var i, j : integer;
begin
  read(i, j);
  while i > j do
    if i > j then i := i - j
    else j := j - i;
  writeln(i)
end.

```

EXAMPLE 1.18

GCD program tokens

Scanning and parsing serve to recognize the structure of the program, without regard to its meaning. The scanner reads characters ('p', 'r', 'o', 'g', 'r', 'a', 'm', ' ', 'g', 'c', 'd', etc.) and groups them into *tokens*, which are the smallest meaningful units of the program. In our example, the tokens are

program	gcd	(input	,	output)	;
var	i	,	j	:	integer	;	begin
read	(i	,	j)	;	while
i	<>	j	do	if	i	>	j
then	i	:=	i	-	j	else	j
:=	j	-	i	;	writeln	(i
)	end	.					

Scanning is also known as *lexical analysis*. The principal purpose of the scanner is to simplify the task of the parser by reducing the size of the input (there are many more characters than tokens) and by removing extraneous characters. The scanner also typically removes comments, produces a listing if desired, and tags tokens with line and column numbers to make it easier to generate good diagnostics in later phases. One could design a parser to take characters instead of tokens as input—dispensing with the scanner—but the result would be awkward and slow.

EXAMPLE 1.19

Context-free grammar and parsing

Parsing organizes tokens into a *parse tree* that represents higher-level constructs in terms of their constituents. The ways in which these constituents combine are defined by a set of potentially recursive rules known as a *context-free grammar*. For example, we know that a Pascal program consists of the keyword *program*, followed by an identifier (the program name), a parenthesized list of files, a semicolon, a series of definitions, and the main *begin ... end* block, terminated by a period:

$$\text{program} \longrightarrow \text{PROGRAM id (id more_ids) ; block} .$$

where

$$\text{block} \longrightarrow \text{labels constants types variables subroutines BEGIN stmt more_stmts END}$$

and

$$\text{more_ids} \longrightarrow , \text{id more_ids}$$

or

$$\text{more_ids} \longrightarrow \epsilon$$

Here ϵ represents the empty string; it indicates that *more_ids* can simply be deleted. Many more grammar rules are needed, of course, to explain the full structure of a program.

A context-free grammar is said to define the *syntax* of the language; parsing is therefore known as *syntactic analysis*. There are many possible grammars for Pascal (an infinite number, in fact); the fragment shown above is based loosely on the “circles-and-arrows” syntax diagrams found in the original Pascal text [JW91]. A full parse tree for our GCD program (based on a full grammar not shown here) appears in Figure 1.3. Much of the complexity of this figure stems from (1) the use of such artificial “constructs” as *more_stmts* and *more_exprs* to represent lists of arbitrary length and (2) the use of the equally artificial *term*, *factor*, and so on, to capture precedence and associativity in arithmetic expressions. Grammars and parse trees will be covered in more detail in Chapter 2.

In the process of scanning and parsing, the compiler checks to see that all of the program’s tokens are well formed and that the sequence of tokens conforms to the syntax defined by the context-free grammar. Any malformed tokens (e.g., 123abc or \$@foo in Pascal) should cause the scanner to produce an error message. Any syntactically invalid token sequence (e.g., A := B C D in Pascal) should lead to an error message from the parser.

1.6.2 Semantic Analysis and Intermediate Code Generation

Semantic analysis is the discovery of *meaning* in a program. The semantic analysis phase of compilation recognizes when multiple occurrences of the same identifier are meant to refer to the same program entity, and ensures that the uses are consistent. In most languages the semantic analyzer tracks the *types* of both identifiers and expressions, both to verify consistent usage and to guide the generation of code in later phases.

To assist in its work, the semantic analyzer typically builds and maintains a *symbol table* data structure that maps each identifier to the information known about it. Among other things, this information includes the identifier’s type, internal structure (if any), and scope (the portion of the program in which it is valid).

Using the symbol table, the semantic analyzer enforces a large variety of rules that are not captured by the hierarchical structure of the context-free grammar and the parse tree. For example, it checks to make sure that

- Every identifier is declared before it is used.
- No identifier is used in an inappropriate context (calling an integer as a subroutine, adding a string to an integer, referencing a field of the wrong type of record, etc.).
- Subroutine calls provide the correct number and types of arguments.

EXAMPLE 1.20

GCD program parse tree

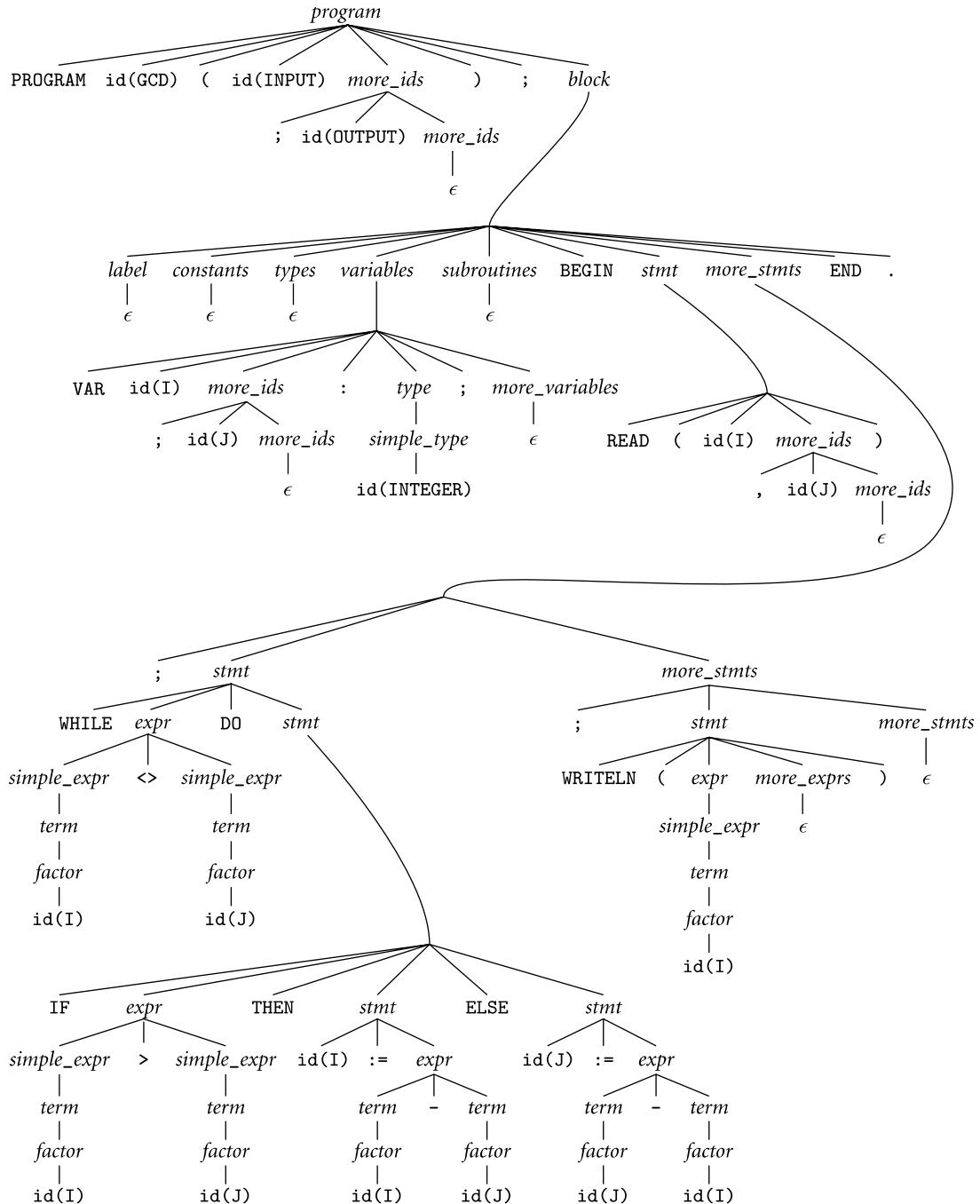


Figure 1.3 Parse tree for the GCD program. The symbol ϵ represents the empty string. The remarkable level of complexity in this figure is an artifact of having to fit the (much simpler) source code into the hierarchical structure of a context-free grammar.

- Labels on the arms of a `case` statement are distinct constants.
- Every function contains at least one statement that specifies a return value.

In many compilers, the work of the semantic analyzer takes the form of *semantic action routines*, invoked by the parser when it realizes that it has reached a particular point within a production.

Of course, not all semantic rules can be checked at compile time. Those that can are referred to as the *static semantics* of the language. Those that must be checked at run time are referred to as the *dynamic semantics* of the language. Examples of rules that must often be checked at run time include

- Variables are never used in an expression unless they have been given a value.⁹
- Pointers are never dereferenced unless they refer to a valid object.
- Array subscript expressions lie within the bounds of the array.
- Arithmetic operations do not overflow.

When it cannot enforce rules statically, a compiler will often produce code to perform appropriate checks at run time, aborting the program or generating an *exception* if one of the checks then fails. (Exceptions will be discussed in Section 8.5.) Some rules, unfortunately, may be unacceptably expensive or impossible to enforce, and the language implementation may simply fail to check them. In Ada, a program that breaks such a rule is said to be *erroneous*; in C its behavior is said to be *undefined*.

A parse tree is sometimes known as a *concrete syntax tree*, because it demonstrates, completely and concretely, how a particular sequence of tokens can be derived under the rules of the context-free grammar. Once we know that a token sequence is valid, however, much of the information in the parse tree is irrelevant to further phases of compilation. In the process of checking static semantic rules, the semantic analyzer typically transforms the parse tree into an *abstract syntax tree* (otherwise known as an *AST*, or simply a *syntax tree*) by removing most of the “artificial” nodes in the tree’s interior. The semantic analyzer also *annotates* the remaining nodes with useful information, such as pointers from identifiers to their symbol table entries. The annotations attached to a particular node are known as its *attributes*. A syntax tree for our GCD program is shown in Figure 1.4. ■

In many compilers, the annotated syntax tree constitutes the intermediate form that is passed from the front end to the back end. In other compilers, semantic analysis ends with a traversal of the tree that generates some other intermediate form. Often this alternative form resembles assembly language for an extremely simple idealized machine. In a suite of related compilers, the front ends

EXAMPLE 1.21

GCD program abstract syntax tree

9 As we shall see in Section 6.1.3, Java and C# actually do enforce initialization at compile time, but only by adopting a conservative set of rules for “definite assignment,” which outlaw programs for which correctness is difficult or impossible to verify at compile time.

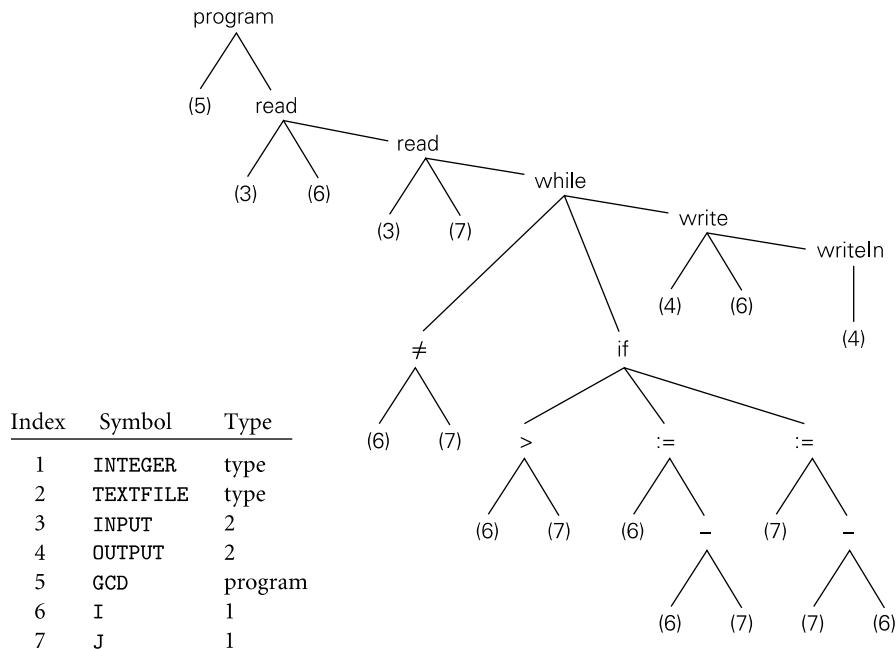


Figure 1.4 Syntax tree and symbol table for the GCD program. Unlike Figure 1.3, the syntax tree retains just the essential structure of the program, omitting detail that was needed only to drive the parsing algorithm.

for several languages and the back ends for several machines would share a common intermediate form.

1.6.3 Target Code Generation

The code generation phase of a compiler translates the intermediate form into the target language. Given the information contained in the syntax tree, generating correct code is usually not a difficult task (generating *good* code is harder, as we shall see in Section 1.6.4). To generate assembly or machine language, the code generator traverses the symbol table to assign locations to variables, and then traverses the syntax tree, generating loads and stores for variable references, interspersed with appropriate arithmetic operations, tests, and branches. Naive code for our GCD example appears in Figure 1.5, in MIPS assembly language. It was generated automatically by a simple pedagogical compiler.

The assembly language mnemonics may appear a bit cryptic, but the comments on each line (not generated by the compiler!) should make the correspondence between Figures 1.4 and 1.5 generally apparent. A few hints: `sp`, `ra`, `at`, `a0`, `v0`, and `t0–t9` are registers (special storage locations, limited in number, that can be accessed very quickly). `28(sp)` refers to the memory location 28 bytes beyond

EXAMPLE 1.22

GCD program assembly code

```

addiu  sp,sp,-32      # reserve room for local variables
sw     ra,20(sp)       # save return address
jal    getint          # read
nop
sw     v0,28(sp)       # store i
jal    getint          # read
nop
sw     v0,24(sp)       # store j
lw     t6,28(sp)       # load i
lw     t7,24(sp)       # load j
nop
beq   t6,t7,D         # branch if i = j
nop
A:   lw     t8,28(sp)   # load i
lw     t9,24(sp)       # load j
nop
slt   at,t9,t8        # determine whether j < i
beq   at,zero,B        # branch if not
nop
lw     t0,28(sp)       # load i
lw     t1,24(sp)       # load j
nop
subu  t2,t0,t1        # t2 := i - j
sw     t2,28(sp)       # store i
b     C
nop
B:   lw     t3,24(sp)   # load j
lw     t4,28(sp)       # load i
nop
subu  t5,t3,t4        # t5 := j - i
sw     t5,24(sp)       # store j
C:   lw     t6,28(sp)   # load i
lw     t7,24(sp)       # load j
nop
bne   t6,t7,A         # branch if i <> j
nop
D:   lw     a0,28(sp)   # load i
jal    putint          # writeln
nop
move  v0,zero          # exit status for program
b     E                # branch to E
nop
b     E                # branch to E
nop
E:   lw     ra,20(sp)   # retrieve return address
addiu sp,sp,32          # deallocate space for local variables
jr    ra                # return to operating system
nop

```

Figure 1.5 Naive MIPS assembly language for the GCD program.

the location whose address is in register `sp`. `Jal` is a subroutine call (“jump and link”); the first argument is passed in register `a0`, and the return value comes back in register `v0`. `Nop` is a “no-op”; it does no useful work but delays the program for one time cycle, allowing a two-cycle load or branch instruction to complete (branch and load delays were a common feature in early RISC machines; we will consider them in Section 5.5.1). Arithmetic operations generally operate on the second and third arguments, and put their result in the first. ■

Often a code generator will save the symbol table for later use by a symbolic debugger—for example, by including it as comments or some other nonexecutable part of the target code.

1.6.4 Code Improvement

Code improvement is often referred to as *optimization*, though it seldom makes anything optimal in any absolute sense. It is an optional phase of compilation whose goal is to transform a program into a new version that computes the same result more efficiently—more quickly or using less memory, or both.

Some improvements are machine independent. These can be performed as transformations on the intermediate form. Other improvements require an understanding of the target machine (or of whatever will execute the program in the target language). These must be performed as transformations on the target program. Thus code improvement often appears as two additional phases of compilation, one immediately after semantic analysis and intermediate code generation, the other immediately after target code generation.

EXAMPLE 1.23

GCD program optimization

Applying a good code improver to the code in Figure 1.5 produces the code shown in Example 1.2 (page 3). Comparing the two programs, we can see that the improved version is quite a lot shorter. Conspicuously absent are most of the loads and stores. The machine-independent code improver is able to verify that `i` and `j` can be kept in registers throughout the execution of the main loop (this would not have been the case if, for example, the loop contained a call to a subroutine that might reuse those registers, or that might try to modify `i` or `j`). The machine-specific code improver is then able to assign `i` and `j` to actual registers of the target machine. In our example the machine-specific improver is also able to *schedule* (reorder) instructions to eliminate several of the no-ops. Careful examination of the instructions following the loads and branches will reveal that they can be executed safely even when the load or branch has not yet completed. For modern microprocessor architectures, particularly those with so-called *superscalar* RISC instruction sets (ones in which separate functional units can execute multiple instructions simultaneously), compilers can usually generate better code than can human assembly language programmers. ■

 **CHECK YOUR UNDERSTANDING**

20. List the principal phases of compilation, and describe the work performed by each.
21. Describe the form in which a program is passed from the scanner to the parser; from the parser to the semantic analyzer; from the semantic analyzer to the intermediate code generator.
22. What distinguishes the front end of a compiler from the back end?
23. What is the difference between a phase and a pass of compilation? Under what circumstances does it make sense for a compiler to have multiple passes?
24. What is the purpose of the compiler's symbol table?
25. What is the difference between static and dynamic semantics?
26. On modern machines, do assembly language programmers still tend to write better code than a good compiler can? Why or why not?

1.7**Summary and Concluding Remarks**

In this chapter we introduced the study of programming language design and implementation. We considered why there are so many languages, what makes them successful or unsuccessful, how they may be categorized for study, and what benefits the reader is likely to gain from that study. We noted that language design and language implementation are intimately related to one another. Obviously an implementation must conform to the rules of the language. At the same time, a language designer must consider how easy or difficult it will be to implement various features, and what sort of performance is likely to result for programs that use those features.

Language implementations are commonly differentiated into those based on interpretation and those based on compilation. We noted, however, that the difference between these approaches is fuzzy, and that most implementations include a bit of each. As a general rule, we say that a language is compiled if execution is preceded by a translation step that (1) fully analyzes both the structure (syntax) and meaning (semantics) of the program and (2) produces an equivalent program in a significantly different form. The bulk of the implementation material in this book pertains to compilation.

Compilers are generally structured as a series of *phases*. The first few phases—scanning, parsing, and semantic analysis—serve to analyze the source program. Collectively these phases are known as the compiler's *front end*. The final few phases—intermediate code generation, code improvement, and target code generation—are known as the *back end*. They serve to build a tar-

get program—preferably a fast one—whose semantics match those of the source.

Chapters 3, 6, 7, 8, and 9 form the core of the rest of this book. They cover fundamental issues of language design, both from the point of view of the programmer and from the point of view of the language implementor. To support the discussion of implementations, Chapters 2 and 4 describe compiler front ends in more detail than has been possible in this introduction. Chapter 5 provides an overview of assembly-level architecture. Chapters 14 and 15 discuss compiler back ends, including assemblers and linkers. Additional language paradigms are covered in Chapters 10 through 13. Appendix A lists the principal programming languages mentioned in the text, together with a genealogical chart and bibliographic references. Appendix B contains a list of “Design and Implementation” sidebars. Appendix C contains a list of numbered examples.

I.8 Exercises

- 1.1 Errors in a computer program can be classified according to when they are detected and, if they are detected at compile time, what part of the compiler detects them. Using your favorite imperative language, give an example of each of the following.
 - (a) A lexical error, detected by the scanner
 - (b) A syntax error, detected by the parser
 - (c) A static semantic error, detected by semantic analysis
 - (d) A dynamic semantic error, detected by code generated by the compiler
 - (e) An error that the compiler can neither catch nor easily generate code to catch (this should be a violation of the language definition, not just a program bug)
- 1.2 Algol family languages are typically compiled, while Lisp family languages, in which many issues cannot be settled until run time, are typically interpreted. Is interpretation simply what one “has to do” when compilation is infeasible, or are there actually some *advantages* to interpreting a language, even when a compiler is available?
- 1.3 The gcd program of Example 1.17 might also be written

```
program gcd(input, output);
var i, j : integer;
begin
  read(i, j);
  while i <> j do
    if i > j then i := i mod j
    else j := j mod i;
  writeln(i)
end.
```

Does this program compute the same result? If not, can you fix it? Under what circumstances would you expect one or the other to be faster?

- 1.4 In your local implementation of C, what is the limit on the size of integers? What happens in the event of arithmetic overflow? What are the implications of size limits on the portability of programs from one machine/compiler to another? How do the answers to these questions differ for Java? For Ada? For Pascal? For Scheme? (You may need to find a manual.)
- 1.5 The Unix `make` utility allows the programmer to specify *dependences* among the separately compiled pieces of a program. If file *A* depends on file *B* and file *B* is modified, `make` deduces that *A* must be recompiled, in case any of the changes to *B* would affect the code produced for *A*. How accurate is this sort of dependence management? Under what circumstances will it lead to unnecessary work? Under what circumstances will it fail to recompile something that needs to be recompiled?
- 1.6 Why is it difficult to tell whether a program is correct? How do you go about finding bugs in your code? What kinds of bugs are revealed by testing? What kinds of bugs are not? (For more formal notions of program correctness, see the bibliographic notes at the end of Chapter 4.)

1.9 Explorations

- 1.7 (a) What was the first programming language you learned? If you chose it, why did you do so? If it was chosen for you by others, why do you think they chose it? What parts of the language did you find the most difficult to learn?
(b) For the language with which you are most familiar (this may or may not be the first one you learned), list three things you wish had been differently designed. Why do you think they were designed the way they were? How would you fix them if you had the chance to do it over? Would there be any negative consequences—for example, in terms of compiler complexity or program execution speed?
- 1.8 Get together with a classmate whose principal programming experience is with a language in a different category of Figure 1.1. (If your experience is mostly in C, for example, you might search out someone with experience in Lisp.) Compare notes. What are the easiest and most difficult aspects of programming, in each of your experiences? Pick some simple problem (e.g., sorting, or identification of connected components in a graph) and solve it using each of your favorite languages. Which solution is more elegant (do the two of you agree)? Which is faster? Why?

- 1.9 (a) If you have access to a Unix system, compile a simple program with the `-S` command-line flag. Add comments to the resulting assembly language file to explain the purpose of each instruction.
(b) Now use the `-o` command-line flag to generate a *relocatable object file*. Using appropriate local tools (look in particular for `nm`, `objdump`, or a symbolic debugger like `gdb` or `dbx`), identify the machine language corresponding to each line of assembler.
(c) Using `nm`, `objdump`, or a similar tool, identify the *undefined external symbols* in your object file. Now run the compiler to completion, to produce an *executable* file. Finally, run `nm` or `objdump` again to see what has happened to the symbols in part (b). Where did they come from, and how did the linker resolve them?
(d) Run the compiler to completion one more time, using the `-v` command-line flag. You should see messages describing the various subprograms invoked during the compilation process (some compilers use a different letter for this option; check the `man` page). The subprograms may include a preprocessor, separate passes of the compiler itself (often two), probably an assembler, and the linker. If possible, run these subprograms yourself, individually. Which of them produce the files described in the previous subquestions? Explain the purpose of the various command-line flags with which the subprograms were invoked.
- 1.10 Write a program that commits a dynamic semantic error (e.g., division by zero, access off the end of an array, dereference of a `nil` pointer). What happens when you run this program? Does the compiler give you options to control what happens? Devise an experiment to evaluate the cost of runtime semantic checks. If possible, try this exercise with more than one language or compiler.
- 1.11 C has a reputation for being a relatively “unsafe” high-level language. In particular, it allows the programmer to mix operands of different sizes and types in many more ways than do its “safer” cousins. The Unix `lint` utility can be used to search for potentially unsafe constructs in C programs. In effect, many of the rules that are enforced by the compiler in other languages are optional in C and are enforced (if desired) by a separate program. What do you think of this approach? Is it a good idea? Why or why not?
- 1.12 Using an Internet search engine or magazine indexing service, read up on the history of Java and C#, including the conflict between Sun and Microsoft over Java standardization. Some have claimed that C# is, at least in part, Microsoft’s attempt to kill Java. Defend or refute this claim.

1.10 Bibliographic Notes

The compiler-oriented chapters of this book attempt to convey a sense of what the compiler does, rather than explaining how to build one. A much greater level of detail can be found in other texts. Leading options include the work of Cooper and Torczon [CT04], Grune et al. [GBJL01], and Appel [App97]. The older texts by Aho, Sethi, and Ullman [ASU86] and Fischer and LeBlanc [FL88] were for many years the standards in the field, but have grown somewhat dated. High-quality texts on programming language design include those of Louden [Lou03], Sebesta [Seb04], and Sethi [Set96].

Some of the best information on the history of programming languages can be found in the proceedings of conferences sponsored by the Association for Computing Machinery in 1978 and 1993 [Wex78, Ass93]. Another excellent reference is Horowitz's 1987 text [Hor87]. A broader range of historical material can be found in the quarterly *IEEE Annals of the History of Computing*. Given the importance of personal taste in programming language design, it is inevitable that some language comparisons should be marked by strongly worded opinions. Examples include the writings of Dijkstra [Dij82], Hoare [Hoa81], Kernighan [Ker81], and Wirth [Wir85a].

Most personal computer software development now takes place in integrated programming environments. Influential precursors to these environments include the Genera Common Lisp environment from Symbolics Corp. [WMWM87] and the Smalltalk [Gol84], Interlisp [TM81], and Cedar [SZBH86] environments at the Xerox Palo Alto Research Center.

2

Programming Language Syntax

Unlike natural languages such as English or Chinese, computer languages must be precise. Both their form (syntax) and meaning (semantics) must be specified without ambiguity so that both programmers and computers can tell what a program is supposed to do. To provide the needed degree of precision, language designers and implementors use formal syntactic and semantic notation. To facilitate the discussion of language features in later chapters, we will cover this notation first: syntax in the current chapter and semantics in Chapter 4.

EXAMPLE 2.1

Syntax of Arabic numerals

As a motivating example, consider the Arabic numerals with which we represent numbers. These numerals are composed of digits, which we can enumerate as follows (“|” means “or”):

$$\text{digit} \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Digits are the syntactic building blocks for numbers. In the usual notation, we say that a natural number is represented by an arbitrary-length (nonempty) string of digits, beginning with a nonzero digit:

$$\begin{aligned}\text{non_zero_digit} &\longrightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \text{natural_number} &\longrightarrow \text{non_zero_digit} \text{ digit } *\end{aligned}$$

Here the “Kleene¹ star” metasymbol (*) is used to indicate zero or more repetitions of the symbol to its left. ■

Of course, digits are only symbols: ink blobs on paper or pixels on a screen. They carry no meaning in and of themselves. We add semantics to digits when we say that they represent the natural numbers from zero to nine, as defined by mathematicians. Alternatively, we could say that they represent colors, or the days of the week in a decimal calendar. These would constitute alternative semantics for the same syntax. In a similar fashion, we define the semantics of natural numbers by associating a base-10, place-value interpretation with each string of

¹ Stephen Kleene (1909–1994), a mathematician at the University of Wisconsin, was responsible for much of the early development of the theory of computation, including much of the material in Section 2.4.

digits. Similar syntax rules and semantic interpretations can be devised for rational numbers, (limited-precision) real numbers, arithmetic, assignments, control flow, declarations, and indeed all of programming languages.

Distinguishing between syntax and semantics is useful for at least two reasons. First, different programming languages often provide features with very similar semantics but very different syntax. It is generally much easier to learn a new language if one is able to identify the common (and presumably familiar) ideas beneath the unfamiliar syntax. Second, there are some very efficient and elegant algorithms that a compiler or interpreter can use to discover the syntactic structure (but not the semantics!) of a computer program, and these algorithms can be used to drive the rest of the compilation or interpretation process.

In the current chapter we focus on syntax: how we specify the structural rules of a programming language, and how a compiler identifies the structure of a given input program. These two tasks—specifying syntax rules and figuring out how (and whether) a given program was built according to those rules—are distinct. The first is of interest mainly to programmers, who want to write valid programs. The second is of interest mainly to compilers, which need to analyze those programs. The first task relies on *regular expressions* and *context-free grammars*, which specify how to generate valid programs. The second task relies on *scanners* and *parsers*, which recognize program structure. We address the first of these tasks in Section 2.1, the second in Sections 2.2 and 2.3.

In Section 2.4 (largely on the PLP CD) we take a deeper look at the formal theory underlying scanning and parsing. In theoretical parlance, a scanner is a *deterministic finite automaton* (DFA) that recognizes the tokens of a programming language. A parser is a deterministic *push-down automaton* (PDA) that recognizes the language's context-free syntax. It turns out that one can generate scanners and parsers automatically from regular expressions and context-free grammars. This task is performed by tools like Unix's `lex` and `yacc`.² Possibly nowhere else in computer science is the connection between theory and practice so clear and so compelling.

2.1 Specifying Syntax: Regular Expressions and Context-Free Grammars

Formal specification of syntax requires a set of rules. How complicated (expressive) the syntax can be depends on the kinds of rules we are allowed to use. It turns out that what we intuitively think of as tokens can be constructed from

2 At many sites, `lex` and `yacc` have been superseded by the GNU `flex` and `bison` tools. These independently developed, noncommercial alternatives are available without charge from the Free Software Foundation at www.gnu.org/software. They provide a superset of the functionality of `lex` and `yacc`.

individual characters using just three kinds of formal rules: concatenation, alternation (choice among a finite set of alternatives), and so-called “Kleene closure” (repetition an arbitrary number of times). Specifying most of the rest of what we intuitively think of as syntax requires one additional kind of rule: recursion (creation of a construct from simpler instances of the same construct). Any set of strings that can be defined in terms of the first three rules is called a *regular set*, or sometimes a *regular language*. Regular sets are generated by *regular expressions* and recognized by scanners. Any set of strings that can be defined if we add recursion is called a *context-free language* (CFL). Context-free languages are generated by *context-free grammars* (CFGs) and recognized by parsers. (Terminology can be confusing here. The meaning of the word *language* varies greatly, depending on whether we’re talking about “formal” languages [e.g., regular or context-free] or programming languages. A formal language is just a set of strings, with no accompanying semantics.)

2.1.1 Tokens and Regular Expressions

Tokens are the basic building blocks of programs. They include keywords, identifiers, numbers, and various kinds of symbols. Pascal, which is a fairly simple language, has 64 kinds of tokens, including 21 symbols (+, -, ;, :=, . . . , etc.), 35 keywords (`begin`, `end`, `div`, `record`, `while`, etc.), integer literals (e.g., 137), real (floating-point) literals (e.g., 6.022e23), quoted character/string literals (e.g., ‘snerk’), identifiers (`MyVariable`, `YourType`, `maxint`, `readln`, etc., 39 of which are predefined), and two different kinds of comments.

Upper- and lowercase letters in identifiers and keywords are considered distinct in some languages (e.g., Modula-2/3 and C and its descendants), and identical in others (e.g., Ada, Common Lisp, Fortran 90, and Pascal). Thus `foo`, `Foo`, and `FOO` all represent the same identifier in Ada but different identifiers in C. Modula-2 and Modula-3 require keywords and predefined (built-in) identifiers to be written in uppercase; C and its descendants require them to be written in lowercase. A few languages (notably Modula-3 and Standard Pascal) allow only letters and digits in identifiers. Most (including many actual implementations of Pascal) allow underscores. A few (notably Lisp) allow a variety of additional characters. Some languages (e.g., Java, C#, and Modula-3) have standard conventions on the use of upper- and lowercase letters in names.³

With the globalization of computing, non-Latin character sets have become increasingly important. Many modern languages, including C99, C++, Ada 95, Java, C#, and Fortran 2003, have explicit support for multibyte character sets, generally based on the Unicode and ISO/IEC 10646 international standards. Most modern programming languages allow non-Latin characters to appear with in

³ For the sake of consistency we do not always obey such conventions in this book. Most examples follow the common practice of C programmers, in which underscores, rather than capital letters, separate the “subwords” of names.

comments and character strings; an increasing number allow them in identifiers as well. Conventions for portability across character sets and for *localization* to a given character set can be surprisingly complex, particularly when various forms of backward compatibility are required (the C99 Rationale devotes five full pages to this subject [Int99, pp. 19–23]); for the most part we ignore such issues here.

Some language implementations impose limits on the maximum length of identifiers, but most avoid such unnecessary restrictions. Most modern languages are also more-or-less *free format*, meaning that a program is simply a sequence of tokens: what matters is their order with respect to one another, not their physical position within a printed line or page. “White space” (blanks, tabs, carriage returns, and line and page feed characters) between tokens is usually ignored, except to the extent that it is needed to separate one token from the next. There are a few exceptions to these rules. Some language implementations limit the maximum length of a line, to allow the compiler to store the current line in a fixed-length buffer. Dialects of Fortran prior to Fortran 90 use a *fixed format*, with 72 characters per line (the width of a paper punch card, on which programs were once stored) and with different columns within the line reserved for different purposes. Line breaks serve to separate statements in several other languages, including Haskell, Occam, SR, Tcl, and Python. Haskell, Occam, and Python also give special significance to indentation. The body of a loop, for example, consists of precisely those subsequent lines that are indented farther than the header of the loop.

To specify tokens, we use the notation of regular expressions. A regular expression is one of the following.

1. A character
2. The empty string, denoted ϵ
3. Two regular expressions next to each other, meaning any string generated by the first one followed by (concatenated with) any string generated by the second one
4. Two regular expressions separated by a vertical bar ($|$), meaning any string generated by the first one *or* any string generated by the second one

DESIGN & IMPLEMENTATION

Formatting restrictions

Formatting limitations inspired by implementation concerns—as in the punch-card-oriented rules of Fortran 77 and its predecessors—have a tendency to become unwanted anachronisms as implementation techniques improve. Given the tendency of certain word processors to “fill” or auto-format text, the line break and indentation rules of languages like Haskell, Occam, and Python are somewhat controversial.

5. A regular expression followed by a Kleene star, meaning the concatenation of zero or more strings generated by the expression in front of the star

Parentheses are used to avoid ambiguity about where the various subexpressions start and end.⁴

Returning to the example of Pascal, numeric literals can be generated by the following regular expressions.⁵

```
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
unsigned_integer → digit digit*
unsigned_number → unsigned_integer (( . unsigned_integer) |  $\epsilon$ )
                    ((( $\epsilon$  | E) (+ | - |  $\epsilon$ ) unsigned_integer) |  $\epsilon$ )
```

To generate a valid string, we scan the regular expression from left to right, choosing among alternatives at each vertical bar, and choosing a number of repetitions at each Kleene star. Within each repetition we may make different choices at vertical bars, generating different substrings. Note that while we have allowed later definitions to build on earlier ones, nothing is ever defined in terms of itself. Such recursive definitions are the distinguishing characteristic of context-free grammars, described in Section 2.1.2. ■

Many readers will be familiar with regular expressions from the grep family of tools in Unix, the search facilities of various text editors (notably emacs), or such scripting languages and tools as Perl, Python, Ruby, awk, and sed. Most of these provide a rich set of extensions to the notation of regular expressions. Some extensions, such as shorthand for “zero or one occurrences” or “anything other than white space” do not change the power of the notation. Others, such as the ability to require a second occurrence later in the input string of the same character sequence that matched an earlier part of the expression, increase the power of the notation, so it is no longer restricted to generating regular sets. Still other extensions are designed not to increase the expressiveness of the notation but rather to tie it to other language facilities. In many tools, for example, one can bracket portions of a regular expression in such a way that when a string is matched against it the contents of the corresponding substrings are assigned into named local variables. We will return to these issues in Section 13.4.2, in the context of scripting languages.

4 Some authors use λ to represent the empty string. Some use a period (.), rather than juxtaposition, to indicate concatenation. Some use a plus sign (+), rather than a vertical bar, to indicate alternation.

5 Numeric literals in many languages are significantly more complex. Java, for example, supports both 32 and 64-bit integer constants, in decimal, octal, and hexadecimal.

EXAMPLE 2.2

Syntax of numbers in Pascal

2.1.2 Context-Free Grammars

EXAMPLE 2.3

Syntactic nesting in expressions

$$\begin{aligned} \text{expr} &\longrightarrow \text{id} \mid \text{number} \mid - \text{expr} \mid (\text{expr}) \\ &\quad \mid \text{expr} \text{ op } \text{expr} \\ \text{op} &\longrightarrow + \mid - \mid * \mid / \end{aligned}$$

Here the ability to define a construct in terms of itself is crucial. Among other things, it allows us to ensure that left and right parentheses are matched, something that cannot be accomplished with regular expressions (see Section 2.4.3 for more details). ■

Each of the rules in a context-free grammar is known as a *production*. The symbols on the left-hand sides of the productions are known as *variables*, or *non-terminals*. There may be any number of productions with the same left-hand side. Symbols that are to make up the strings derived from the grammar are known as *terminals* (shown here in typewriter font). They cannot appear on the left-hand side of any production. In a programming language, the terminals of the context-free grammar are the language’s tokens. One of the nonterminals, usually the one on the left-hand side of the first production, is called the *start symbol*. It names the construct defined by the overall grammar.

The notation for context-free grammars is sometimes called Backus-Naur Form (BNF), in honor of John Backus and Peter Naur, who devised it for the definition of the Algol 60 programming language [NBB⁺63].⁶ Strictly speaking, the Kleene star and meta-level parentheses of regular expressions are not allowed in BNF, but they do not change the expressive power of the notation and are commonly included for convenience. Sometimes one sees a “Kleene plus” (+) as well; it indicates one or more instances of the symbol or group of symbols in front of it.⁷ When augmented with these extra operators, the notation is often called extended BNF (EBNF). The construct

$$\text{id_list} \longrightarrow \text{id} (, \text{id})^*$$

is shorthand for

$$\begin{aligned} \text{id_list} &\longrightarrow \text{id} \\ \text{id_list} &\longrightarrow \text{id_list} , \text{id} \end{aligned}$$

⁶ John Backus (1924–), is also the inventor of Fortran. He spent most of his professional career at IBM Corporation, and was named an IBM Fellow in 1987. He received the ACM Turing Award in 1977.

⁷ Some authors use curly braces ({}) to indicate zero or more instances of the symbols inside. Some use square brackets ([]) to indicate zero or one instance of the symbols inside—that is, to indicate that those symbols are optional.

EXAMPLE 2.4

Extended BNF (EBNF)

“Kleene plus” is analogous. The vertical bar is also in some sense superfluous, though it was provided in the original BNF. The construct

$$op \longrightarrow + \mid - \mid * \mid /$$

can be considered shorthand for

$$\begin{aligned} op &\longrightarrow + \\ op &\longrightarrow - \\ op &\longrightarrow * \\ op &\longrightarrow / \end{aligned}$$

which is also sometimes written

$$\begin{aligned} op &\longrightarrow + \\ &\longrightarrow - \\ &\longrightarrow * \\ &\longrightarrow / \end{aligned}$$

Many tokens, such as `id` and `number` above, have many possible spellings (i.e., may be represented by many possible strings of characters). The parser is oblivious to these; it does not distinguish one identifier from another. The semantic analyzer does distinguish them, however, so the scanner must save the spelling of each “interesting” token for later use.

2.1.3 Derivations and Parse Trees

A context-free grammar shows us how to generate a syntactically valid string of terminals: begin with the start symbol. Choose a production with the start symbol on the left-hand side; replace the start symbol with the right-hand side of that production. Now choose a nonterminal A in the resulting string, choose a production P with A on its left-hand side, and replace A with the right-hand side of P . Repeat this process until no nonterminals remain.

EXAMPLE 2.5

Derivation of `slope * x + intercept`

$$\begin{aligned} \text{expr} &\implies \text{expr } op \text{ } \underline{\text{expr}} \\ &\implies \text{expr } op \text{ id} \\ &\implies \underline{\text{expr}} + \text{id} \\ &\implies \text{expr } op \underline{\text{expr}} + \text{id} \\ &\implies \text{expr } op \text{ id} + \text{id} \\ &\implies \underline{\text{expr}} * \text{id} + \text{id} \\ &\implies \text{id} * \text{id} + \text{id} \\ &\quad (\text{slope}) \quad (\text{x}) \quad (\text{intercept}) \end{aligned}$$

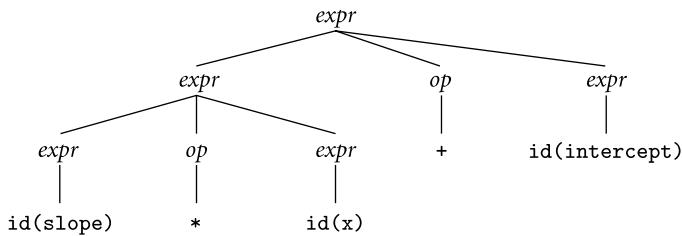


Figure 2.1 Parse tree for $\text{slope} * \text{x} + \text{intercept}$ (grammar in Example 2.3).

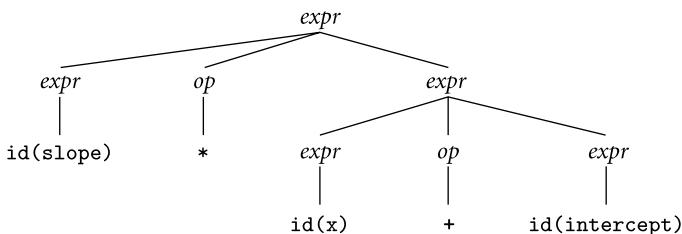


Figure 2.2 Alternative (less desirable) parse tree for $\text{slope} * \text{x} + \text{intercept}$ (grammar in Example 2.3). The fact that more than one tree exists implies that our grammar is ambiguous.

The \Rightarrow metasymbol indicates that the right-hand side was obtained by using a production to replace some nonterminal in the left-hand side. At each line we have underlined the symbol A that is replaced in the following line. ■

A series of replacement operations that shows how to derive a string of terminals from the start symbol is called a *derivation*. Each string of symbols along the way is called a *sentential form*. The final sentential form, consisting of only terminals, is called the *yield* of the derivation. We sometimes elide the intermediate steps and write $\text{expr} \Rightarrow^* \text{slope} * \text{x} + \text{intercept}$, where the metasymbol \Rightarrow^* means “yields after zero or more replacements.” In this particular derivation, we have chosen at each step to replace the right-most nonterminal with the right-hand side of some production. This replacement strategy leads to a *right-most* derivation, also called a *canonical* derivation. There are many other possible derivations, including *left-most* and options in-between. Most parsers are designed to find a particular derivation (usually the left-most or right-most).

We saw in Chapter 1 that we can represent a derivation graphically as a *parse tree*. The root of the parse tree is the start symbol of the grammar. The leaves of the tree are its yield. Each internal node, together with its children, represents the use of a production.

A parse tree for our example expression appears in Figure 2.1. This tree is not unique. At the second level of the tree, we could have chosen to turn the operator into a $*$ instead of a $+$, and to further expand the expression on the right, rather than the one on the left (see Figure 2.2). The fact that some strings are the yield of more than one parse tree tells us that our grammar is *ambiguous*. Ambiguity

EXAMPLE 2.6

Parse trees for $\text{slope} * \text{x} + \text{intercept}$

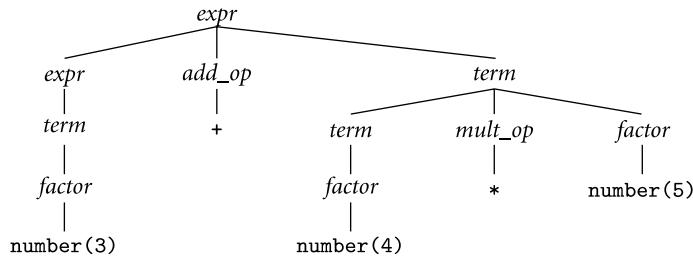


Figure 2.3 Parse tree for $3 + 4 * 5$, with precedence (grammar in Example 2.7).

turns out to be a problem when trying to build a parser: it requires some extra mechanism to drive a choice between equally acceptable alternatives. ■

A moment's reflection will reveal that there are infinitely many context-free grammars for any given context-free language. Some of these grammars are much more useful than others. In this text we will avoid the use of ambiguous grammars (though most parser generators allow them, by means of *disambiguating* rules). We will also avoid the use of so-called *useless* symbols: nonterminals that cannot generate any string of terminals, or terminals that cannot appear in the yield of any derivation.

When designing the grammar for a programming language, we generally try to find one that reflects the internal structure of programs in a way that is useful to the rest of the compiler. (We shall see in Section 2.3.2 that we also try to find one that can be parsed efficiently, which can be a bit of a challenge.) One place in which structure is particularly important is in arithmetic expressions, where we can use productions to capture the *associativity* and *precedence* of the various operators. Associativity tells us that the operators in most languages group left-to-right, so $10 - 4 - 3$ means $(10 - 4) - 3$ rather than $10 - (4 - 3)$. Precedence tells us that multiplication and division in most languages group more tightly than addition and subtraction, so $3 + 4 * 5$ means $3 + (4 * 5)$ rather than $(3 + 4) * 5$. (These rules are not universal; we will consider them again in Section 6.1.1.)

Here is a better version of our expression grammar.

1. $\text{expr} \rightarrow \text{term} \mid \text{expr } \text{add_op } \text{term}$
2. $\text{term} \rightarrow \text{factor} \mid \text{term } \text{mult_op } \text{factor}$
3. $\text{factor} \rightarrow \text{id} \mid \text{number} \mid - \text{factor} \mid (\text{expr})$
4. $\text{add_op} \rightarrow + \mid -$
5. $\text{mult_op} \rightarrow * \mid /$

This grammar is unambiguous. It captures precedence in the way *factor*, *term*, and *expr* build on one another, with different operators appearing at each level. It captures associativity in the second halves of lines 1 and 2, which build sub*exprs* and sub*terms* to the left of the operator, rather than to the right. In Figure 2.3, we can see how building the notion of precedence into the grammar makes it clear

EXAMPLE 2.7

Expression grammar with precedence and associativity

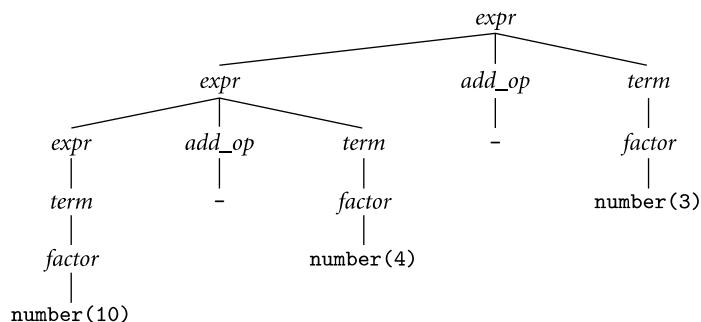


Figure 2.4 Parse tree for $10 - 4 - 3$, with left associativity (grammar in Example 2.7).

that multiplication groups more tightly than addition in $3 + 4 * 5$, even without parentheses. In Figure 2.4, we can see that subtraction groups more tightly to the left, so $10 - 4 - 3$ would evaluate to 3 rather than to 9. ■

CHECK YOUR UNDERSTANDING

1. What is the difference between syntax and semantics?
2. What are the three basic operations that can be used to build complex regular expressions from simpler regular expressions?
3. What additional operation (beyond the three of regular expressions) is provided in context-free grammars?
4. What is *Backus-Naur form*? When and why was it devised?
5. Name a language in which indentation affects program syntax.
6. When discussing context-free languages, what is a *derivation*? What is a *sentential form*?
7. What is the difference between a *right-most* derivation and a *left-most* derivation? Which one of them is also called *canonical*?
8. What does it mean for a context-free grammar to be *ambiguous*?
9. What are *associativity* and *precedence*? Why are they significant in parse trees?

2.2 Scanning

Together, the scanner and parser for a programming language are responsible for discovering the syntactic structure of a program. This process of discovery, or *syntax analysis*, is a necessary first step toward translating the program into

an equivalent program in the target language. (It's also the first step toward interpreting the program directly. In general, we will focus on compilation, rather than interpretation, for the remainder of the book. Most of what we shall discuss either has an obvious application to interpretation or is obviously irrelevant to it.)

By grouping input characters into tokens, the scanner dramatically reduces the number of individual items that must be inspected by the more computationally intensive parser. In addition, the scanner typically removes comments (so the parser doesn't have to worry about them appearing throughout the context-free grammar); saves the text of “interesting” tokens like identifiers, strings, and numeric literals; and tags tokens with line and column numbers to make it easier to generate high-quality error messages in later phases.

Suppose for a moment that we are writing a scanner for Pascal.⁸ We might sketch the process as shown in Figure 2.5. The structure of the code is entirely up to the programmer, but it seems reasonable to check the simpler and more common cases first, to peek ahead when we need to, and to embed loops for comments and for long tokens such as identifiers, numbers, and strings.

After announcing a token the scanner returns to the parser. When invoked again it repeats the algorithm from the beginning, using the next available characters of input (including any look-ahead that was peeked at but not consumed the last time). ■

As a rule, we accept the longest possible token in each invocation of the scanner. Thus `foobar` is always `foobar` and never `f` or `foo` or `foob`. More to the point, `3.14159` is a real number and never `3`, `.`, and `14159`. White space (blanks,

DESIGN & IMPLEMENTATION

Nested comments

Nested comments can be handy for the programmer (e.g., for temporarily “commenting out” large blocks of code). Scanners normally deal only with nonrecursive constructs, however, so nested comments require special treatment. Some languages disallow them. Others require the language implementor to augment the scanner with special purpose comment-handling code. C++ and C99 strike a compromise: `/* ... */` style comments are not allowed to nest, but `/* ... */` and `//...` style comments can appear inside each other. The programmer can thus use one style for “normal” comments and the other for “commenting out.” (The C99 designers note, however, that conditional compilation (`#if`) is preferable [Int03a, p. 58].)

⁸ As in Example 1.17, we use Pascal for this example because its lexical structure is significantly simpler than that of most modern imperative languages.

```

we skip any initial white space (spaces, tabs, and newlines)
we read the next character
if it is a ( we look at the next character
    if that is a * we have a comment;
        we skip forward through the terminating *)
        otherwise we return a left parenthesis and reuse the look-ahead
if it is one of the one-character tokens ([ ] , ; = + - etc.)
    we return that token
if it is a . we look at the next character
    if that is a . we return .. †
    otherwise we return . and reuse the look-ahead
if it is a < we look at the next character
    if that is a = we return <=
    otherwise we return < and reuse the look-ahead
etc.
if it is a letter we keep reading letters and digits
    and maybe underscores until we can't anymore;
    then we check to see if it is a keyword
        if so we return the keyword
        otherwise we return an identifier
    in either case we reuse the character beyond the end of the token
if it is a digit we keep reading until we find a nondigit
    if that is not a . we return an integer and reuse the nondigit
    otherwise we keep looking for a real number
        if the character after the . is not a digit we return an integer
        and reuse the . and the look-ahead
etc.

```

Figure 2.5 Outline of an ad hoc Pascal scanner. Only a fraction of the code is shown.

†The double-dot .. token is used to specify ranges in Pascal (e.g., `type day = 1..31`).

tabs, carriage returns, comments) is generally ignored, except to the extent that it separates tokens (e.g., `foo bar` is different from `foobar`).

It is not difficult to flesh out Figure 2.5 by hand to produce code in some programming language. This ad hoc style of scanner is often used in production compilers; the code is fast and compact. In some cases, however, it makes sense to build a scanner in a more structured way, as an explicit representation of a *finite automaton*. An example of such an automaton, for part of a Pascal scanner, appears in Figure 2.6. The automaton starts in a distinguished initial state. It then moves from state to state based on the next available character of input. When it reaches one of a designated set of final states it recognizes the token associated with that state. The “longest possible token” rule means that the scanner returns to the parser only when the next character cannot be used to continue the current token. ■

EXAMPLE 2.9

Finite automaton for part
of a Pascal scanner

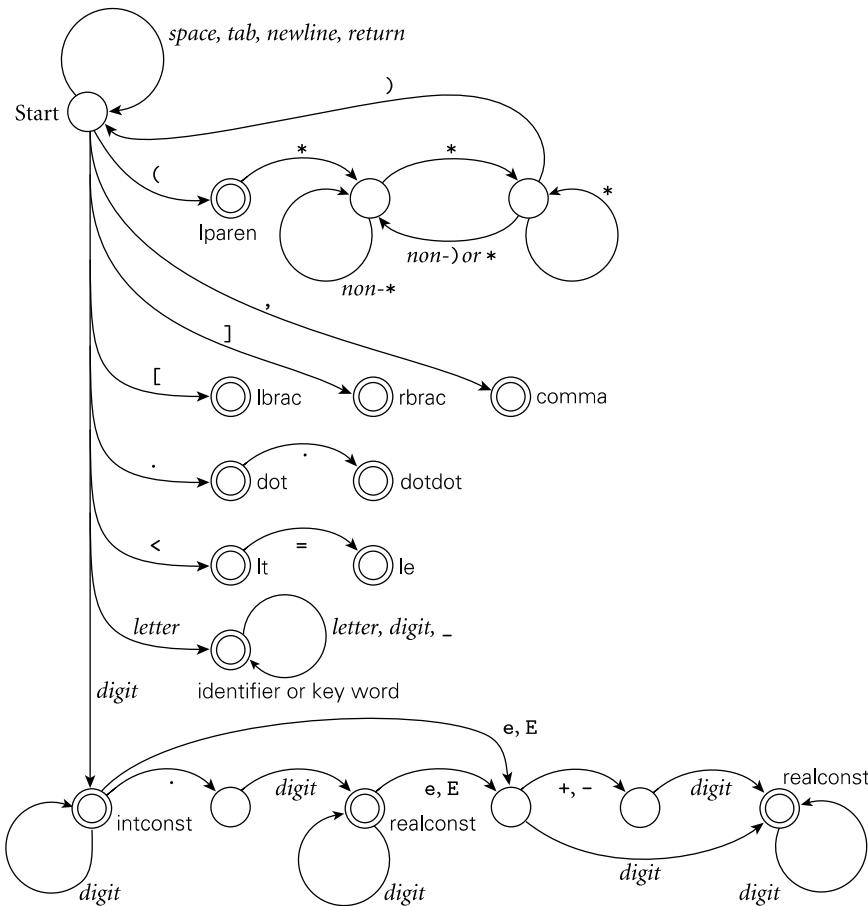


Figure 2.6 Pictorial representation of (part of) a Pascal scanner as a finite automaton. Scanning for each token begins in the state marked "Start." The final states, in which a token is recognized, are indicated by double circles.

2.2.1 Generating a Finite Automaton

While a finite automaton can in principle be written by hand, it is more common to build one automatically from a set of regular expressions, using a *scanner generator* tool. Because regular expressions are significantly easier to write and modify than an ad hoc scanner is, automatically generated scanners are often used during language or compiler development, or when ease of implementation is more important than the last little bit of run-time performance. In effect, regular expressions constitute a declarative programming language for a limited problem domain: namely, that of scanning.

The example automaton of Figure 2.6 is *deterministic*: there is never any ambiguity about what it ought to do, because in a given state with a given in-

put character there is never more than one possible outgoing transition (arrow) labeled by that character. As it turns out, however, there is no obvious one-step algorithm to convert a set of regular expressions into an equivalent deterministic finite automaton (DFA). The typical scanner generator implements the conversion as a series of three separate steps.

The first step converts the regular expressions into a *nondeterministic* finite automaton (NFA). An NFA is like a DFA except that (a) there may be more than one transition out of a given state labeled by a given character, and (b) there may be so-called *epsilon transitions*: arrows labeled by the empty string symbol, ϵ . The NFA is said to accept an input string (token) if there exists a path from the start state to a final state whose non- ϵ transitions are labeled, in order, by the characters of the token.

To avoid the need to search all possible paths for one that “works,” the second step of a scanner generator translates the NFA into an equivalent DFA: an automaton that accepts the same language, but in which there are no epsilon transitions and no states with more than one outgoing transition labeled by the same character. The third step is a space optimization that generates a final DFA with the minimum possible number of states.

From a Regular Expression to an NFA

EXAMPLE 2.10

Constructing an NFA for a given regular expression

A trivial regular expression consisting of a single character a is equivalent to a simple two-state NFA (in fact, a DFA), illustrated in part (a) of Figure 2.7. Similarly, the regular expression ϵ is equivalent to a two-state NFA whose arc is labeled by ϵ . Starting with this base we can use three subconstructions, illustrated in parts (b)–(d) of the same figure, to build larger NFAs to represent the concatenation, alternation, or Kleene closure of the regular expressions represented by smaller NFAs. Each step preserves three invariants: there are no transitions into the initial state, there is a single final state, and there are no transitions out of the final state. These invariants allow smaller machines to be joined into larger machines without any ambiguity about where to create the connections, and without creating any unexpected paths. ■

EXAMPLE 2.11

NFA for $(1^*01^*0)^*1^*$

To make these constructions concrete, we consider a small but nontrivial example. Suppose we wish to generate all strings of zeros and ones in which the number of zeros is even. To generate exactly two zeros we could use the expression 00 . We must allow these to be preceded, followed, or separated by an arbitrary number of ones: $1^*01^*01^*$. This whole construct can then be repeated an arbitrary number of times: $(1^*01^*01^*)^*$. Finally, we observe that there is no point in beginning and ending the parenthesized expression with 1^* . If we move one of the occurrences outside the parentheses we get an arguably simpler expression: $(1^*01^*0)^*1^*$.

Starting with this regular expression and using the constructions of Figure 2.7, we illustrate the construction of an equivalent NFA in Figure 2.8. In this particular example alternation is not required. ■

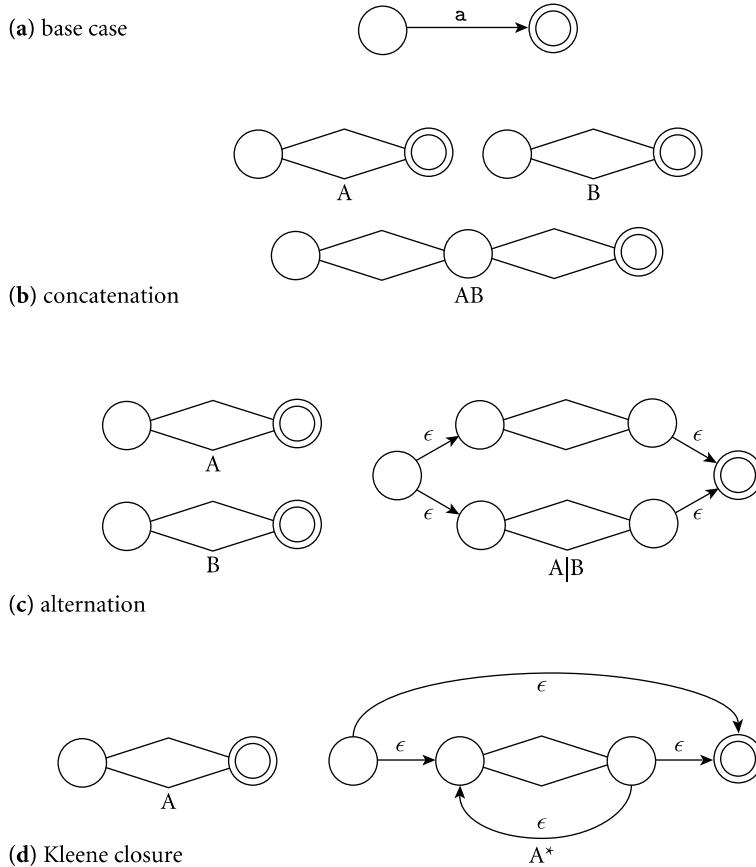


Figure 2.7 Construction of an NFA equivalent to a given regular expression. Part (a) shows the base case: the automaton for the single letter a . Parts (b), (c), and (d), respectively, show the constructions for concatenation, alternation, and Kleene closure. Each construction retains a unique start state and a single final state. Internal detail is hidden in the diamond-shaped center regions.

From an NFA to a DFA

EXAMPLE 2.12

DFA for $(1^*01^*0)^*1^*$

With no way to “guess” the right transition to take from any given state, any practical implementation of an NFA would need to explore all possible transitions, concurrently or via backtracking. To avoid such a complex and time-consuming strategy, we can use a “set of subsets” construction to transform the NFA into an equivalent DFA. The key idea is for the state of the DFA after reading a given input to represent the *set* of states that the NFA might have reached on the same input. We illustrate the construction in Figure 2.9 using the NFA from Figure 2.8. Initially, before it consumes any input, the NFA may be in State 1, or it may make epsilon transitions to States 2, 3, 5, 11, 12, or 14. We thus create an initial State A for our DFA to represent this set. On an input of 1, our NFA may move from

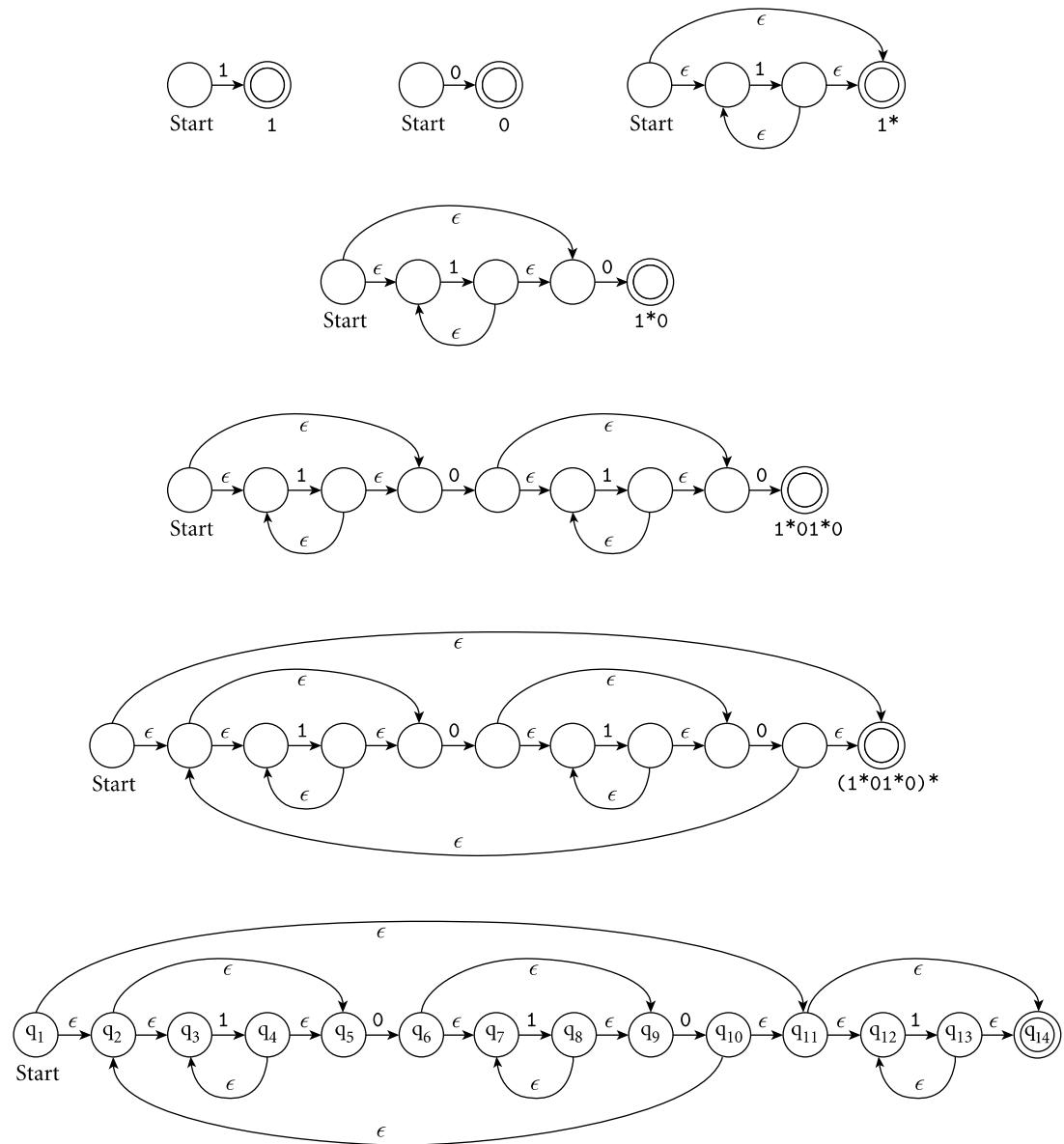


Figure 2.8 Construction of an NFA equivalent to the regular expression $(1^*01^*)^*1^*$. In the top line are the primitive automata for 1 and 0, and the Kleene closure construction for 1^* . In the second and third rows we have used the concatenation construction to build 1^*0 and 1^*01^* . The fourth row uses Kleene closure again to construct $(1^*01^*)^*$; the final line uses concatenation to complete the NFA. We have labeled the states in the final automaton for reference in subsequent figures.

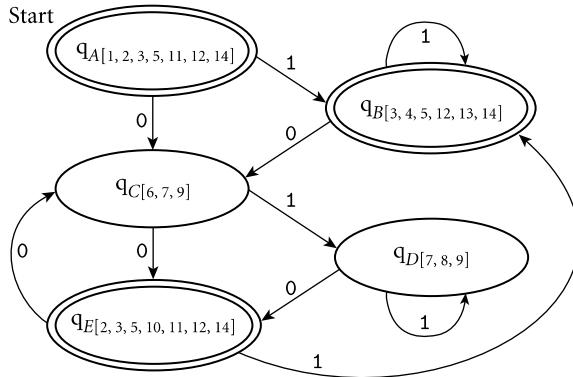


Figure 2.9 A DFA equivalent to the NFA at the bottom of Figure 2.8. Each state of the DFA represents the set of states that the NFA could be in after seeing the same input.

State 3 to State 4, or from State 12 to State 13. It has no other transitions on this input from any of the states in A. From States 4 and 13, however, the NFA may make epsilon transitions to any of States 3, 5, 12, or 14. We therefore create DFA State B as shown. On a 0, our NFA may move from State 5 to State 6, from which it may reach States 7 and 9 by epsilon transitions. We therefore create DFA State C as shown, with a transition from A to C on 0. Careful inspection reveals that a 1 will leave the DFA in State B, while a 0 will move it from B to C. Continuing in this fashion, we end up creating three additional states. Each state that “contains” the final state (State 14) of the NFA is marked as a final state of the DFA. ■

In our example, the DFA ends up being smaller than the NFA, but this is only because our regular language is so simple. In theory, the number of states in the DFA may be exponential in the number of states in the NFA, but this extreme is also uncommon in practice. For a programming language scanner, the DFA tends to be larger than the NFA, but not outlandishly so.

Minimizing the DFA

EXAMPLE 2.13

Minimal DFA for
 $(1^*01^*0)^*1^*$

Starting from a regular expression we have now constructed an equivalent DFA. Though this DFA has five states, a bit of thought suggests that it should be possible to build an automaton with only two states: one that will be reached after consuming input containing an odd number of zeros and one that will be reached after consuming input containing an even number of zeros. We can obtain this machine by performing the following inductive construction. Initially we place the states of the (not necessarily minimal) DFA into two equivalence classes: final states and nonfinal states. We then repeatedly search for an equivalence class C and an input symbol a such that when given a as input, the states in C make transitions to states in $k > 1$ different equivalence classes. We then partition C into k classes in such a way that all states in a given new class would move to a member of the same old class on a . When we are unable to find a class to partition in this fashion we are done. In our example, the original placement puts

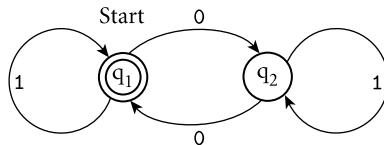


Figure 2.10 Minimal DFA for the language consisting of all strings of zeros and ones in which the number of zeros is even. State q_1 represents the merger of states q_A , q_B , and q_E in Figure 2.9; state q_2 represents the merger of states q_C and q_D .

States A , B , and E in one class (final states) and C and D in another. In all cases, a 1 leaves us in the current class, while a 0 takes us to the other class. Consequently, no class requires partitioning, and we are left with the two-state DFA of Figure 2.10. ■

2.2.2 Scanner Code

We can implement a scanner that explicitly captures the “circles-and-arrows” structure of a DFA in either of two main ways. One embeds the automaton in the control flow of the program using `gosub` or nested case (`switch`) statements; the other, described in the following subsection, uses a table and a driver. As a general rule, handwritten scanners tend to use nested case statements, while most (but not all [BC93]) automatically generated scanners use tables. Tables are hard to create by hand but easier than code to create from within a program. Unix’s `lex/flex` tool produces C language output containing tables and a customized driver. Some other scanner generators produce tables for use with a handwritten driver, which can be written in any language.

EXAMPLE 2.14

Nested case statement automaton

The nested case statement style of automaton is illustrated in Figure 2.11. The outer case statement covers the states of the finite automaton. The inner case statements cover the transitions out of each state. Most of the inner clauses simply set a new state. Some return from the scanner with the current token. ■

Two aspects of the code do not strictly follow the form of a finite automaton. One is the handling of keywords. The other is the need to peek ahead in order to distinguish between the dot in the middle of a real number and a double dot that follows an integer.

Keywords in most languages (including Pascal) look just like identifiers, but they are reserved for a special purpose (some authors use the term *reserved word* instead of keyword⁹). It is possible to write a finite automaton that distinguishes

⁹ Keywords (reserved words) are not the same as predefined identifiers. Predefined identifiers can be redefined to have a different meaning; keywords cannot. The scanner does not distinguish between predefined and other identifiers. It does distinguish between identifiers and keywords. In Pascal, keywords include `begin`, `div`, `record`, and `while`. Predefined identifiers include `integer`, `writeln`, `true`, and `ord`.

```

state := start
loop
  case state of
    start :
      erase text of current token
      case input_char of
        ' ', '\t', '\n', '\r' : no_op
        '[' : state := got_lbrac
        ']' : state := got_rbrac
        ',' : state := got_comma
        ...
        '(' : state := saw_lparen
        '.' : state := saw_dot
        '<' : state := saw_lthan
        ...
        'a'..'z', 'A'..'Z' :
          state := in_ident
        '0'..'9' : state := in_int
        ...
        else error
        ...
      saw_lparen: case input_char of
        '*' : state := in_comment
        else return lparen
      in_comment: case input_char of
        '*' : state := leaving_comment
        else no_op
      leaving_comment: case input_char of
        ')' : state := start
        else state := in_comment
        ...
      saw_dot : case input_char of
        '.' : state := got_dotdot
        else return dot
        ...
      saw_lthan : case input_char of
        '=' : state := got_le
        else return lt
        ...

```

Figure 2.11 Outline of a Pascal scanner written as an explicit finite automaton, in the form of nested `case` statements in a loop. (*continued*)

```

in_ident : case input_char of
  'a'..'z', 'A'..'Z', '0'..'9', '_' : no_op
  else
    look up accumulated token in keyword table
    if found, return keyword
    else return id
  ...
in_int : case input_char of
  '0'..'9' : no_op
  '.' :
    peek at character beyond input_char;
    if '0'..'9', state := saw_real_dot
    else
      unread peeked-at character
      return intconst
  'a'..'z', 'A'..'Z', '_' : error
  else return intconst
  ...
saw_real_dot : ...
  ...
got_lbrac : return lbrac
got_rbrac : return rbrac
got_comma : return comma
got_dotdot : return dotdot
got_le : return le
  ...
append input_char to text of current token
read new input_char

```

Figure 2.11 (continued)

between keywords and identifiers, but it requires a lot of states. To begin with, there must be a separate state, reachable from the initial state, for each letter that might begin a keyword. For each of these, there must then be a state for each possible second character of a keyword (e.g., to distinguish between `file`, `for`, and `from`). It is a nuisance (and a likely source of errors) to enumerate these states by hand. Likewise, while it is easy to write a regular expression that represents a keyword (`b e g i n | e n d | w h i l e | ...`), it is not at all easy to write an expression that represents a (non-keyword) identifier (Exercise 2.3). Most scanners, both handwritten and automatically generated, therefore treat keywords as “exceptions” to the rule for identifiers. Before returning an identifier to the parser, the scanner looks it up in a hash table or trie (a tree of branching paths) to make sure it isn’t really a keyword. This convention is reflected in the `in_ident` arm of Figure 2.11.

Whenever one legitimate token is a prefix of another, the “longest possible token” rule says that we should continue scanning. If some of the intermediate

EXAMPLE 2.15

The “dot-dot problem” in Pascal

strings are not valid tokens, however, we can’t tell whether a longer token is possible without looking more than one character ahead. This problem arises in Pascal in only one case, sometimes known as the “dot-dot problem.” If the scanner has seen a 3 and has a dot coming up in the input, it needs to peek at the character beyond the dot in order to distinguish between 3.14 (a single token designating a real number), 3 . . 5 (three tokens designating a range), and 3 . foo (three tokens that the scanner should accept, even though the parser will object to seeing them in that order). ■

EXAMPLE 2.16

Look-ahead in Fortran scanning

In messier languages, a scanner may need to look an arbitrary distance ahead. In Fortran IV, for example, `DO 5 I = 1,25` is the header of a loop (it executes the statements up to the one labeled 5 for values of `I` from 1 to 25), while `DO 5 I = 1.25` is an assignment statement that places the value 1.25 into the variable `D05I`. Spaces are ignored in (pre-'90) Fortran input, even in the middle of variable names. Moreover, variables need not be declared, and the terminator for a `DO` loop is simply a label, which the parser can ignore. After seeing `DO`, the scanner cannot tell whether the 5 is part of the current token until it reaches the comma or dot. It has been widely (but apparently incorrectly) claimed that NASA’s Mariner 1 space probe was lost due to accidental replacement of a comma with a dot in a case similar to this one in flight control software.¹⁰ Dialects of Fortran starting with Fortran 77 allow (in fact encourage) the use of alternative syntax for loop headers, in which an extra comma makes misinterpretation less likely: `DO 5,I = 1,25`. ■

In Pascal, the dot-dot problem can be handled as a special case, as shown in the `in_int` arm of Figure 2.11. In languages requiring larger amounts of look-ahead, the scanner can take a more general approach. In any case of ambiguity, it assumes that a longer token will be possible but remembers that a shorter token could have been recognized at some point in the past. It also buffers all characters read beyond the end of the shorter token. If the optimistic assumption leads the

DESIGN & IMPLEMENTATION**Longest possible tokens**

A little care in syntax design—avoiding tokens that are nontrivial prefixes of other tokens—can dramatically simplify scanning. In straightforward cases of prefix ambiguity the scanner can enforce the “longest possible token” rule automatically. In Fortran, however, the rules are sufficiently complex that no purely lexical solution suffices. Some of the problems, and a possible solution, are discussed in an article by Dyadkin [Dya95].

10 In actuality, the faulty software for Mariner 1 appears to have stemmed from a missing “bar” punctuation mark (indicating an average) in handwritten notes from which the software was derived [Cer89, pp. 202–203]. The Fortran `DO` loop error does appear to have occurred in at least one piece of NASA software, but no serious harm resulted [Web89].

scanner into an error state, it “unread”s the buffered characters so that they will be seen again later, and returns the shorter token.

2.2.3 Table-Driven Scanning

EXAMPLE 2.17

Table-driven scanning

Figure 2.11 uses control flow—a loop and nested case statements—to represent a finite automaton. An alternative approach represents the automaton as a data structure: a two-dimensional *transition table*. A driver program uses the current state and input character to index into the table (Figure 2.12). Each entry in the table specifies whether to move to a new state (and if so, which one), return a token, or announce an error. A second table indicates, for each state, whether we might be at the end of a token (and if so, which one). Separating this second table from the first allows us to notice when we pass a state that might have been the end of a token, so we can back up if we hit an error state.

Like a handwritten scanner, the table-driven code of Figure 2.12 looks tokens up in a table of keywords immediately before returning. An outer loop serves to filter out comments and “white space”—spaces, tabs, and newlines. These character sequences are not meaningful to the parser, and would in fact be very difficult to represent in a grammar (Exercise 2.15). ■

2.2.4 Lexical Errors

The code in Figure 2.12 explicitly recognizes the possibility of *lexical errors*. In some cases the next character of input may be neither an acceptable continuation of the current token nor the start of another token. In such cases the scanner must print an error message and perform some sort of recovery so that compilation can continue, if only to look for additional errors. Fortunately, lexical errors are relatively rare—most character sequences do correspond to token sequences—and relatively easy to handle. The most common approach is simply to (1) throw away the current, invalid token, (2) skip forward until a character is found that can legitimately begin a new token, (3) restart the scanning algorithm, and (4) count on the error-recovery mechanism of the parser to cope with any cases in which the resulting sequence of tokens is not syntactically valid. Of course the need for error recovery is not unique to table-driven scanners; any scanner must cope with errors. We did not show the code in Figures 2.5 and 2.11, but it would have to be there in practice.

The code in Figure 2.12 also shows that the scanner must return both the kind of token found and its character-string image (spelling); again this requirement applies to all types of scanners. For some tokens the character-string image is redundant: all semicolons look the same, after all, as do all `while` keywords. For other tokens, however (e.g., identifiers, character strings, and numeric constants), the image is needed for semantic analysis. It is also useful for error messages: “undeclared identifier” is not as nice as “`foo` has not been declared.”

```

state = 0..number_of_states
token = 0..number_of_tokens
scan_tab : array [char, state] of record
    action : (move, recognize, error)
    new_state : state
token_tab : array [state] of token      -- what to recognize
keyword_tab : set of record
    k_image : string
    k_token : token
-- these three tables are created by a scanner generator tool

tok : token
cur_char : char
remembered_chars : list of char
repeat
    cur_state : state := start_state
    image : string := null
    remembered_state : state := 0      -- none
loop
    read cur_char
    case scan_tab[cur_char, cur_state].action
        move:
            if token_tab[cur_state] ≠ 0
                -- this could be a final state
                remembered_state := cur_state
                remembered_chars := ε
            add cur_char to remembered_chars
            cur_state := scan_tab[cur_char, cur_state].new_state
        recognize:
            tok := token_tab[cur_state]
            unread cur_char      -- push back into input stream
            exit inner loop
        error:
            if remembered_state ≠ 0
                tok := token_tab[remembered_state]
                unread remembered_chars
                exit inner loop
            -- else print error message and recover; probably start over
            append cur_char to image
    -- end inner loop
until tok ∉ {white_space, comment}
look image up in keyword_tab and replace tok with appropriate keyword if found
return {tok, image}

```

Figure 2.12 Driver for a table-driven scanner, with code to handle the ambiguous case in which one valid token is a prefix of another; but some intermediate string is not.

2.2.5 Pragmas

Some languages and language implementations allow a program to contain constructs called *pragmas* that provide directives or hints to the compiler. Pragmas are sometimes called *significant comments* because, in most cases, they do not affect the meaning (semantics) of the program—only the compilation process. In many languages the name is also appropriate because, like comments, pragmas can appear anywhere in the source program. In this case they are usually handled by the scanner: allowing them anywhere in the grammar would greatly complicate the parser. In other languages (Ada, for example), pragmas are permitted only at certain well-defined places in the grammar. In this case they are best handled by the parser or semantic analyzer.

Examples of directives include the following.

- Turn various kinds of run-time checks (e.g., pointer or subscript checking) on or off.
- Turn certain code improvements on or off (e.g., on in inner loops to improve performance; off otherwise to improve compilation speed).
- Turn performance profiling on or off.

Some directives “cross the line” and change program semantics. In Ada, for example, the `unchecked` pragma can be used to disable type checking.

Hints provide the compiler with information about the source program that may allow it to do a better job:

- Variable `x` is very heavily used (it may be a good idea to keep it in a register).
- Subroutine `F` is a pure function: its only effect on the rest of the program is the value it returns.
- Subroutine `S` is not (indirectly) recursive (its storage may be statically allocated).
- 32 bits of precision (instead of 64) suffice for floating-point variable `x`.

The compiler may ignore these in the interest of simplicity, or in the face of contradictory information.

CHECK YOUR UNDERSTANDING

10. List the tasks performed by the typical scanner.
11. What are the advantages of an automatically generated scanner, in comparison to a handwritten one? Why do many commercial compilers use a handwritten scanner anyway?
12. Explain the difference between deterministic and nondeterministic finite automata. Why do we prefer the deterministic variety for scanning?

13. Outline the constructions used to turn a set of regular expressions into a minimal DFA.
 14. What is the “longest possible token” rule?
 15. Why must a scanner sometimes “peek” at upcoming characters?
 16. What is the difference between a *keyword* and an *identifier*?
 17. Why must a scanner save the text of tokens?
 18. How does a scanner identify lexical errors? How does it respond?
 19. What is a *pragma*?
-

2.3 Parsing

The parser is the heart of a typical compiler. It calls the scanner to obtain the tokens of the input program, assembles the tokens together into a syntax tree, and passes the tree (perhaps one subroutine at a time) to the later phases of the compiler, which perform semantic analysis and code generation and improvement. In effect, the parser is “in charge” of the entire compilation process; this style of compilation is sometimes referred to as *syntax-directed translation*.

As noted in the introduction to this chapter, a context-free grammar (CFG) is a *generator* for a CF language. A parser is a *language recognizer*. It can be shown that for any CFG we can create a parser that runs in $O(n^3)$ time, where n is the length of the input program.¹¹ There are two well-known parsing algorithms that achieve this bound: Earley’s algorithm [Ear70] and the Cocke-Younger-Kasami (CYK) algorithm [Kas65, You67]. Cubic time is much too slow for parsing sizable programs, but fortunately not all grammars require such a general and slow parsing algorithm. There are large classes of grammars for which we can build parsers that run in linear time. The two most important of these classes are called LL and LR.

LL stands for “Left-to-right, Left-most derivation.” LR stands for “Left-to-right, Right-most derivation.” In both classes the input is read left-to-right. An LL parser discovers a left-most derivation; an LR parser discovers a right-most derivation. We will cover LL parsers first. They are generally considered to be simpler and easier to understand. They can be written by hand or generated automatically from an appropriate grammar by a parser-generating tool. The class of LR grammars is larger, and some people find the structure of the grammars more intuitive, especially in the part of the grammar that deals with arithmetic

¹¹ In general, an algorithm is said to run in time $O(f(n))$, where n is the length of the input, if its running time $t(n)$ is proportional to $f(n)$ in the worst case. More precisely, we say $t(n) = O(f(n)) \iff \exists c, m \ [n > m \implies t(n) < cf(n)]$.

expressions. LR parsers are almost always constructed by a parser-generating tool. Both classes of parsers are used in production compilers, though LR parsers are more common.

LL parsers are also called “top-down” or “predictive” parsers. They construct a parse tree from the root down, predicting at each step which production will be used to expand the current node, based on the next available token of input. LR parsers are also called “bottom-up” parsers. They construct a parse tree from the leaves up, recognizing when a collection of leaves or other nodes can be joined together as the children of a single parent.

EXAMPLE 2.18

Top-down and bottom-up parsing

We can illustrate the difference between top-down and bottom-up parsing by means of a simple example. Consider the following grammar for a comma-separated list of identifiers, terminated by a semicolon.

```

 $id\_list \longrightarrow id \ id\_list\_tail$ 
 $id\_list\_tail \longrightarrow , \ id \ id\_list\_tail$ 
 $id\_list\_tail \longrightarrow ;$ 

```

These are the productions that would normally be used for an identifier list in a top-down parser. They can also be parsed bottom-up (most top-down grammars can be). In practice they would not be used in a bottom-up parser, for reasons that will become clear in a moment, but the ability to handle them either way makes them good for this example.

Progressive stages in the top-down and bottom-up construction of a parse tree for the string A, B, C; appear in Figure 2.13. The top-down parser begins by predicting that the root of the tree (id_list) will be replaced by $id \ id_list_tail$. It then matches the id against a token obtained from the scanner. (If the scanner produced something different, the parser would announce a syntax error.) The parser then moves down into the first (in this case only) nonterminal child and predicts that id_list_tail will be replaced by $, \ id \ id_list_tail$. To make this prediction it needs to peek at the upcoming token (a comma), which allows it to choose between the two possible expansions for id_list_tail . It then matches the comma and the id and moves down into the next id_list_tail . In a similar, recursive fashion, the top-down parser works down the tree, left-to-right, predicting and expanding nodes and tracing out a left-most derivation of the fringe of the tree.

The bottom-up parser, by contrast, begins by noting that the left-most leaf of the tree is an id . The next leaf is a comma and the one after that is another id . The parser continues in this fashion, shifting new leaves from the scanner into a forest of partially completed parse tree fragments, until it realizes that some of those fragments constitute a complete right-hand side. In this grammar, that doesn’t occur until the parser has seen the semicolon—the right-hand side of $id_list_tail \longrightarrow ;$. With this right-hand side in hand, the parser reduces the semicolon to an id_list_tail . It then reduces $, \ id \ id_list_tail$ into another id_list_tail . After doing this one more time it is able to reduce $id \ id_list_tail$ into the root of the parse tree, id_list .

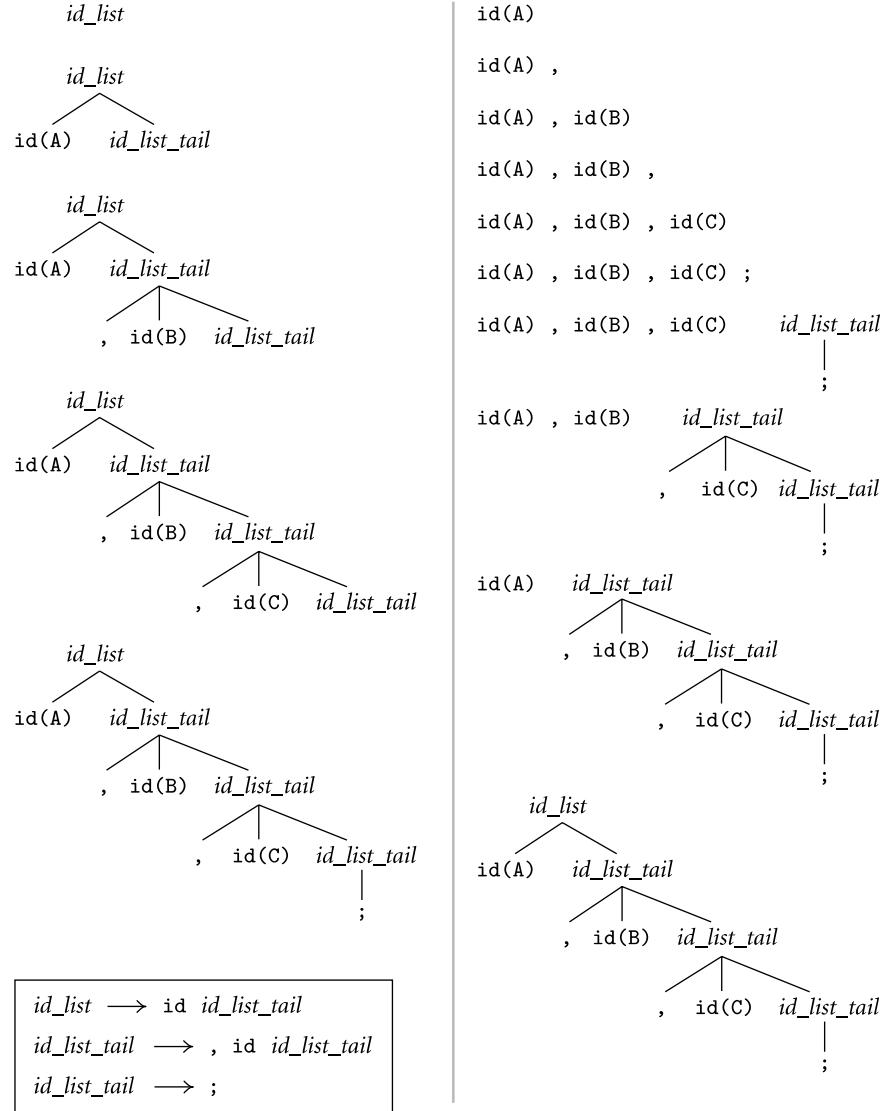


Figure 2.13 Top-down (left) and bottom-up parsing (right) of the input string A, B, C;. Grammar appears at lower left.

At no point does the bottom-up parser predict what it will see next. Rather, it shifts tokens into its forest until it recognizes a right-hand side, which it then reduces to a left-hand side. Because of this behavior, bottom-up parsers are sometimes called shift-reduce parsers. Looking up the figure, from bottom to top, we can see that the shift-reduce parser traces out a right-most (canonical) derivation, in reverse. ■

There are several important subclasses of LR parsers, including SLR, LALR, and “full LR.” SLR and LALR are important for their ease of implementation, full LR for its generality. LL parsers can also be grouped into SLL and “full LL” subclasses. We will cover the differences among them only briefly here; for further information see any of the standard compiler-construction or parsing theory textbooks [App97, ASU86, AU72, CT04, FL88].

One commonly sees LL or LR (or whatever) written with a number in parentheses after it: LL(2) or LALR(1), for example. This number indicates how many tokens of look-ahead are required in order to parse. Most real compilers use just one token of look-ahead, though more can sometimes be helpful. Terrence Parr’s open-source ANTLR tool, in particular, uses multi-token look-ahead to enlarge the class of languages amenable to top-down parsing [PQ95]. In Section 2.3.1 we will look at LL(1) grammars and handwritten parsers in more detail. In Sections 2.3.2 and 2.3.3 we will consider automatically generated LL(1) and LR(1) (actually SLR(1)) parsers.

EXAMPLE 2.19

Bounding space with a bottom-up grammar

The problem with our example grammar, for the purposes of bottom-up parsing, is that it forces the compiler to shift all the tokens of an *id_list* into its forest before it can reduce any of them. In a very large program we might run out of space. Sometimes there is nothing that can be done to avoid a lot of shifting. In this case, however, we can use an alternative grammar that allows the parser to reduce prefixes of the *id_list* into nonterminals as it goes along:

```
id_list → id_list_prefix ;
id_list_prefix → id_list_prefix , id
                    → id
```

This grammar cannot be parsed top-down, because when we see an *id* on the input and we’re expecting an *id_list_prefix*, we have no way to tell which of the two possible productions we should predict (more on this dilemma in Section 2.3.2). As shown in Figure 2.14, however, the grammar works well bottom-up. ■

2.3.1 Recursive Descent

EXAMPLE 2.20

Top-down grammar for a calculator language

To illustrate top-down (predictive) parsing, let us consider the grammar for a simple “calculator” language, shown in Figure 2.15. The calculator allows values to be read into (numeric) variables, which may then be used in expressions. Expressions in turn can be written to the output. Control flow is strictly linear (no loops, *if* statements, or other jumps). The end-marker (\$\$) pseudo-token is produced by the scanner at the end of the input. This token allows the parser to terminate cleanly once it has seen the entire program. As in regular expressions, we use the symbol ϵ to denote the empty string. A production with ϵ on the right-hand side is sometimes called an *epsilon production*.

It may be helpful to compare the *expr* portion of Figure 2.15 to the expression grammar of Example 2.7 (page 45). Most people find that previous, LR grammar to be significantly more intuitive. It suffers, however, from a problem

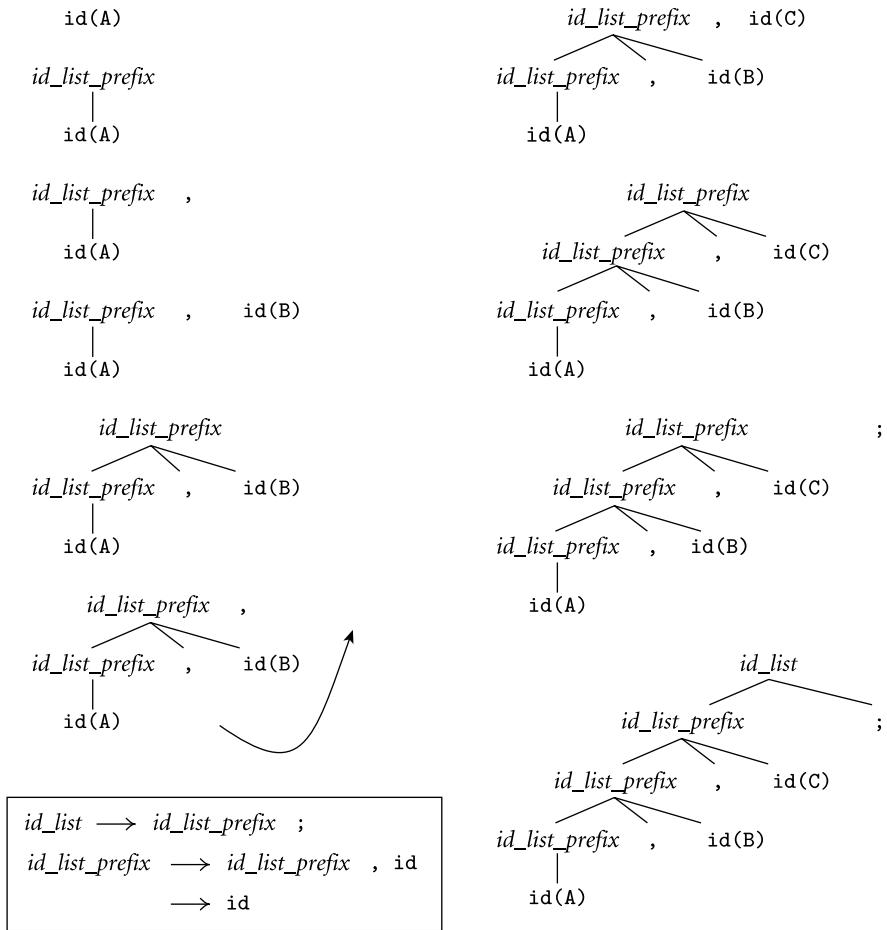


Figure 2.14 Bottom-up parse of A, B, C; using a grammar (lower left) that allows lists to be collapsed incrementally.

similar to that of the *id_list* grammar of Example 2.19: if we see an *id* on the input when expecting an *expr*, we have no way to tell which of the two possible productions to predict. The grammar of Figure 2.15 avoids this problem by merging the common prefixes of right-hand sides into a single production, and by using new symbols (*term_tail* and *factor_tail*) to generate additional operators and operands as required. The transformation has the unfortunate side effect of placing the operands of a given operator in separate right-hand sides. In effect, we have sacrificed grammatical elegance in order to be able to parse predictively. ■

So how do we parse a string with our calculator grammar? We saw the basic idea in Figure 2.13. We start at the top of the tree and predict needed productions on the basis of the current left-most nonterminal in the tree and the current in-

```

program → stmt_list $$ 
stmt_list → stmt stmt_list | ε
stmt → id := expr | read id | write expr
expr → term term_tail
term_tail → add_op term term_tail | ε
term → factor factor_tail
factor_tail → mult_op factor factor_tail | ε
factor → ( expr ) | id | number
add_op → + | -
mult_op → * | /

```

Figure 2.15 LL(1) grammar for a simple calculator language.

put token. We can formalize this process in one of two ways. The first, described in the remainder of this subsection, is to build a *recursive descent parser* whose subroutines correspond, one-to-one, to the nonterminals of the grammar. Recursive descent parsers are typically constructed by hand, though the ANTLR parser generator constructs them automatically from an input grammar. The second approach, described in Section 2.3.2, is to build an *LL parse table*, which is then read by a driver program. Table-driven parsers are almost always constructed automatically by a parser generator. These two options—recursive descent and table-driven—are reminiscent of the nested case statements and table-driven approaches to building a scanner that we saw in Sections 2.2.2 and 2.2.3. Handwritten recursive descent parsers are most often used when the language to be parsed is relatively simple, or when a parser-generator tool is not available.

EXAMPLE 2.21

Recursive descent parser
for the calculator language

Pseudocode for a recursive descent parser for our calculator language appears in Figure 2.16. It has a subroutine for every nonterminal in the grammar. It also has a mechanism `input_token` to inspect the next token available from the scanner and a subroutine (`match`) to consume this token and in the process verify that it is the one that was expected (as specified by an argument). If `match` or any of the other subroutines sees an unexpected token, then a syntax error has occurred. For the time being let us assume that the `parse_error` subroutine simply prints a message and terminates the parse. In Section 2.3.4 we will consider how to recover from such errors and continue to parse the remainder of the input. ■

EXAMPLE 2.22

Recursive descent parse of
a “sum and average”
program

Suppose now that we are to parse a simple program to read two numbers and print their sum and average:

```

read A
read B
sum := A + B
write sum
write sum / 2

```

```

procedure match(expected)
    if input_token = expected
        consume input_token
    else parse_error

-- this is the start routine:
procedure program
    case input_token of
        id, read, write, $$ :
            stmt_list
            match($$)
        otherwise parse_error

procedure stmt_list
    case input_token of
        id, read, write : stmt; stmt_list
        $$ : skip      -- epsilon production
    otherwise parse_error

procedure stmt
    case input_token of
        id : match(id); match(:=); expr
        read : match(read); match(id)
        write : match(write); expr
    otherwise parse_error

procedure expr
    case input_token of
        id, number, ( : term; term_tail
    otherwise parse_error

procedure term_tail
    case input_token of
        +, - : add_op; term; term_tail
        ), id, read, write, $$ :
            skip      -- epsilon production
    otherwise parse_error

procedure term
    case input_token of
        id, number, ( : factor; factor_tail
    otherwise parse_error

```

Figure 2.16 Recursive descent parser for the calculator language. Execution begins in procedure program. The recursive calls trace out a traversal of the parse tree. Not shown is code to save this tree (or some similar structure) for use by later phases of the compiler. (*continued*)

```

procedure factor_tail
    case input_token of
        *, / : mult_op; factor; factor_tail
        +, -, ), id, read, write, $$ :
            skip      -- epsilon production
        otherwise parse_error

procedure factor
    case input_token of
        id : match(id)
        number : match(number)
        ( : match(); expr; match())
        otherwise parse_error

procedure add_op
    case input_token of
        + : match(+)
        - : match(-)
        otherwise parse_error

procedure mult_op
    case input_token of
        * : match(*)
        / : match(/)
        otherwise parse_error

```

Figure 2.16 (continued)

The parse tree for this program appears in Figure 2.17. The parser begins by calling the subroutine `program`. After noting that the initial token is a `read`, `program` calls `stmt_list` and then attempts to match the end-of-file pseudo-token. (In the parse tree, the root, `program`, has two children, `stmt_list` and `$$`.) Procedure `stmt_list` again notes that the upcoming token is a `read`. This observation allows it to determine that the current node (`stmt_list`) generates *stmt stmt_list* (rather than ϵ). It therefore calls `stmt` and `stmt_list` before returning. Continuing in this fashion, the execution path of the parser traces out a left-to-right depth-first traversal of the parse tree. This correspondence between the dynamic execution trace and the structure of the parse tree is the distinguishing characteristic of recursive descent parsing. Note that because the `stmt_list` non-terminal appears in the right-hand side of a `stmt_list` production, the `stmt_list` subroutine must call itself. This recursion accounts for the name of the parsing technique. ■

Without additional code (not shown in Figure 2.16), the parser merely verifies that the program is syntactically correct (i.e., that none of the otherwise `parse_error` clauses in the case statements are executed and that `match` always sees what it expects to see). To be of use to the rest of the compiler—which must produce an equivalent target program in some other language—the parser must

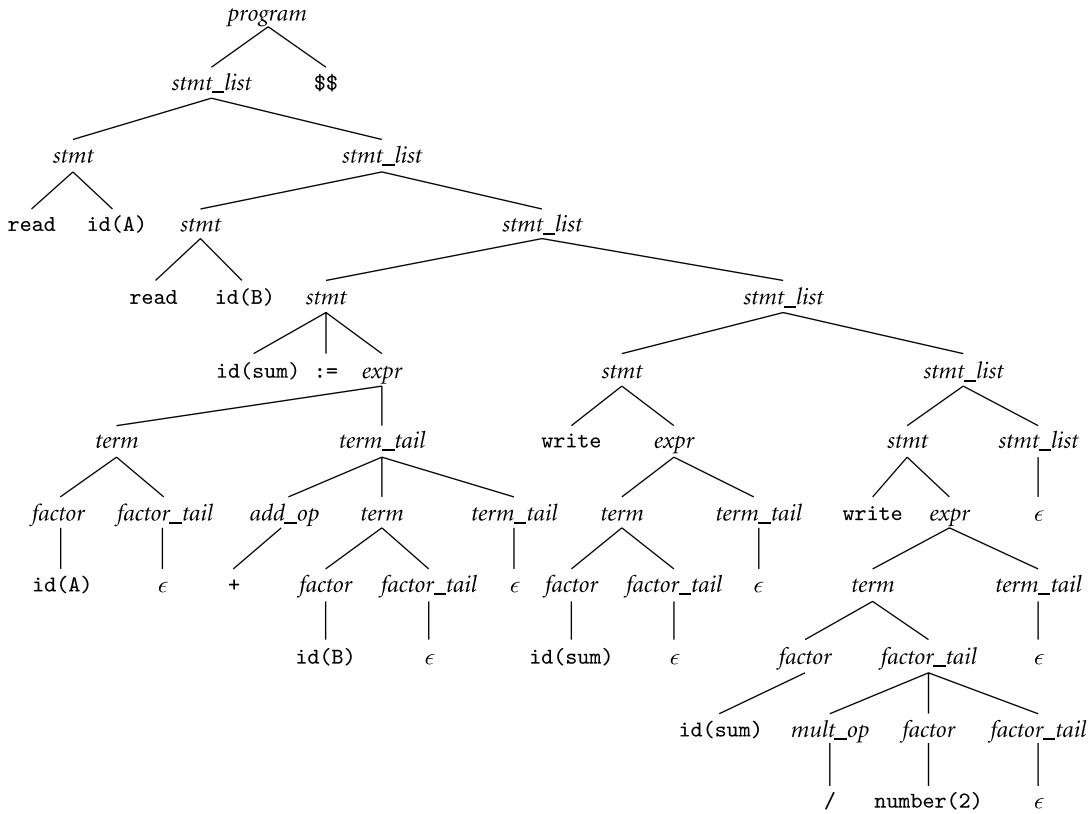


Figure 2.17 Parse tree for the sum-and-average program of Example 2.22, using the grammar of Figure 2.15.

save the parse tree or some other representation of program fragments as an explicit data structure. To save the parse tree itself, we can allocate and link together records to represent the children of a node immediately before executing the recursive subroutines and match invocations that represent those children. We shall need to pass each recursive routine an argument that points to the record that is to be expanded (i.e., whose children are to be discovered). Procedure *match* will also need to save information about certain tokens (e.g., character-string representations of identifiers and literals) in the leaves of the tree.

As we saw in Chapter 1, the parse tree contains a great deal of irrelevant detail that need not be saved for the rest of the compiler. It is therefore rare for a parser to construct a full parse tree explicitly. More often it produces an abstract syntax tree or some other more terse representation. In a recursive descent compiler, a syntax tree can be created by allocating and linking together records in only a subset of the recursive calls.

Perhaps the trickiest part of writing a recursive descent parser is figuring out which tokens should label the arms of the case statements. Each arm represents

one production: one possible expansion of the symbol for which the subroutine was named. The tokens that label a given arm are those that *predict* the production. A token X may predict a production for either of two reasons: (1) the right-hand side of the production, when recursively expanded, may yield a string beginning with X , or (2) the right-hand side may yield nothing (i.e., it is ϵ , or a string of nonterminals that may recursively yield ϵ), and X may begin the yield of what comes *next*. In the following subsection we will formalize this notion of prediction using sets called FIRST and FOLLOW, and show how to derive them automatically from an LL(1) CFG.

CHECK YOUR UNDERSTANDING

20. What is the inherent “big-O” complexity of parsing? What is the complexity of parsers used in real compilers?
21. Summarize the difference between LL and LR parsing. Which one of them is also called “bottom-up”? “Top-down”? Which one is also called “predictive”? “Shift-reduce”? What do “LL” and “LR” stand for?
22. What kind of parser (top-down or bottom-up) is most common in production compilers?
23. What is the significance of the “1” in LR(1)?
24. Why might we want (or need) different grammars for different parsing algorithms?
25. What is an *epsilon production*?
26. What are *recursive descent* parsers? Why are they used mostly for small languages?
27. How might a parser construct an explicit parse tree or syntax tree?

2.3.2 Table-Driven Top-Down Parsing

EXAMPLE 2.23

Driver and table for top-down parsing

In a recursive descent parser, each arm of a case statement corresponds to a production, and contains parsing routine and match calls corresponding to the symbols on the right-hand side of that production. At any given point in the parse, if we consider the calls beyond the program counter (the ones that have yet to occur) in the parsing routine invocations currently in the call stack, we obtain a list of the symbols that the parser expects to see between here and the end of the program. A table-driven top-down parser maintains an explicit stack containing this same list of symbols.

Pseudocode for such a parser appears in Figure 2.18. The code is language independent. It requires a language *dependent* parsing table, generally produced

```

terminal = 1 .. number_of_terminals
non_terminal = number_of_terminals + 1 .. number_of_symbols
symbol = 1 .. number_of_symbols
production = 1 .. number_of_productions

parse_tab : array [non_terminal, terminal] of
    action : (predict, error)
    prod : production
prod_tab : array [production] of list of symbol
-- these two tables are created by a parser generator tool

parse_stack : stack of symbol

parse_stack.push(start_symbol)
loop
    expected_sym : symbol := parse_stack.pop
    if expected_sym ∈ terminal
        match(expected_sym)           -- as in Figure 2.16
        if expected_sym = $$ return   -- success!
    else
        if parse_tab[expected_sym, input_token].action = error
            parse_error
        else
            prediction : production := parse_tab[expected_sym, input_token].prod
            foreach sym : symbol in reverse prod_tab[prediction]
                parse_stack.push(sym)

```

Figure 2.18 Driver for a table-driven LL(1) parser.

EXAMPLE 2.24

Table-driven parse of the “sum and average” program

by an automatic tool. For the calculator grammar of Figure 2.15, the table appears as shown in Figure 2.19.

To illustrate the algorithm, Figure 2.20 shows a trace of the stack and the input over time for the sum-and-average program of Example 2.22. The parser iterates around a loop in which it pops the top symbol off the stack and performs the following actions. If the popped symbol is a terminal, the parser attempts to match it against an incoming token from the scanner. If the match fails, the parser announces a syntax error and initiates some sort of error recovery (see Section 2.3.4). If the popped symbol is a nonterminal, the parser uses that nonterminal together with the next available input token to index into a two-dimensional table that tells it which production to predict (or whether to announce a syntax error and initiate recovery).

Initially, the parse stack contains the start symbol of the grammar (in our case, *program*). When it predicts a production, the parser pushes the right-hand-side symbols onto the parse stack in reverse order, so the first of those symbols ends up at top-of-stack. The parse completes successfully when we match the end token, $\$\$$. Assuming that $\$\$$ appears only once in the grammar, at the end of the first production, and that the scanner returns this token only at end-of-file, any syntax

Top-of-stack nonterminal	id	Current input token												
		number	read	write	$:$ =	()	+	-	*	/	\$\$		
<i>program</i>	1	—	1	1	—	—	—	—	—	—	—	—	1	
<i>stmt_list</i>	2	—	2	2	—	—	—	—	—	—	—	—	3	
<i>stmt</i>	4	—	5	6	—	—	—	—	—	—	—	—	—	
<i>expr</i>	7	7	—	—	—	7	—	—	—	—	—	—	—	
<i>term_tail</i>	9	—	9	9	—	—	9	8	8	—	—	—	9	
<i>term</i>	10	10	—	—	—	10	—	—	—	—	—	—	—	
<i>factor_tail</i>	12	—	12	12	—	—	12	12	12	11	11	12		
<i>factor</i>	14	15	—	—	—	13	—	—	—	—	—	—	—	
<i>add_op</i>	—	—	—	—	—	—	—	16	17	—	—	—	—	
<i>mult_op</i>	—	—	—	—	—	—	—	—	—	18	19	—	—	

Figure 2.19 LL(1) parse table for the calculator language. Table entries indicate the production to predict (as numbered in Figure 2.22). A dash indicates an error. When the top-of-stack symbol is a terminal, the appropriate action is always to match it against an incoming token from the scanner. An auxiliary table, not shown here, gives the right-hand side symbols for each production.

error is guaranteed to manifest itself either as a failed match or as an error entry in the table. ■

Predict Sets

As we hinted at the end of Section 2.3.1, predict sets are defined in terms of simpler sets called FIRST and FOLLOW, where $\text{FIRST}(A)$ is the set of all tokens that could be the start of an A , plus ϵ if $A \xrightarrow{*} \epsilon$, and $\text{FOLLOW}(A)$ is the set of all tokens that could come after an A in some valid program, plus ϵ if A can be the final token in the program. If we extend the domain of FIRST in the obvious way to include strings of symbols, we then say that the predict set of a production $A \xrightarrow{} \beta$ is $\text{FIRST}(\beta)$ (except for ϵ), plus $\text{FOLLOW}(A)$ if $\beta \xrightarrow{*} \epsilon$.¹²

We can illustrate the algorithm to construct these sets using our calculator grammar (Figure 2.15). We begin with “obvious” facts about the grammar and build on them inductively. If we recast the grammar in plain BNF (no EBNF ‘|’ constructs), then it has 19 productions. The “obvious” facts arise from adjacent pairs of symbols in right-hand sides. In the first production, we can see that $\$\$$

EXAMPLE 2.25

Predict sets for the calculator language

12 Following conventional notation, we use uppercase Roman letters near the beginning of the alphabet to represent nonterminals, uppercase Roman letters near the end of the alphabet to represent arbitrary grammar symbols (terminals or nonterminals), lowercase Roman letters near the beginning of the alphabet to represent terminals (tokens), lowercase Roman letters near the end of the alphabet to represent token strings, and lowercase Greek letters to represent strings of arbitrary symbols.

Parse stack	Input stream	Comment
<i>program</i>	read A read B ...	initial stack contents
<i>stmt_list \$\$</i>	predict <i>program</i> \rightarrow <i>stmt_list</i> \$\$	
<i>stmt stmt_list \$\$</i>	predict <i>stmt_list</i> \rightarrow <i>stmt stmt_list</i> \$\$	
<i>read id stmt_list \$\$</i>	predict <i>stmt</i> \rightarrow <i>read id</i>	
<i>id stmt_list \$\$</i>	match <i>read</i>	
<i>stmt_list \$\$</i>	match <i>id</i>	
<i>stmt stmt_list \$\$</i>	predict <i>stmt_list</i> \rightarrow <i>stmt stmt_list</i> \$\$	
<i>read id stmt_list \$\$</i>	predict <i>stmt</i> \rightarrow <i>read id</i>	
<i>id stmt_list \$\$</i>	match <i>read</i>	
<i>stmt_list \$\$</i>	match <i>id</i>	
<i>stmt stmt_list \$\$</i>	predict <i>stmt_list</i> \rightarrow <i>stmt stmt_list</i> \$\$	
<i>read id stmt_list \$\$</i>	predict <i>stmt</i> \rightarrow <i>read id</i>	
<i>id stmt_list \$\$</i>	match <i>read</i>	
<i>stmt_list \$\$</i>	match <i>id</i>	
<i>id := expr stmt_list \$\$</i>	predict <i>stmt_list</i> \rightarrow <i>stmt stmt_list</i> \$\$	
<i>:= expr stmt_list \$\$</i>	predict <i>stmt</i> \rightarrow <i>id := expr</i>	
<i>expr stmt_list \$\$</i>	match <i>id</i>	
<i>term term_tail stmt_list \$\$</i>	match <i>:=</i>	
<i>factor factor_tail term_tail stmt_list \$\$</i>	predict <i>expr</i> \rightarrow <i>term term_tail</i>	
<i>id factor_tail term_tail stmt_list \$\$</i>	predict <i>term</i> \rightarrow <i>factor factor_tail</i>	
<i>factor_tail term_tail stmt_list \$\$</i>	predict <i>factor</i> \rightarrow <i>id</i>	
<i>term tail stmt_list \$\$</i>	match <i>id</i>	
<i>add_op term term_tail stmt_list \$\$</i>	predict <i>factor_tail</i> \rightarrow ϵ	
<i>+ term term_tail stmt_list \$\$</i>	predict <i>term_tail</i> \rightarrow <i>add_op term term_tail</i>	
<i>term term_tail stmt_list \$\$</i>	predict <i>add_op</i> \rightarrow <i>+</i>	
<i>term term_tail stmt_list \$\$</i>	match <i>+</i>	
<i>factor factor_tail term_tail stmt_list \$\$</i>	predict <i>term</i> \rightarrow <i>factor factor_tail</i>	
<i>id factor_tail term_tail stmt_list \$\$</i>	predict <i>factor</i> \rightarrow <i>id</i>	
<i>factor_tail term_tail stmt_list \$\$</i>	match <i>id</i>	
<i>term tail stmt_list \$\$</i>	predict <i>term</i> \rightarrow <i>factor tail</i>	
<i>stmt_list \$\$</i>	predict <i>factor</i> \rightarrow <i>id</i>	
<i>stmt stmt_list \$\$</i>	match <i>id</i>	
<i>write expr stmt_list \$\$</i>	predict <i>term tail</i> \rightarrow ϵ	
<i>expr stmt_list \$\$</i>	predict <i>stmt_list</i> \rightarrow <i>stmt stmt_list</i> \$\$	
<i>term term_tail stmt_list \$\$</i>	predict <i>stmt</i> \rightarrow <i>write expr</i>	
<i>factor factor_tail term_tail stmt_list \$\$</i>	predict <i>sum write sum / 2</i>	match <i>write</i>
<i>id factor_tail term_tail stmt_list \$\$</i>	predict <i>expr</i> \rightarrow <i>term term_tail</i>	
<i>factor_tail term_tail stmt_list \$\$</i>	predict <i>term</i> \rightarrow <i>factor factor_tail</i>	
<i>term tail stmt_list \$\$</i>	predict <i>factor</i> \rightarrow <i>id</i>	
<i>stmt_list \$\$</i>	predict <i>sum write sum / 2</i>	match <i>id</i>
<i>stmt stmt_list \$\$</i>	predict <i>factor tail</i> \rightarrow ϵ	
<i>write sum / 2</i>	predict <i>term tail</i> \rightarrow ϵ	
<i>sum / 2</i>	predict <i>stmt_list</i> \rightarrow <i>stmt stmt_list</i> \$\$	
<i>sum / 2</i>	predict <i>stmt</i> \rightarrow <i>write expr</i>	
<i>sum / 2</i>	match <i>write</i>	
<i>sum / 2</i>	predict <i>expr</i> \rightarrow <i>term term_tail</i>	
<i>sum / 2</i>	predict <i>term</i> \rightarrow <i>factor factor_tail</i>	
<i>sum / 2</i>	predict <i>factor</i> \rightarrow <i>id</i>	
<i>/ 2</i>	match <i>id</i>	
<i>mult_op factor factor_tail term_tail stmt_list \$\$</i>	predict <i>factor tail</i> \rightarrow <i>mult_op factor factor_tail</i>	
<i>/ factor factor_tail term_tail stmt_list \$\$</i>	predict <i>mult_op</i> \rightarrow <i>/</i>	
<i>factor factor_tail term_tail stmt_list \$\$</i>	match <i>/</i>	
<i>number factor_tail term_tail stmt_list \$\$</i>	predict <i>factor</i> \rightarrow <i>number</i>	
<i>factor_tail term_tail stmt_list \$\$</i>	match <i>number</i>	
<i>term tail stmt_list \$\$</i>	predict <i>factor tail</i> \rightarrow ϵ	
<i>stmt_list \$\$</i>	predict <i>term tail</i> \rightarrow ϵ	
<i>\$\$</i>	predict <i>stmt_list</i> \rightarrow ϵ	

Figure 2.20 Trace of a table-driven LL(1) parse of the sum-and-average program of Example 2.22.

$program \rightarrow stmt_list \ \$\$$	$\$\$ \in FOLLOW(stmt_list),$ $\epsilon \in FOLLOW(\$\$)$, and $\epsilon \in FOLLOW(program)$
$stmt_list \rightarrow stmt \ stmt_list$	
$stmt \rightarrow \epsilon$	$\epsilon \in FIRST(stmt_list)$
$stmt \rightarrow id := expr$	$id \in FIRST(stmt)$ and $:= \in FOLLOW(id)$
$stmt \rightarrow read \ id$	$read \in FIRST(stmt)$ and $id \in FOLLOW(read)$
$stmt \rightarrow write \ expr$	$write \in FIRST(stmt)$
$expr \rightarrow term \ term_tail$	
$term_tail \rightarrow add_op \ term \ term_tail$	
$term_tail \rightarrow \epsilon$	$\epsilon \in FIRST(term_tail)$
$term \rightarrow factor \ factor_tail$	
$factor_tail \rightarrow mult_op \ factor \ factor_tail$	
$factor_tail \rightarrow \epsilon$	$\epsilon \in FIRST(factor_tail)$
$factor \rightarrow (\ expr \)$	$(\in FIRST(factor)$ and $) \in FOLLOW(expr)$
$factor \rightarrow id$	$id \in FIRST(factor)$
$factor \rightarrow number$	$number \in FIRST(factor)$
$add_op \rightarrow +$	$+ \in FIRST(add_op)$
$add_op \rightarrow -$	$- \in FIRST(add_op)$
$mult_op \rightarrow *$	$* \in FIRST(mult_op)$
$mult_op \rightarrow /$	$/ \in FIRST(mult_op)$

Figure 2.21 “Obvious” facts about the LL(1) calculator grammar.

$\in FOLLOW(stmt_list)$. In the fourth ($stmt \rightarrow id := expr$), $id \in FIRST(stmt)$, and $:= \in FOLLOW(id)$. In the fifth and sixth productions ($stmt \rightarrow read \ id$ | $write \ expr$), $\{read, write\} \subset FIRST(stmt)$, and $id \in FOLLOW(read)$. The complete set of “obvious” facts appears in Figure 2.21.

From the “obvious” facts we can deduce a larger set of facts during a second pass over the grammar. For example, in the second production ($stmt_list \rightarrow stmt \ stmt_list$) we can deduce that $\{id, read, write\} \subset FIRST(stmt_list)$, because we already know that $\{id, read, write\} \subset FIRST(stmt)$, and a $stmt_list$ can begin with a $stmt$. Similarly, in the first production, we can deduce that $\$\$ \in FIRST(program)$, because we already know that $\epsilon \in FIRST(stmt_list)$.

In the eleventh production ($factor_tail \rightarrow mult_op \ factor \ factor_tail$), we can deduce that $\{(, id, number\} \subset FOLLOW(mult_op)$, because we already know that $\{(, id, number\} \subset FIRST(factor)$, and $factor$ follows $mult_op$ in the right-hand side. In the seventh production ($expr \rightarrow term \ term_tail$), we can deduce that $) \in FOLLOW(term_tail)$, because we already know that $) \in FOLLOW(expr)$, and a $term_tail$ can be the last part of an $expr$. In this same production, we can also deduce that $) \in FOLLOW(term)$, because the $term_tail$ can generate ϵ ($\epsilon \in FIRST(term_tail)$), allowing a $term$ to be the last part of an $expr$.

There is more that we can learn from our second pass through the grammar, but these examples cover all the different kinds of cases. To complete our calculation, we continue with additional passes over the grammar until we don’t learn any more (i.e., we don’t add anything to any of the FIRST and FOLLOW sets). We

FIRST

```

program {id, read, write, $$}
stmt_list {id, read, write,  $\epsilon$ }
stmt {id, read, write}
expr {(), id, number}
term_tail {+, -,  $\epsilon$ }
term {(), id, number}
factor_tail {*, /,  $\epsilon$ }
factor {(), id, number}
add_op {+, -}
mult_op {*}

```

Also note that $\text{FIRST}(a) = \{a\} \forall \text{tokens } a.$

FOLLOW

```

id {+, -, *, /, ), :=, id, read, write, $$}
number {+, -, *, /, ), id, read, write, $$}
read {id}
write {(), id, number}
( {(), id, number}
) {+, -, *, /, ), id, read, write, $$}
:= {(), id, number}
+ {(), id, number}
- {(), id, number}
* {(), id, number}
/ {(), id, number}
$$ { $\epsilon$ }
program { $\epsilon$ }
stmt_list {$$}
stmt {id, read, write, $$}

```

expr {}, *id*, *read*, *write*, \$\$

```

term_tail {}, id, read, write, $$
```

```
term {+, -, ), id, read, write, $$}
```

```
factor_tail {+, -, *, /, ), id, read, write, $$}
```

```
factor {+, -, *, /, ), id, read, write, $$}
```

```
add_op {(), id, number}
```

```
mult_op {(), id, number}
```

PREDICT

1. $\text{program} \longrightarrow \text{stmt_list } \$\$ \{ \text{id}, \text{read}, \text{write}, \$\$ \}$
2. $\text{stmt_list} \longrightarrow \text{stmt } \text{stmt_list} \{ \text{id}, \text{read}, \text{write} \}$
3. $\text{stmt_list} \longrightarrow \epsilon \{ \$\$ \}$
4. $\text{stmt} \longrightarrow \text{id} := \text{expr} \{ \text{id} \}$
5. $\text{stmt} \longrightarrow \text{read } \text{id} \{ \text{read} \}$
6. $\text{stmt} \longrightarrow \text{write } \text{expr} \{ \text{write} \}$
7. $\text{expr} \longrightarrow \text{term } \text{term_tail} \{ (), \text{id}, \text{number} \}$
8. $\text{term_tail} \longrightarrow \text{add_op } \text{term } \text{term_tail} \{ +, - \}$
9. $\text{term_tail} \longrightarrow \epsilon \{ (), \text{id}, \text{read}, \text{write}, \$\$ \}$
10. $\text{term} \longrightarrow \text{factor } \text{factor_tail} \{ (), \text{id}, \text{number} \}$
11. $\text{factor_tail} \longrightarrow \text{mult_op } \text{factor } \text{factor_tail} \{ *, / \}$
12. $\text{factor_tail} \longrightarrow \epsilon \{ +, -, (), \text{id}, \text{read}, \text{write}, \$\$ \}$
13. $\text{factor} \longrightarrow (\text{expr}) \{ () \}$
14. $\text{factor} \longrightarrow \text{id} \{ \text{id} \}$
15. $\text{factor} \longrightarrow \text{number} \{ \text{number} \}$
16. $\text{add_op} \longrightarrow + \{ + \}$
17. $\text{add_op} \longrightarrow - \{ - \}$
18. $\text{mult_op} \longrightarrow * \{ * \}$
19. $\text{mult_op} \longrightarrow / \{ / \}$

Figure 2.22 FIRST, FOLLOW, and PREDICT sets for the calculator language.

then construct the PREDICT sets. Final versions of all three sets appear in Figure 2.22. The parse table of Figure 2.19 follows directly from PREDICT. ■

The algorithm to compute FIRST, FOLLOW, and PREDICT sets appears, a bit more formally, in Figure 2.23. It relies on the following definitions.

$$\text{FIRST}(\alpha) \equiv \{a : \alpha \xrightarrow{*} a \beta\} \cup (\text{if } \alpha \xrightarrow{*} \epsilon \text{ then } \{\epsilon\} \text{ else } \emptyset)$$

$$\text{FOLLOW}(A) \equiv \{a : S \xrightarrow{+} \alpha A a \beta\} \cup (\text{if } S \xrightarrow{*} \alpha A \text{ then } \{\epsilon\} \text{ else } \emptyset)$$

$$\text{PREDICT}(A \longrightarrow \alpha) \equiv (\text{FIRST}(\alpha) \setminus \{\epsilon\}) \cup (\text{if } \alpha \xrightarrow{*} \epsilon \text{ then } \text{FOLLOW}(A) \text{ else } \emptyset)$$

Note that FIRST sets for strings of length greater than one are calculated on demand; they are not stored explicitly. The algorithm is guaranteed to terminate

First sets for all symbols:

```

for all terminals  $a$ ,  $\text{FIRST}(a) := \{a\}$ 
for all nonterminals  $X$ ,  $\text{FIRST}(X) := \emptyset$ 
for all productions  $X \rightarrow \epsilon$ , add  $\epsilon$  to  $\text{FIRST}(X)$ 
repeat
  (outer) for all productions  $X \rightarrow Y_1 Y_2 \dots Y_k$ ,
    (inner) for  $i$  in  $1 \dots k$ 
      add  $(\text{FIRST}(Y_i) \setminus \{\epsilon\})$  to  $\text{FIRST}(X)$ 
      if  $\epsilon \notin \text{FIRST}(Y_i)$  (yet)
        continue outer loop
      add  $\epsilon$  to  $\text{FIRST}(X)$ 
    until no further progress
  
```

First set subroutine for string $X_1 X_2 \dots X_n$, similar to inner loop above:

```

return_value :=  $\emptyset$ 
for  $i$  in  $1 \dots n$ 
  add  $(\text{FIRST}(X_i) \setminus \{\epsilon\})$  to return_value
  if  $\epsilon \notin \text{FIRST}(X_i)$ 
    return
  add  $\epsilon$  to return_value
  
```

Follow sets for all symbols:

```

 $\text{FOLLOW}(S) := \{\epsilon\}$ , where  $S$  is the start symbol
for all other symbols  $X$ ,  $\text{FOLLOW}(X) := \emptyset$ 
repeat
  for all productions  $A \rightarrow \alpha B \beta$ ,
    add  $(\text{FIRST}(\beta) \setminus \{\epsilon\})$  to  $\text{FOLLOW}(B)$ 
  for all productions  $A \rightarrow \alpha B$ 
    or  $A \rightarrow \alpha B \beta$ , where  $\epsilon \in \text{FIRST}(\beta)$ ,
    add  $\text{FOLLOW}(A)$  to  $\text{FOLLOW}(B)$ 
  until no further progress
  
```

Predict sets for all productions:

```

for all productions  $A \rightarrow \alpha$ 
   $\text{PREDICT}(A \rightarrow \alpha) := (\text{FIRST}(\alpha) \setminus \{\epsilon\})$ 
   $\cup (\text{if } \epsilon \in \text{FIRST}(\alpha) \text{ then } \text{FOLLOW}(A) \text{ else } \emptyset)$ 
  
```

Figure 2.23 Algorithm to calculate FIRST, FOLLOW, and PREDICT sets. The grammar is LL(1) if and only if the PREDICT sets are disjoint.

(i.e., converge on a solution), because the sizes of the sets are bounded by the number of terminals in the grammar.

If in the process of calculating PREDICT sets we find that some token belongs to the PREDICT set of more than one production with the same left-hand side, then the grammar is not LL(1), because we will not be able to choose which of the productions to employ when the left-hand side is at the top of the parse stack (or we are in the left-hand side's subroutine in a recursive descent parser) and we see the token coming up in the input. This sort of ambiguity is known as a *predict-predict conflict*; it can arise either because the same token can begin

more than one right-hand side, or because it can begin one right-hand side and can also appear after the left-hand side in some valid program, and one possible right-hand side can generate ϵ .

Writing an LL(1) Grammar

When working with a top-down parser generator, one has to acquire a certain facility in writing and modifying LL(1) grammars. The two most common obstacles to “LL(1)-ness” are *left recursion* and *common prefixes*.

EXAMPLE 2.26

Left recursion

Left recursion occurs when the first symbol on the right-hand side of a production is the same as the symbol on the left-hand side. Here again is the grammar from Example 2.19, which cannot be parsed top-down:

```
id_list → id_list_prefix ;
id_list_prefix → id_list_prefix , id
→ id
```

The problem is in the second and third productions; with *id_list_prefix* at top-of-stack and an *id* on the input, a predictive parser cannot tell which of the productions it should use. (Recall that left recursion is *desirable* in bottom-up grammars, because it allows recursive constructs to be discovered incrementally, as in Figure 2.14.)

Common prefixes occur when two different productions with the same left-hand side begin with the same symbol or symbols. Here is an example that commonly appears in Algol-family languages:

```
stmt → id := expr
→ id ( argument_list ) -- procedure call
```

Clearly *id* is in the FIRST set of both right-hand sides, and therefore in the PREDICT set of both productions.

Both left recursion and common prefixes can be removed from a grammar mechanically. The general case is a little tricky (Exercise 2.17), because the prediction problem may be an indirect one (e.g., $S \rightarrow A \alpha$ and $A \rightarrow S \beta$, or $S \rightarrow A \alpha$, $S \rightarrow B \beta$, $A \Rightarrow^* a \gamma$, and $B \Rightarrow^* a \delta$). We can see the general idea in the examples above, however.

Our left-recursive definition of *id_list* can be replaced by the right-recursive variant we saw in Example 2.18:

```
id_list → id id_list_tail
id_list_tail → , id id_list_tail
id_list_tail → ;
```

EXAMPLE 2.28

Eliminating left recursion

Our common-prefix definition of *stmt* can be made LL(1) by a technique called *left factoring*:

```
stmt → id stmt_list_tail
stmt_list_tail → := expr | ( argument_list )
```

EXAMPLE 2.29

Left factoring

Of course, simply eliminating left recursion and common prefixes is *not* guaranteed to make a grammar LL(1). There are infinitely many non-LL *languages*—languages for which no LL grammar exists—and the mechanical transformations to eliminate left recursion and common prefixes work on their grammars just fine. Fortunately, the few non-LL languages that arise in practice can generally be handled by augmenting the parsing algorithm with one or two simple heuristics.

EXAMPLE 2.30

Parsing a “dangling else”

The best known example of a “not quite LL” construct arises in languages like Pascal, in which the `else` part of an `if` statement is optional. The natural grammar fragment

$$\begin{aligned} \text{stmt} &\longrightarrow \text{if condition then_clause else_clause } | \text{other_stmt} \\ \text{then_clause} &\longrightarrow \text{then stmt} \\ \text{else_clause} &\longrightarrow \text{else stmt } | \epsilon \end{aligned}$$

is ambiguous (and thus neither LL nor LR); it allows the `else` in `if C1 then if C2 then S1 else S2` to be paired with either `then`. The less natural grammar fragment

$$\begin{aligned} \text{stmt} &\longrightarrow \text{balanced_stmt } | \text{unbalanced_stmt} \\ \text{balanced_stmt} &\longrightarrow \text{if condition then balanced_stmt else balanced_stmt } \\ &\quad | \text{other_stmt} \\ \text{unbalanced_stmt} &\longrightarrow \text{if condition then stmt } \\ &\quad | \text{if condition then balanced_stmt else unbalanced_stmt} \end{aligned}$$

can be parsed bottom-up but not top-down (there is *no* pure top-down grammar for Pascal `else` statements). A *balanced_stmt* is one with the same number of `thens` and `elses`. An *unbalanced_stmt* has more `thens`. ■

The usual approach, whether parsing top-down or bottom-up, is to use the ambiguous grammar together with a “disambiguating rule,” which says that in the case of a conflict between two possible productions, the one to use is the one that occurs first, textually, in the grammar. In the ambiguous fragment above, the fact that $\text{else_clause} \rightarrow \text{else stmt}$ comes before $\text{else_clause} \rightarrow \epsilon$ ends up pairing the `else` with the nearest `then`, as desired.

EXAMPLE 2.31

“Dangling else” program bug

Better yet, a language designer can avoid this sort of problem by choosing different syntax. The ambiguity of the *dangling else* problem in Pascal leads to problems not only in parsing but in writing and maintaining correct programs. Most Pascal programmers have at one time or another written a program like this one:

```
if P <> nil then
  if P^.val = goal then
    foundIt := true
  else
    endOfList := true
```

Indentation notwithstanding, the Pascal manual states that an `else` clause matches the closest unmatched `then`—in this case the inner one—which is

clearly not what the programmer intended. To get the desired effect, the Pascal programmer must write

```
if P <> nil then begin
    if P^.val = goal then
        foundIt := true
    end
else
    endOfList := true
```

EXAMPLE 2.32

End markers for structured statements

Many other Algol-family languages (including Modula, Modula-2, and Oberon, all more recent inventions of Pascal's designer, Niklaus Wirth) require explicit *end markers* on all structured statements. The grammar fragment for *if* statements in Modula-2 looks something like this:

```
stmt → IF condition then_clause else_clause END | other_stmt
then_clause → THEN stmt_list
else_clause → ELSE stmt_list | ε
```

The addition of the END eliminates the ambiguity.

Modula-2 uses END to terminate all its structured statements. Ada and Fortran 77 end an *if* with *end if* (and a *while* with *end while*, etc.). Algol 68 creates its terminators by spelling the initial keyword backward (*if...fi*, *case...esac*, *do...od*, etc.).

EXAMPLE 2.33

The need for *elsif*

One problem with end markers is that they tend to bunch up. In Pascal one can write

```
if A = B then ...
else if A = C then ...
else if A = D then ...
else if A = E then ...
else ...
```

With end markers this becomes

```
if A = B then ...
else if A = C then ...
else if A = D then ...
else if A = E then ...
else ...
end end end end
```

DESIGN & IMPLEMENTATION

The dangling *else*

A simple change in language syntax—eliminating the dangling *else*—not only reduces the chance of programming errors but also significantly simplifies parsing. For more on the dangling *else* problem, see Exercise 2.23 and Section 6.4.

To avoid this awkwardness, languages with end markers generally provide an `elsif` keyword (sometimes spelled `elif`):

```
if A = B then ...
elsif A = C then ...
elsif A = D then ...
elsif A = E then ...
else ...
end
```

With `elsif` clauses added, the Modula-2 grammar fragment for `if` statements looks like this:

```
stmt → IF condition then_clause elsif_clauses else_clause END | other_stmt
then_clause → THEN stmt_list
elsif_clauses → ELSIF condition then_clause elsif_clauses | ε
else_clause → ELSE stmt_list | ε
```

CHECK YOUR UNDERSTANDING

28. Discuss the similarities and differences between recursive descent and table-driven top-down parsing.
29. What are FIRST and FOLLOW sets? What are they used for?
30. Under what circumstances does a top-down parser predict the production $A \rightarrow \alpha$?
31. What sorts of “obvious” facts form the basis of FIRST set and FOLLOW set construction?
32. Outline the algorithm used to complete the construction of FIRST and FOLLOW sets. How do we know when we are done?
33. How do we know when a grammar is not LL(1)?
34. Describe two common idioms in context-free grammars that cannot be parsed top-down.
35. What is the “dangling else” problem? How is it avoided in modern languages?

2.3.3 Bottom-Up Parsing

Conceptually, as we saw at the beginning of Section 2.3, a bottom-up parser works by maintaining a forest of partially completed subtrees of the parse tree, which it joins together whenever it recognizes the symbols on the right-hand side

of some production used in the right-most derivation of the input string. It creates a new internal node and makes the roots of the joined-together trees the children of that node.

In practice, a bottom-up parser is almost always table-driven. It keeps the roots of its partially completed subtrees on a stack. When it accepts a new token from the scanner, it *shifts* the token into the stack. When it recognizes that the top few symbols on the stack constitute a right-hand side, it *reduces* those symbols to their left-hand side by popping them off the stack and pushing the left-hand side in their place. The role of the stack is the first important difference between top-down and bottom-up parsing: a top-down parser's stack contains a list of what the parser expects to see in the future; a bottom-up parser's stack contains a record of what the parser has already seen in the past.

Canonical Derivations

We also noted earlier that the actions of a bottom-up parser trace out a right-most (canonical) derivation in reverse. The roots of the partial subtrees, left-to-right, together with the remaining input, constitute a sentential form of the right-most derivation. On the right-hand side of Figure 2.13, for example, we have the following series of steps.

EXAMPLE 2.34

Derivation of an *id* list

stack contents (roots of partial trees)	remaining input
ϵ	A, B, C;
<i>id</i> (A)	, B, C;
<i>id</i> (A) ,	B, C;
<i>id</i> (A) , <i>id</i> (B)	, C;
<i>id</i> (A) , <i>id</i> (B) ,	C;
<i>id</i> (A) , <i>id</i> (B) , <i>id</i> (C)	;
<i>id</i> (A) , <i>id</i> (B) , <i>id</i> (C) <u>i</u>	
<i>id</i> (A) , <i>id</i> (B) , <u><i>id</i> (C) <i>id_list_tail</i></u>	
<i>id</i> (A) , <u><i>id</i> (B) <i>id_list_tail</i></u>	
<u><i>id</i> (A) <i>id_list_tail</i></u>	
<i>id_list</i>	

The last four lines (the ones that don't just shift tokens into the forest) correspond to the right-most derivation:

$$\begin{aligned}
 id_list &\implies id\ id_list_tail \\
 &\implies id\ ,\ id\ id_list_tail \\
 &\implies id\ ,\ id\ ,\ id\ id_list_tail \\
 &\implies id\ ,\ id\ ,\ id\ ;
 \end{aligned}$$

The symbols that need to be joined together at each step of the parse to represent the next step of the backward derivation are called the *handle* of the sentential form. In the preceding parse trace, the handles are underlined. ■

EXAMPLE 2.35

Bottom-up grammar for the calculator language

In our *id_list* example, no handles were found until the entire input had been shifted onto the stack. In general this will not be the case. We can obtain a more realistic example by examining an LR version of our calculator language, shown

1. $program \rightarrow stmt_list \ \$\$$
2. $stmt_list \rightarrow stmt_list \ stmt$
3. $stmt_list \rightarrow stmt$
4. $stmt \rightarrow id := expr$
5. $stmt \rightarrow read \ id$
6. $stmt \rightarrow write \ expr$
7. $expr \rightarrow term$
8. $expr \rightarrow expr \ add_op \ term$
9. $term \rightarrow factor$
10. $term \rightarrow term \ mult_op \ factor$
11. $factor \rightarrow (\ expr \)$
12. $factor \rightarrow id$
13. $factor \rightarrow number$
14. $add_op \rightarrow +$
15. $add_op \rightarrow -$
16. $mult_op \rightarrow *$
17. $mult_op \rightarrow /$

Figure 2.24 LR(1) grammar for the calculator language. Productions have been numbered for reference in future figures.

in Figure 2.24. While the LL grammar of Figure 2.15 can be parsed bottom-up, the version in Figure 2.24 is preferable for two reasons. First, it uses a left-recursive production for $stmt_list$. Left recursion allows the parser to collapse long statement lists as it goes along, rather than waiting until the entire list is on the stack and then collapsing it from the end. Second, it uses left-recursive productions for $expr$ and $term$. These productions capture left associativity while still keeping an operator and its operands together in the same right-hand side, something we were unable to do in a top-down grammar. ■

Modeling a Parse with LR Items

EXAMPLE 2.36

Bottom-up parse of the “sum and average” program

```
read A
read B
sum := A + B
write sum
write sum / 2
```

The key to success will be to figure out when we have reached the end of a right-hand side—that is, when we have a handle at the top of the parse stack. The trick is to keep track of the set of productions we might be “in the middle of” at any

particular time, together with an indication of where in those productions we might be.

When we begin execution, the parse stack is empty and we are at the beginning of the production for *program*. (In general, we can assume that there is only one production with the start symbol on the left-hand side; it is easy to modify any grammar to make this the case.) We can represent our location—more specifically, the location represented by the top of the parse stack—with a • in the right-hand side of the production:

$$\text{program} \longrightarrow \bullet \text{stmt_list} \ \$\$$$

When augmented with a •, a production is called an LR *item*. Since the • in this item is immediately in front of a nonterminal—namely *stmt_list*—we may be about to see the yield of that nonterminal coming up on the input. This possibility implies that we may be at the beginning of some production with *stmt_list* on the left-hand side:

$$\begin{aligned} \text{program} &\longrightarrow \bullet \text{stmt_list} \ \$\$ \\ \text{stmt_list} &\longrightarrow \bullet \text{stmt_list} \ \text{stmt} \\ \text{stmt_list} &\longrightarrow \bullet \text{stmt} \end{aligned}$$

And, since *stmt* is a nonterminal, we may also be at the beginning of any production whose left-hand side is *stmt*:

$$\begin{aligned} \text{program} &\longrightarrow \bullet \text{stmt_list} \ \$\$ && (\text{State 0}) \\ \text{stmt_list} &\longrightarrow \bullet \text{stmt_list} \ \text{stmt} \\ \text{stmt_list} &\longrightarrow \bullet \text{stmt} \\ \text{stmt} &\longrightarrow \bullet \text{id} := \text{expr} \\ \text{stmt} &\longrightarrow \bullet \text{read id} \\ \text{stmt} &\longrightarrow \bullet \text{write expr} \end{aligned}$$

Since all of these last productions begin with a terminal, no additional items need to be added to our list. The original item ($\text{program} \longrightarrow \bullet \text{stmt_list} \ \$\$$) is called the *basis* of the list. The additional items are its *closure*. The list represents the initial state of the parser. As we shift and reduce, the set of items will change, always indicating which productions *may* be the right one to use next in the derivation of the input string. If we reach a state in which some item has the • at the end of the right-hand side, we can reduce by that production. Otherwise, as in the current situation, we must shift. Note that if we need to shift, but the incoming token cannot follow the • in any item of the current state, then a syntax error has occurred. We will consider error recovery in more detail in Section 2.3.4.

Our upcoming token is a *read*. Once we shift it onto the stack, we know we are in the following state:

$$\text{stmt} \longrightarrow \text{read} \ \bullet \text{id} && (\text{State 1})$$

This state has a single basis item and an empty closure—the • precedes a terminal. After shifting the A, we have

$$stmt \longrightarrow \text{read id } \bullet \quad (\text{State } 1')$$

We now know that `read id` is the handle, and we must reduce. The reduction pops two symbols off the parse stack and pushes a *stmt* in their place, but what should the new state be? We can see the answer if we imagine moving back in time to the point at which we shifted the `read`—the first symbol of the right-hand side. At that time we were in the state labeled “State 0” above, and the upcoming tokens on the input (though we didn’t look at them at the time) were `read id`. We have now consumed these tokens, and we know that they constituted a *stmt*. By pushing a *stmt* onto the stack, we have in essence replaced `read id` with *stmt* on the input stream, and have then “shifted” the nonterminal, rather than its yield, into the stack. Since one of the items in State 0 was

$$stmt_list \longrightarrow \bullet \cdot stmt$$

we now have

$$stmt_list \longrightarrow stmt \bullet \quad (\text{State } 0')$$

Again we must reduce. We remove the *stmt* from the stack and push a *stmt_list* in its place. Again we can see this as “shifting” a *stmt_list* when in State 0. Since two of the items in State 0 have a *stmt_list* after the \bullet , we don’t know (without looking ahead) which of the productions will be the next to be used in the derivation, but we don’t have to know. The key advantage of bottom-up parsing over top-down parsing is that we don’t need to predict ahead of time which production we shall be expanding.

Our new state is as follows:

$$\begin{aligned} program &\longrightarrow stmt_list \bullet \text{ $$} & (\text{State } 2) \\ stmt_list &\longrightarrow stmt_list \bullet stmt \\ stmt &\longrightarrow \bullet id := expr \\ stmt &\longrightarrow \bullet \text{read id} \\ stmt &\longrightarrow \bullet \text{write expr} \end{aligned}$$

The first two productions are the basis; the others are the closure. Since no item has a \bullet at the end, we shift the next token, which happens again to be a `read`, taking us back to State 1. Shifting the B takes us to State 1’ again, at which point we reduce. This time however, we go back to State 2 rather than State 0 before shifting the left-hand side *stmt*. Why? Because we were in State 2 when we began to read the right-hand side. ■

The Characteristic Finite State Machine and LR Parsing Variants

An LR-family parser keeps track of the states it has traversed by pushing them into the parse stack along with the grammar symbols. It is in fact the states (rather than the symbols) that drive the parsing algorithm: they tell us what state we were in at the beginning of a right-hand side. Specifically, when the combination of state and input tells us we need to reduce using production $A \longrightarrow \alpha$, we pop $\text{length}(\alpha)$ symbols off the stack, together with the record of states we moved

through while shifting those symbols. These pops expose the state we were in immediately prior to the shifts, allowing us to return to that state and proceed as if we had seen A in the first place.

We can think of the shift rules of an LR-family parser as the transition function of a finite automaton, much like the automata we used to model scanners. Each state of the automaton corresponds to a list of items that indicate where the parser might be at some specific point in the parse. The transition for input symbol X (which may be either a terminal or a nonterminal) moves to a state whose basis consists of items in which the \bullet has been moved across an X in the right-hand side, plus whatever items need to be added as closure. The lists are constructed by a bottom-up parser generator in order to build the automaton but are not needed during parsing.

It turns out that the simpler members of the LR family of parsers—LR(0), SLR(1), and LALR(1)—all use the same automaton, called the *characteristic finite-state machine*, or CFSM. Full LR parsers use a machine with (for most grammars) a much larger number of states. The differences between the algorithms lie in how they deal with states that contain a *shift-reduce conflict*—one item with the \bullet in the middle (suggesting the need for a shift) and another with the \bullet at the end (suggesting the need for a reduction). An LR(0) parser works only when there are no such states. It can be proven that with the addition of an end-marker (i.e., $\$\$$), any language that can be deterministically parsed bottom-up has an LR(0) grammar. Unfortunately, the LR(0) grammars for real programming languages tend to be prohibitively large and unintuitive.

SLR (simple LR) parsers peek at upcoming input and use FOLLOW sets to resolve conflicts. An SLR parser will call for a reduction via $A \rightarrow \alpha$ only if the upcoming token(s) are in $\text{FOLLOW}(\alpha)$. It will still see a conflict, however, if the tokens are also in the FIRST set of any of the symbols that follow a \bullet in other items of the state. As it turns out, there are important cases in which a token may follow a given nonterminal somewhere in a valid program, but never in a context described by the current state. For these cases global FOLLOW sets are too crude. LALR (look-ahead LR) parsers improve on SLR by using *local* (state-specific) look-ahead instead.

Conflicts can still arise in an LALR parser when the same set of items can occur on two different paths through the CFSM. Both paths will end up in the same state, at which point state-specific look-ahead can no longer distinguish between them. A full LR parser duplicates states in order to keep paths disjoint when their local look-aheads are different.

LALR parsers are the most common bottom-up parsers in practice. They are the same size and speed as SLR parsers, but are able to resolve more conflicts. Full LR parsers for real programming languages tend to be very large. Several researchers have developed techniques to reduce the size of full-LR tables, but LALR works sufficiently well in practice that the extra complexity of full LR is usually not required. Yacc/bison produces C code for an LALR parser.

Bottom-Up Parsing Tables

Like a table-driven LL(1) parser, an SLR(1), LALR(1), or LR(1) parser executes a loop in which it repeatedly inspects a two-dimensional table to find out what action to take. However, instead of using the current input token and top-of-stack nonterminal to index into the table, an LR-family parser uses the current input token and the current parser state (which can be found at the top of the stack). “Shift” table entries indicate the state that should be pushed. “Reduce” table entries indicate the number of states that should be popped and the non-terminal that should be pushed back onto the input stream, to be shifted by the state uncovered by the pops. There is always one popped state for every symbol on the right-hand side of the reducing production. The state to be pushed next can be found by indexing into the table using the uncovered state and the newly recognized nonterminal.

EXAMPLE 2.37

CFSM for the bottom-up calculator grammar

The CFSM for our bottom-up version of the calculator grammar appears in Figure 2.25. States 6, 7, 9, and 13 contain potential shift-reduce conflicts, but all of these can be resolved with global FOLLOW sets. SLR parsing therefore suffices. In State 6, for example, $\text{FIRST}(add_op) \cap \text{FOLLOW}(stmt) = \emptyset$. In addition to shift and reduce rules, we allow the parse table as an optimization to contain rules of the form “shift and then reduce.” This optimization serves to eliminate trivial states such as 1' and 0' in Example 2.36, which had only a single item, with the • at the end.

A pictorial representation of the CFSM appears in Figure 2.26. A tabular representation, suitable for use in a table-driven parser, appears in Figure 2.27. Pseudocode for the (language independent) parser driver appears in Figure 2.28. A trace of the parser’s actions on the sum-and-average program appears in Figure 2.29. ■

Handling Epsilon Productions

EXAMPLE 2.38

Epsilon productions in the bottom-up calculator grammar

The careful reader may have noticed that the grammar of Figure 2.24, in addition to using left-recursive rules for *stmt_list*, *expr*, and *term*, differs from the grammar of Figure 2.15 in one other way: it defines a *stmt_list* to be a sequence of one or more *stmts*, rather than zero or more. (This means, of course, that it defines a different language.) To capture the same language as Figure 2.15, the productions

```
program → stmt_list $$  
stmt_list → stmt_list stmt | stmt
```

in Figure 2.24 would need to be replaced with

```
program → stmt_list $$  
stmt_list → stmt_list stmt | ε
```

State	Transitions
0. $\underline{\text{program} \rightarrow \bullet \text{stmt_list} \ \$\$}$ $\text{stmt_list} \rightarrow \bullet \text{stmt_list} \ \text{stmt}$ $\text{stmt_list} \rightarrow \bullet \text{stmt}$ $\text{stmt} \rightarrow \bullet \text{id} := \text{expr}$ $\text{stmt} \rightarrow \bullet \text{read id}$ $\text{stmt} \rightarrow \bullet \text{write expr}$	on stmt_list shift and goto 2 on stmt shift and reduce (pop 1 state, push stmt_list on input) on id shift and goto 3 on read shift and goto 1 on write shift and goto 4
1. $\text{stmt} \rightarrow \text{read } \bullet \text{id}$	on id shift and reduce (pop 2 states, push stmt on input)
2. $\underline{\text{program} \rightarrow \text{stmt_list} \bullet \ \$\$}$ $\underline{\text{stmt_list} \rightarrow \text{stmt_list} \bullet \ \text{stmt}}$ $\text{stmt} \rightarrow \bullet \text{id} := \text{expr}$ $\text{stmt} \rightarrow \bullet \text{read id}$ $\text{stmt} \rightarrow \bullet \text{write expr}$	on $\$\$$ shift and reduce (pop 2 states, push program on input) on stmt shift and reduce (pop 2 states, push stmt_list on input) on id shift and goto 3 on read shift and goto 1 on write shift and goto 4
3. $\text{stmt} \rightarrow \text{id } \bullet := \text{expr}$	on $:=$ shift and goto 5
4. $\underline{\text{stmt} \rightarrow \text{write } \bullet \ \text{expr}}$ $\text{expr} \rightarrow \bullet \text{term}$ $\text{expr} \rightarrow \bullet \text{expr add_op term}$ $\text{term} \rightarrow \bullet \text{factor}$ $\text{term} \rightarrow \bullet \text{term mult_op factor}$ $\text{factor} \rightarrow \bullet (\ \text{expr} \)$ $\text{factor} \rightarrow \bullet \text{id}$ $\text{factor} \rightarrow \bullet \text{number}$	on expr shift and goto 6 on term shift and goto 7 on factor shift and reduce (pop 1 state, push term on input) on $($ shift and goto 8 on id shift and reduce (pop 1 state, push factor on input) on number shift and reduce (pop 1 state, push factor on input)
5. $\underline{\text{stmt} \rightarrow \text{id } := \bullet \ \text{expr}}$ $\text{expr} \rightarrow \bullet \text{term}$ $\text{expr} \rightarrow \bullet \text{expr add_op term}$ $\text{term} \rightarrow \bullet \text{factor}$ $\text{term} \rightarrow \bullet \text{term mult_op factor}$ $\text{factor} \rightarrow \bullet (\ \text{expr} \)$ $\text{factor} \rightarrow \bullet \text{id}$ $\text{factor} \rightarrow \bullet \text{number}$	on expr shift and goto 9 on term shift and goto 7 on factor shift and reduce (pop 1 state, push term on input) on $($ shift and goto 8 on id shift and reduce (pop 1 state, push factor on input) on number shift and reduce (pop 1 state, push factor on input)
6. $\underline{\text{stmt} \rightarrow \text{write } \text{expr} \bullet}$ $\underline{\text{stmt} \rightarrow \text{expr } \bullet \ \text{add_op term}}$ $\text{add_op} \rightarrow \bullet +$ $\text{add_op} \rightarrow \bullet -$	on FOLLOW(stmt) = { id , read , write , $\$\$$ } reduce (pop 2 states, push stmt on input) on add_op shift and goto 10 on $+$ shift and reduce (pop 1 state, push add_op on input) on $-$ shift and reduce (pop 1 state, push add_op on input)

Figure 2.25 CFSM for the calculator grammar (Figure 2.24). Basis and closure items in each state are separated by a horizontal rule. Trivial reduce-only states have been eliminated by use of “shift and reduce” transitions. (continued)

State	Transitions
7. $\begin{array}{l} expr \longrightarrow term \bullet \\ term \longrightarrow term \bullet mult_op \ factor \end{array}$	on FOLLOW(expr) = {id, read, write, \$\$, +, -} \text{ reduce } (pop 1 state, push } expr \text{ on input) on } mult_op \text{ shift and goto 11}
<hr/>	<hr/>
mult_op $\longrightarrow \bullet *$	on * shift and reduce (pop 1 state, push } mult_op \text{ on input)
mult_op $\longrightarrow \bullet /$	on / shift and reduce (pop 1 state, push } mult_op \text{ on input)
8. $\begin{array}{l} factor \longrightarrow (\bullet expr) \end{array}$	on } expr \text{ shift and goto 12}
<hr/>	<hr/>
expr $\longrightarrow \bullet term$	on } term \text{ shift and goto 7}
expr $\longrightarrow \bullet expr \ add_op \ term$	on } factor \text{ shift and reduce (pop 1 state, push } term \text{ on input)}
term $\longrightarrow \bullet factor$	on (shift and goto 8
term $\longrightarrow \bullet term \ mult_op \ factor$	on id shift and reduce (pop 1 state, push } factor \text{ on input)
factor $\longrightarrow \bullet (\ expr)$	on number shift and reduce (pop 1 state, push } factor \text{ on input)
factor $\longrightarrow \bullet id$	
factor $\longrightarrow \bullet number$	
9. $\begin{array}{l} stmt \longrightarrow id := expr \bullet \\ expr \longrightarrow expr \bullet add_op \ term \end{array}$	on FOLLOW(stmt) = {id, read, write, \$\$} \text{ reduce } (pop 3 states, push } stmt \text{ on input) on } add_op \text{ shift and goto 10}
<hr/>	<hr/>
add_op $\longrightarrow \bullet +$	on + shift and reduce (pop 1 state, push } add_op \text{ on input)
add_op $\longrightarrow \bullet -$	on - shift and reduce (pop 1 state, push } add_op \text{ on input)
10. $\begin{array}{l} expr \longrightarrow expr \ add_op \bullet term \end{array}$	on } term \text{ shift and goto 13}
<hr/>	<hr/>
term $\longrightarrow \bullet factor$	on } factor \text{ shift and reduce (pop 1 state, push } term \text{ on input)}
term $\longrightarrow \bullet term \ mult_op \ factor$	on (shift and goto 8
factor $\longrightarrow \bullet (\ expr)$	on id shift and reduce (pop 1 state, push } factor \text{ on input)
factor $\longrightarrow \bullet id$	on number shift and reduce (pop 1 state, push } factor \text{ on input)
factor $\longrightarrow \bullet number$	
11. $\begin{array}{l} term \longrightarrow term \ mult_op \bullet factor \end{array}$	on } factor \text{ shift and reduce (pop 3 states, push } term \text{ on input)}
<hr/>	<hr/>
factor $\longrightarrow \bullet (\ expr)$	on (shift and goto 8
factor $\longrightarrow \bullet id$	on id shift and reduce (pop 1 state, push } factor \text{ on input)
factor $\longrightarrow \bullet number$	on number shift and reduce (pop 1 state, push } factor \text{ on input)
12. $\begin{array}{l} factor \longrightarrow (\ expr \bullet) \\ expr \longrightarrow expr \bullet add_op \ term \end{array}$	on) shift and reduce (pop 3 states, push } factor \text{ on input) on } add_op \text{ shift and goto 10}
<hr/>	<hr/>
add_op $\longrightarrow \bullet +$	on + shift and reduce (pop 1 state, push } add_op \text{ on input)
add_op $\longrightarrow \bullet -$	on - shift and reduce (pop 1 state, push } add_op \text{ on input)
13. $\begin{array}{l} expr \longrightarrow expr \ add_op \ term \bullet \\ term \longrightarrow term \bullet mult_op \ factor \end{array}$	on FOLLOW(expr) = {id, read, write, \$\$, +, -} \text{ reduce } (pop 3 states, push } expr \text{ on input) on } mult_op \text{ shift and goto 11}
<hr/>	<hr/>
mult_op $\longrightarrow \bullet *$	on * shift and reduce (pop 1 state, push } mult_op \text{ on input)
mult_op $\longrightarrow \bullet /$	on / shift and reduce (pop 1 state, push } mult_op \text{ on input)

Figure 2.25 (continued)

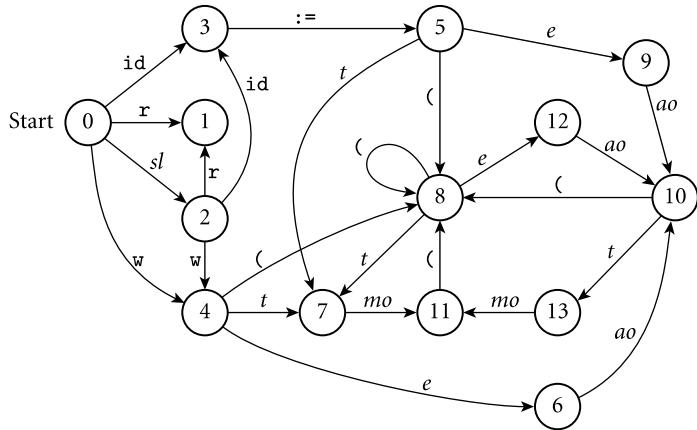


Figure 2.26 Pictorial representation of the CFSM of Figure 2.25. Symbol names have been abbreviated for clarity. Reduce actions are not shown.

Top-of-stack state	Current input symbol																		
	sl	s	e	t	f	ao	mo	id	lit	r	w	:=	()	+	-	*	/	\$\$
0	s2	b3	—	—	—	—	—	s3	—	s1	s4	—	—	—	—	—	—	—	—
1	—	—	—	—	—	—	—	b5	—	—	—	—	—	—	—	—	—	—	—
2	—	b2	—	—	—	—	—	s3	—	s1	s4	—	—	—	—	—	—	—	b1
3	—	—	—	—	—	—	—	—	—	—	—	s5	—	—	—	—	—	—	—
4	—	—	s6	s7	b9	—	—	b12	b13	—	—	—	s8	—	—	—	—	—	—
5	—	—	s9	s7	b9	—	—	b12	b13	—	—	—	s8	—	—	—	—	—	—
6	—	—	—	—	—	s10	—	r6	—	r6	r6	—	—	b14	b15	—	—	r6	—
7	—	—	—	—	—	—	s11	r7	—	r7	r7	—	—	r7	r7	r7	b16	b17	r7
8	—	—	s12	s7	b9	—	—	b12	b13	—	—	—	s8	—	—	—	—	—	—
9	—	—	—	—	—	s10	—	r4	—	r4	r4	—	—	b14	b15	—	—	r4	—
10	—	—	—	—	—	s13	b9	—	—	b12	b13	—	—	s8	—	—	—	—	—
11	—	—	—	—	—	b10	—	—	b12	b13	—	—	s8	—	—	—	—	—	—
12	—	—	—	—	—	s10	—	—	—	—	—	—	—	b11	b14	b15	—	—	—
13	—	—	—	—	—	—	s11	r8	—	r8	r8	—	—	r8	r8	r8	b16	b17	r8

Figure 2.27 SLR(1) parse table for the calculator language. Table entries indicate whether to shift (s), reduce (r), or shift and then reduce (b). The accompanying number is the new state when shifting, or the production that has been recognized when (shifting and) reducing. Production numbers are given in Figure 2.24. Symbol names have been abbreviated for the sake of formatting. A dash indicates an error. An auxiliary table, not shown here, gives the left-hand side symbol and right-hand side length for each production.

Note that it does in general make sense to have an empty statement list. In the calculator language it simply permits an empty program, which is admittedly silly. In real languages, however, it allows the body of a structured statement to be empty, which can be very useful. One frequently wants one arm of a case or multiway if... then... else statement to be empty, and an empty while loop allows a parallel program (or the operating system) to wait for a signal from another process or an I/O device.

```

state = 1 .. number_of_states
symbol = 1 .. number_of_symbols
production = 1 .. number_of_productions
action_rec = record
    action : (shift, reduce, shift_reduce, error)
    new_state : state
    prod : production

parse_tab : array [symbol, state] of action_rec
prod_tab : array [production] of record
    lhs : symbol
    rhs_len : integer
-- these two tables are created by a parser generator tool

parse_stack : stack of record
    sym : symbol
    st : state

parse_stack.push((null, start_state))
cur_sym : symbol := scan                         -- get new token from scanner
loop
    cur_state : state := parse_stack.top.st      -- peek at state at top of stack
    if cur_state = start_state
        and cur_sym = start_symbol return -- success!
    ar : action_rec := parse_tab[cur_state, cur_sym]
    case ar.action
        shift:
            parse_stack.push((cur_sym, ar.new_state))
            cur_sym := scan                         -- get new token from scanner
        reduce:
            cur_sym := prod_tab[ar.prod].lhs
            parse_stack.pop(prod_tab[ar.prod].rhs_len)
        shift_reduce:
            cur_sym := prod_tab[ar.prod].lhs
            parse_stack.pop(prod_tab[ar.prod].rhs_len - 1)
        error:
            parse_error

```

Figure 2.28 Driver for a table-driven SLR(1) parser. We call the scanner directly, rather than using the global input_token of Figures 2.16 and 2.18, so that we can set cur_sym to be an arbitrary symbol.

EXAMPLE 2.39

CFSM with epsilon productions

If we look at the CFSM for the calculator language, we discover that State 0 is the only state that needs to be changed in order to allow empty statement lists. The item

stmt_list → • *stmt*

becomes

Parse stack	Input stream	Comment
0	read A read B ...	
0 read 1	A read B ...	shift read
0	stmt read B ...	shift id(A) & reduce by <i>stmt</i> \rightarrow read id
0	stmt_list read B ...	shift <i>stmt</i> & reduce by <i>stmt_list</i> \rightarrow stmt
0 stmt_list 2	read B sum ...	shift <i>stmt_list</i>
0 stmt_list 2 read 1	B sum := ...	shift read
0 stmt_list 2	stmt sum := ...	shift <i>stmt</i> & reduce by <i>stmt_list</i> \rightarrow stmt_list stmt
0	stmt_list sum := ...	shift <i>stmt_list</i>
0 stmt_list 2	sum := A ...	shift id(sum)
0 stmt_list 2 id 3	:= A + ...	shift :=
0 stmt_list 2 id 3 := 5	A + B ...	shift <i>id(A)</i> & reduce by <i>factor</i> \rightarrow id
0 stmt_list 2 id 3 := 5	factor + B ...	shift <i>factor</i> & reduce by <i>term</i> \rightarrow factor
0 stmt_list 2 id 3 := 5	term + B ...	shift <i>term</i>
0 stmt_list 2 id 3 := 5 term 7	+ B write ...	reduce by <i>expr</i> \rightarrow term
0 stmt_list 2 id 3 := 5	expr + B write ...	shift <i>expr</i>
0 stmt_list 2 id 3 := 5 expr 9	+ B write ...	shift + & reduce by <i>add_op</i> \rightarrow +
0 stmt_list 2 id 3 := 5 expr 9	add_op B write ...	shift <i>add_op</i>
0 stmt_list 2 id 3 := 5 expr 9 add_op 10	B write sum ...	shift <i>id(B)</i> & reduce by <i>factor</i> \rightarrow id
0 stmt_list 2 id 3 := 5 expr 9 add_op 10	factor write sum ...	shift <i>factor</i> & reduce by <i>term</i> \rightarrow factor
0 stmt_list 2 id 3 := 5 expr 9 add_op 10	term write sum ...	shift <i>term</i>
0 stmt_list 2 id 3 := 5 expr 9 add_op 10 term 13	write sum ...	reduce by <i>expr</i> \rightarrow expr add_op term
0 stmt_list 2 id 3 := 5	expr write sum ...	shift <i>expr</i>
0 stmt_list 2 id 3 := 5 expr 9	write sum ...	reduce by <i>stmt</i> \rightarrow id := expr
0 stmt_list 2	stmt write sum ...	shift <i>stmt</i> & reduce by <i>stmt_list</i> \rightarrow stmt
0	stmt_list write sum ...	shift <i>stmt_list</i>
0 stmt_list 2	write sum ...	shift write
0 stmt_list 2 write 4	sum write sum ...	shift <i>id(sum)</i> & reduce by <i>factor</i> \rightarrow id
0 stmt_list 2 write 4	factor write sum ...	shift <i>factor</i> & reduce by <i>term</i> \rightarrow factor
0 stmt_list 2 write 4	term write sum ...	shift <i>term</i>
0 stmt_list 2 write 4 term 7	write sum ...	reduce by <i>expr</i> \rightarrow term
0 stmt_list 2 write 4	expr write sum ...	shift <i>expr</i>
0 stmt_list 2 write 4 expr 6	write sum ...	reduce by <i>stmt</i> \rightarrow write expr
0 stmt_list 2	stmt write sum ...	shift <i>stmt</i> & reduce by <i>stmt_list</i> \rightarrow stmt_list stmt
0	stmt_list write sum ...	shift <i>stmt_list</i>
0 stmt_list 2	write sum / ...	shift write
0 stmt_list 2 write 4	sum / 2 ...	shift <i>id(sum)</i> & reduce by <i>factor</i> \rightarrow id
0 stmt_list 2 write 4	factor / 2 ...	shift <i>factor</i> & reduce by <i>term</i> \rightarrow factor
0 stmt_list 2 write 4	term / 2 ...	shift <i>term</i>
0 stmt_list 2 write 4 term 7	/ 2 \$\$	shift / & reduce by <i>mult_op</i> \rightarrow /
0 stmt_list 2 write 4 term 7	mult_op 2 \$\$	shift <i>mult_op</i>
0 stmt_list 2 write 4 term 7 mult_op 11	2 \$\$	shift number(2) & reduce by <i>factor</i> \rightarrow number
0 stmt_list 2 write 4 term 7 mult_op 11	factor \$\$	shift <i>factor</i> & reduce by <i>term</i> \rightarrow term mult_op factor
0 stmt_list 2 write 4	term \$\$	shift <i>term</i>
0 stmt_list 2 write 4 term 7	\$\$	reduce by <i>expr</i> \rightarrow term
0 stmt_list 2 write 4	expr \$\$	shift <i>expr</i>
0 stmt_list 2 write 4 expr 6	\$\$	reduce by <i>stmt</i> \rightarrow write expr
0 stmt_list 2	stmt \$\$	shift <i>stmt</i> & reduce by <i>stmt_list</i> \rightarrow stmt_list stmt
0	stmt_list \$\$	shift <i>stmt_list</i>
0 stmt_list 2	\$\$	shift \$\$ & reduce by <i>program</i> \rightarrow stmt_list \$\$
0	program	
[done]		

Figure 2.29 Trace of a table-driven SLR(1) parse of the sum-and-average program. States in the parse stack are shown in boldface type. Symbols in the parse stack are for clarity only; they are not needed by the parsing algorithm. Parsing begins with the initial state of the CFSM (State 0) in the stack. It ends when we reduce by *program* \rightarrow *stmt_list* \$\$, uncovering State 0 again and pushing *program* onto the input stream.

$$\text{stmt_list} \longrightarrow \bullet \epsilon$$

which is equivalent to

$$\text{stmt_list} \longrightarrow \epsilon \bullet$$

or simply

$$\text{stmt_list} \longrightarrow \bullet$$

The entire state is then

$\text{program} \longrightarrow \bullet \text{stmt_list} \text{ $$}$	on stmt_list shift and goto 2
<hr/>	
$\text{stmt_list} \longrightarrow \bullet \text{stmt_list} \text{ stmt}$	
$\text{stmt_list} \longrightarrow \bullet$	on $\text{ $$}$ reduce (pop 0 states, push stmt_list on input)
$\text{stmt} \longrightarrow \bullet \text{id} := \text{expr}$	on id shift and goto 3
$\text{stmt} \longrightarrow \bullet \text{read id}$	on read shift and goto 1
$\text{stmt} \longrightarrow \bullet \text{write expr}$	on write shift and goto 4

The look-ahead for item

$$\text{stmt_list} \longrightarrow \bullet$$

is $\text{FOLLOW}(\text{stmt_list})$, which is the end-marker, $\text{ $$}$. Since $\text{ $$}$ does not appear in the look-aheads for any other item in this state, our grammar is still SLR(1). It is worth noting that epsilon productions prevent a grammar from being LR(0), since one can never tell whether to “recognize” ϵ without peeking ahead. An LR(0) grammar never has epsilon productions. ■

CHECK YOUR UNDERSTANDING

36. What is the *handle* of a right sentential form?
37. Explain the significance of the characteristic finite state machine in LR parsing.
38. What is the significance of the dot (\bullet) in an LR item?
39. What distinguishes the *basis* from the *closure* of an LR state?
40. What is a *shift-reduce conflict*? How is it resolved in the various kinds of LR-family parsers?
41. Outline the steps performed by the driver of a bottom-up parser.
42. What kind of parser is produced by yacc/bison? By ANTLR?
43. Why are there never any epsilon productions in an LR(0) grammar?

2.3.4 Syntax Errors

EXAMPLE 2.40

A syntax error in C

```
A = B : C + D;
```

We will detect a syntax error immediately after the B, when the colon appears from the scanner. At this point the simplest thing to do is just to print an error message and halt. This naive approach is generally not acceptable, however: it would mean that every run of the compiler reveals no more than one syntax error. Since most programs, at least at first, contain numerous such errors, we really need to find as many as possible now (we'd also like to continue looking for semantic errors). To do so, we must modify the state of the parser and/or the input stream so that the upcoming token(s) are acceptable. We shall probably want to turn off code generation, disabling the back end of the compiler: since the input is not a valid program, the code will not be of use, and there's no point in spending time creating it.

In general, the term *syntax error recovery* is applied to any technique that allows the compiler, in the face of a syntax error, to continue looking for other errors later in the program. High-quality syntax error recovery is essential in any production-quality compiler. The better the recovery technique, the more likely the compiler will be to recognize additional errors (especially nearby errors) correctly, and the less likely it will be to become confused and announce spurious *cascading errors* later in the program.

IN MORE DEPTH

There are many possible approaches to syntax error recovery. In *panic mode*, the compiler writer defines a small set of “safe symbols” that delimit clean points in the input. When an error occurs, the compiler deletes input tokens until it finds a safe symbol, and then “backs the parser out” (e.g., returns from recursive descent subroutines) until it finds a context in which that symbol might appear. *Phrase-level recovery* improves on this technique by employing different sets of “safe” symbols in different productions of the grammar. *Context-sensitive look-ahead* obtains additional improvements by differentiating among the various contexts in which a given production might appear in a syntax tree. To respond gracefully to certain common programming errors, the compiler writer may augment the grammar with *error productions* that capture language-specific idioms that are incorrect but are often written by mistake.

Niklaus Wirth published an elegant implementation of phrase-level and context-sensitive recovery for recursive descent parsers in 1976 [Wir76, Sec. 5.9]. *Exceptions* (to be discussed further in Section 8.5.3) provide a simpler alternative if supported by the language in which the compiler is written. For table-driven top-down parsers, Fischer, Milton, and Quiring published an algorithm in 1980 that automatically implements a well-defined notion of *locally least-cost syntax*.

repair. Locally least-cost repair is also possible in bottom-up parsers, but it is significantly more difficult. Most bottom-up parsers rely on more straightforward phrase-level recovery; a typical example can be found in yacc/bison.

2.4 Theoretical Foundations

Our understanding of the relative roles and computational power of scanners, parsers, regular expressions, and context-free grammars is based on the formalisms of *automata theory*. In automata theory, a *formal language* is a set of strings of symbols drawn from a finite *alphabet*. A formal language can be specified either by a set of rules (such as regular expressions or a context-free grammar) that generate the language or by a *formal machine* that *accepts* (*recognizes*) the language. A formal machine takes strings of symbols as input and outputs either “yes” or “no.” A machine is said to accept a language if it says “yes” to all and only those strings that are in the language. Alternatively, a language can be defined as the set of strings for which a particular machine says “yes.”

Formal languages can be grouped into a series of successively larger classes known as the *Chomsky hierarchy*.¹³ Most of the classes can be characterized in two ways: by the types of rules that can be used to generate the set of strings or by the type of formal machine that is capable of recognizing the language. As we have seen, *regular languages* are defined by using concatenation, alternation, and Kleene closure, and are recognized by a scanner. *Context-free languages* are a proper superset of the regular languages. They are defined by using concatenation, alternation, and recursion (which subsumes Kleene closure), and are recognized by a parser. A scanner is a concrete realization of a *finite automaton*, a type of formal machine. A parser is a concrete realization of a *push-down automaton*. Just as context-free grammars add recursion to regular expressions, push-down automata add a stack to the memory of a finite automaton. There are additional levels in the Chomsky hierarchy, but they are less directly applicable to compiler construction, and are not covered here.

It can be proven, constructively, that regular expressions and finite automata are equivalent: one can construct a finite automaton that accepts the language defined by a given regular expression, and vice versa. Similarly, it is possible to construct a push-down automaton that accepts the language defined by a given context-free grammar, and vice versa. The grammar-to-automaton constructions are in fact performed by scanner and parser generators such as lex and yacc. Of course, a real scanner does not accept just one token; it is called in a loop so that it keeps accepting tokens repeatedly. This detail is accommodated by having the

13 Noam Chomsky (1928–), a linguist and social philosopher at the Massachusetts Institute of Technology, developed much of the early theory of formal languages.

scanner accept the alternation of all the tokens in the language, and by having it continue to consume characters until no longer token can be constructed.

🕒 IN MORE DEPTH

On the PLP CD we consider finite and pushdown automata in more detail. We give an algorithm to convert a DFA into an equivalent regular expression. Combined with the constructions in Section 2.2.1, this algorithm demonstrates the equivalence of regular expressions and finite automata. We also consider the sets of grammars and languages that can and cannot be parsed by the various linear-time parsing algorithms.

2.5 Summary and Concluding Remarks

In this chapter we have introduced the formalisms of regular expressions and context-free grammars, and the algorithms that underlie scanning and parsing in practical compilers. We also mentioned syntax error recovery, and presented a quick overview of relevant parts of automata theory. Regular expressions and context-free grammars are language *generators*: they specify how to construct valid strings of characters or tokens. Scanners and parsers are language *recognizers*: they indicate whether a given string is valid. The principal job of the scanner is to reduce the quantity of information that must be processed by the parser, by grouping characters together into tokens, and by removing comments and white space. Scanner and parser generators automatically translate regular expressions and context-free grammars into scanners and parsers.

Practical parsers for programming languages (parsers that run in linear time) fall into two principal groups: top-down (also called LL or predictive) and bottom-up (also called LR or shift-reduce). A top-down parser constructs a parse tree starting from the root and proceeding in a left-to-right depth-first traversal. A bottom-up parser constructs a parse tree starting from the leaves, again working left-to-right, and combining partial trees together when it recognizes the children of an internal node. The stack of a top-down parser contains a prediction of what will be seen in the future; the stack of a bottom-up parser contains a record of what has been seen in the past.

Top-down parsers tend to be simple, both in the parsing of valid strings and in the recovery from errors in invalid strings. Bottom-up parsers are more powerful, and in some cases lend themselves to more intuitively structured grammars, though they suffer from the inability to embed action routines at arbitrary points in a right-hand side (we discuss this point in more detail in Section 4.5.1). Both varieties of parser are used in real compilers, though bottom-up parsers are more common. Top-down parsers tend to be smaller in terms of code and data size, but modern machines provide ample memory for either.

Both scanners and parsers can be built by hand if an automatic tool is not available. Hand-built scanners are simple enough to be relatively common. Hand-built parsers are generally limited to top-down recursive descent, and are generally used only for comparatively simple languages (e.g., Pascal but not Ada). Automatic generation of the scanner and parser has the advantage of increased reliability, reduced development time, and easy modification and enhancement.

Various features of language design can have a major impact on the complexity of syntax analysis. In many cases, features that make it difficult for a compiler to scan or parse also make it difficult for a human being to write correct, maintainable code. Examples include the lexical structure of Fortran and the `if...then...else` statement of languages like Pascal. This interplay among language design, implementation, and use will be a recurring theme throughout the remainder of the book.

2.6 Exercises

2.1 Write regular expressions to capture

- (a) Strings in C. These are delimited by double quotes ("), and may not contain newline characters. They may contain double quote or backslash characters if and only if those characters are “escaped” by a preceding backslash. You may find it helpful to introduce shorthand notation to represent any character that is *not* a member of a small specified set.
- (b) Comments in Pascal. These are delimited by (* and *), as shown in Figure 2.6, or by { and }.
- (c) Floating-point constants in Ada. These are the same as in Pascal (see the definition of `unsigned_number` in Example 2.2 [page 41]), except that (1) an underscore is permitted between digits, and (2) an alternative numeric base may be specified by surrounding the non-exponent part of the number with pound signs, preceded by a base in decimal (e.g., `16#6.a7#e+2`). In this latter case, the letters a..f (both upper- and lowercase) are permitted as digits. Use of these letters in an inappropriate (e.g., decimal) number is an error but need not be caught by the scanner.
- (d) Inexact constants in Scheme. Scheme allows real numbers to be explicitly *inexact* (imprecise). A programmer who wants to express all constants using the same number of characters can use sharp signs (#) in place of any lower-significance digits whose values are not known. A base-ten constant without exponent consists of one or more digits followed by zero or more sharp signs. An optional decimal point can be placed at the beginning, the end, or anywhere in between. (For the record, numbers in Scheme are actually a good bit more complicated than this. For the

purposes of this exercise, please ignore anything you may know about sign, exponent, radix, exactness and length specifiers, and complex or rational values.)

- (e) Financial quantities in American notation. These have a leading dollar sign (\$), an optional string of asterisks (*—used on checks to discourage fraud), a string of decimal digits, and an optional fractional part consisting of a decimal point (.) and two decimal digits. The string of digits to the left of the decimal point may consist of a single zero (0). Otherwise it must not start with a zero. If there are more than three digits to the left of the decimal point, groups of three (counting from the right) must be separated by commas (,). Example: \$**2,345.67. (Feel free to use “productions” to define abbreviations, so long as the language remains regular.)
- 2.2 Show (as “circles-and-arrows” diagrams) the finite automata for parts (a) and (c) of Exercise 2.1.
- 2.3 Build a regular expression that captures all nonempty sequences of letters other than `file`, `for`, and `from`. For notational convenience, you may assume the existence of a **not** operator that takes a set of letters as argument and matches any *other* letter. Comment on the practicality of constructing a regular expression for all sequences of letters other than the keywords of a large programming language.
- 2.4 (a) Show the NFA that results from applying the construction of Figure 2.8 to the regular expression *letter* (*letter* | *digit*)^{*}.
- (b) Apply the transformation illustrated by Example 2.12 to create an equivalent DFA.
- (c) Apply the transformation illustrated by Example 2.13 to minimize the DFA.
- 2.5 Build an ad hoc scanner for the calculator language. As output, have it print a list, in order, of the input tokens. For simplicity, feel free to simply halt in the event of a lexical error.
- 2.6 Build a nested-case-statements finite automaton that converts all letters in its input to lowercase, except within Pascal-style comments and strings. A Pascal comment is delimited by { and }, or by {*} and {*}. Comments do not nest. A Pascal string is delimited by single quotes (' ... '). A quote character can be placed in a string by doubling it (''Madam, I ''m Adam.''). This upper-to-lower mapping can be useful if feeding a program written in standard Pascal (which ignores case) to a compiler that considers upper- and lowercase letters to be distinct.
- 2.7 Give an example of a grammar that captures right associativity for an exponentiation operator (e.g., `**` in Fortran).
- 2.8 Prove that the following grammar is LL(1).

$$\begin{aligned} \textit{decl} &\longrightarrow \text{ ID } \textit{decl_tail} \\ \textit{decl_tail} &\longrightarrow , \text{ decl} \\ &\longrightarrow : \text{ ID } ; \end{aligned}$$

(The final ID is meant to be a type name.)

- 2.9 Consider the following grammar.

$$\begin{aligned} G &\longrightarrow S \text{ } \$\$ \\ S &\longrightarrow A \text{ } M \\ M &\longrightarrow S \mid \epsilon \\ A &\longrightarrow \text{a } E \mid \text{b } A \text{ } A \\ E &\longrightarrow \text{a } B \mid \text{b } A \mid \epsilon \\ B &\longrightarrow \text{b } E \mid \text{a } B \text{ } B \end{aligned}$$

- (a) Describe in English the language that the grammar generates.
 - (b) Show a parse tree for the string a b a a.
 - (c) Is the grammar LL(1)? If so, show the parse table; if not, identify a prediction conflict.
- 2.10 Consider the language consisting of all strings of properly balanced parentheses and brackets.
- (a) Give LL(1) and SLR(1) grammars for this language.
 - (b) Give the corresponding LL(1) and SLR(1) parsing tables.
 - (c) For each grammar, show the parse tree for ([]([])) [](()).
 - (d) Give a trace of the actions of the parsers on this input.
- 2.11 Give an example of a grammar that captures all the levels of precedence for arithmetic expressions in C. (*Hint:* This exercise is somewhat tedious. You probably want to attack it with a text editor rather than a pencil, so you can cut, paste, and replace. You can find a summary of C precedence in Figure 6.1 [page 237]; you may want to consult a manual for further details.)
- 2.12 Extend the grammar of Figure 2.24 to include if statements and while loops, along the lines suggested by the following examples.

```

abs := n
if n < 0 then abs := 0 - abs fi

sum := 0
read count
while count > 0 do
    read n
    sum := sum + n
    count := count - 1
od
write sum

```

Your grammar should support the six standard comparison operations in conditions, with arbitrary expressions as operands. It should allow an arbitrary number of statements in the body of an `if` or `while` statement.

- 2.13 Consider the following LL(1) grammar for a simplified subset of Lisp.

$$\begin{array}{l} P \longrightarrow E \ \$\$ \\ E \longrightarrow \text{atom} \\ \quad \longrightarrow ' \ E \\ \quad \longrightarrow (\ E \ Es \) \\ Es \longrightarrow E \ Es \\ \quad \longrightarrow \end{array}$$

- (a) What is $\text{FIRST}(Es)$? $\text{FOLLOW}(E)$? $\text{PREDICT}(Es \longrightarrow \epsilon)$?
 - (b) Give a parse tree for the string `(cdr ' (a b c)) $$`.
 - (c) Show the left-most derivation of `(cdr ' (a b c)) $$`.
 - (d) Show a trace, in the style of Figure 2.20, of a table-driven top-down parse of this same input.
 - (e) Now consider a recursive descent parser running on the same input. At the point where the quote token `'` is matched, which recursive descent routines will be active (i.e., what routines will have a frame on the parser's run-time stack)?
- 2.14 Write top-down and bottom-up grammars for the language consisting of all well-formed regular expressions. Arrange for all operators to be left-associative. Give Kleene closure the highest precedence and alternation the lowest precedence.
- 2.15 Suppose that the expression grammar in Example 2.7 were to be used in conjunction with a scanner that did *not* remove comments from the input but rather returned them as tokens. How would the grammar need to be modified to allow comments to appear at arbitrary places in the input?
- 2.16 Build a complete recursive descent parser for the calculator language. As output, have it print a trace of its matches and predictions.
- 2.17 Flesh out the details of an algorithm to eliminate left recursion and common prefixes in an arbitrary context-free grammar.
- 2.18 In some languages an assignment can appear in any context in which an expression is expected: the value of the expression is the right-hand side of the assignment, which is placed into the left-hand side as a side effect. Consider the following grammar fragment for such a language. Explain why it is not LL(1), and discuss what might be done to make it so.

$$\begin{array}{l} \text{expr} \longrightarrow \text{id} := \text{expr} \\ \text{expr} \longrightarrow \text{term} \ \text{term_tail} \\ \text{term_tail} \longrightarrow + \ \text{term} \ \text{term_tail} \mid \epsilon \end{array}$$

$$\begin{aligned} \text{term} &\longrightarrow \text{factor factor_tail} \\ \text{factor_tail} &\longrightarrow * \text{ factor factor_tail} \mid \epsilon \\ \text{factor} &\longrightarrow (\text{expr}) \mid \text{id} \end{aligned}$$

- 2.19 Construct a trace over time of the forest of partial parse trees manipulated by a bottom-up parser for the string A, B, C;, using the grammar in Example 2.19 (the one that is able to collapse prefixes of the *id_list* as it goes along).
- 2.20 Construct the CFSM for the *id_list* grammar in Example 2.18 (page 62) and verify that it can be parsed bottom-up with zero tokens of look-ahead.
- 2.21 Modify the grammar in Exercise 2.20 to allow an *id_list* to be empty. Is the grammar still LR(0)?
- 2.22 Consider the following grammar for a declaration list.

$$\begin{aligned} \text{decl_list} &\longrightarrow \text{decl_list decl} \mid \text{decl} \\ \text{decl} &\longrightarrow \text{id} : \text{type} \\ \text{type} &\longrightarrow \text{int} \mid \text{real} \mid \text{char} \\ &\longrightarrow \text{array const} \dots \text{const of type} \\ &\longrightarrow \text{record decl_list end} \end{aligned}$$

Construct the CFSM for this grammar. Use it to trace out a parse (as in Figure 2.29) for the following input program.

```
foo : record
    a : char;
    b : array 1..2 of real;
end;
```

- 2.23 The dangling *else* problem of Pascal is not shared by Algol 60. To avoid ambiguity regarding which *then* is matched by an *else*, Algol 60 prohibits *if* statements immediately inside a *then* clause. The Pascal fragment

```
if C1 then if C2 then S1 else S2
```

must be written as either

```
if C1 then begin if C2 then S1 end else S2
```

or

```
if C1 then begin if C2 then S1 else S2 end
```

in Algol 60. Show how to write a grammar for conditional statements that enforces this rule. (*Hint:* You will want to distinguish in your grammar between conditional statements and nonconditional statements; some contexts will accept either, some only the latter.)

- © 2.24–2.28 In More Depth.

2.7 Explorations

- 2.29 Some languages (e.g., C) distinguish between upper- and lowercase letters in identifiers. Others (e.g., Ada) do not. Which convention do you prefer? Why?
- 2.30 The syntax for type casts in C and its descendants introduces potential ambiguity: is $(x)-y$ a subtraction, or the unary negation of y , cast to type x ? Find out how C, C++, Java, and C# answer this question. Discuss how you would implement the answer(s).
- 2.31 What do you think of Haskell, Occam, and Python’s use of indentation to delimit control constructs (Section 2.1.1)? Would you expect this convention to make program construction and maintenance easier or harder? Why?
- 2.32 Skip ahead to Section 13.4.2 and learn about the “regular expressions” used in scripting languages, editors, search tools, and so on. Are these really regular? What can they express that cannot be expressed in the notation introduced in Section 2.1.1?
- 2.33 Rebuild the automaton of Exercise 2.6 using `lex/flex`.
- 2.34 Find a manual for `yacc/bison`, or consult a compiler textbook [ASU86] to learn about *operator precedence parsing*. Explain how it could be used to simplify the grammar of Exercise 2.11.
- 2.35 Use `lex/flex` and `yacc/bison` to construct a parser for the calculator language. Have it output a trace of its shifts and reductions.
- 2.36 Repeat the previous exercise using ANTLR.

© 2.37–2.38 In More Depth.

2.8 Bibliographic Notes

Our coverage of scanning and parsing in this chapter has of necessity been brief. Considerably more detail can be found in texts on parsing theory [AU72] and compiler construction [App97, ASU86, CT04, FL88, GBJL01]. Many compilers of the early 1960s employed recursive descent parsers. Lewis and Stearns [LS68] and Rosenkrantz and Stearns [RS70] published early formal studies of LL grammars and parsing. The original formulation of LR parsing is due to Knuth [Knu65]. Bottom-up parsing became practical with DeRemer’s discovery of the SLR and LALR algorithms [DeR69, DeR71]. W. L. Johnson et al. [JPAR68] describe an early scanner generator. The Unix `lex` tool is due to Lesk [Les75]. Yacc is due to S. C. Johnson [Joh75].

Further details on formal language theory can be found in a variety of textbooks, including those of Hopcroft, Motwani, and Ullman [HMU01] and

Sipser [Sip97]. Kleene [Kle56] and Rabin and Scott [RS59] proved the equivalence of regular expressions and finite automata.¹⁴ The proof that finite automata are unable to recognize nested constructs is based on a theorem known as the *pumping lemma*, due to Bar-Hillel, Perles, and Shamir [BHP61]. Context-free grammars were first explored by Chomsky [Cho56] in the context of natural language. Independently, Backus and Naur developed BNF for the syntactic description of Algol 60 [NBB⁺63]. Ginsburg and Rice [GR62] recognized the equivalence of the two notations. Chomsky [Cho62] and Evey [Eve63] demonstrated the equivalence of context-free grammars and push-down automata.

Fischer and LeBlanc's text [FL88] contains an excellent survey of error recovery and repair techniques, with references to other work. The phrase-level recovery mechanism for recursive descent parsers described in Section 2.3.4 is due to Wirth [Wir76, Sec. 5.9]. The locally least-cost recovery mechanism for table-driven LL parsers described in Section 2.3.4 is due to Fischer, Milton, and Quiring [FMQ80]. Dion published a locally least-cost bottom-up repair algorithm in 1978 [Dio78]. It is quite complex, and requires very large precomputed tables. More recently, McKenzie, Yeatman, and De Vere have shown how to effect the same repairs without the precomputed tables, at a higher but still acceptable cost in time [MYD95].

14 Dana Scott (1932–), Professor Emeritus at Carnegie Mellon University, is known principally for inventing domain theory and launching the field of denotational semantics, which provides a mathematically rigorous way to formalize the meaning of programming languages. Michael Rabin (1931–), of Harvard University, has made seminal contributions to the concepts of non-determinism and randomization in computer science. Scott and Rabin shared the ACM Turing Award in 1976.

3

Names, Scopes, and Bindings

“High-level” programming languages take their name from the relatively high level, or degree of abstraction, of the features they provide, relative to those of the assembly languages that they were originally designed to replace. The adjective *abstract*, in this context, refers to the degree to which language features are separated from the details of any particular computer architecture. The early development of languages like Fortran, Algol, and Lisp was driven by a pair of complementary goals: machine independence and ease of programming. By abstracting the language away from the hardware, designers not only made it possible to write programs that would run well on a wide variety of machines, but also made the programs easier for human beings to understand.

Machine independence is a fairly simple concept. Basically it says that a programming language should not rely on the features of any particular instruction set for its efficient implementation. Machine dependences still become a problem from time to time (standards committees for C, for example, have only recently agreed on how to accommodate machines with 64-bit arithmetic), but with a few noteworthy exceptions (Java comes to mind) it has probably been 30 years since the desire for greater machine independence has really driven language design. Ease of programming, on the other hand, is a much more elusive and compelling goal. It affects every aspect of language design, and has historically been less a matter of science than of aesthetics and trial and error.

This chapter is the first of five to address core issues in language design. (The others are Chapters 6–9.) In Chapter 6 we will look at control-flow constructs, which allow the programmer to specify the order in which operations are to occur. In contrast to the jump-based control flow of assembly languages, high-level control flow relies heavily on the lexical nesting of constructs. In Chapter 7 we will look at types, which allow the programmer to organize program data and the operations on them. In Chapters 8 and 9 we will look at subroutines and classes. In this current chapter we look at *names*.

A name is a mnemonic character string used to represent something else. Names in most languages are identifiers (alpha-numeric tokens), though certain other symbols, such as + or :=, can also be names. Names allow us to refer to variables, constants, operations, types, and so on using symbolic identifiers rather than low-level concepts like addresses. Names are also essential in the context of a second meaning of the word *abstraction*. In this second meaning, abstraction is a process by which the programmer associates a name with a potentially complicated program fragment, which can then be thought of in terms of its purpose or function, rather than in terms of how that function is achieved. By hiding irrelevant details, abstraction reduces conceptual complexity, making it possible for the programmer to focus on a manageable subset of the program text at any particular time. Subroutines are *control abstractions*: they allow the programmer to hide arbitrarily complicated code behind a simple interface. Classes are *data abstractions*: they allow the programmer to hide data representation details behind a (comparatively) simple set of operations.

We will look at several major issues related to names. Section 3.1 introduces the notion of *binding time*, which refers not only to the binding of a name to the thing it represents, but also in general to the notion of resolving any design decision in a language implementation. Section 3.2 outlines the various mechanisms used to allocate and deallocate storage space for objects, and distinguishes between the lifetime of an object and the lifetime of a binding of a name to that object.¹ Most name-to-object bindings are usable only within a limited region of a given high-level program. Section 3.3 explores the *scope* rules that define this region; Section 3.4 (mostly on the PLP CD) considers their implementation.

The complete set of bindings in effect at a given point in a program is known as the current *referencing environment*. Section 3.5 expands on the notion of scope rules by considering the ways in which a referencing environment may be bound to a subroutine that is passed as a parameter, returned from a function, or stored in a variable. Section 3.6 discusses aliasing, in which more than one name may refer to a given object in a given scope; overloading, in which a name may refer to more than one object in a given scope, depending on the context of the reference; and polymorphism, in which a single object may have more than one type, depending on context or execution history. Finally, Section 3.7 (mostly on the PLP CD) discusses separate compilation.

3.1 The Notion of Binding Time

A *binding* is an association between two things, such as a name and the thing it names. *Binding time* is the time at which a binding is created or, more generally,

I For want of a better term, we will use the term *object* throughout Chapters 3–8 to refer to anything that might have a name: variables, constants, types, subroutines, modules, and others. In many modern languages *object* has a more formal meaning, which we will consider in Chapter 9.

the time at which any implementation decision is made (we can think of this as binding an answer to a question). There are many different times at which decisions may be bound:

Language design time: In most languages, the control flow constructs, the set of fundamental (primitive) types, the available *constructors* for creating complex types, and many other aspects of language semantics are chosen when the language is designed.

Language implementation time: Most language manuals leave a variety of issues to the discretion of the language implementor. Typical (though by no means universal) examples include the precision (number of bits) of the fundamental types, the coupling of I/O to the operating system's notion of files, the organization and maximum sizes of stack and heap, and the handling of run-time exceptions such as arithmetic overflow.

Program writing time: Programmers, of course, choose algorithms, data structures, and names.

Compile time: Compilers choose the mapping of high-level constructs to machine code, including the layout of statically defined data in memory.

Link time: Since most compilers support *separate compilation*—compiling different modules of a program at different times—and depend on the availability of a library of standard subroutines, a program is usually not complete until the various modules are joined together by a linker. The linker chooses the overall layout of the modules with respect to one another. It also resolves intermodule references. When a name in one module refers to an object in another module, the binding between the two was not finalized until link time.

Load time: Load time refers to the point at which the operating system loads the program into memory so that it can run. In primitive operating systems, the choice of machine addresses for objects within the program was not finalized until load time. Most modern operating systems distinguish between virtual and physical addresses. Virtual addresses are chosen at link time; physical addresses can actually change at run time. The processor's memory management hardware translates virtual addresses into physical addresses during each individual instruction at run time.

DESIGN & IMPLEMENTATION

Binding time

It is difficult to overemphasize the importance of binding times in the design and implementation of programming languages. In general, early binding times are associated with greater efficiency, while later binding times are associated with greater flexibility. The tension between the goals provides a recurring theme for later chapters of this book.

Run time: Run time is actually a very broad term that covers the entire span from the beginning to the end of execution. Bindings of values to variables occur at run time, as do a host of other decisions that vary from language to language. Run time subsumes program start-up time, module entry time, elaboration time (the point at which a declaration is first “seen”), subroutine call time, block entry time, and statement execution time.

The terms *static* and *dynamic* are generally used to refer to things bound before run time and at run time, respectively. Clearly *static* is a coarse term. So is *dynamic*.

Compiler-based language implementations tend to be more efficient than interpreter-based implementations because they make earlier decisions. For example, a compiler analyzes the syntax and semantics of global variable declarations once, before the program ever runs. It decides on a layout for those variables in memory, and generates efficient code to access them wherever they appear in the program. A pure interpreter, by contrast, must analyze the declarations every time the program begins execution. In the worst case, an interpreter may reanalyze the local declarations within a subroutine each time that subroutine is called. If a call appears in a deeply nested loop, the savings achieved by a compiler that is able to analyze the declarations only once may be very large. As we shall see in the following section, a compiler will not usually be able to predict the address of a local variable at compile time, since space for the variable will be allocated dynamically on a stack, but it can arrange for the variable to appear at a fixed offset from the location pointed to by a certain register at run time.

Some languages are difficult to compile because their definitions require certain fundamental decisions to be postponed until run time, generally in order to increase the flexibility or expressiveness of the language. Smalltalk, for example, delays all type checking until run time. All operations in Smalltalk are cast in the form of “messages” to “objects.” A message is acceptable if and only if the object provides a handler for it. References to objects of arbitrary types (classes) can then be assigned into arbitrary named variables, as long as the program never ends up sending a message to an object that is not prepared to handle it. This form of *polymorphism*—allowing a variable name to refer to objects of multiple types—allows the Smalltalk programmer to write very general purpose code, which will correctly manipulate objects whose types had yet to be fully defined at the time the code was written. We will mention polymorphism again in Section 3.6.3, and discuss it further in Chapters 7 and 9.

3.2 Object Lifetime and Storage Management

In any discussion of names and bindings, it is important to distinguish between names and the objects to which they refer, and to identify several key events:

- The creation of objects
- The creation of bindings
- References to variables, subroutines, types, and so on, all of which use bindings
- The deactivation and reactivation of bindings that may be temporarily unusable
- The destruction of bindings
- The destruction of objects

The period of time between the creation and the destruction of a name-to-object binding is called the binding's *lifetime*. Similarly, the time between the creation and destruction of an object is the object's lifetime. These lifetimes need not necessarily coincide. In particular, an object may retain its value and the potential to be accessed even when a given name can no longer be used to access it. When a variable is passed to a subroutine by *reference*, for example (as it typically is in Fortran or with var parameters in Pascal or "&" parameters in C++), the binding between the parameter name and the variable that was passed has a lifetime shorter than that of the variable itself. It is also possible, though generally a sign of a program bug, for a name-to-object binding to have a lifetime *longer* than that of the object. This can happen, for example, if an object created via the C++ new operator is passed as a & parameter and then deallocated (delete-ed) before the subroutine returns. A binding to an object that is no longer live is called a *dangling reference*. Dangling references will be discussed further in Sections 3.5 and 7.7.2.

Object lifetimes generally correspond to one of three principal *storage allocation* mechanisms, used to manage the object's space:

1. *Static* objects are given an absolute address that is retained throughout the program's execution.
2. *Stack* objects are allocated and deallocated in last-in, first-out order, usually in conjunction with subroutine calls and returns.
3. *Heap* objects may be allocated and deallocated at arbitrary times. They require a more general (and expensive) storage management algorithm.

3.2.1 Static Allocation

Global variables are the obvious example of static objects, but not the only one. The instructions that constitute a program's machine-language translation can also be thought of as statically allocated objects. In addition, we shall see examples in Section 3.3.1 of variables that are local to a single subroutine but retain their values from one invocation to the next; their space is statically allocated. Numeric and string-valued constant literals are also statically allocated, for statements such as `A = B/14.7` or `printf("hello, world\n")`. (Small constants

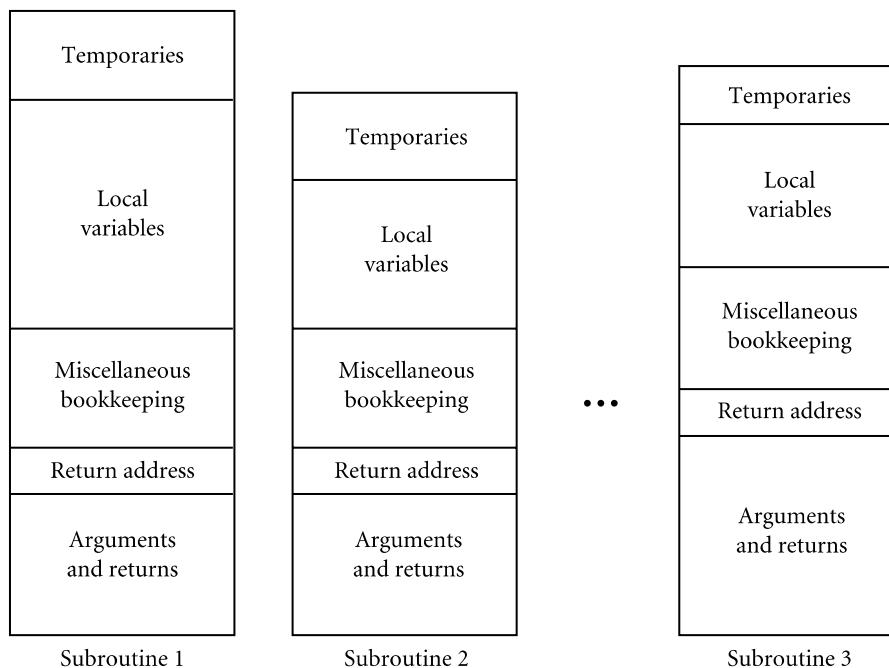


Figure 3.1 Static allocation of space for subroutines in a language or program without recursion.

are often stored within the instruction itself; larger ones are assigned a separate location.) Finally, most compilers produce a variety of tables that are used by run-time support routines for debugging, dynamic type checking, garbage collection, exception handling, and other purposes; these are also statically allocated. Statically allocated objects whose value should not change during program execution (e.g., instructions, constants, and certain run-time tables) are often allocated in protected, read-only memory so that any inadvertent attempt to write to them will cause a processor interrupt, allowing the operating system to announce a run-time error.

Logically speaking, local variables are created when their subroutine is called and destroyed when it returns. If the subroutine is called repeatedly, each invocation is said to create and destroy a separate *instance* of each local variable. It is not always the case, however, that a language implementation must perform work at run time corresponding to these create and destroy operations. Recursion was not originally supported in Fortran (it was added in Fortran 90). As a result, there can never be more than one invocation of a subroutine active at any given time, and a compiler may choose to use static allocation for local variables, effectively arranging for the variables of different invocations to share the same locations, and thereby avoiding any run-time overhead for creation and destruction (Figure 3.1). ■

EXAMPLE 3.1

Static allocation of local variables

In many languages a constant is required to have a value that can be determined at compile time. Usually the expression that specifies the constant's value is permitted to include only literal (*manifest*) constants and built-in functions and arithmetic operators. These sorts of *compile-time constants* can always be allocated statically, even if they are local to a recursive subroutine: multiple instances can share the same location. In other languages (e.g., C and Ada), constants are simply variables that cannot be changed after elaboration time. Their values, though unchanging, can depend on other values that are not known until run time. These *elaboration-time constants*, when local to a recursive subroutine, must be allocated on the stack. C# provides both options, explicitly, with the `const` and `readonly` keywords.

Along with local variables and elaboration-time constants, the compiler typically stores a variety of other information associated with the subroutine, including the following.

Arguments and return values. Modern compilers tend to keep these in registers when possible, but sometimes space in memory is needed.

Temporaries. These are usually intermediate values produced in complex calculations. Again, a good compiler will keep them in registers whenever possible.

Bookkeeping information. This may include the subroutine's return address, a reference to the stack frame of the caller (also called the *dynamic link*), additional saved registers, debugging information, and various other values that we will study later.

3.2.2 Stack-Based Allocation

EXAMPLE 3.2

Layout of the run-time stack

If a language permits recursion, static allocation of local variables is no longer an option, since the number of instances of a variable that may need to exist at the same time is conceptually unbounded. Fortunately, the natural nesting of subroutine calls makes it easy to allocate space for locals on a stack. A simplified picture of a typical stack appears in Figure 3.2. Each instance of a subroutine at run time has its own *frame* (also called an *activation record*) on the stack, containing arguments and return values, local variables, temporaries, and bookkeeping

DESIGN & IMPLEMENTATION

Recursion in Fortran

The lack of recursion in (pre-Fortran 90) Fortran is generally attributed to the expense of stack manipulation on the IBM 704, on which the language was first implemented. Many (perhaps most) Fortran implementations choose to use a stack for local variables, but because the language definition permits the use of static allocation instead, Fortran programmers were denied the benefits of language-supported recursion for over 30 years.

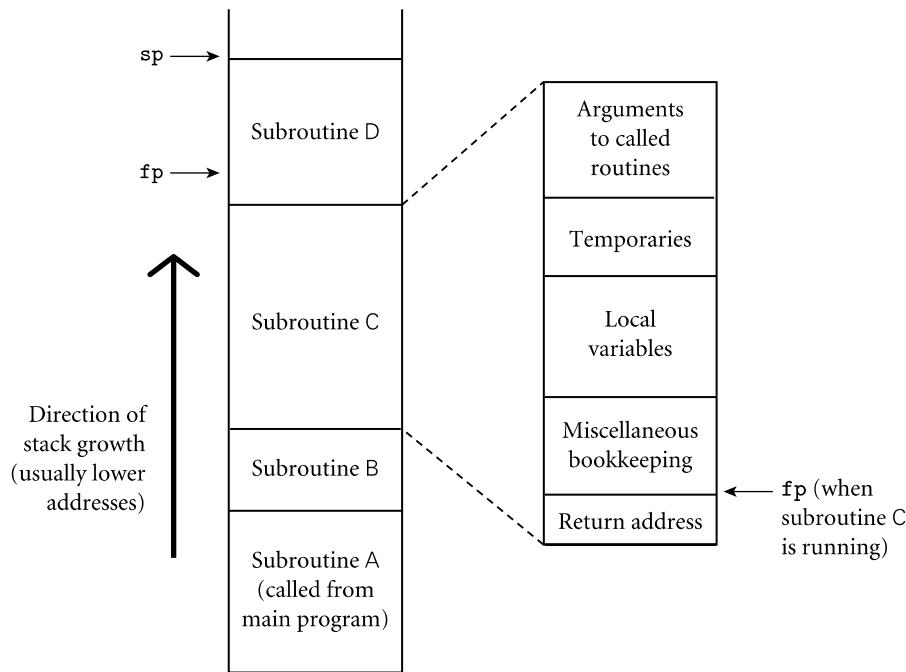


Figure 3.2 Stack-based allocation of space for subroutines. We assume here that subroutine A has been called by the main program and that it then calls subroutine B. Subroutine B subsequently calls C, which in turn calls D. At any given time, the stack pointer (**sp**) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (**fp**) register points to a known location within the frame (activation record) of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

information. Arguments to be passed to subsequent routines lie at the top of the frame, where the callee can easily find them. The organization of the remaining information is implementation-dependent: it varies from one language and compiler to another. ■

Maintenance of the stack is the responsibility of the subroutine *calling sequence*—the code executed by the caller immediately before and after the call—and of the *prologue* (code executed at the beginning) and *epilogue* (code executed at the end) of the subroutine itself. Sometimes the term “calling sequence” is used to refer to the combined operations of the caller, the prologue, and the epilogue. We will study calling sequences in more detail in Section 8.2.

While the location of a stack frame cannot be predicted at compile time (the compiler cannot in general tell what other frames may already be on the stack), the offsets of objects *within* a frame usually *can* be statically determined. Moreover, the compiler can arrange (in the calling sequence or prologue) for a particular register, known as the *frame pointer*, to always point to a known location

within the frame of the current subroutine. Code that needs to access a local variable within the current frame, or an argument near the top of the calling frame, can do so by adding a predetermined offset to the value in the frame pointer. As we shall see in Section 5.3.1, almost every processor provides an *addressing mode* that allows this addition to be specified implicitly as part of an ordinary *load* or *store* instruction. The stack grows “downward” toward lower addresses in most language implementations. Some machines provide special *push* and *pop* instructions that assume this direction of growth. Arguments and returns typically have positive offsets from the frame pointer; local variables, temporaries, and bookkeeping information typically have negative offsets.

Even in a language without recursion, it can be advantageous to use a stack for local variables, rather than allocating them statically. In most programs the pattern of potential calls among subroutines does not permit all of those subroutines to be active at the same time. As a result, the total space needed for local variables of currently active subroutines is seldom as large as the total space across *all* subroutines, active or not. A stack may therefore require substantially less memory at run time than would be required for static allocation.

3.2.3 Heap-Based Allocation

A *heap* is a region of storage in which subblocks can be allocated and deallocated at arbitrary times.² Heaps are required for the dynamically allocated pieces of linked data structures and for dynamically resized objects, such as fully general character strings, lists, and sets, whose size may change as a result of an assignment statement or other update operation.

There are many possible strategies to manage space in a heap. We review the major alternatives here; details can be found in any data-structures textbook. The principal concerns are speed and space, and as usual there are tradeoffs between them. Space concerns can be further subdivided into issues of internal and external *fragmentation*. Internal fragmentation occurs when a storage-management algorithm allocates a block that is larger than required to hold a given object; the extra space is then unused. External fragmentation occurs when the blocks that have been assigned to active objects are scattered through the heap in such a way that the remaining, unused space is composed of multiple blocks: there may be quite a lot of free space, but no one piece of it may be large enough to satisfy some future request (see Figure 3.3). ■

Many storage-management algorithms maintain a single linked list—the *free list*—of heap blocks not currently in use. Initially the list consists of a single block comprising the entire heap. At each allocation request the algorithm searches the list for a block of appropriate size. With a *first fit* algorithm we select the

EXAMPLE 3.3

External fragmentation in the heap

2 Unfortunately, the term *heap* is also used for a common tree-based implementation of a priority queue. These two uses of the term have nothing to do with one another.

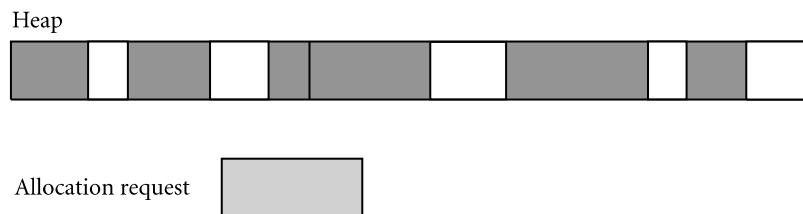


Figure 3.3 External fragmentation. The shaded blocks are in use; the clear blocks are free. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

first block on the list that is large enough to satisfy the request. With a *best fit* algorithm we search the entire list to find the smallest block that is large enough to satisfy the request. In either case, if the chosen block is significantly larger than required, then we divide it in two and return the unneeded portion to the free list as a smaller block. (If the unneeded portion is below some minimum threshold in size, we may leave it in the allocated block as internal fragmentation.) When a block is deallocated and returned to the free list, we check to see whether either or both of the physically adjacent blocks are free; if so, we coalesce them.

Intuitively, one would expect a best fit algorithm to do a better job of reserving large blocks for large requests. At the same time, it has a higher allocation cost than a first fit algorithm, because it must always search the entire list, and it tends to result in a larger number of very small “leftover” blocks. Which approach—first fit or best fit—results in lower external fragmentation depends on the distribution of size requests.

In any algorithm that maintains a single free list, the cost of allocation is linear in the number of free blocks. To reduce this cost to a constant, some storage management algorithms maintain separate free lists for blocks of different sizes. Each request is rounded up to the next standard size (at the cost of internal fragmentation) and allocated from the appropriate list. In effect, the heap is divided into “pools,” one for each standard size. The division may be static or dynamic. Two common mechanisms for dynamic pool adjustment are known as the *buddy system* and the *Fibonacci heap*. In the buddy system, the standard block sizes are powers of two. If a block of size 2^k is needed, but none is available, a block of size 2^{k+1} is split in two. One of the halves is used to satisfy the request; the other is placed on the k th free list. When a block is deallocated, it is coalesced with its “buddy”—the other half of the split that created it—if that buddy is free. Fibonacci heaps are similar, but they use Fibonacci numbers for the standard sizes, instead of powers of two. The algorithm is slightly more complex but leads to slightly lower internal fragmentation because the Fibonacci sequence grows more slowly than 2^k .

The problem with external fragmentation is that the ability of the heap to satisfy requests may degrade over time. Multiple free lists may help, by clustering small blocks in relatively close physical proximity, but they do not eliminate the

problem. It is always possible to devise a sequence of requests that cannot be satisfied, even though the total space required is less than the size of the heap. If size pools are statically allocated, one need only exceed the maximum number of requests of a given size. If pools are dynamically readjusted, one can “checkerboard” the heap by allocating a large number of small blocks and then deallocating every other one, in order of physical address, leaving an alternating pattern of small free and allocated blocks. To eliminate external fragmentation, we must be prepared to *compact* the heap, by moving already-allocated blocks. This task is complicated by the need to find and update all outstanding references to a block that is being moved. We will discuss compaction further in Sections 7.7.2 and 7.7.3.

3.2.4 Garbage Collection

Allocation of heap-based objects is always triggered by some specific operation in a program: instantiating an object, appending to the end of a list, assigning a long value into a previously short string, and so on. Deallocation is also explicit in some languages (e.g., C, C++, and Pascal.) As we shall see in Section 7.7, however, many languages specify that objects are to be deallocated implicitly when it is no longer possible to reach them from any program variable. The run-time library for such a language must then provide a *garbage collection* mechanism to identify and reclaim unreachable objects. Most functional languages require garbage collection, as do many more recent imperative languages, including Modula-3, Java, C#, and all the major scripting languages.

The traditional arguments in favor of explicit deallocation are implementation simplicity and execution speed. Even naive implementations of automatic garbage collection add significant complexity to the implementation of a language with a rich type system, and even the most sophisticated garbage collector can consume nontrivial amounts of time in certain programs. If the programmer can correctly identify the end of an object’s lifetime, without too much run-time bookkeeping, the result is likely to be faster execution.

The argument in favor of automatic garbage collection, however, is compelling: manual deallocation errors are among the most common and costly bugs in real-world programs. If an object is deallocated too soon, the program may follow a *dangling reference*, accessing memory now used by another object. If an object is *not* deallocated at the end of its lifetime, then the program may “leak memory,” eventually running out of heap space. Deallocation errors are notoriously difficult to identify and fix. Over time, both language designers and programmers have increasingly come to consider automatic garbage collection an essential language feature. Garbage-collection algorithms have improved, reducing their run-time overhead; language implementations have become more complex in general, reducing the marginal complexity of automatic collection; and leading-edge applications have become larger and more complex, making the benefits of automatic collection ever more appealing.

 **CHECK YOUR UNDERSTANDING**

1. What is *binding time*?
2. Explain the distinction between decisions that are bound statically and those that are bound dynamically.
3. What is the advantage of binding things as early as possible? What is the advantage of delaying bindings?
4. Explain the distinction between the *lifetime* of a name-to-object binding and its *visibility*.
5. What determines whether an object is allocated statically, on the stack, or in the heap?
6. List the objects and information commonly found in a stack frame.
7. What is a *frame pointer*? What is it used for?
8. What is a *calling sequence*?
9. What are internal and external *fragmentation*?
10. What is *garbage collection*?
11. What is a *dangling reference*?

3.3 Scope Rules

The textual region of the program in which a binding is active is its *scope*. In most modern languages, the scope of a binding is determined statically—that is, at compile time. In C, for example, we introduce a new scope upon entry to a subroutine. We create bindings for local objects and deactivate bindings for global objects that are “hidden” by local objects of the same name. On subroutine exit, we destroy bindings for local variables and reactivate bindings for any global objects that were hidden. These manipulations of bindings may at first glance appear to be run-time operations, but they do not require the execution of any code: the portions of the program in which a binding is active are completely determined at compile time. We can look at a C program and know which names refer to which objects at which points in the program based on purely textual rules. For this reason, C is said to be *statically scoped* (some authors say *lexically scoped*³).

3 *Lexical scope* is actually a better term than *static scope*, because scope rules based on nesting can be enforced at run time instead of compile time if desired. In fact, in Common Lisp and Scheme it is possible to pass the unevaluated text of a subroutine declaration into some other subroutine as a parameter, and then use the text to create a lexically nested declaration at run time.

Other languages, including APL, Snobol, and early dialects of Lisp, are *dynamically scoped*: their bindings depend on the flow of execution at run time. We will examine static and dynamic scope in more detail in Sections 3.3.1 and 3.3.6.

In addition to talking about the “scope of a binding,” we sometimes use the word *scope* as a noun all by itself, without a specific binding in mind. Informally, a scope is a program region of maximal size in which no bindings change (or at least none are destroyed—more on this in Section 3.3.3). Typically, a scope is the body of a module, class, subroutine, or structured control flow statement, sometimes called a *block*. In C family languages it would be delimited with { . . . } braces.

Algol 68 and Ada use the term *elaboration* to refer to the process by which declarations become active when control first enters a scope. Elaboration entails the creation of bindings. In many languages, it also entails the allocation of stack space for local objects, and possibly the assignment of initial values. In Ada it can entail a host of other things, including the execution of error-checking or heap-space-allocating code, the propagation of exceptions, and the creation of concurrently executing *tasks* (to be discussed in Chapter 12).

At any given point in a program’s execution, the set of active bindings is called the current *referencing environment*. The set is principally determined by static or dynamic *scope rules*. We shall see that a referencing environment generally corresponds to a sequence of scopes that can be examined (in order) to find the current binding for a given name.

In some cases, referencing environments also depend on what are (in a confusing use of terminology) called *binding rules*. Specifically, when a reference to a subroutine *S* is stored in a variable, passed as a parameter to another subroutine, or returned as a function value, one needs to determine when the referencing environment for *S* is chosen—that is, when the binding between the reference to *S* and the referencing environment of *S* is made. The two principal options are *deep binding*, in which the choice is made when the reference is first created, and *shallow binding*, in which the choice is made when the reference is finally used. We will examine these options in more detail in Section 3.5.

3.3.1 Static Scope

In a language with static (lexical) scoping, the bindings between names and objects can be determined at compile time by examining the text of the program, without consideration of the flow of control at run time. Typically, the “current” binding for a given name is found in the matching declaration whose block most closely surrounds a given point in the program, though as we shall see there are many variants on this basic theme.

The simplest static scope rule is probably that of early versions of Basic, in which there was only a single, global scope. In fact, there were only a few hundred possible names, each of which consisted of a letter optionally followed by a digit.

There were no explicit declarations; variables were declared implicitly by virtue of being used.

Scope rules are somewhat more complex in Fortran, though not much more.⁴ Fortran distinguishes between global and local variables. The scope of a local variable is limited to the subroutine in which it appears; it is not visible elsewhere. Variable declarations are optional. If a variable is not declared, it is assumed to be local to the current subroutine and to be of type `integer` if its name begins with the letters I–N, or `real` otherwise. (Different conventions for implicit declarations can be specified by the programmer. In Fortran 90, the programmer can also turn off implicit declarations, so that use of an undeclared variable becomes a compile-time error.)

Global variables in Fortran may be partitioned into `common` blocks, which are then “imported” by subroutines. `Common` blocks are designed to support separate compilation: they allow a subroutine to import only a subset of the global environment. Unfortunately, Fortran requires each subroutine to declare the names and types of the variables in each of the `common` blocks it uses, and there is no standard mechanism to ensure that the declarations in different subroutines are the same. In fact, Fortran explicitly allows the declarations to be different. A programmer who knows the data layout rules employed by the compiler can use a completely different set of names and types in one subroutine to refer to the data defined in another subroutine. The underlying bits will be shared, but the effect of this sharing is highly implementation-dependent. A similar effect can be achieved through the (mis)use of `equivalence` statements, which allow the programmer to specify that a set of variables share the same location(s). `Equivalence` statements are a precursor of the variant records and unions of languages like Pascal and C. Their intended purpose is to save space in programs in which only one of the `equivalence`-ed variables is in use at any one time.

Semantically, the lifetime of a local Fortran variable (both the object itself and the name-to-object binding) encompasses a single execution of the variable’s subroutine. Programmers can override this rule by using an explicit `save` statement. A `save`-ed variable has a lifetime that encompasses the entire execution of the program. Instead of a logically separate object for every invocation of the subroutine, the `save` statement creates a single object that retains its value from one invocation of the subroutine to the next. (The name-to-variable binding, of course, is inactive when the subroutine is not executing, because the name is out of scope.)

In early implementations of Fortran, it was common for all local variables to behave as if they were `save`-ed, because language implementations employed the static allocation strategy described in Section 3.2. It is a dangerous practice to

⁴ Fortran and C have evolved considerably over the years. Unless otherwise noted, comments in this text apply to the Fortran 77 dialect [Ame78a] (still more widely used than the newer Fortran 90). Comments on C refer to all versions of the language (including the C99 standard [Int99]) unless otherwise noted. Comments on Ada, likewise, refer to both Ada 83 [Ame83] and Ada 95 [Int95b] unless otherwise noted.

depend on this implementation artifact, however, because it is not guaranteed by the language definition. In a Fortran compiler that uses a stack to save space, or that exploits knowledge of the patterns of calls among subroutines to overlap statically allocated space (Exercise 3.10), non-`save`-ed variables may *not* retain their values from one invocation to the next.

3.3.2 Nested Subroutines

The ability to nest subroutines inside each other, introduced in Algol 60, is a feature of many modern languages, including Pascal, Ada, ML, Scheme, and Common Lisp. Other languages, including C and its descendants, allow classes or other scopes to nest. Just as the local variables of a Fortran subroutine are not visible to other subroutines, any constants, types, variables, or subroutines declared within a block are not visible outside that block in Algol-family languages. More formally, Algol-style nesting gives rise to the *closest nested scope rule* for resolving bindings from names to objects: a name that is introduced in a declaration is known in the scope in which it is declared, and in each internally nested scope, unless it is *hidden* by another declaration of the same name in one or more nested scopes. To find the object referenced by a given use of a name, we look for a declaration with that name in the current, innermost scope. If there is one, it defines the active binding for the name. Otherwise, we look for a declaration in the immediately surrounding scope. We continue outward, examining successively surrounding scopes, until we reach the outer nesting level of the program, where global objects are declared. If no declaration is found at any level, then the program is in error.

Many languages provide a collection of *built-in*, or *predefined*, objects, such as I/O routines, trigonometric functions, and in some cases types such as `integer` and `char`. It is common to consider these to be declared in an extra, invisible, outermost scope, which surrounds the scope in which global objects are declared. The search for bindings described in the previous paragraph terminates at this extra, outermost scope, if it exists, rather than at the scope in which global objects are declared. This outermost scope convention makes it possible for a programmer to define a global object whose name is the same as that of some predefined object (whose “declaration” is thereby hidden, making it unusable).

An example of nested scopes appears in Figure 3.4.⁵ In this example, procedure P2 is called only by P1, and need not be visible outside. It is therefore declared inside P1, limiting its scope (its region of visibility) to the portion of the program shown here. In a similar fashion, P4 is visible only within P1, P3 is visible only within P2, and F1 is visible only within P4. Under the standard rules for nested scopes, F1 could call P2, and P4 could call F1, but P2 could not call F1.

EXAMPLE 3.4
Nested scopes

⁵ This code is not contrived; it was extracted from an implementation of the FMQ error repair algorithm described in Section 2.3.4.

```

procedure P1(A1 : T1);
var X : real;
...
procedure P2(A2 : T2);
...
procedure P3(A3 : T3);
...
begin
    ...
    (* body of P3 *)
end;
...
begin
    ...
    (* body of P2 *)
end;
...
procedure P4(A4 : T4);
...
function F1(A5 : T5) : T6;
var X : integer;
...
begin
    ...
    (* body of F1 *)
end;
...
begin
    ...
    (* body of P4 *)
end;
...
begin
    ...
    (* body of P1 *)
end

```

Figure 3.4 Example of nested subroutines in Pascal.

Though they are hidden from the rest of the program, nested subroutines are able to access the parameters and local variables (and other local objects) of the surrounding scope(s). In our example, P3 can name (and modify) A1, X, and A2, in addition to A3. Because P1 and F1 both declare local variables named X, the inner declaration hides the outer one within a portion of its scope. Uses of X in F1 refer to the inner X; uses of X in other regions of the code shown here refer to the outer X. ■

A name-to-object binding that is hidden by a nested declaration of the same name is said to have a *hole* in its scope. In most languages the object whose name is hidden is inaccessible in the nested scope (unless it has more than one name). Some languages allow the programmer to access the outer meaning of a name by applying a *qualifier* or *scope resolution operator*. In Ada, for example, a name may

be prefixed by the name of the scope in which it is declared, using syntax that resembles the specification of fields in a record. `My_proc.X`, for example, refers to the declaration of `X` in subroutine `My_proc`, regardless of whether some other `X` has been declared in a lexically closer scope. In C++, which does not allow subroutines to nest, `::X` refers to a global declaration of `X`, regardless of whether the current subroutine also has an `X`.⁶

Access to Nonlocal Objects

We have already seen that the compiler can arrange for a frame pointer register to point to the frame of the currently executing subroutine at run time. Target code can use this register to access local objects, as well as any objects in surrounding scopes that are still within the same subroutine. But what about objects in lexically surrounding subroutines? To find these we need a way to find the frames corresponding to those scopes at run time. Since a deeply nested subroutine may call a routine in an outer scope, it is *not* the case that the lexically surrounding scope corresponds to the caller's scope at run time. At the same time, we can be sure that there *is* some frame for the surrounding scope somewhere below in the stack, since the current subroutine could not have been called unless it was visible, and it could not have been visible unless the surrounding scope was active. (It is actually possible in some languages to save a reference to a nested subroutine and then call it when the surrounding scope is no longer active. We defer this possibility to Section 3.5.2.)

The simplest way in which to find the frames of surrounding scopes is to maintain a *static link* in each frame that points to the “parent” frame: the frame of the most recent invocation of the lexically surrounding subroutine. If a subroutine is declared at the outermost nesting level of the program, then its frame will have a null static link at run time. If a subroutine is nested k levels deep, then its frame's static link, and those of its parent, grandparent, and so on, will form a *static chain* of length k at run time. To find a variable or parameter declared j subroutine scopes outward, target code at run time can dereference the static chain j times, and then add the appropriate offset. Static chains are illustrated in Figure 3.5. We will discuss the code required to maintain them in Section 8.2.

EXAMPLE 3.5
Static chains

3.3.3 Declaration Order

In our discussion so far we have glossed over an important subtlety: suppose an object `x` is declared somewhere within block `B`. Does the scope of `x` include the portion of `B` before the declaration, and if so, can `x` actually be used in that portion of the code? Put another way, can an expression `E` refer to any name

6 The C++ `::` operator is also used to name members (fields or methods) of a base class that are hidden by members of a derived class; we will consider this use in Section 9.2.2.

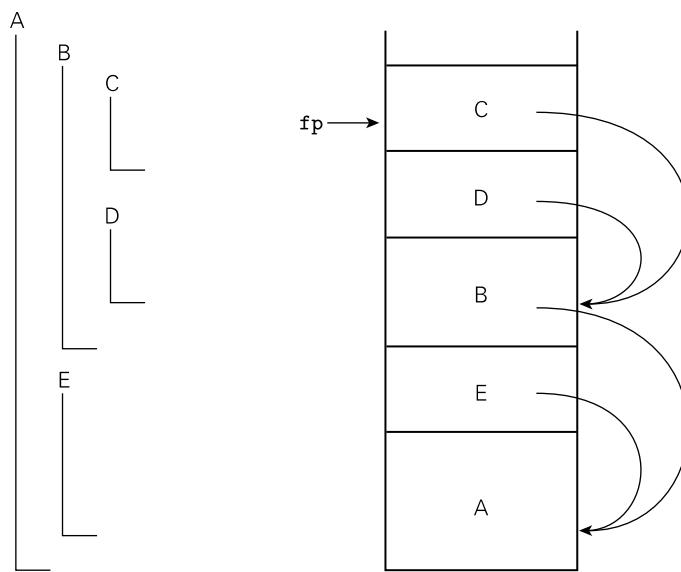


Figure 3.5 Static chains. Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

declared in the current scope, or only to names that are declared *before E* in the scope?

Several early languages, including Algol 60 and Lisp, required that all declarations appear at the beginning of their scope. One might at first think that this rule would avoid the questions in the preceding paragraph, but it does not, because declarations may refer to one another.⁷

DESIGN & IMPLEMENTATION

Mutual recursion

Some Algol 60 compilers were known to process the declarations of a scope in program order. This strategy had the unfortunate effect of implicitly outlawing mutually recursive subroutines and types, something the language designers clearly did not intend [Atk73].

⁷ We saw an example of mutually recursive subroutines in the recursive descent parsing of Section 2.3.1. Mutually recursive types frequently arise in linked data structures, where nodes of two types may need to point to each other.

EXAMPLE 3.6

A “gotcha” in
declare-before-use

In an apparent attempt at simplification, Pascal modified the requirement to say that names must be declared before they are used (with special-case mechanisms to accommodate recursive types and subroutines). At the same time, however, Pascal retained the notion that the scope of a declaration is the entire surrounding block. These two rules can interact in surprising ways:

```

1. const N = 10;
2. ...
3. procedure foo;
4. const
5.     M = N; (* static semantic error! *)
6. ...
7.     N = 20; (* additional constant declaration; hides the outer N *)

```

Pascal says that the second declaration of `N` covers all of `foo`, so the semantic analyzer should complain on line 5 that `N` is being used before its declaration. The error has the potential to be highly confusing, particularly if the programmer meant to use the outer `N`:

```

const N = 10;
...
procedure foo;
const
    M = N;          (* static semantic error! *)
var
    A : array [1..M] of integer;
    N : real;        (* hiding declaration *)

```

Here the pair of messages “`N` used before declaration” and “`N` is not a constant” are almost certainly not helpful.

In order to determine the validity of any declaration that appears to use a name from a surrounding scope, a Pascal compiler must scan the remainder of the scope’s declarations to see if the name is hidden. To avoid this complication, most Pascal successors (and some dialects of Pascal itself) specify that the scope of an identifier is not the entire block in which it is declared (excluding holes), but rather the portion of that block from the declaration to the end (again excluding holes). If our program fragment had been written in Ada, for example, or in C, C++, or Java, no semantic errors would be reported. The declaration of `M` would refer to the first (outer) declaration of `N`. ■

C++ and Java further relax the rules by dispensing with the define-before-use requirement in many cases. In both languages, members of a class (including those that are not defined until later in the program text) are visible inside all of the class’s methods. In Java, classes themselves can be declared in any order. Interestingly, while C# echos Java in requiring declaration before use for local variables (but not for classes and members), it returns to the Pascal notion of whole-block scope. Thus the following is invalid in C#.

EXAMPLE 3.7

Whole-block scope in C#

```
class A {
    const int N = 10;
    void foo() {
        const int M = N;      // uses inner N before it is declared
        const int N = 20;
```

Perhaps the simplest approach to declaration order, from a conceptual point of view, is that of Modula-3, which says that the scope of a declaration is the entire block in which it appears (minus any holes created by nested declarations) and that the order of declarations doesn't matter. The principal objection to this approach is that programmers may find it counterintuitive to use a local variable before it is declared. Python takes the "whole block" scope rule one step further by dispensing with variable declarations altogether. In their place it adopts the unusual convention that the local variables of subroutine S are precisely those variables that are written by some statement in the (static) body of S. If S is nested inside of T, and the name x appears on the left-hand side of assignment statements in both S and T, then the x's are distinct: there is one in S and one in T. Nonlocal variables are read-only unless explicitly imported (using Python's `global` statement).

EXAMPLE 3.8

"Local if written" in Python

EXAMPLE 3.9

Declaration order in Scheme

In the interest of flexibility, modern Lisp dialects tend to provide several options for declaration order. In Scheme, for example, the `letrec` and `let*` constructs define scopes with, respectively, whole-block and declaration-to-end-of-block semantics. The most frequently used construct, `let`, provides yet another option:

```
(let ((A 1))          ; outer scope, with A defined to be 1
  (let ((A 2))        ; inner scope, with A defined to be 2
    (B A))           ;           and B defined to be A
  B))                ; return the value of B
```

Here the nested declarations of A and B don't take effect until after the end of the declaration list. Thus B is defined to be the *outer* A, and the code as a whole returns 1.

Declarations and Definitions

Given the requirement that names be declared before they can be used, languages like Pascal, C, and C++ require special mechanisms for recursive types and subroutines. Pascal handles the former by making pointers an exception to the rules and the latter by introducing so-called *forward declarations*. C and C++ handle both cases uniformly, by distinguishing between the *declaration* of an object and its *definition*. Informally, a declaration introduces a name and indicates its scope. A definition describes the thing to which the name is bound. If a declaration is not complete enough to be a definition, then a separate definition must appear elsewhere in the scope. In C we can write

EXAMPLE 3.10

Declarations v. definitions in C

```

    struct manager;           /* declaration only */
    struct employee {
        struct manager *boss;
        struct employee *next_employee;
        ...
    };
    struct manager {          /* definition */
        struct employee *first_employee;
        ...
    };

```

and

```

void list_tail(follow_set fs);      /* declaration only */
void list(follow_set fs)
{
    switch (input_token) {
        case id : match(id); list_tail(fs);
        ...
    }
    void list_tail(follow_set fs)      /* definition */
    {
        switch (input_token) {
            case comma : match(comma); list(fs);
            ...
        }
    }
}

```

Nested Blocks

In many languages, including Algol 60, C89, and Ada, local variables can be declared not only at the beginning of any subroutine, but also at the top of any

DESIGN & IMPLEMENTATION

Redeclarations

Some languages, particularly those that are intended for interactive use, permit the programmer to redeclare an object: to create a new binding for a given name in a given scope. Interactive programmers commonly use redeclarations to fix bugs. In most interactive languages, the new meaning of the name replaces the old in all contexts. In ML, however, the old meaning of the name may remain accessible to functions that were elaborated before the name was redeclared. This design choice in ML can sometimes be counterintuitive. It probably reflects the fact that ML is usually compiled, bit by bit on the fly, rather than interpreted. A language like Scheme, which is lexically scoped but usually interpreted, stores the binding for a name in a known location. A program accesses the meaning of the name indirectly through that location: if the meaning of the name changes, all accesses to the name will use the new meaning. In ML, previously elaborated functions have already been compiled into a form (often machine code) that accesses the meaning of the name directly.

`begin...end` (`{...}`) block. Other languages, including Algol 68, C99, and all of C's descendants, are even more flexible, allowing declarations wherever a statement may appear. In most languages a nested declaration hides any outer declaration with the same name (Java and C# make it a static semantic error if the outer declaration is local to the current subroutine).

EXAMPLE 3.11

Inner declarations in C

```
{
    int temp = a;
    a = b;
    b = temp;
}
```

Keeping the declaration of `temp` lexically adjacent to the code that uses it makes the program easier to read, and eliminates any possibility that this code will interfere with another variable named `temp`. ■

No run-time work is needed to allocate or deallocate space for variables declared in nested blocks; their space can be included in the total space for local variables allocated in the subroutine prologue and deallocated in the epilogue. Exercise 3.9 considers how to minimize the total space required.

CHECK YOUR UNDERSTANDING

12. What do we mean by the *scope* of a name-to-object binding?
13. Describe the difference between static and dynamic scope.
14. What is *elaboration*?
15. What is a *referencing environment*?
16. Explain the *closest nested scope rule*.
17. What is the purpose of a *scope resolution operator*?
18. What is a *static chain*? What is it used for?
19. What are *forward references*? Why are they prohibited or restricted in many programming languages?
20. Explain the difference between a *declaration* and a *definition*. Why is the distinction important?

3.3.4 Modules

A major challenge in the construction of any large body of software is how to divide the effort among programmers in such a way that work can proceed on multiple fronts simultaneously. This modularization of effort depends critically

```

/*
Place into *s a new name beginning with the letter l and
continuing with the ascii representation of an integer guaranteed
to be distinct in each separate call. s is assumed to point to
space large enough to hold any such name; for the short ints used
here, seven characters suffice. l is assumed to be an upper or
lower-case letter. sprintf 'prints' formatted output to a string.
*/
void gen_new_name(char *s, char l) {
    static short int name_nums[52];
    /* C guarantees that static local variables without explicit
       initial values are initialized as if explicitly set to zero. */
    int index = (l >= 'a' && l <= 'z') ? l-'a' : 26 + l-'A';
    name_nums[index]++;
    sprintf(s, "%c%d\0", 1, name_nums[index]);
}

```

Figure 3.6 C code to illustrate the use of static variables.

on the notion of *information hiding*, which makes objects and algorithms invisible, whenever possible, to portions of the system that do not need them. Properly modularized code reduces the “cognitive load” on the programmer by minimizing the amount of information required to understand any given portion of the system. In a well-designed program the interfaces between modules are as “narrow” (i.e., simple) as possible, and any design decision that is likely to change is hidden inside a single module. This latter point is crucial, since maintenance (bug fixes and enhancement) consumes many more programmer years than does initial construction for most commercial software.

In addition to reducing cognitive load, information hiding has several more pedestrian benefits. First, it reduces the risk of name conflicts: with fewer visible names, there is less chance that a newly introduced name will be the same as one already in use. Second, it safeguards the integrity of data abstractions: any attempt to access objects outside of the subroutine(s) to which they belong will cause the compiler to issue an “undefined symbol” error message. Third, it helps to compartmentalize run-time errors: if a variable takes on an unexpected value, we can generally be sure that the code that modified it is in the variable’s scope.

Unfortunately, the information hiding provided by nested subroutines is limited to objects whose lifetime is the same as that of the subroutine in which they are hidden. When control returns from a subroutine, its local variables will no longer be live: their values will be discarded. We have seen a partial solution to this problem in the form of the `save` statement in Fortran. A similar directive exists in several other languages: the `own` variables of Algol and the `static` variables of C, for example, retain their values from one invocation of a subroutine to the next.

EXAMPLE 3.12

Static variables in C

As an example of the use of static variables, consider the code in Figure 3.6. The subroutine `gen_new_name` can be used to generate a series of distinct

character-string names. A compiler could use these in its assembly language output. Labels, for example, might be named L1, L2, L3, and so on; subroutines could be named S1, S2, S3, and so on.

Static variables allow a subroutine like `gen_new_name` to have “memory”—to retain information from one invocation to the next—while protecting that memory from accidental access or modification by other parts of the program. Put another way, static variables allow programmers to build single-subroutine abstractions. Unfortunately, they do not allow the construction of abstractions whose interface needs to consist of more than one subroutine. Suppose, for example, that we wish to construct a *stack* abstraction. We should like to hide the representation of the stack—its internal structure—from the rest of the program, so that it can be accessed only through its push and pop routines. We can achieve this goal in many languages through use of a *module* construct.

A module allows a collection of objects—subroutines, variables, types, and so on—to be encapsulated in such a way that (1) objects inside are visible to each other, but (2) objects on the inside are not visible on the outside unless explicitly *exported*, and (3) (in many languages) objects outside are not visible on the inside unless explicitly *imported*. Modules can be found in Clu (which calls them *clusters*), Modula (1, 2, and 3), Turing, Ada (which calls them *packages*), C++ (which calls them *namespaces*), and many other modern languages. They can also be emulated to some degree through use of the separate compilation facilities of C; we discuss this possibility in Section 3.7.

EXAMPLE 3.13

Stack module in Modula-2

As an example of the use of modules, consider the stack abstraction shown in Figure 3.7. This stack can be embedded anywhere a subroutine might appear in a Modula-2 program. Bindings to variables declared in a module are inactive outside the module, not destroyed. In our stack example, `s` and `top` have the same lifetime they would have had if not enclosed in the module. If `stack` is declared at the program’s outermost nesting level, then `s` and `top` retain their values throughout the execution of the program, though they are visible only to the code inside `push` and `pop`. If `stack` is declared inside some subroutine `sub`, then `s` and `top` have the same lifetime as the local variables of `sub`. If `stack` is declared inside some other module `mod`, then `s` and `top` have the same lifetime as they would have had if not enclosed in either module. Type `stack_index`, which is also declared inside `stack`, is likewise visible only inside `push` and `pop`. The issue of lifetime is not relevant for types or constants, since they have no mutable state.

Our stack abstraction has two imports: the type (`element`) and maximum number (`stack_size`) of elements to be placed in the stack. `Element` and `stack_size` must be declared in a surrounding scope; the compiler will complain if they are not. With one exception, `element` and `stack_size` are the *only* names from surrounding scopes that will be visible inside `stack`. The exception is that predefined (*pervasive*) names, such as `integer` and `arctan`, are visible without being imported. Our stack also has two exports: `push` and `pop`. These are the only names inside of `stack` that will be visible in the surrounding scope.

```

CONST stack_size = ...
TYPE element = ...
...
MODULE stack;
IMPORT element, stack_size;
EXPORT push, pop;
TYPE
    stack_index = [1..stack_size];
VAR
    s : ARRAY stack_index OF element;
    top : stack_index;          (* first unused slot *)

PROCEDURE error; ...

PROCEDURE push(elem : element);
BEGIN
    IF top = stack_size THEN
        error;
    ELSE
        s[top] := elem;
        top := top + 1;
    END;
END push;

PROCEDURE pop() : element;      (* A Modula-2 function is just a *)
BEGIN                                (* procedure with a return type. *)
    IF top = 1 THEN
        error;
    ELSE
        top := top - 1;
        RETURN s[top];
    END;
END pop;

BEGIN
    top := 1;
END stack;
    VAR x, y : element;
    ...
    push(x);
    ...
    y := pop;

```

Figure 3.7 Stack abstraction in Modula-2.

Most module-based languages allow the programmer to specify that certain exported names are usable only in restricted ways. Variables may be exported read-only, for example, or types may be exported *opaquely*, meaning that variables of that type may be declared, passed as arguments to the module's subroutines, and possibly compared or assigned to one another, but not manipulated in any other way. To facilitate separate compilation, many module-based languages (Modula-2 among them) also allow a module to be divided into a declaration part (or *header*) and an implementation part (or *body*). Code that uses the ex-

ports of a given module can then be compiled as soon as the header exists; it is not dependent on the body.

Modules into which names must be explicitly imported are said to be *closed scopes*. Modules are closed in Modula (1, 2, and 3). By extension, modules that do not require imports are said to be *open scopes*. An increasingly common option, found in the modules of Ada, Java, C#, and Python, among others, might be called *selectively open scopes*. In these languages a name `foo` exported from module A is automatically visible in peer module B as `A.foo`. It becomes visible as merely `foo` if B explicitly imports it.

Nested subroutines are open scopes in most Algol family languages. Important exceptions are Euclid, in which both module and subroutine scopes are closed, Turing, Modula (1), and Perl, in which subroutines are optionally closed, and Clu, which outlaws the use of nonlocal variables entirely. A subroutine in Euclid must explicitly import any nonpervasive name that it uses from a surrounding scope. A subroutine in Turing or Modula can also import names explicitly; if it does so then no other nonlocal names are visible. Import lists serve to document the program: the use of names from surrounding scopes is really part of the interface between a subroutine and the rest of the program. Requiring explicit imports forces the programmer to document this interface more precisely than is required in other languages. Outlawing nonlocal variables serves a similar purpose in Clu, though nonlocal constants and subroutines can still be named, without explicit import.

In addition to making programs easier to understand and maintain, import lists help a Euclid or Turing compiler to enforce language rules that prohibit the creation of *aliases*—multiple names that refer to the same object in a given scope. Modula has no similar prohibition; its import lists are simply for documentation and information hiding. We will return to the subject of aliases in Section 3.6.1.

3.3.5 Module Types and Classes

Modules facilitate the construction of abstractions by allowing data to be made private to the subroutines that use them. As defined in Modula-2, Turing, or Ada 83, however, modules are most naturally suited to creating only a single instance of a given abstraction. The code in Figure 3.7, for example, does not lend itself to applications that require several stacks. For such an application, the programmer must either replicate the code (giving the new copy another name) or adopt an alternative organization in which the module becomes a “manager” for instances of a stack *type*, which is then exported (see Figure 3.8). This latter organization requires additional subroutines to create/initialize and possibly destroy stack instances, and it requires that every subroutine (`push`, `pop`, `create`) take an extra parameter, to specify the stack in question. Clu addresses this problem by automatically making *every* module (“cluster”) the manager for a type. In fact, the only variables that may appear in a cluster (other than static variables in subroutines) are the representation of that type. ■

EXAMPLE 3.14

Module as “manager” for a type

```

CONST stack_size = ...;
TYPE element = ...;
...
MODULE stack_manager;
IMPORT element, stack_size;
EXPORT stack, init_stack, push, pop;
TYPE
  stack_index = [1..stack_size];
  STACK = RECORD
    s : ARRAY stack_index OF element;
    top : stack_index;           (* first unused slot *)
  END;

PROCEDURE init_stack(VAR stk : stack);
BEGIN
  stk.top := 1;
END init_stack;

PROCEDURE push(VAR stk : stack; elem : element);
BEGIN
  IF stk.top = stack_size THEN
    error;
  ELSE
    stk.s[stk.top] := elem;
    stk.top := stk.top + 1;
  END;
END push;

PROCEDURE pop(VAR stk : stack) : element;
BEGIN
  IF stk.top = 1 THEN
    error;
  ELSE
    stk.top := stk.top - 1;
    return stk.s[stk.top];
  END;
END pop;

```

var A, B : stack;
var x, y : element;
...
init_stack(A);
init_stack(B);
...
push(A, x);
...
y := pop(B);

Figure 3.8 Manager module for stacks in Modula-2.

An alternative solution to the multiple instance problem can be found in Simula, Euclid, and (in a slightly different sense) ML, which treat modules as *types*, rather than simple encapsulation constructs. Given a module type, the programmer can declare an arbitrary number of similar module objects. The skeleton of a Euclid stack appears in Figure 3.9. As in the (single) Modula-2 stack of Figure 3.7, Euclid allows the programmer to provide initialization code that is executed whenever a new stack is created. Euclid also allows the programmer to

EXAMPLE 3.15

Module types in Euclid

```

const stack_size := ...
type element : ...
...
type stack = module
    imports (element, stack_size)
    exports (push, pop)
type
    stack_index = 1..stack_size
var
    s    : array stack_index of element
    top : stack_index

procedure push(elem : element) = ...
function pop returns element = ...
...
initially
    top := 1
end stack

```

```

var A, B : stack
var x, y : element
...
A.push(x)
...
y := B.pop

```

Figure 3.9 Module type for stacks in Euclid. Unlike the code in Figure 3.7, the code here can be used to create an arbitrary number of stacks.

specify finalization code that will be executed at the end of a module’s lifetime. This feature is not needed for an array-based stack, but it would be useful if elements were allocated from a heap and needed to be reclaimed. ■

The difference between the module-as-manager and module-as-type approaches to abstraction is reflected in the lower right of Figures 3.8 and 3.9. With module types, the programmer can think of the module’s subroutines as “belonging” to the stack in question (`A.push(x)`), rather than as outside entities to which the stack can be passed as an argument (`push(A, x)`). Conceptually, there is a separate pair of push and pop operations for every stack. In practice, of course, it would be highly wasteful to create multiple copies of the code. As we shall see in Chapter 9, all stacks share a single pair of push and pop operations, and the compiler arranges for a pointer to the relevant stack to be passed to the operation as an extra, hidden parameter. The implementation turns out to be very similar to the implementation of Figure 3.8, but the programmer need not think of it that way.⁸

As an extension of the module-as-type approach to data abstraction, many languages now provide a *class* construct for *object-oriented programming*. To first approximation, classes can be thought of as module types that have been augmented with an *inheritance* mechanism. Inheritance allows new classes to be defined as extensions or refinements of existing classes. Inheritance facilitates a pro-

8 It is interesting to note that Turing, which was derived from Euclid, reverts to Modula-2 style modules, in order to avoid implementation complexity [HMRC88, p. 9].

gramming style in which all or most operations are thought of as belonging to objects, and in which new objects can inherit most of their operations from existing objects, without the need to rewrite code. Classes have their roots in Simula-67, and are the central innovation of object-oriented languages such as Smalltalk, Eiffel, C++, Java, and C#. Inheritance mechanisms can also be found in several languages that are not usually considered object-oriented, including Modula-3, Ada 95, and Oberon. We will examine inheritance and its impact on scope rules in Chapter 9.

Module types and classes (ignoring issues related to inheritance) require only simple changes to the scope rules defined for modules in the previous subsection. Every instance A of a module type or class (e.g., every stack) has a separate copy of the module or class's variables. These variables are then visible when executing one of A's operations. They may also be indirectly visible to the operations of some other instance B if A is passed as a parameter to one of those operations. This rule makes it possible in most object-oriented languages to construct binary (or more-ary) operations that can manipulate the variables of more than one instance of a class. In C++, for example, we could create an operation that determines which of two stacks contains a larger number of elements:

```
class stack {  
    ...  
    bool deeper(stack other) {      // function declaration  
        return (top > other.top);  
    }  
    ...  
};  
...  
if (A.deeper(B)) ...
```

Within the `deeper` operation of stack A, `top` refers to `A.top`. Because `deeper` is an operation of class `stack`, however, it is able to refer not only to the variables of A (which it can access directly by name), but also to the variables of any *other* stack that is passed to it as an argument. Because these variables belong to a different stack, `deeper` must name that stack explicitly—for example, as in `other.top`. In a module-as-manager style program, of course, module subroutines would access all instance variables via parameters. ■

3.3.6 Dynamic Scope

In a language with dynamic scoping, the bindings between names and objects depend on the flow of control at run time and, in particular, on the order in which subroutines are called. In comparison to the static scope rules discussed in the previous section, dynamic scope rules are generally quite simple: the “current” binding for a given name is the one encountered most recently *during execution*, and not yet destroyed by returning from its scope.

```

1. a : integer           -- global declaration
2. procedure first
3.     a := 1
4. procedure second
5.     a : integer           -- local declaration
6.     first()
7. a := 2
8. if read_integer() > 0
9.     second()
10. else
11.     first()
12. write_integer(a)

```

Figure 3.10 Static versus dynamic scope. Program output depends on both scope rules and, in the case of dynamic scope, a value read at run time.

Languages with dynamic scoping include APL [Ive62], Snobol [GPP71], and early dialects of Lisp [MAE⁺65, Moo78, TM81] and Perl.⁹ Because the flow of control cannot in general be predicted in advance, the bindings between names and objects in a language with dynamic scope cannot in general be determined by a compiler. As a result, many semantic rules in a language with dynamic scope become a matter of dynamic semantics rather than static semantics. Type checking in expressions and argument checking in subroutine calls, for example, must in general be deferred until run time. To accommodate all these checks, languages with dynamic scoping tend to be interpreted rather than compiled.

EXAMPLE 3.17

Static v. dynamic scope

As an example of dynamic scope, consider the program in Figure 3.10. If static scoping is in effect, this program prints a 1. If dynamic scoping is in effect, the program prints either a 1 or a 2, depending on the value read at line 8 at run time. Why the difference? At issue is whether the assignment to the variable *a* at line 3 refers to the global variable declared at line 1 or to the local variable declared at line 5. Static scope rules require that the reference resolve to the closest lexically enclosing declaration—namely the global *a*. Procedure *first* changes *a* to 1, and line 12 prints this value.

Dynamic scope rules, on the other hand, require that we choose the most recent, active binding for *a* at run time. We create a binding for *a* when we enter the main program. We create another when and if we enter procedure *second*. When we execute the assignment statement at line 3, the *a* to which we are referring will depend on whether we entered *first* through *second* or directly from

⁹ Scheme and Common Lisp are statically scoped, though the latter allows the programmer to specify dynamic scoping for individual variables. Static scoping was added to Perl in version 5. The programmer now chooses static or dynamic scoping explicitly in each variable declaration.

```

max_score : integer      -- maximum possible score

function scaled_score(raw_score : integer) : real
    return raw_score / max_score * 100
    ...

procedure foo
    max_score : real := 0      -- highest percentage seen so far
    ...
    foreach student in class
        student.percent := scaled_score(student.points)
        if student.percent > max_score
            max_score := student.percent

```

Figure 3.11 The problem with dynamic scoping. Procedure scaled_score probably does not do what the programmer intended when dynamic scope rules allow procedure foo to change the meaning of max_score.

the main program. If we entered through second, we will assign the value 1 to second's local a. If we entered from the main program, we will assign the value 1 to the global a. In either case, the write at line 12 will refer to the global a, since second's local a will be destroyed, along with its binding, when control returns to the main program.

With dynamic scoping in effect, no program fragment that makes use of non-local names is guaranteed a predictable referencing environment. In Figure 3.11, for example, the declaration of a local variable in procedure foo accidentally redefines a global variable used by function scaled_score, which is then called from foo. Since the global max_score is an integer, while the local max_score is a floating-point number, dynamic semantic checks in at least some languages will result in a type clash message at run time. If the local max_score had been an integer, no error would have been detected, but the program would almost certainly have produced incorrect results. This sort of error can be very hard to find. ■

DESIGN & IMPLEMENTATION

Dynamic scoping

It is not entirely clear whether the use of dynamic scoping in Lisp and other early interpreted languages was deliberate or accidental. One reason to think that it may have been deliberate is that it makes it very easy for an interpreter to look up the meaning of a name: all that is required is a stack of declarations (we examine this stack more closely in Section ⑩ 3.4.2). Unfortunately, this simple implementation has a very high run-time cost, and experience indicates that dynamic scoping makes programs harder to understand. The modern consensus seems to be that dynamic scoping is usually a bad idea (see Exercise 3.15 and Exploration 3.29 for two exceptions).

EXAMPLE 3.18

Customization via dynamic scope

The principal argument in *favor* of dynamic scoping is that it facilitates the customization of subroutines. Suppose, for example, that we have a library routine `print_integer` that is capable of printing its argument in any of several bases (decimal, binary, hexadecimal, etc.). Suppose further that we want the routine to use decimal notation most of the time, and to use other bases only in a few special cases; we do not want to have to specify a base explicitly on each individual call. We can achieve this result with dynamic scoping by having `print_integer` obtain its base from a nonlocal variable `print_base`. We can establish the default behavior by declaring a variable `print_base` and setting its value to 10 in a scope encountered early in execution. Then, any time we want to change the base temporarily, we can write

```
begin      -- nested block
  print_base : integer := 16      -- use hexadecimal
  print_integer(n)
```

EXAMPLE 3.19

Multiple interface alternative

The problem with this argument is that there are usually other ways to achieve the same effect, without dynamic scoping. One option would be to have `print_integer` use decimal notation in all cases, and create another routine, `print_integer_with_base`, that takes a second argument. In a language like Ada or C++, one could make the base an optional (default) parameter of a single `print_integer` routine, or use overloading to give the same name to both routines. (We will consider default parameters in Section 8.3.3; overloading is discussed in Section 3.6.2.)

Unfortunately, using two different routines for printing (or one routine with two calling sequences) requires that the caller know what is going on. In our example, alternative routines work fine if the calls are all made in the scope in which the local `print_base` variable would have been declared. If that scope calls subroutines that in turn call `print_integer`, however, we cannot in general arrange for the called routines to use the alternative interface. A second alternative to dynamic scoping solves this problem: we can create a static variable, either global or encapsulated with `print_integer` inside an appropriate module, that controls the base. To change the print base temporarily, we can then write

```
begin      -- nested block
  print_base_save : integer := print_base
  print_base := 16      -- use hexadecimal
  print_integer(n)
  print_base := print_base_save
```

The possibility that we may forget to restore the original value, of course, is a potential source of bugs. With dynamic scoping the value is restored automatically.

3.4 Implementing Scope

To keep track of the names in a statically scoped program, a compiler relies on a data abstraction called a *symbol table*. In essence, the symbol table is a dictionary: it maps names to the information the compiler knows about them. The most basic operations serve to place a new mapping (a name-to-object binding) into the table and to retrieve (nondestructively) the information held in the mapping for a given name. Static scope rules in most languages impose additional complexity by requiring that the referencing environment be different in different parts of the program.

In a language with dynamic scoping, an interpreter (or the output of a compiler) must perform operations at run time that correspond to the insert, lookup, enter_scope, and leave_scope symbol table operations in the implementation of a statically scoped language. In principle, any organization used for a symbol table in a compiler could be used to track name-to-object bindings in an interpreter, and vice versa. In practice, implementations of dynamic scoping tend to adopt one of two specific organizations: an *association list* or a *central reference table*.

IN MORE DEPTH

Most variations on static scoping can be handled by augmenting a basic dictionary-style symbol table with enter_scope and leave_scope operations to keep track of visibility. Nothing is ever deleted from the table; the entire structure is retained throughout compilation, and then saved for the debugger. A symbol table with visibility support can be implemented in several different ways. One appealing approach, due to LeBlanc and Cook [CL83], is described on the PLP CD.

An association list (or *A-list* for short) is simply a list of name/value pairs. When used to implement dynamic scope it functions as a stack: new declarations are pushed as they are encountered, and popped at the end of the scope in which they appeared. Bindings are found by searching down the list from the top. A central reference table avoids the need for linear-time search by maintaining an explicit mapping from names to their current meanings. Lookup is faster, but scope entry and exit are somewhat more complex, and it becomes substantially more difficult to save a referencing environment for future use (we discuss this issue further in Section 3.5.1).

CHECK YOUR UNDERSTANDING

21. Explain the importance of information hiding.
22. What is an *opaque* export?

23. Why might it be useful to distinguish between the *header* and the *body* of a module?
 24. What does it mean for a scope to be *closed*?
 25. Explain the distinction between “modules as managers” and “modules as types.”
 26. How do classes differ from modules?
 27. Why does the use of dynamic scoping imply the need for run-time type checking?
 28. Give an argument in favor of dynamic scoping. Describe how similar benefits can be achieved in a language without dynamic scoping.
 29. Explain the purpose of a compiler’s symbol table.
-

3.5

The Binding of Referencing Environments

We have seen in the previous section how scope rules determine the referencing environment of a given statement in a program. Static scope rules specify that the referencing environment depends on the lexical nesting of program blocks in which names are declared. Dynamic scope rules specify that the referencing environment depends on the order in which declarations are encountered at run time. An additional issue that we have not yet considered arises in languages that allow one to create a *reference* to a subroutine—for example, by passing it as a parameter. When should scope rules be applied to such a subroutine: when the reference is first created, or when the routine is finally called? The answer is particularly important for languages with dynamic scoping, though we shall see that it matters even in languages with static scoping. As an example of the former, consider the program fragment shown in Figure 3.12. (As in Figure 3.10, we use an Algol-like syntax, even though Algol-family languages are usually statically scoped.)

EXAMPLE 3.21

Deep and shallow binding

Procedure print_selected_records in our example is assumed to be a general purpose routine that knows how to traverse the records in a database, regardless of whether they represent people, sprockets, or salads. It takes as parameters a database, a predicate to make print/don’t print decisions, and a subroutine that knows how to format the data in the records of this particular database. In Section 3.3.6 we hypothesized a print_integer library routine that would print in any of several bases, depending on the value of a nonlocal variable print_base. Here we have hypothesized in a similar fashion that print_person uses the value of nonlocal variable line_length to calculate the number and width of columns in its output. In a language with dynamic scope, it is natural for procedure print_selected_records to declare and initialize this variable locally, knowing that code inside print_routine will pick it up if needed. For this coding technique to work,

```

type person = record
  ...
    age : integer
  ...
threshold : integer
people : database

function older_than(p : person) : boolean
  return p.age ≥ threshold

procedure print_person(p : person)
  -- Call appropriate I/O routines to print record on standard output.
  -- Make use of nonlocal variable line_length to format data in columns.
  ...

procedure print_selected_records(db : database;
  predicate, print_routine : procedure)
line_length : integer

if device_type(stdout) = terminal
  line_length := 80
else      -- Standard output is a file or printer.
  line_length := 132
foreach record r in db
  -- Iterating over these may actually be
  -- a lot more complicated than a 'for' loop.
  if predicate(r)
    print_routine(r)

-- main program
...
threshold := 35
print_selected_records(people, older_than, print_person)

```

Figure 3.12 Program to illustrate the importance of binding rules. One might argue that deep binding is appropriate for the environment of function older_than (for access to threshold), while shallow binding is appropriate for the environment of procedure print_person (for access to line_length).

the referencing environment of print_routine must not be created until the routine is actually called by print_selected_records. This late binding of the referencing environment of a subroutine that has been passed as a parameter is known as *shallow binding*. It is usually the default in languages with dynamic scoping.

For function older_than, by contrast, shallow binding may not work well. If, for example, procedure print_selected_records happens to have a local variable named threshold, then the variable set by the main program to influence the behavior of older_than will not be visible when the function is finally called, and

the predicate will be unlikely to work correctly. In such a situation, the code that originally passes the function as a parameter has a particular referencing environment (the current one) in mind; it does not want the routine to be called in any other environment. It therefore makes sense to bind the environment at the time the routine is first passed as a parameter, and then restore that environment when the routine is finally called. This early binding of the referencing environment is known as *deep binding*. The need for deep binding is sometimes referred to as the *funarg problem* in Lisp. ■

3.5.1 Subroutine Closures

Deep binding is implemented by creating an explicit representation of a referencing environment (generally the one in which the subroutine would execute if called at the present time) and bundling it together with a reference to the subroutine. The bundle as a whole is referred to as a *closure*. Usually the subroutine itself can be represented in the closure by a pointer to its code. If an association list is used to represent the referencing environment of a program with dynamic scoping, then the referencing environment in a closure can be represented by a top-of-stack (beginning of A-list) pointer. When a subroutine is called through a closure, the main pointer to the referencing environment A-list is temporarily replaced by the saved pointer, making any bindings created since the closure was created temporarily invisible. New bindings created *within* the subroutine are pushed using the temporary pointer. Because the A-list is represented by pointers (rather than an array), the effect is to have two lists—one representing the temporary referencing environment resulting from use of the closure and the other the main referencing environment that will be restored when the subroutine returns—that share their older entries.

If a central reference table is used to represent the referencing environment of a program with dynamic scoping, then the creation of a closure is more complicated. In the general case, it may be necessary to copy the entire main array of the central table and the first entry on each of its lists. Space and time overhead may be reduced if the compiler or interpreter is able to determine that only some of the program's names will be used by the subroutine in the closure (or by things that the subroutine may call). In this case, the environment can be saved by copying the first entries of the lists for only the "interesting" names. When the subroutine is called through the closure, these entries can then be pushed onto the beginnings of the appropriate lists in the central reference table.

Deep binding is often available as an option in languages with dynamic scope. In early dialects of Lisp, for example, the built-in primitive function takes a function as its argument and returns a closure whose referencing environment is the one in which the function would execute if called at the present time. This closure can then be passed as a parameter to another function. If and when it is eventually called, it will execute in the saved environment. (Closures work slightly

```

program binding_example(input, output);

procedure A(I : integer; procedure P);

procedure B;
begin
    writeln(I);
end;

begin (* A *)
    if I > 1 then
        P
    else
        A(2, B);
end;

procedure C; begin end;

begin (* main *)
    A(1, C);
end.

```

Figure 3.13 Deep binding in Pascal. When `B` is called via formal parameter `P`, two instances of `I` exist. Because the closure for `P` was created in the initial invocation of `A`, it uses that invocation's instance of `I`, and prints a 1.

differently from “bare” functions in most Lisp dialects: they must be called by passing them to the built-in primitives `funcall` or `apply`.)

Deep binding is generally the default in languages with static (lexical) scoping. At first glance, one might be tempted to think that the binding time of referencing environments would not matter in languages with static scoping. After all, the meaning of a statically scoped name depends on its lexical nesting, not on the flow of execution, and this nesting is the same whether it is captured at the time a subroutine is passed as a parameter or at the time the subroutine is called. The catch is that a running program may have more than one *instance* of an object that is declared within a recursive subroutine. A closure in a language with static scoping captures the current instance of every object, at the time the closure is created. When the closure's subroutine is called, it will find these captured instances, even if newer instances have subsequently been created by recursive calls.

One could imagine combining static scoping with shallow binding [VF82], but the combination does not seem to make much sense, and it does not appear to have been adopted in any language. Figure 3.13 contains a Pascal program that illustrates the impact of binding rules in the presence of static scoping. This program prints a 1. With shallow binding it would print a 2. ■

EXAMPLE 3.22

Binding rules with static
scoping

It should be noted that binding rules matter with static scoping only when accessing objects that are neither local nor global. If an object is local to the currently executing subroutine, then it does not matter whether the subroutine was called directly or through a closure; in either case local objects will have been created when the subroutine started running. If an object is global, there will never be more than one instance, since the main body of the program is not recursive. Binding rules are therefore irrelevant in languages like C, which has no nested subroutines, or Modula-2, which allows only outermost subroutines to be passed as parameters. (They are also irrelevant in languages like PL/I and Ada 83, which do not permit subroutines to be passed as parameters at all.)

Suppose then that we have a language with static scoping in which nested subroutines can be passed as parameters, with deep binding. To represent a closure for subroutine S, we can simply save a pointer to S's code together with the static link that S would use if it were called right now, in the current environment. When S is finally called, we temporarily restore the saved static link, rather than creating a new one. When S follows its static chain to access a nonlocal object, it will find the object instance that was current at the time the closure was created.

3.5.2 First- and Second-Class Subroutines

In general, a value in a programming language is said to have *first-class* status if it can be passed as a parameter, returned from a subroutine, or assigned into a variable. Simple types such as integers and characters are first-class values in most programming languages. By contrast, a “second-class” value can be passed as a parameter, but not returned from a subroutine or assigned into a variable, and a “third-class” value cannot even be passed as a parameter. As we shall see in Section 8.3.2, labels are third-class values in most programming languages but second-class values in Algol. Subroutines are second-class values in most imperative languages but third-class values in Ada 83. They are first-class values in all functional programming languages, in C#, Perl, and Python, and, with certain restrictions, in several other imperative languages, including Fortran, Modula-2 and -3, Ada 95, C, and C++.¹⁰

So far in this subsection we have considered the ramifications of second-class subroutines. First-class subroutines in a language with nested scopes introduce an additional level of complexity: they raise the possibility that a reference to a subroutine may outlive the execution of the scope in which that routine was declared. Consider the following example in Scheme.

EXAMPLE 3.23

Returning a first-class subroutine in Scheme

10 Some authors would say that first-class status requires the ability to create new functions at run time. C#, Perl, Python, and all functional languages meet this requirement, but most imperative languages do not.

```

1. (define plus_x (lambda (x)
2.   (lambda (y) (+ x y))))
3. ...
4. (let ((f (plus_x 2)))
5.   (f 3)) ; returns 5

```

Here the `let` construct on line 4 declares a new function, `f`, which is the result of calling `plus_x` with argument 2. (Like all Lisp dialects, Scheme puts the function name *inside* the parentheses, right in front of the arguments. The `lambda` keyword introduces the parameter list and body of a function.) When `f` is called at line 5, it must use the 2 that was passed to `plus_x`, despite the fact that `plus_x` has already returned. ■

If local objects were destroyed (and their space reclaimed) at the end of each scope's execution, then the referencing environment captured in a long-lived closure might become full of dangling references. To avoid this problem, most functional languages specify that local objects have *unlimited extent*: their lifetimes continue indefinitely. Their space can be reclaimed only when the garbage collection system is able to prove that they will never be used again. Local objects (other than `own/static` variables) in Algol-family languages generally have *limited extent*: they are destroyed at the end of their scope's execution. Space for local objects with limited extent can be allocated on a stack. Space for local objects with unlimited extent must generally be allocated on a heap.

Given the desire to maintain stack-based allocation for the local variables of subroutines, imperative languages with first-class subroutines must generally adopt alternative mechanisms to avoid the dangling reference problem for closures. C, C++, and Fortran, of course, do not have nested subroutines. Modula-2 allows references to be created only to outermost subroutines (outermost routines are first-class values; nested routines are third-class values). Modula-3 allows nested subroutines to be passed as parameters, but only outermost routines to be returned or stored in variables (outermost routines are first-class values; nested routines are second-class values). Ada 95 allows a nested routine to be returned, but only if the scope in which it was declared is at least as wide as that of the declared return type. This containment rule, while more conservative than strictly necessary (it forbids the Ada equivalent of Figure 3.13), makes it impossi-

DESIGN & IMPLEMENTATION

Binding rules and extent

Binding mechanisms and the notion of extent are closely tied to implementation issues. A-lists make it easy to build closures, but so do the non-nested subroutines of C and the rule against passing non-global subroutines as parameters in Modula-2. In a similar vein, the lack of first-class subroutines in most imperative languages reflects in large part the desire to avoid heap allocation, which would be needed for local variables with unlimited extent.

ble to propagate a subroutine reference to a portion of the program in which the routine's referencing environment is not active.

3.6 Binding Within a Scope

So far in our discussion of naming and scopes we have assumed that every name must refer to a distinct object in every scope. This is not necessarily the case. Two or more names that refer to a single object in a given scope are said to be *aliases*. A name that can refer to more than one object in a given scope is said to be *overloaded*.

3.6.1 Aliases

EXAMPLE 3.24

Aliasing with parameters

Simple examples of aliases occur in the common blocks and equivalence statements of Fortran (Section 3.3.1) and in the variant records and unions of languages like Pascal and C#. They also arise naturally in programs that make use of pointer-based data structures. A more subtle way to create aliases in many languages is to pass a variable by reference to a subroutine that also accesses that variable directly (consider variable `sum` in Figure 3.14). As we noted in Section 3.3.4, Euclid and Turing use explicit and implicit subroutine import lists to catch and prohibit precisely this case. ■

As a general rule, aliases tend to make programs more confusing than they otherwise would be. They also make it much more difficult for a compiler to perform certain important code improvements. Consider the following C code.

EXAMPLE 3.25

Aliases and code improvement

DESIGN & IMPLEMENTATION

Pointers in C and Fortran

The tendency of pointers to introduce aliases is one of the reasons why Fortran compilers have tended, historically, to produce faster code than C compilers: pointers are heavily used in C but missing from Fortran 77 and its predecessors. It is only in recent years that sophisticated alias analysis algorithms have allowed C compilers to rival their Fortran counterparts in speed of generated code. Pointer analysis is sufficiently important that the designers of the C99 standard decided to add a new keyword to the language. The `restrict` qualifier, when attached to a pointer declaration, is an assertion on the part of the programmer that the object to which the pointer refers has no alias in the current scope. It is the programmer's responsibility to ensure that the assertion is correct; the compiler need not attempt to check it.

```

double sum, sum_of_squares;
...
void accumulate(double& x)      // x passed by reference
{
    sum += x;
    sum_of_squares += x * x;
}
...
accumulate(sum);

```

Figure 3.14 Example of a potentially problematic alias in C++. Procedure `accumulate` probably does not do what the programmer intended when `sum` is passed as a parameter.

```

int a, b, *p, *q;
...
a = *p;      /* read from the variable referred to by p */
*q = 3;      /* assign to the variable referred to by q */
b = *p;      /* read from the variable referred to by p */

```

The initial assignment to `a` will, on most machines, require that `*p` be loaded into a register. Since accessing memory is expensive, the compiler will want to hang onto the loaded value and reuse it in the assignment to `b`. It will be unable to do so, however, unless it can verify that `p` and `q` cannot refer to the same object. While verification of this sort is possible in many common cases, in general it's uncomputable. ■

3.6.2 Overloading

Most programming languages provide at least a limited form of overloading. In C, for example, the plus sign (+) is used to name two different functions: integer and floating-point addition. Most programmers don't worry about the distinction between these two functions—both are based on the same mathematical concept, after all—but they take arguments of different types and perform very different operations on the underlying bits. A slightly more sophisticated form of overloading appears in the enumeration constants of Ada. In Figure 3.15, the constants `oct` and `dec` refer either to months or to numeric bases, depending on the context in which they appear. ■

Within the symbol table of a compiler, overloading must be handled by arranging for the *lookup* routine to return a *list* of possible meanings for the requested name. The semantic analyzer must then choose from among the elements of the list based on context. When the context is not sufficient to decide, as in the call to `print` in Figure 3.15, then the semantic analyzer must announce an error. Most languages that allow overloaded enumeration constants allow the programmer to provide appropriate context explicitly. In Ada, for example, one can say

EXAMPLE 3.26

Overloaded enumeration constants in Ada

EXAMPLE 3.27

Resolving ambiguous overloads

```

declare
    type month is (jan, feb, mar, apr, may, jun,
                   jul, aug, sep, oct, nov, dec);
    type print_base is (dec, bin, oct, hex);
    mo : month;
    pb : print_base;
begin
    mo := dec;      -- the month dec
    pb := oct;      -- the print_base oct
    print(oct);     -- error! insufficient context to decide

```

Figure 3.15 Overloading of enumeration constants in Ada.

```
print(month'(oct));
```

In Modula-3, and C#, *every* use of an enumeration constant must be prefixed with a type name, even when there is no chance of ambiguity:

```
mo := month.dec;
pb := print_base.oct;
```

In C, C++, and standard Pascal, one cannot overload enumeration constants at all; every constant visible in a given scope must be distinct.

Both Ada and C++ have elaborate facilities for overloading subroutine names. (Most of the C++ facilities carry over to Java and C#.) A given name may refer to an arbitrary number of subroutines in the same scope, so long as the subroutines differ in the number or types of their arguments. C++ examples appear in Figure 3.16.¹¹

EXAMPLE 3.28

Overloading in Ada and C++

EXAMPLE 3.29

Overloading built-in operators

Ada, C++, C#, and Fortran 90 also allow the built-in arithmetic operators (+, -, *, etc.) to be overloaded with user-defined functions. Ada, C++, and C# do this by defining alternative *prefix* forms of each operator, and defining the usual infix forms to be abbreviations (or “syntactic sugar”) for the prefix forms. In Ada, A + B is short for "+"(A, B). If "+" is overloaded, it must be possible to determine the intended meaning from the types of A and B. In C++ and C#, A + B is short for A.operator+(B), where A is an instance of a class (module type) that defines an operator+ function. The class-based style of abbreviation in C++ and C# resembles a similar facility in Clu. Since the abbreviation expands to an unambiguous name (i.e., A’s operator+; not any other), one might be tempted to say that no “real” overloading is involved, and this is in fact the case in Clu. In C++ and C#, however, there may be more than one definition of A.operator+, allowing the second argument to be of several types. Fortran 90 provides a special interface construct that can be used to associate an operator with some named binary function.

11 C++ actually provides more elegant ways to handle both I/O and user-defined types such as complex. We examine these in Section 7.9 and Chapter 9.

```

struct complex {
    double real, imaginary;
};

enum base {dec, bin, oct, hex};

int i;
complex x;

void print_num(int n) ...
void print_num(int n, base b) ...
void print_num(complex c) ...

print_num(i);           // uses the first function above
print_num(i, hex);     // uses the second function above
print_num(x);          // uses the third function above

```

Figure 3.16 Simple example of overloading in C++. In each case the compiler can tell which function is intended by the number and types of arguments.

3.6.3 Polymorphism and Related Concepts

In the case of subroutine names, it is worth distinguishing overloading from the closely related concepts of coercion and polymorphism. All three can be used, in certain circumstances, to pass arguments of multiple types to (or return values of multiple types from) a given named routine. The syntactic similarity, however, hides significant differences in semantics and pragmatics.

Suppose, for example, that we wish to be able to compute the minimum of two values of either integer or floating-point type. In Ada we might obtain this capability using overloaded functions:

```

function min(a, b : integer) return integer is ...
function min(x, y : real) return real is ...

```

In Fortran, however, we could get by with a single function:

```

real function min(x, y)
real x, y
...

```

If the Fortran function is called in a context that expects an integer (e.g., `i = min(j, k)`), the compiler will automatically convert the integer arguments (`j` and `k`) to floating-point numbers, call `min`, and then convert the result back to an integer (via truncation). So long as `real` variables have at least as many significant bits as `integers` (which they do in the case of 32-bit integers and 64-bit double-precision floating-point), the result will be numerically correct. ■

Coercion is the process by which a compiler automatically converts a value of one type into a value of another type when that second type is required by the surrounding context. As we shall see in Section 7.2.2, coercion is somewhat controversial. Pascal provides a limited number of coercions. Fortran and C provide

EXAMPLE 3.30

Overloading v. coercion

more. C++ provides an extremely rich set, and allows the programmer to define more. Ada as a matter of principle coerces nothing but explicit constants, subranges, and in certain cases arrays with the same type of elements.

In our example, overloading allows the Ada compiler to choose between two different versions of `min`, depending on the types of the arguments. Coercion allows the Fortran compiler to modify the arguments to fit a *single* subroutine. *Polymorphism* provides yet another option: it allows a single subroutine to accept *unconverted* arguments of multiple types.

The term *polymorphic* is from the Greek, meaning “having multiple forms.” It is applied to code—both data structures and subroutines—that can work with values of multiple types. For this concept to make sense, the types must generally have certain characteristics in common, and the code must not depend on any other characteristics. The commonality is usually captured in one of two main ways. In *parametric polymorphism* the code takes a type (or set of types) as a parameter, either explicitly or implicitly. In *subtype polymorphism* the code is designed to work with values of some specific type T , but the programmer can define additional types to be extensions or refinements of T , and the polymorphic code will work with these *subtypes* as well.

Explicit parametric polymorphism is also known as *genericity*. Generic facilities appear in Ada, C++, Clu, Eiffel, Modula-3, and recent versions of Java and C#, among others. Readers familiar with C++ will know them by the name of *templates*. We will consider them further in Sections 8.4 and 9.4.4. Implicit parametric polymorphism appears in the Lisp and ML families of languages, and in various scripting languages; we will consider it further in Sections ⑩ 7.2.4 and 10.3. Subtype polymorphism is fundamental to object-oriented languages, in which subtypes (classes) are said to *inherit* the methods of their parent types. We will consider inheritance further in Section 9.4.

Generics (explicit parametric polymorphism) are usually, though not always, implemented by creating multiple copies of the polymorphic code, one specialized for each needed concrete type. Inheritance (subtype polymorphism) is almost always implemented by creating a single copy of the code, and by inserting sufficient “metadata” in the representation of objects that the code can tell when to treat them differently. Implicit parametric polymorphism can be imple-

DESIGN & IMPLEMENTATION

Coercion and overloading

In addition to their semantic differences, coercion and overloading can have very different costs. Calling an integer-specific version of `min` would be much more efficient than calling the floating-point version with integer arguments: it would use integer arithmetic for the comparison (which is cheaper in and of itself) and would avoid four conversion operations. One of the arguments against supporting coercion in a language is that it tends to impose hidden costs.

```

generic
    type T is private;
        with function "<"(x, y : T) return Boolean;
function min(x, y : T) return T;

function min(x, y : T) return T is
begin
    if x < y then return x;
    else return y;
    end if;
end min;

function string_min is new min(string, "<");
function date_min is new min(date, date_precedes);

```

Figure 3.17 Use of a generic subroutine in Ada.

mented either way. Most Lisp implementations use a single copy of the code, and delay all semantic checks until run time. ML and its descendants perform all type checking at compile time. They typically generate a single copy of the code where possible (e.g., when all the types in question are records that share a similar representation) and generate multiple copies when necessary (e.g., when polymorphic arithmetic must operate on both integer and floating-point numbers). Object-oriented languages that perform type checking at compile time, including C++, Eiffel, Java, and C#, generally provide both generics *and* inheritance. Smalltalk (Section ⑩ 9.6.1), Objective-C, Python, and Ruby use a single mechanism (with run-time checking) to provide both parametric and subtype polymorphism.

EXAMPLE 3.31

Generic `min` function in Ada

As a concrete example of generics, consider the overloaded `min` functions of Example 3.30. The code for the integer and floating-point versions is likely to be very similar. We can exploit this similarity to define a single version that works not only for integers and reals, but for any type whose values are totally ordered. This code appears in Figure 3.17. The initial (bodyless) declaration of `min` is preceded by a `generic` clause specifying that two things are required in order to create a concrete instance of a minimum function: a type, `T`, and a correspond-

DESIGN & IMPLEMENTATION

Generics as macros

In some sense, the local stack module of Figure 3.7 (page 127) is a primitive sort of generic module. Because it imports the `element` type and `stack_size` constant, it can be inserted (with a text editor) into any context in which these names are declared, and will produce a “customized” stack for that context when compiled. Early versions of C++ formalized this mechanism by using macros to implement templates. Later versions of C++ have made templates (generics) a fully supported language feature.

ing comparison routine. This declaration is followed by the actual code for `min`. Given appropriate declarations of `string` and `date` types (not shown), we can create functions to return the lesser of pairs of objects of these types as shown in the last two lines. (The "`<`" operation mentioned in the definition of `string_min` is presumably overloaded; the compiler resolves the overloading by finding the version of "`<`" that takes arguments of type `T`, where `T` is already known to be `string`.) ■

EXAMPLE 3.32

Implicit polymorphism in Scheme

With the *implicit* parametric polymorphism of Lisp, ML, and their descendants, the programmer need not specify a type parameter. The Scheme definition of `min` looks like this:

```
(define min (lambda (a b) (if (< a b) a b)))
```

It makes no mention of types. The typical Scheme implementation employs an interpreter that examines the arguments to `min` and determines, at run time, whether they support a `<` operator. Given the preceding definition, the expression `(min 123 456)` evaluates to 123; `(min 3.14159 2.71828)` evaluates to 2.71828. The expression `(min "abc" "def")` produces a run-time error when evaluated, because the string comparison operator is named `string<?`, not `<`. ■

EXAMPLE 3.33

Implicit polymorphism in Haskell

The Haskell version of `min` is even simpler and more general:

```
min a b = if a < b then a else b
```

This version works for values of any totally ordered type, including strings. It is type-checked at compile time, using a sophisticated system of *type inference* (to be described in Section ⑦.2.4). ■

So what exactly is the difference between the overloaded `min` functions of Example 3.30 and the generic version of Figure 3.17? The answer lies in the generality of the code. With overloading the programmer must write a separate copy of the code, by hand, for every type with a `min` operation. Generics allow the compiler (in the typical implementation) to create a copy automatically for every needed type. The similarity of the calling syntax and of the generated code has led some authors to refer to overloading as *ad hoc (special case) polymorphism*. There is no particular reason, however, for the programmer to think of generics in terms of multiple copies: from a semantic (conceptual) point of view, overloaded subroutines use a single name for more than one thing; a polymorphic subroutine *is* a single thing.

CHECK YOUR UNDERSTANDING

30. Describe the difference between deep and shallow *binding* of referencing environments.
31. Why are binding rules particularly important for languages with dynamic scoping?
32. What is a *closure*? What is it used for? How is it implemented?
33. What are *first-class* subroutines? What languages support them?

34. Explain the distinction between limited and unlimited *extent* of objects in a local scope.
 35. What are *aliases*? Why are they considered a problem in language design and implementation?
 36. Explain the value of the `restrict` qualifier in C99.
 37. Explain the differences between *overloading*, *coercion*, and *polymorphism*.
 38. Define *parametric* and *subtype* polymorphism. Explain the distinction between *explicit* and *implicit* parametric polymorphism. Which is also known as *genericity*?
 39. Why is overloading sometimes referred to as *ad hoc* polymorphism?
-

3.7 Separate Compilation

Since most large programs are constructed and tested incrementally, and since the compilation of a very large program can be a multihour operation, any language designed to support large programs must provide a separate compilation facility.

IN MORE DEPTH

Because they are designed for encapsulation and provide a narrow interface, modules are the natural choice for the “compilation units” of many programming languages. The separate module headers and bodies of Modula-3 and Ada, for example, are explicitly intended for separate compilation, and reflect experience gained with more primitive facilities in other languages. C and C++, by contrast, must maintain backward compatibility with mechanisms designed in the early 1970s. C++ includes a *namespace* mechanism that provides module-like data hiding, but names must still be declared before they are used in every compilation unit, and the mechanisms used to accommodate this rule are purely a matter of convention. Java and C# break with the C tradition by requiring the compiler to infer header information automatically from separately compiled class definitions; no header files are required.

3.8 Summary and Concluding Remarks

This chapter has addressed the subject of names, and the *binding* of names to objects (in a broad sense of the word). We began with a general discussion of the notion of *binding time*: the time at which a name is associated with a particular object or, more generally, the time at which an answer is associated with any open

question in language or program design or implementation. We defined the notion of *lifetime* for both objects and name-to-object bindings, and noted that they need not be the same. We then introduced the three principal storage allocation mechanisms—static, stack, and heap—used to manage space for objects.

In Section 3.3 we described how the binding of names to objects is governed by *scope rules*. In some languages, scope rules are dynamic: the meaning of a name is found in the most recently entered scope that contains a declaration and that has not yet been exited. In most modern languages, however, scope rules are static, or *lexical*: the meaning of a name is found in the closest lexically surrounding scope that contains a declaration. We found that lexical scope rules vary in important but sometimes subtle ways from one language to another. We considered what sorts of scopes are allowed to nest, whether scopes are *open* or *closed*, whether the scope of a name encompasses the entire block in which it is declared, and whether a name must be declared before it is used. We explored the implementation of scope rules in Section 3.4. In Section 3.5 we considered the question of when to bind a referencing environment to a subroutine that is passed as a parameter, returned from a function, or stored in a variable.

Some of the more complicated aspects of lexical scoping illustrate the evolution of language support for data abstraction, a subject to which we will return in Chapter 9. We began by describing the *own* or *static* variables of languages like Fortran, Algol 60, and C, which allow a variable that is local to a subroutine to retain its value from one invocation to the next. We then noted that simple modules can be seen as a way to make long-lived objects local to a group of subroutines, in such a way that they are not visible to other parts of the program. At the next level of complexity, we noted that some languages treat modules as types, allowing the programmer to create an arbitrary number of instances of the abstraction defined by a module. We contrasted this module-as-abstraction style of programming with the module-as-manager approach. Finally, we noted that object-oriented languages extend the module-as-abstraction approach by providing an inheritance mechanism that allows new abstractions (classes) to be defined as extensions or refinements of existing classes.

In Section 3.6 we examined several ways in which bindings relate to one another. Aliases arise when two or more names in a given scope are bound to the same object. Overloading arises when one name is bound to multiple objects. Polymorphism allows a single body of code to operate on objects of more than one type, depending on context or execution history. We noted that while similar effects can sometimes be achieved through overloading, coercion, and polymorphism, the underlying mechanisms are really very different. In Section 3.7 we considered rules for separate compilation.

Among the topics considered in this chapter, we saw several examples of useful features (recursion, static scoping, forward references, first-class subroutines, unlimited extent) that have been omitted from certain languages because of concern for their implementation complexity or run-time cost. We also saw an example of a feature (the private part of a module specification) introduced expressly to facilitate a language’s implementation, and another (separate compila-

tion in C) whose design was clearly intended to mirror a particular implementation. In several additional aspects of language design (late versus early binding, static versus dynamic scope, support for coercions and conversions, toleration of pointers and other aliases), we saw that implementation issues play a major role.

In a similar vein, apparently simple language rules can have surprising implications. In Section 3.3.3, for example, we considered the interaction of whole-block scope with the requirement that names be declared before they can be used. Like the do loop syntax and white space rules of Fortran (Section 2.2.2) or the if... then ... else syntax of Pascal (Section 2.3.2), poorly chosen scoping rules can make program analysis difficult not only for the compiler, but for human beings as well. In future chapters we shall see several additional examples of features that are both confusing and hard to compile. Of course, semantic utility and ease of implementation do not always go together. Many easy-to-compile features (goto statements, for example) are of questionable value at best. We will also see several examples of highly useful and (conceptually) simple features, such as garbage collection (Section 7.7.3) and unification (Sections 7.2.4 and 11.2.1), whose implementations are quite complex.

3.9 Exercises

- 3.1 Indicate the binding time (e.g., when the language is designed, when the program is linked, when the program begins execution, etc.) for each of the following decisions in your favorite programming language and implementation. Explain any answers you think are open to interpretation.
 - The number of built-in functions (math, type queries, etc.)
 - The variable declaration that corresponds to a particular variable reference (use)
 - The maximum length allowed for a constant (literal) character string
 - The referencing environment for a subroutine that is passed as a parameter
 - The address of a particular library routine
 - The total amount of space occupied by program code and data
- 3.2 In Fortran 77, local variables are typically allocated statically. In Algol and its descendants (e.g., Pascal and Ada), they are typically allocated in the stack. In Lisp they are typically allocated at least partially in the heap. What accounts for these differences? Give an example of a program in Pascal or Ada that would not work correctly if local variables were allocated statically. Give an example of a program in Scheme or Common Lisp that would not work correctly if local variables were allocated on the stack.

- 3.3** Give two examples in which it might make sense to delay the binding of an implementation decision, even though sufficient information exists to bind it early.
- 3.4** Give three concrete examples drawn from programming languages with which you are familiar in which a variable is live but not in scope.
- 3.5** Consider the following pseudocode, assuming nested subroutines and static scope.

```

procedure main
    g : integer

    procedure B(a : integer)
        x : integer

        procedure A(n : integer)
            g := n

        procedure R(m : integer)
            write_integer(x)
            x /= 2 -- integer division
            if x > 1
                R(m + 1)
            else
                A(m)

        -- body of B
        x := a × a
        R(1)

    -- body of main
    B(3)
    write_integer(g)

```

- (a) What does this program print?
- (b) Show the frames on the stack when *A* has just been called. For each frame, show the static and dynamic links.
- (c) Explain how *A* finds *g*.
- 3.6** As part of the development team at MumbleTech.com, Janet has written a list manipulation library for C that contains, among other things, the code in Figure 3.18.

- (a) Accustomed to Java, new team member Brad includes the following code in the main loop of his program.

```

list_node *L = 0;
while (more_widgets()) {
    insert(next_widget(), L);
}
L = reverse(L);

```

```

typedef struct list_node {
    void *data;
    struct list_node *next;
} list_node;

list_node *insert(void *d, list_node *L) {
    list_node *t = (list_node *) malloc(sizeof(list_node));
    t->data = d;
    t->next = L;
    return t;
}

list_node *reverse(list_node *L) {
    list_node *rtn = 0;
    while (L) {
        rtn = insert(L->data, rtn);
        L = L->next;
    }
    return rtn;
}

void delete_list(list_node *L) {
    while (L) {
        list_node *t = L;
        L = L->next;
        free(t->data);
        free(t);
    }
}

```

Figure 3.18 List management routines for Exercise 3.6.

Sadly, after running for a while, Brad's program always runs out of memory and crashes. Explain what's going wrong.

- (b) After Janet patiently explains the problem to him, Brad gives it another try:

```

list_node *L = 0;
while (more_widgets()) {
    insert(next_widget(), L);
}
list_node *T = reverse(L);
delete_list(L);
L = T;

```

This seems to solve the insufficient memory problem, but where the program used to produce correct results (before running out of memory), now its output is strangely corrupted, and Brad goes back to Janet for advice. What will she tell him this time?

- 3.7** Rewrite Figures 3.7 and 3.8 in C.
- 3.8** Modula-2 provides no way to divide the header of a module into a public part and a private part: everything in the header is visible to the users of the module. Is this a major shortcoming? Are there disadvantages to the public/private division (e.g., as in Ada)? (For hints, see Section 9.2.)
- 3.9** Consider the following fragment of code in C.

```
{   int a, b, c;
...
{   int d, e;
...
{   int f;
...
}
...
}
...
{   int g, h, i;
...
}
...
}
```

Assume that each integer variable occupies four bytes. How much total space is required for the variables in this code? Describe an algorithm that a compiler could use to assign stack frame offsets to the variables of arbitrary nested blocks, in a way that minimizes the total space required.

- 3.10** Consider the design of a Fortran 77 compiler that uses static allocation for the local variables of subroutines. Expanding on the solution to the previous question, describe an algorithm to minimize the total space required for these variables. You may find it helpful to construct a *call graph* data structure in which each node represents a subroutine and each directed arc indicates that the subroutine at the tail may sometimes call the subroutine at the head.
- 3.11** Consider the following pseudocode.

```
procedure P(A, B : real)
  X : real

  procedure Q(B, C : real)
    Y : real
    ...

  procedure R(A, C : real)
    Z : real
    ...
    -- (*)
  ...

```

Assuming static scope, what is the referencing environment at the location marked by (*)?

- 3.12** Write a simple program in Scheme that displays three different behaviors, depending on whether we use `let`, `let*`, or `letrec` to declare a given set of names. (*Hint:* To make good use of `letrec`, you will probably want your names to be functions [`lambda` expressions].)

- 3.13** Consider the following pseudocode.

```

x : integer      -- global

procedure set_x(n : integer)
    x := n

procedure print_x
    write_integer(x)

procedure first
    set_x(1)
    print_x

procedure second
    x : integer
    set_x(2)
    print_x

set_x(0)
first()
print_x
second()
print_x

```

What does this program print if the language uses static scoping? What does it print with dynamic scoping? Why?

- 3.14** Consider the programming idiom illustrated in Example 3.20. One of the reviewers for this book suggests that we think of this idiom as a way to implement a central reference table for dynamic scope. Explain what is meant by this suggestion.

- 3.15** If you are familiar with structured exception-handling, as provided in Ada, Modula-3, C++, Java, C#, ML, Python, or Ruby, consider how this mechanism relates to the issue of scoping. Conventionally, a `raise` or `throw` statement is thought of as referring to an exception, which it passes as a parameter to a handler-finding library routine. In each of the languages mentioned, the exception itself must be declared in some surrounding scope, and is subject to the usual static scope rules. Describe an alternative point of view, in which the `raise` or `throw` is actually a reference to a *handler*, to which it transfers control directly. Assuming this point of view, what are the scope rules for handlers? Are these rules consistent with the rest of the language? Explain. (For further information on exceptions, see Section 8.5.)

3.16 Consider the following pseudocode.

```

x : integer      -- global

procedure set_x(n : integer)
    x := n

procedure print_x
    write_integer(x)

procedure foo(S, P : function; n : integer)
    x : integer := 5
    if n in {1, 3}
        set_x(n)
    else
        S(n)
    if n in {1, 2}
        print_x
    else
        P

set_x(0); foo(set_x, print_x, 1); print_x
set_x(0); foo(set_x, print_x, 2); print_x
set_x(0); foo(set_x, print_x, 3); print_x
set_x(0); foo(set_x, print_x, 4); print_x

```

Assume that the language uses dynamic scoping. What does the program print if the language uses shallow binding? What does it print with deep binding? Why?

3.17 Consider the following pseudocode.

```

x : integer := 1
y : integer := 2

procedure add
    x := x + y

procedure second(P : procedure)
    x : integer := 2
    P()

procedure first
    y : integer := 3
    second(add)

first()
write_integer(x)

```

(a) What does this program print if the language uses static scoping?

- (b) What does it print if the language uses dynamic scoping with deep binding?
 - (c) What does it print if the language uses dynamic scoping with shallow binding?
- 3.18** In Section 3.6.3 we noted that while a single `min` function in Fortran would work for both integer and floating-point numbers, overloading would be more efficient because it would avoid the cost of type conversions. Give an example in which overloading does not seem advantageous—one in which it makes more sense to have a single function with floating-point parameters, and perform coercion when integers are supplied.
- 3.19** (a) Write a polymorphic sorting routine in Scheme.
 (b) Write a generic sorting routine in C++, Java, or C#. (For hints, see Section 8.4.)
 (c) Write a nongeneric sorting routine using subtype polymorphism in your favorite object-oriented language. Assume that the elements to be sorted are members of some class derived from class `ordered`, which has a method `precedes` such that `a.precedes(b)` is true if and only if `a` comes before `b` in some canonical total order. (For hints, see Section 9.4.)
- © **3.20–3.25** In More Depth.

3.10 Explorations

- 3.26** Experiment with naming rules in your favorite programming language. Read the manual, and write and compile some test programs. Does the language use lexical or dynamic scope? Can scopes nest? Are they open or closed? Does the scope of a name encompass the entire block in which it is declared, or only the portion after the declaration? How does one declare mutually recursive types or subroutines? Can subroutines be passed as parameters, returned from functions, or stored in variables? If so, when are referencing environments bound?
- 3.27** List the keywords (reserved words) of one or more programming languages. List the predefined identifiers. (Recall that every keyword is a separate token. An identifier cannot have the same spelling as a keyword.) What criteria do you think were used to decide which names should be keywords and which should be predefined identifiers? Do you agree with the choices? Why or why not?
- 3.28** If you have experience with a language like C, C++, or Pascal, in which dynamically allocated space must be manually reclaimed, describe your experience with dangling references or memory leaks. How often do these bugs

arise? How do you find them? How much effort does it take? Learn about open source or commercial tools for finding storage bugs (IBM's *Purify* is a popular example). Do such tools weaken the argument for automatic garbage collection?

- 3.29 We learned in Section 3.3.6 that modern languages have generally abandoned dynamic scoping. One place it can still be found is in the so-called *environment variables* of the Unix programming environment. If you are not familiar with these, read the manual page for your favorite shell (command interpreter—csh/tcsh, ksh/bash, etc.) to learn how these behave. Explain why the usual alternatives to dynamic scoping (default parameters and static variables) are not appropriate in this case.
- 3.30 Compare the mechanisms for overloading of enumeration names in Ada and Modula-3 (Section 3.6.2). One might argue that the (historically more recent) Modula-3 approach moves responsibility from the compiler to the programmer: it requires even an unambiguous use of an enumeration constant to be annotated with its type. Why do you think this approach was chosen by the language designers? Do you agree with the choice? Why or why not?
- 3.31 Write a program in C++ or Ada that creates at least two concrete types or subroutines from the same template/generic. Compile your code to assembly language and look at the result. Describe the mapping from source to target code.
- 3.32 Do you think coercion is a good idea? Why or why not?
- 3.33 Give three examples of features that are *not* provided in some language with which you are familiar, but that are common in other languages. Why do you think these features are missing? Would they complicate the implementation of the language? If so, would the complication (in your judgment) be justified?

© 3.34–3.38 In More Depth.

3.11 Bibliographic Notes

This chapter has traced the evolution of naming and scoping mechanisms through many different languages, including Fortran (several versions), Basic, Algol 60 and 68, Pascal, Simula, C and C++, Euclid, Turing, Modula (1, 2, and 3), Ada (83 and 95), Oberon, Eiffel, Java, and C#. Bibliographic references for all of these can be found in Appendix A.

Both modules and objects trace their roots to Simula, which was developed by Dahl, Nygaard, Myhrhaug, and others at the Norwegian Computing Centre in the mid-1960s. (Simula I was implemented in 1964; descriptions in this book pertain to Simula 67.) The encapsulation mechanisms of Simula were refined in

the 1970s by the developers of Clu, Modula, Euclid, and related languages. Other Simula innovations—inheritance and dynamic method binding in particular—provided the inspiration for Smalltalk, the original and arguably purest of the object-oriented languages. Modern object-oriented languages, including Eiffel, C++, Java, and C#, represent to a large extent a reintegration of the evolutionary lines of encapsulation on the one hand and inheritance and dynamic method binding on the other.

The notion of information hiding originates in Parnas's classic paper “On the Criteria to Be Used in Decomposing Systems into Modules” [Par72]. Comparative discussions of naming, scoping, and abstraction mechanisms can be found, among other places, in Liskov et al.'s discussion of Clu [LSAS77], Liskov and Guttag's text [LG86, Chap. 4], the Ada Rationale [IBFW91, Chaps. 9–12], Harbison's text on Modula-3 [Har92, Chaps. 8–9], Wirth's early work on modules [Wir80], and his later discussion of Modula and Oberon [Wir88a]. Further information on object-oriented languages can be found in Chapter 9.

For a detailed discussion of overloading and polymorphism, see the survey by Cardelli and Wegner [CW85]. Cailliau [Cai82] provides a lighthearted discussion of many of the scoping pitfalls noted in Section 3.3.3. Abelson and Sussman [AS96, p. 11n] attribute the term “syntactic sugar” to Peter Landin.

Semantic Analysis

In Chapter 2 we considered the topic of programming language syntax. In the current chapter we turn to the topic of semantics. Informally, syntax concerns the *form* of a valid program, while semantics concerns its *meaning*. Meaning is important for at least two reasons: it allows us to enforce rules (e.g., type consistency) that go beyond mere form, and it provides the information we need in order to generate an equivalent output program.

It is conventional to say that the syntax of a language is precisely that portion of the language definition that can be described conveniently by a context-free grammar, while the semantics is that portion of the definition that cannot. This convention is useful in practice, though it does not always agree with intuition. When we require, for example, that the number of arguments contained in a call to a subroutine match the number of formal parameters in the subroutine definition, it is tempting to say that this requirement is a matter of syntax. After all, we can count arguments without knowing what they mean. Unfortunately, we cannot count them with context-free rules. Similarly, while it is possible to write a context-free grammar in which every function must contain at least one `return` statement, the required complexity makes this strategy very unattractive. In general, any rule that requires the compiler to compare things that are separated by long distances, or to count things that are not properly nested, ends up being a matter of semantics.

Semantic rules are further divided into *static* and *dynamic* semantics, though again the line between the two is somewhat fuzzy. The compiler enforces static semantic rules at compile time. It generates code to enforce dynamic semantic rules at run time (or to call library routines that do so). Certain errors, such as division by zero, or attempting to index into an array with an out-of-bounds subscript, cannot in general be caught at compile time, since they may occur only for certain input values, or certain behaviors of arbitrarily complex code. In special cases, a compiler may be able to tell that a certain error will always or never occur, regardless of run-time input. In these cases, the compiler can

generate an error message at compile time, or refrain from generating code to perform the check at run time, as appropriate. Basic results from computability theory, however, tell us that no algorithm can make these predictions correctly for arbitrary programs. There will inevitably be cases in which an error will always occur, but the compiler cannot tell, and must delay the error message until run time. There will also be cases in which an error can never occur, but the compiler cannot tell, and must incur the cost of unnecessary run-time checks.

Both semantic analysis and intermediate code generation can be described in terms of annotation, or *decoration*, of a parse tree or syntax tree. The annotations themselves are known as *attributes*. Numerous examples of static and dynamic semantic rules will appear in subsequent chapters. In this current chapter we focus primarily on the mechanisms a compiler uses to enforce the static rules. We will consider intermediate code generation in Chapter 14.

In Section 4.1 we consider the role of the semantic analyzer in more detail, considering both the rules it needs to enforce and its relationship to other phases of compilation. Most of the rest of the chapter is then devoted to the subject of *attribute grammars*. Attribute grammars provide a formal framework for the decoration of a tree. This framework is a useful conceptual tool even in compilers that do not build a parse tree or syntax tree as an explicit data structure. We introduce the notion of an attribute grammar in Section 4.2. We then consider various ways in which such grammars can be applied in practice. Section 4.3 discusses the issue of *attribute flow*, which constrains the order(s) in which nodes of a tree can be decorated. In practice, most compilers require decoration of the parse tree (or the evaluation of attributes that would reside in a parse tree if there were one) to occur in the process of an LL or LR parse. Section 4.4 presents *action routines* as an ad hoc mechanism for such on-the-fly evaluation. In Section 4.5 (mostly on the PLP CD) we consider the management of space for parse tree attributes.

One particularly common compiler organization uses action routines during parsing solely for the purpose of constructing a syntax tree. The syntax tree is then decorated during a separate traversal, which can be formalized, if desired, with a separate attribute grammar. We consider the decoration of syntax trees in Section 4.6.

4.1 The Role of the Semantic Analyzer

Programming languages vary dramatically in their choice of semantic rules. In Section 3.6.3, for example, we saw a range of approaches to coercion, from languages like Fortran and C, which allow operands of many types to be intermixed in expressions, to languages like Ada, which do not. Languages also vary in the extent to which they require their implementations to perform dynamic checks. At one extreme, C requires no checks at all, beyond those that come “free” with the hardware (e.g., division by zero or attempted access to memory outside the

bounds of the program). At the other extreme, Java takes great pains to check as many rules as possible, in part to ensure that an untrusted program cannot do anything to damage the memory or files of the machine on which it runs.

In the typical compiler, the interface between semantic analysis and intermediate code generation defines the boundary between the *front end* and the *back end*. The exact division of labor varies a bit from compiler to compiler: it can be hard to say exactly where analysis (figuring out what the program means) ends and synthesis (expressing that meaning in some new form) begins. Many compilers actually carry a program through more than one intermediate form. In one common organization, described in more detail in Chapter 14, the semantic analyzer creates an annotated syntax tree, which the intermediate code generator then translates into a linear form reminiscent of the assembly language for some idealized machine. After machine-independent code improvement, this linear form is then translated into yet another form, patterned more closely on the assembly language of the target machine. That form may then undergo machine-specific code improvement.

Compilers also vary in the extent to which semantic analysis and intermediate code generation are interleaved with parsing. With fully separated phases, the parser passes a full parse tree on to the semantic analyzer, which converts it to a syntax tree, fills in the symbol table, performs semantic checks, and passes it on to the code generator. With fully interleaved phases, there may be no need to build either the parse tree or the syntax tree in its entirety: the parser can call semantic check and code generation routines “on-the-fly” as it parses each expression, statement, or subroutine of the source. We will focus on an organization in which construction of the syntax tree is interleaved with parsing (and the parse tree is not built), but semantic analysis occurs during a separate traversal of the syntax tree.

Many compilers that implement dynamic checks provide the option of disabling them if desired. It is customary in some organizations to enable dynamic checks during program development and testing, and then disable them for production use, to increase execution speed. The wisdom of this practice is questionable: Tony Hoare, one of the key figures in programming language design,¹ has likened the programmer who disables semantic checks to a sailing enthusiast who wears a life jacket when training on dry land but removes it when going to sea [Hoa89, p. 198]. Errors may be less likely in production use than they are in testing, but the consequences of an undetected error are significantly worse. Moreover, with the increasing use of multi-issue, superscalar processors (described in Section 5.4.3), it is often possible for dynamic checks to execute in instruction slots that would otherwise go unused, making them virtually free. On

1 Among other things, C. A. R. Hoare (1934–) invented the quicksort algorithm and the `case` statement, contributed to the design of Algol W, and was one of the leaders in the development of axiomatic semantics. In the area of concurrent programming, he refined and formalized the `monitor` construct (to be described in Section 12.3.4), and designed the CSP programming model and notation. He received the ACM Turing Award in 1980.

the other hand, some dynamic checks (e.g., for use of uninitialized variables) are sufficiently expensive that they are rarely implemented.

Assertions

EXAMPLE 4.1

Assertions in Euclid

A few programming languages (e.g., Euclid and Eiffel) allow the programmer to specify logical *assertions*, *invariants*, *preconditions*, and *postconditions* that must be verified by dynamic semantic checks. An assertion is a statement that a specified condition is expected to be true when execution reaches a certain point in the code. In Euclid one can write

```
assert denominator not= 0
```

An invariant is a condition that is expected to be true at all “clean points” of a given body of code. In Eiffel the programmer can specify an invariant on the data inside a class: the invariant is expected to be true at the beginning and end of all of the class’s methods (subroutines). Similar invariants for loops are expected to be true before and after every iteration. Pre- and postconditions are expected to be true at the beginning and end of subroutines, respectively.

Invariants, preconditions, and postconditions are essentially structured assertions. A postcondition, specified once in the header of a Euclid subroutine, will be checked not only at the end of the subroutine’s text, but at every `return` statement as well, automatically.

EXAMPLE 4.2

Assertions in C

Many languages support assertions via standard library routines or macros. In C, for example, one can write

```
assert(denominator != 0);
```

If the assertion fails, the program will terminate abruptly with the message

```
myprog.c:42: failed assertion 'denominator != 0'
```

The C manual requires `assert` to be implemented as a macro (or built into the compiler) so that it has access to the textual representation of its argument, and to the file name and line number on which the call appears.

DESIGN & IMPLEMENTATION

Dynamic semantic checks

In the past, language theorists and researchers in programming methodology and software engineering tended to argue for more extensive semantic checks, while “real world” programmers “voted with their feet” for languages like C and Fortran, which omitted those checks in the interest of execution speed. As computers have become more powerful, and as companies have come to appreciate the enormous costs of software maintenance, the “real world” camp has become much more sympathetic to checking. Languages like Ada and Java have been designed from the outset with safety in mind, and languages like C and C++ have evolved (to the extent possible) toward increasingly strict definitions.

Assertions, of course, could be used to cover the other three sorts of checks, but not as clearly or succinctly. Invariants, preconditions, and postconditions are a prominent part of the header of the code to which they apply, and can cover a potentially large number of places where an assertion would otherwise be required. Euclid and Eiffel implementations allow the programmer to disable assertions and related constructs when desired, to eliminate their run-time cost.

Static Analysis

In general, compile-time algorithms that predict run-time behavior are known as *static analysis*. Such analysis is said to be *precise* if it allows the compiler to determine whether a given program will always follow the rules. Type checking, for example, is static and precise in languages like Ada, C, and ML: the compiler ensures that no variable will ever be used at run time in a way that is inappropriate for its type. By contrast, languages like Lisp and Smalltalk obtain greater flexibility, while remaining completely type-safe, by accepting the run-time overhead of dynamic type checks. (We will cover type checking in more detail in Chapter 7.)

Static analysis can also be useful when it isn't precise. Compilers will often check what they can at compile time and then generate code to check the rest dynamically. In Java, for example, type checking is mostly static, but dynamically loaded classes and type casts may require run-time checks. In a similar vein, many compilers perform extensive static analysis in an attempt to eliminate the need for dynamic checks on array subscripts, variant record tags, or potentially dangling pointers (again, to be discussed in Chapter 7).

If we think of the omission of unnecessary dynamic checks as a performance optimization, it is natural to look for other ways in which static analysis may enable code improvement. We will consider this topic in more detail in Chapter 15. Examples include *alias analysis*, which determines when values can be safely cached in registers, computed "out of order," or accessed by concurrent threads; *escape analysis*, which determines when all references to a value will be confined to a given context, allowing it to be allocated on the stack instead of the heap, or to be accessed without locks; and *subtype analysis*, which determines when a variable in an object-oriented language is guaranteed to have a certain subtype, so that its methods can be called without dynamic dispatch.

An optimization is said to be *unsafe* if it may lead to incorrect code in certain programs. It is said to be *speculative* if it usually improves performance but may degrade it in certain cases. A compiler is said to be *conservative* if it applies optimizations only when it can guarantee that they will be both safe and effective. By contrast, an *optimistic* compiler may make liberal use of speculative optimizations. It may also pursue unsafe optimizations by generating two versions of the code, with a dynamic check that chooses between them based on information not available at compile time. Examples of speculative optimization include *nonbinding prefetches*, which try to bring data into the cache before they are needed, and *trace scheduling*, which rearranges code in hopes of improving the performance of the processor pipeline and the instruction cache.

To eliminate dynamic checks, language designers may choose to tighten semantic rules, banning programs for which conservative analysis fails. The ML type system (Section 7.2.4), for example, avoids the dynamic type checks of Lisp but disallows certain useful programming idioms that Lisp supports. Similarly, the *definite assignment* rules of Java and C# (Section 6.1.3) allow the compiler to ensure that a variable is always given a value before it is used in an expression, but disallow certain programs that are legal (and correct) in C.

4.2 Attribute Grammars

EXAMPLE 4.3

Bottom-up CFG for constant expressions

In Chapter 2 we learned how to use a context-free grammar to specify the syntax of a programming language. Here, for example, is an LR (bottom-up) grammar for arithmetic expressions composed of constants, with precedence and associativity:

$$\begin{array}{l} E \longrightarrow E + T \\ E \longrightarrow E - T \\ E \longrightarrow T \\ T \longrightarrow T * F \\ T \longrightarrow T / F \\ T \longrightarrow F \\ F \longrightarrow - F \\ F \longrightarrow (E) \\ F \longrightarrow \text{const} \end{array}$$



EXAMPLE 4.4

Bottom-up AG for constant expressions

This grammar will generate all properly formed constant expressions over the basic arithmetic operators, but it says nothing about their meaning. To tie these expressions to mathematical concepts (as opposed to, say, floor tile patterns or dance steps), we need additional notation. The most common is based on *attributes*. In our expression grammar, we can associate a *val* attribute with each *E*, *T*, *F*, and *const* in the grammar. The intent is that for any symbol *S*, *S.val* will be the meaning, as an arithmetic value, of the token string derived from *S*. We assume that the *val* of a *const* is provided to us by the scanner. We must then invent a set of rules for each production to specify how the *vals* of different symbols are related. The resulting *attribute grammar* is shown in Figure 4.1.

In this simple grammar, every production has a single rule. We shall see more complicated grammars later in which productions can have several rules. The rules come in two forms. Those in productions 3, 6, 8, and 9 are known as *copy rules*; they specify that one attribute should be a copy of another. The other rules invoke *semantic functions* (*sum*, *quotient*, *additive_inverse*, etc.). In this example, the semantic functions are all familiar arithmetic operations. In general, they can be arbitrarily complex functions specified by the language designer. Each seman-

1. $E_1 \longrightarrow E_2 + T$
▷ $E_1.\text{val} := \text{sum}(E_2.\text{val}, T.\text{val})$
2. $E_1 \longrightarrow E_2 - T$
▷ $E_1.\text{val} := \text{difference}(E_2.\text{val}, T.\text{val})$
3. $E \longrightarrow T$
▷ $E.\text{val} := T.\text{val}$
4. $T_1 \longrightarrow T_2 * F$
▷ $T_1.\text{val} := \text{product}(T_2.\text{val}, F.\text{val})$
5. $T_1 \longrightarrow T_2 / F$
▷ $T_1.\text{val} := \text{quotient}(T_2.\text{val}, F.\text{val})$
6. $T \longrightarrow F$
▷ $T.\text{val} := F.\text{val}$
7. $F_1 \longrightarrow - F_2$
▷ $F_1.\text{val} := \text{additive_inverse}(F_2.\text{val})$
8. $F \longrightarrow (E)$
▷ $F.\text{val} := E.\text{val}$
9. $F \longrightarrow \text{const}$
▷ $F.\text{val} := \text{const}.val$

Figure 4.1 A simple attribute grammar for constant expressions, using the standard arithmetic operations.

tic function takes an arbitrary number of arguments (each of which must be an attribute of a symbol in the current production: no constants, global variables, etc.), and each computes a single result, which must likewise be assigned into an attribute of a symbol in the current production. When more than one symbol of a production has the same name, subscripts are used to distinguish them. These subscripts are solely for the benefit of the semantic functions; they are not part of the context-free grammar itself. ■

In a strict definition of attribute grammars, copy rules and semantic function calls are the only two kinds of permissible rules. In practice, it is common to allow rules to consist of small fragments of code in some well-defined notation (e.g., the language in which a compiler is being written) so that simple semantic functions can be written out “in-line.” These code fragments are not allowed to refer to any variables or attributes outside the current production (we will relax this restriction when we discuss action routines in Section 4.4). In our examples we use a ▷ symbol to introduce each code fragment corresponding to a single semantic function.

Semantic functions must be written in some already-existing notation, because attribute grammars do not really specify the meaning of a program; rather, they provide a way to associate a program with something else that presumably has meaning. Neither the notation for semantic functions nor the types of the

attributes themselves (i.e., the domain of values passed to and returned from semantic functions) is intrinsic to the attribute grammar notion. In the preceding example, we have used an attribute grammar to associate numeric values with the symbols in our grammar, using semantic functions drawn from ordinary arithmetic. In the code generation phase of a compiler, we might associate fragments of target machine code with our symbols, using semantic functions written in some existing programming language. If we were interested in defining the meaning of a programming language in a machine-independent way, our attributes might be domain theory *denotations* (these are the basis of *denotational semantics*). If we were interested in proving theorems about the behavior of programs in our language, our attributes might be logical formulas (this is the basis of *axiomatic semantics*).² These more formal concepts are beyond the scope of this text (but see the Bibliographic Notes at the end of the chapter). We will use attribute grammars primarily as a framework for building a syntax tree, checking semantic rules, and (in Chapter 14) generating code.

4.3 Evaluating Attributes

EXAMPLE 4.5

Decoration of a parse tree

The process of evaluating attributes is called *annotation* or *decoration* of the parse tree. Figure 4.2 shows how to decorate the parse tree for the expression $(1 + 3) * 2$, using the attribute grammar of Figure 4.1. Once decoration is complete, the value of the overall expression can be found in the *val* attribute of the root of the tree. ■

Synthesized Attributes

The attribute grammar of Figure 4.1 is very simple. Each symbol has at most one attribute (the punctuation marks have none). Moreover, they are all so-called *synthesized attributes*: their values are calculated (synthesized) only in productions in which their symbol appears on the left-hand side. For annotated parse trees like the one in Figure 4.2, this means that the *attribute flow*—the pattern in which information moves from node to node—is entirely bottom-up.

An attribute grammar in which all attributes are synthesized is said to be *S-attributed*. The arguments to semantic functions in an S-attributed grammar are always attributes of symbols on the right-hand side of the current production, and the return value is always placed into an attribute of the left-hand side of the production. Tokens (terminals) often have intrinsic properties (e.g., the character-string representation of an identifier or the value of a numeric

2 It's actually stretching things a bit to discuss axiomatic semantics in the context of attribute grammars. Axiomatic semantics is intended not so much to define the meaning of programs as to permit one to prove that a given program satisfies some desired property (e.g., computes some desired function).

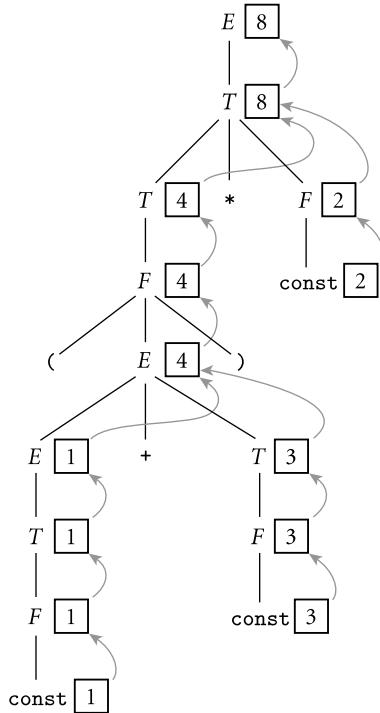


Figure 4.2 Decoration of a parse tree for $(1 + 3) * 2$. The val attributes of symbols are shown in boxes. Curved arrows represent the attribute flow, which is strictly upward in this case.

constant); in a compiler these are synthesized attributes initialized by the scanner.

Inherited Attributes

In general, we can imagine (and will in fact have need of) attributes whose values are calculated when their symbol is on the right-hand side of the current production. Such attributes are said to be *inherited*. They allow contextual information to flow into a symbol from above or from the side, so that the rules of that production can be enforced in different ways (or generate different values) depending on surrounding context. Symbol table information is commonly passed from symbol to symbol by means of inherited attributes. Inherited attributes of the root of the parse tree can also be used to represent the external environment (characteristics of the target machine, command-line arguments to the compiler, etc.).

EXAMPLE 4.6

Top-down CFG and parse tree for subtraction

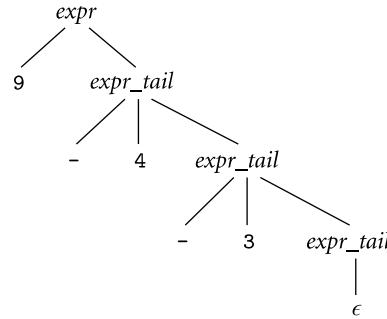
As a simple example of inherited attributes, consider the following simplified fragment of an LL(1) expression grammar (here covering only subtraction):

```

expr → const expr_tail
expr_tail → - const expr_tail | ε.

```

For the expression $9 - 4 - 3$, we obtain the following parse tree:



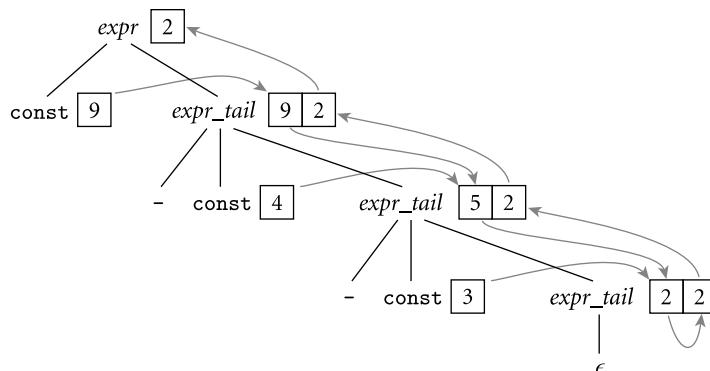
■

If we want to create an attribute grammar that accumulates the value of the overall expression into the root of the tree, we have a problem: because subtraction is left-associative, we cannot summarize the right subtree of the root with a single numeric value. If we want to decorate the tree bottom-up, with an S-attributed grammar, we must be prepared to describe an arbitrary number of right operands in the attributes of the top-most *expr_tail* node (see Exercise 4.4). This is indeed possible, but it defeats the purpose of the formalism: in effect, it requires us to embed the entire tree into the attributes of a single node, and do all the real work inside a single semantic function.

EXAMPLE 4.7

Decoration with left-to-right attribute flow

If, however, we are allowed to pass attribute values not only bottom-up but also left-to-right in the tree, then we can pass the 9 into the top-most *expr_tail* node, where it can be combined (in proper left-associative fashion) with the 4. The resulting 5 can then be passed into the middle *expr_tail* node, combined with the 3 to make 2, and then passed upward to the root:



■

1. $E \longrightarrow T \text{ } TT$
 $\triangleright \text{TT.st} := T.\text{val}$ $\triangleright E.\text{val} := \text{TT.val}$
2. $TT_1 \longrightarrow + \text{ } T \text{ } TT_2$
 $\triangleright \text{TT}_2.\text{st} := \text{TT}_1.\text{st} + T.\text{val}$ $\triangleright \text{TT}_1.\text{val} := \text{TT}_2.\text{val}$
3. $TT_1 \longrightarrow - \text{ } T \text{ } TT_2$
 $\triangleright \text{TT}_2.\text{st} := \text{TT}_1.\text{st} - T.\text{val}$ $\triangleright \text{TT}_1.\text{val} := \text{TT}_2.\text{val}$
4. $TT \longrightarrow \epsilon$
 $\triangleright \text{TT.val} := \text{TT.st}$
5. $T \longrightarrow F \text{ } FT$
 $\triangleright \text{FT.st} := F.\text{val}$ $\triangleright T.\text{val} := \text{FT.val}$
6. $FT_1 \longrightarrow * \text{ } F \text{ } FT_2$
 $\triangleright \text{FT}_2.\text{st} := \text{FT}_1.\text{st} \times F.\text{val}$ $\triangleright \text{FT}_1.\text{val} := \text{FT}_2.\text{val}$
7. $FT_1 \longrightarrow / \text{ } F \text{ } FT_2$
 $\triangleright \text{FT}_2.\text{st} := \text{FT}_1.\text{st} \div F.\text{val}$ $\triangleright \text{FT}_1.\text{val} := \text{FT}_2.\text{val}$
8. $FT \longrightarrow \epsilon$
 $\triangleright \text{FT.val} := \text{FT.st}$
9. $F_1 \longrightarrow - \text{ } F_2$
 $\triangleright F_1.\text{val} := - F_2.\text{val}$
10. $F \longrightarrow (\text{ } E \text{ })$
 $\triangleright F.\text{val} := E.\text{val}$
11. $F \longrightarrow \text{const}$
 $\triangleright F.\text{val} := \text{const.val}$

Figure 4.3 An attribute grammar for constant expressions based on an LL(1) CFG.

EXAMPLE 4.8

Top-down AG for subtraction

To effect this style of decoration, we need the following attribute rules:

- ```

 $expr \longrightarrow \text{const } expr_tail$
 $\triangleright \text{expr_tail.st} := \text{const.val}$
 $\triangleright \text{expr.val} := \text{expr_tail.val}$

 $expr_tail_1 \longrightarrow - \text{ const } expr_tail_2$
 $\triangleright \text{expr_tail}_2.\text{st} := \text{expr_tail}_1.\text{st} - \text{const.val}$
 $\triangleright \text{expr_tail}_1.\text{val} := \text{expr_tail}_2.\text{val}$

 $expr_tail \longrightarrow \epsilon$
 $\triangleright \text{expr_tail.val} := \text{expr_tail.st}$

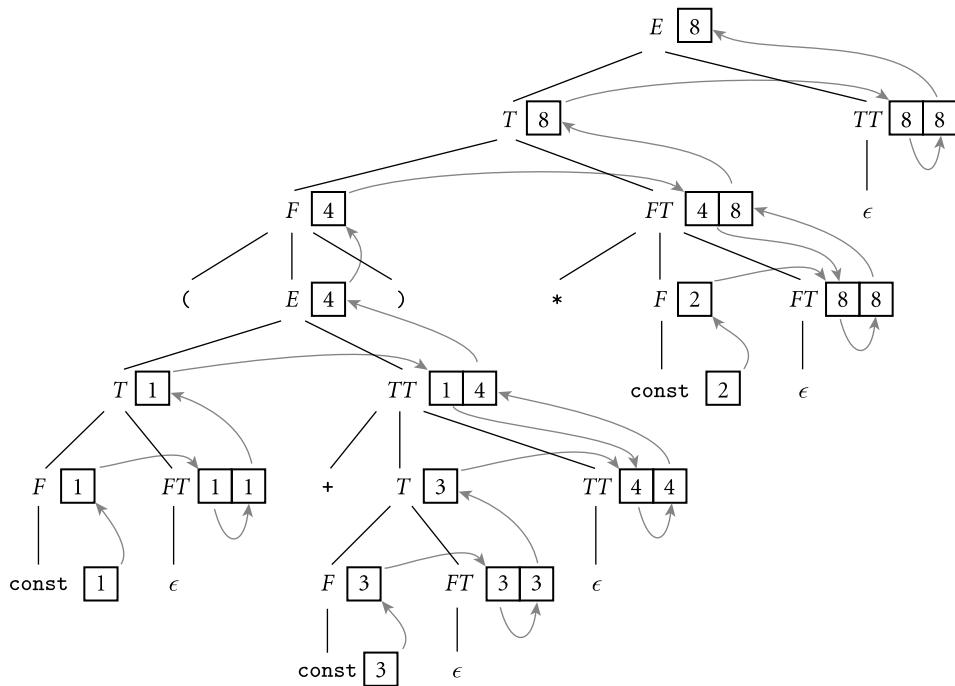
```

In each of the first two productions, the first rule serves to copy the left context (value of the expression so far) into a “subtotal” (st) attribute; the second rule copies the final value from the right-most leaf back up to the root. ■

#### EXAMPLE 4.9

Top-down AG for constant expressions

We can flesh out the grammar fragment of Example 4.6 to produce a more complete expression grammar, as shown in Figure 4.3. The underlying CFG for this grammar accepts the same language as the one in Figure 4.1, but where that one was SLR(1), this one is LL(1). Attribute flow for a parse of  $(1 + 3) * 2$ ,



**Figure 4.4** Decoration of a top-down parse tree for  $(1 + 3) * 2$ , using the attribute grammar of Figure 4.3. Curved arrows again represent attribute flow, which is no longer bottom-up, but is still left-to-right.

using the LL(1) grammar, appears in Figure 4.4. As in the grammar fragment of Example 4.6, the value of the left operand of each operator is carried into the  $TT$  and  $FT$  productions by the  $st$  (subtotal) attribute. The relative complexity of the attribute flow arises from the fact that operators are left associative, but the grammar cannot be left recursive: the left and right operands of a given operator are thus found in separate productions. Grammars to perform semantic analysis for practical languages generally require some non-S-attributed flow. ■

#### Attribute Flow

Just as a context-free grammar does not specify how it should be parsed, an attribute grammar does not specify the order in which attribute rules should be invoked. Put another way, both notations are *declarative*: they define a set of valid trees, but they don't say how to build or decorate them. Among other things, this means that the order in which attribute rules are listed for a given production is immaterial; attribute flow may require them to execute in any order. If in Figure 4.3 we were to reverse the order in which the rules appear in productions 1, 2, 3, 5, 6, and/or 7 (listing the rule for `symbol.val` first), it would be a purely cosmetic change; the grammar would not be altered.

We say an attribute grammar is *well defined* if its rules determine a unique set of values for the attributes of every possible parse tree. An attribute grammar is *noncircular* if it never leads to a parse tree in which there are cycles in the attribute flow graph—that is, if no attribute, in any parse tree, ever depends (transitively) on itself. (A grammar can be circular and still be well defined if attributes are guaranteed to converge to a unique value.) As a general rule, practical attribute grammars tend to be noncircular.

An algorithm that decorates parse trees by invoking the rules of an attribute grammar in an order that respects the tree's attribute flow is called a *translation scheme*. Perhaps the simplest scheme is one that makes repeated passes over a tree, invoking any semantic function whose arguments have all been defined, and stopping when it completes a pass in which no values change. Such a scheme is said to be *oblivious*, in the sense that it exploits no special knowledge of either the parse tree or the grammar. It will halt only if the grammar is well defined. Better performance, at least for noncircular grammars, may be achieved by a *dynamic* scheme that tailors the evaluation order to the structure of a given parse tree—for example, by constructing a topological sort of the attribute flow graph and then invoking rules in an order consistent with the sort.

The fastest translation schemes, however, tend to be *static*—based on an analysis of the structure of the attribute grammar itself, and then applied mechanically to any tree arising from the grammar. Like LL and LR parsers, linear-time static translation schemes can be devised only for certain restricted classes of grammars. S-attributed grammars, such as the one in Figure 4.1, form the simplest such class. Because attribute flow in an S-attributed grammar is strictly bottom-up, attributes can be evaluated by visiting the nodes of the parse tree in exactly the same order that those nodes were generated by the parser. In fact, the attributes can be evaluated on-the-fly during a bottom-up parse, thereby interleaving parsing and semantic analysis (attribute evaluation).

The attribute grammar of Figure 4.3 is a good bit messier than that of Figure 4.1, but it is still *L-attributed*: its attributes can be evaluated by visiting the nodes of the parse tree in a single left-to-right, depth-first traversal (the same order in which they are visited during a top-down parse). If we say that an attribute  $A.s$  *depends on* an attribute  $B.t$  if  $B.t$  is ever passed to a semantic function that returns a value for  $A.s$ , then we can define L-attributed grammars more formally with the following two rules: (1) each synthesized attribute of a left-hand side symbol depends only on that symbol's own inherited attributes or on attributes (synthesized or inherited) of the production's right-hand side symbols; and (2) each inherited attribute of a right-hand side symbol depends only on inherited attributes of the left-hand side symbol or on attributes (synthesized or inherited) of symbols to its left in the right-hand side.

S-attributed grammars are the most general class of attribute grammars for which evaluation can be implemented on-the-fly during an LR parse. L-attributed grammars are a proper superset of S-attributed grammars. They are the most general class of attribute grammars for which evaluation can be implemented on-the-fly during an LL parse. If we interleave semantic analysis (and

possibly intermediate code generation) with parsing, then a bottom-up parser must in general be paired with an S-attributed translation scheme; a top-down parser must be paired with an L-attributed translation scheme. (Depending on the structure of the grammar, it is often possible for a bottom-up parser to accommodate some non-S-attributed attribute flow; we consider this possibility in Section 4.5.1.) If we choose to separate parsing and semantic analysis into separate passes, then the code that builds the parse tree or syntax tree must still use an S-attributed or L-attributed translation scheme (as appropriate), but the semantic analyzer can use a more powerful scheme if desired. There are certain tasks, such as the generation of code for “short-circuit” Boolean expressions (to be discussed in Sections 6.1.5 and 6.4.1), that are easiest to accomplish with a non-L-attributed scheme.

### One-Pass Compilers

A compiler that interleaves semantic analysis and code generation with parsing is said to be a *one-pass compiler*.<sup>3</sup> It is unclear whether interleaving semantic analysis with parsing makes a compiler simpler or more complex; it’s mainly a matter of taste. If intermediate code generation is interleaved with parsing, one need not build a syntax tree at all (unless of course the syntax tree *is* the intermediate code). Moreover, it is often possible to write the intermediate code to an output file on-the-fly, rather than accumulating it in the attributes of the root of the parse tree. The resulting space savings were important for previous generations of computers, which had very small main memories. On the other hand, semantic analysis is easier to perform during a separate traversal of

## DESIGN & IMPLEMENTATION

### Forward references

In Sections 3.3.3 and 3.4.1 we noted that the scope rules of many languages require names to be declared before they are used, and provide special mechanisms to introduce the forward references needed for recursive definitions. While these rules may help promote the creation of clear, maintainable code, an equally important motivation, at least historically, was to facilitate the construction of one-pass compilers. With increases in memory size, processing speed, and programmer expectations regarding the quality of code improvement, multipass compilers have become ubiquitous, and language designers have felt free (as, for example, in the class declarations of C++, Java, and C#) to abandon the requirement that declarations precede uses.

---

**3** Most authors use the term *one-pass* only for compilers that translate all the way from source to target code in a single pass. Some authors insist only that intermediate code be generated in a single pass, and permit additional pass(es) to translate intermediate code to target code.

```

 $E_1 \longrightarrow E_2 + T$
 ▷ $E_1.\text{ptr} := \text{make_bin_op}("+", E_2.\text{ptr}, T.\text{ptr})$
 $E_1 \longrightarrow E_2 - T$
 ▷ $E_1.\text{ptr} := \text{make_bin_op}("-", E_2.\text{ptr}, T.\text{ptr})$
 $E \longrightarrow T$
 ▷ $E.\text{ptr} := T.\text{ptr}$
 $T_1 \longrightarrow T_2 * F$
 ▷ $T_1.\text{ptr} := \text{make_bin_op}("\times", T_2.\text{ptr}, F.\text{ptr})$
 $T_1 \longrightarrow T_2 / F$
 ▷ $T_1.\text{ptr} := \text{make_bin_op}("\div", T_2.\text{ptr}, F.\text{ptr})$
 $T \longrightarrow F$
 ▷ $T.\text{ptr} := F.\text{ptr}$
 $F_1 \longrightarrow - F_2$
 ▷ $F_1.\text{ptr} := \text{make_un_op}("+/_-", F_2.\text{ptr})$
 $F \longrightarrow (E)$
 ▷ $F.\text{ptr} := E.\text{ptr}$
 $F \longrightarrow \text{const}$
 ▷ $F.\text{ptr} := \text{make_leaf}(\text{const}.val)$

```

**Figure 4.5** Bottom-up attribute grammar to construct a syntax tree. The symbol  $+/_-$  is used (as it is on calculators) to indicate change of sign.

a syntax tree, because that tree reflects the program’s semantic structure better than the parse tree does, especially with a top-down parser, and because one has the option of traversing the tree in an order other than that chosen by the parser.

### Building a Syntax Tree

If we choose not to interleave parsing and semantic analysis, we still need to add attribute rules to the context-free grammar, but they serve only to create the syntax tree—not to enforce semantic rules or generate code. Figures 4.5 and 4.6 contain bottom-up and top-down attribute grammars, respectively, to build a syntax tree for constant expressions. The attributes in these grammars hold neither numeric values nor target code fragments; instead they point to nodes of the syntax tree. Function `make_leaf` returns a pointer to a newly allocated syntax tree node containing the value of a constant. Functions `make_un_op` and `make_bin_op` return pointers to newly allocated syntax tree nodes containing a unary or binary operator, respectively, and pointers to the supplied operand(s). Figures 4.7 and 4.8 show stages in the decoration of parse trees for  $(1 + 3) * 2$ , using the grammars of Figures 4.5 and 4.6, respectively. ■

#### EXAMPLE 4.10

Bottom-up and top-down AGs to build a syntax tree

```

 $E \longrightarrow T \; TT$
 ▷ $TT.\text{st} := T.\text{ptr}$
 ▷ $E.\text{ptr} := TT.\text{ptr}$

 $TT_1 \longrightarrow + \; T \; TT_2$
 ▷ $TT_2.\text{st} := \text{make_bin_op}("+", TT_1.\text{st}, T.\text{ptr})$
 ▷ $TT_1.\text{ptr} := TT_2.\text{ptr}$

 $TT_1 \longrightarrow - \; T \; TT_2$
 ▷ $TT_2.\text{st} := \text{make_bin_op}("-", TT_1.\text{st}, T.\text{ptr})$
 ▷ $TT_1.\text{ptr} := TT_2.\text{ptr}$

 $TT \longrightarrow \epsilon$
 ▷ $TT.\text{ptr} := TT.\text{st}$

 $T \longrightarrow F \; FT$
 ▷ $FT.\text{st} := F.\text{ptr}$
 ▷ $T.\text{ptr} := FT.\text{ptr}$

 $FT_1 \longrightarrow * \; F \; FT_2$
 ▷ $FT_2.\text{st} := \text{make_bin_op}("\times", FT_1.\text{st}, F.\text{ptr})$
 ▷ $FT_1.\text{ptr} := FT_2.\text{ptr}$

 $FT_1 \longrightarrow / \; F \; FT_2$
 ▷ $FT_2.\text{st} := \text{make_bin_op}("\div", FT_1.\text{st}, F.\text{ptr})$
 ▷ $FT_1.\text{ptr} := FT_2.\text{ptr}$

 $FT \longrightarrow \epsilon$
 ▷ $FT.\text{ptr} := FT.\text{st}$

 $F_1 \longrightarrow - \; F_2$
 ▷ $F_1.\text{ptr} := \text{make_un_op}("+/-", F_2.\text{ptr})$

 $F \longrightarrow (\; E \;)$
 ▷ $E.\text{ptr} := E.\text{ptr}$

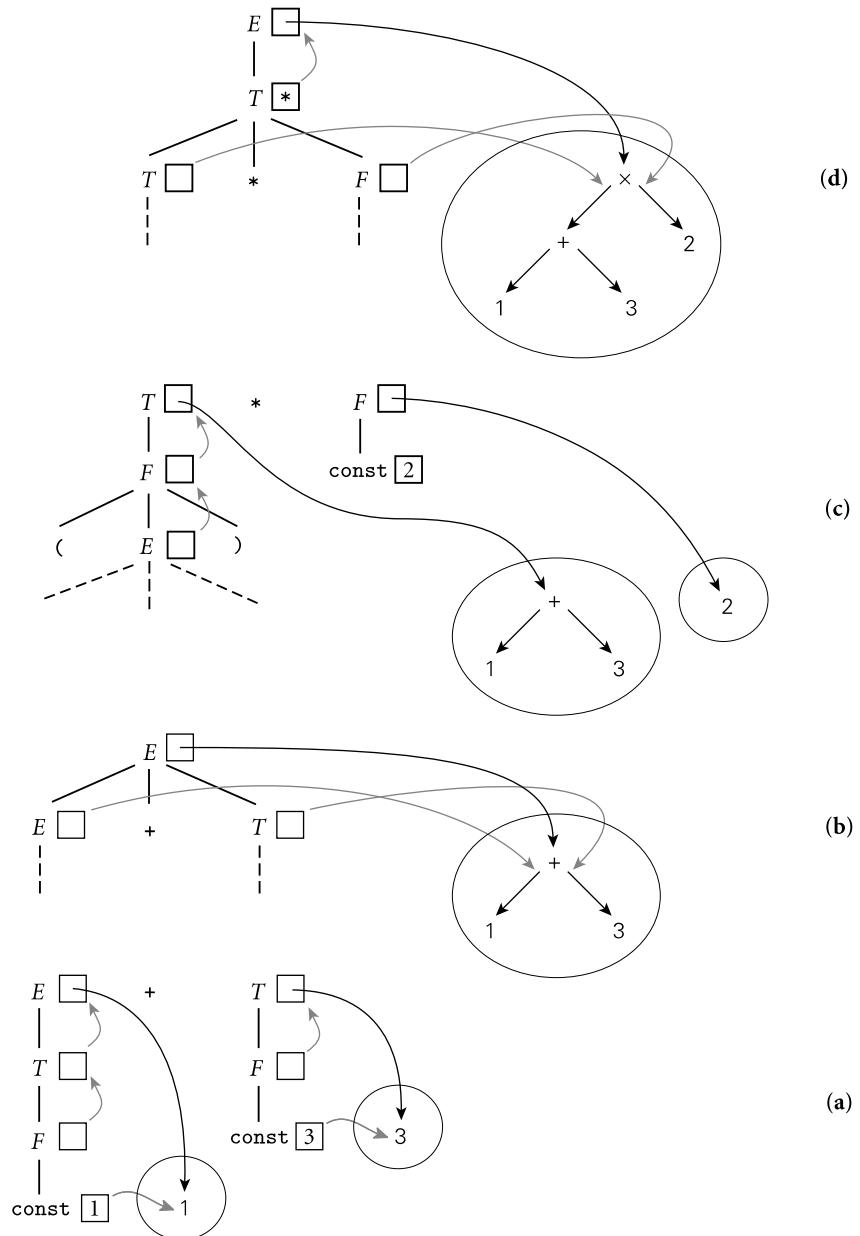
 $F \longrightarrow \text{const}$
 ▷ $F.\text{ptr} := \text{make_leaf}(\text{const}.val)$

```

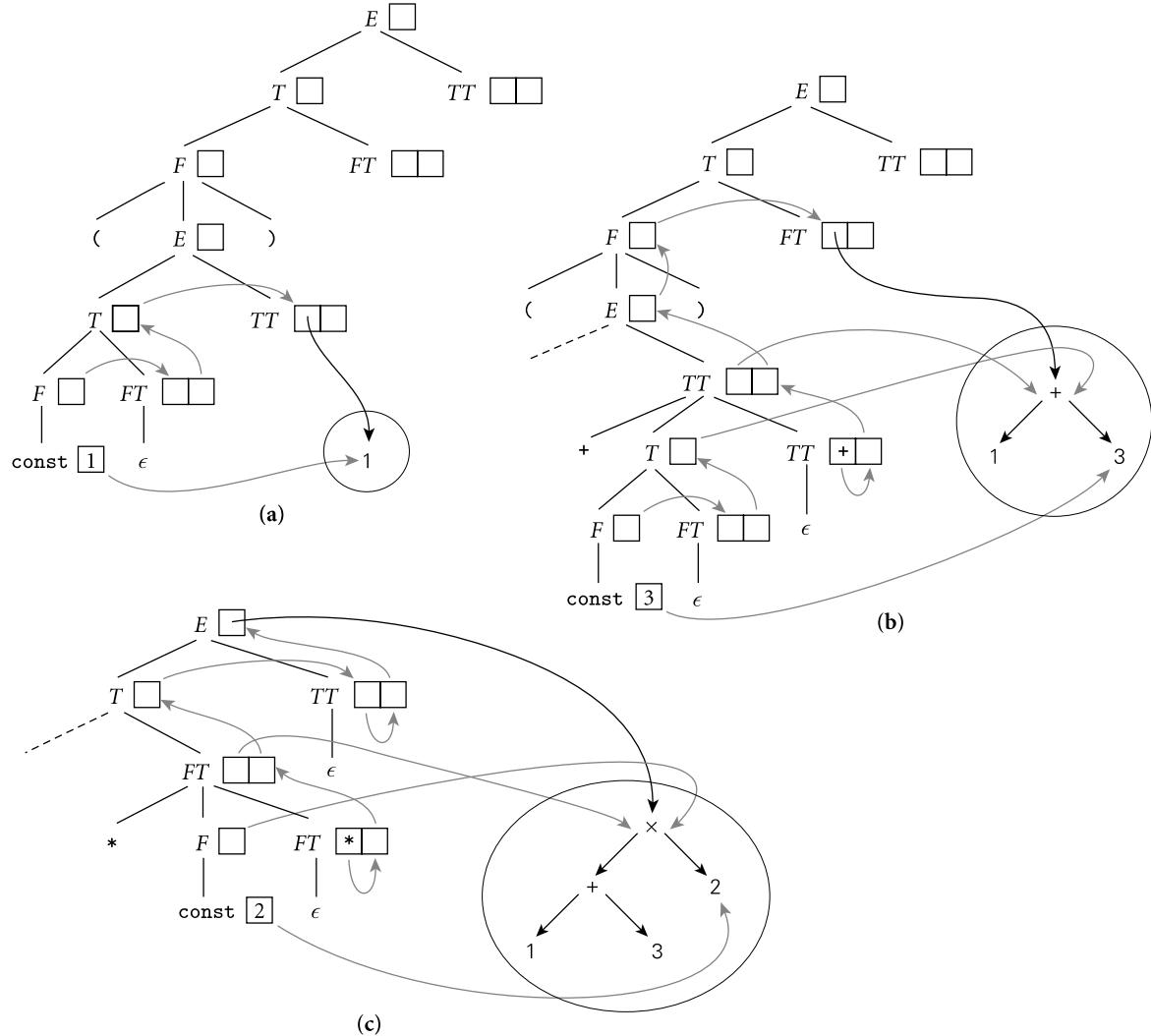
**Figure 4.6** Top-down attribute grammar to construct a syntax tree. Here the st attribute, like the ptr attribute (and unlike the st attribute of Figure 4.3), is a pointer to a syntax tree node.

### CHECK YOUR UNDERSTANDING

1. What determines whether a language rule is a matter of syntax or of static semantics?
2. Why is it impossible to detect certain program errors at compile time, even though they can be detected at run time?
3. What is an *attribute grammar*?
4. What are programming *assertions*? What is their purpose?
5. What is the difference between *synthesized* and *inherited* attributes?



**Figure 4.7** Construction of a syntax tree via decoration of a bottom-up parse tree, using the grammar of Figure 4.5. In diagram (a), the values of the constants 1 and 3 have been placed in new syntax tree leaves. Pointers to these leaves propagate up into the attributes of  $E$  and  $T$ . In (b), the pointers to these leaves become child pointers of a new internal  $+$  node. In (c) the pointer to this node propagates up into the attributes of  $T$ , and a new leaf is created for 2. Finally, in (d), the pointers from  $T$  and  $F$  become child pointers of a new internal  $\times$  node, and a pointer to this node propagates up into the attributes of  $E$ .



**Figure 4.8** Construction of a syntax tree via decoration of a top-down parse tree, using the grammar of Figure 4.6. In the top diagram, (a), the value of the constant 1 has been placed in a new syntax tree leaf. A pointer to this leaf then propagates to the st attribute of  $TT$ . In (b), a second leaf has been created to hold the constant 3. Pointers to the two leaves then become child pointers of a new internal + node, a pointer to which propagates from the st attribute of the bottom-most  $TT$ , where it was created, all the way up and over to the st attribute of the top-most  $FT$ . In (c), a third leaf has been created for the constant 2. Pointers to this leaf and to the + node then become the children of a new  $x$  node, a pointer to which propagates from the st of the lower  $FT$ , where it was created, all the way to the root of the tree.

6. Give two examples of information that is typically passed through inherited attributes.
7. What is *attribute flow*?
8. What is a *one-pass* compiler?
9. What does it mean for an attribute grammar to be *S-attributed*? *L-attributed*? *Noncircular*? What is the significance of these grammar classes?

## 4.4 Action Routines

Just as there are automatic tools that will construct a parser for a given context-free grammar, there are automatic tools that will construct a semantic analyzer (attribute evaluator) for a given attribute grammar. Attribute evaluator generators are heavily used in syntax-based editors [RT88], incremental compilers [SDB84], and programming language research. Most production compilers, however, use an ad hoc, handwritten translation scheme, interleaving parsing with at least the initial construction of a syntax tree, and possibly all of semantic analysis and intermediate code generation. Because they are able to evaluate the attributes of each production as it is parsed, they do not need to build the full parse tree.

An ad hoc translation scheme that is interleaved with parsing takes the form of a set of *action routines*. An action routine is a semantic function that the programmer (grammar writer) instructs the compiler to execute at a particular point in the parse. Most parser generators allow the programmer to specify action routines. In an LL parser generator, an action routine can appear anywhere within a right-hand side. A routine at the beginning of a right-hand side will be called as soon as the parser predicts the production. A routine embedded in the middle of a right-hand side will be called as soon as the parser has matched (the yield of) the symbol to the left. The implementation mechanism is simple: when

### DESIGN & IMPLEMENTATION

#### Attribute evaluators

Automatic evaluators based on formal attribute grammars are popular in language research projects because they save developer time when the language definition changes. They are popular in syntax-based editors and incremental compilers because they save execution time: when a small change is made to a program, the evaluator may be able to “patch up” tree decorations significantly faster than it could rebuild them from scratch. For the typical compiler, however, semantic analysis based on a formal attribute grammar is overkill: it has higher overhead than action routines, and doesn’t really save the compiler writer that much work.

```


$$\begin{aligned} E &\longrightarrow T \{ TT.st := T.ptr \} TT \{ E.ptr := TT.ptr \} \\ TT_1 &\longrightarrow + T \{ TT_2.st := \text{make_bin_op}("+", TT_1.st, T.ptr) \} TT_2 \{ TT_1.ptr := TT_2.ptr \} \\ TT_1 &\longrightarrow - T \{ TT_2.st := \text{make_bin_op}("-", TT_1.st, T.ptr) \} TT_2 \{ TT_1.ptr := TT_2.ptr \} \\ TT &\longrightarrow \epsilon \{ TT.ptr := TT.st \} \\ T &\longrightarrow F \{ FT.st := F.ptr \} FT \{ T.ptr := FT.ptr \} \\ FT_1 &\longrightarrow * F \{ FT_2.st := \text{make_bin_op}("\times", FT_1.st, F.ptr) \} FT_2 \{ FT_1.ptr := FT_2.ptr \} \\ FT_1 &\longrightarrow / F \{ FT_2.st := \text{make_bin_op}("\div", FT_1.st, F.ptr) \} FT_2 \{ FT_1.ptr := FT_2.ptr \} \\ FT &\longrightarrow \epsilon \{ FT.ptr := FT.st \} \\ F_1 &\longrightarrow - F_2 \{ F_1.ptr := \text{make_un_op}("+/-", F_2.ptr) \} \\ F &\longrightarrow (E) \{ F.ptr := E.ptr \} \\ F &\longrightarrow \text{const} \{ F.ptr := \text{make_leaf}(\text{const}.ptr) \} \end{aligned}$$


```

**Figure 4.9** LL(1) grammar with action routines to build a syntax tree.

it predicts a production, the parser pushes *all* of the right-hand side onto the stack—terminals (to be matched), nonterminals (to drive future predictions), and pointers to action routines. When it finds a pointer to an action routine at the top of the parse stack, the parser simply calls it.

#### EXAMPLE 4.11

Top-down action routines  
to build a syntax tree

To make this process more concrete, consider again our LL(1) grammar for constant expressions. Action routines to build a syntax tree while parsing this grammar appear in Figure 4.9. The only difference between this grammar and the one in Figure 4.6 is that the action routines (delimited here with curly braces) are embedded among the symbols of the right-hand sides; the work performed is the same. The ease with which the attribute grammar can be transformed into the grammar with action routines is due to the fact that the attribute grammar is L-attributed. If it required more complicated flow, we would not be able to cast it in the form of action routines. ■

#### Bottom-Up Evaluation

In an LR parser generator, one cannot in general embed action routines at arbitrary places in a right-hand side, since the parser does not in general know what production it is in until it has seen all or most of the yield. LR parser generators therefore permit action routines only after the point at which the production being parsed can be identified unambiguously (this is known as the *trailing part* of the right-hand side; the ambiguous part is the *left corner*). If the attribute flow of the action routines is strictly bottom-up (as it is in an S-attributed attribute grammar), then execution at the end of right-hand sides is all that is needed. The attribute grammars of Figures 4.1 and 4.5, in fact, are essentially identical to the action routine versions. If the action routines are responsible for a significant part of semantic analysis, however (as opposed to simply building a syntax

tree), then they will often need contextual information in order to do their job. To obtain and use this information in an LR parse, they will need some (necessarily limited) access to inherited attributes or to information outside the current production. We consider this issue further in Section  4.5.1.

## 4.5 Space Management for Attributes

Any attribute evaluation method requires space to hold the attributes of the grammar symbols. If we are building an explicit parse tree, then the obvious approach is to store attributes in the nodes of the tree themselves. If we are not building a parse tree, then we need to find a way to keep track of the attributes for the symbols we have seen (or predicted) but not yet finished parsing. The details differ in bottom-up and top-down parsers.

For a bottom-up parser with an S-attributed grammar, the obvious approach is to maintain an *attribute stack* that directly mirrors the parse stack: next to every state number on the parse stack is an attribute record for the symbol we shifted when we entered that state. Entries in the attribute stack are pushed and popped automatically by the parser driver; space management is not an issue for the writer of action routines. Complications arise if we try to achieve the effect of inherited attributes, but these can be accommodated within the basic attribute-stack framework.

For a top-down parser with an L-attributed grammar, we have two principal options. The first option is automatic, but more complex than for bottom-up grammars. It still uses an attribute stack, but one that does not mirror the parse stack. The second option has lower space overhead, and saves time by “short-cutting” copy rules, but requires action routines to allocate and deallocate space for attributes explicitly.

In both families of parsers, it is common for some of the contextual information for action routines to be kept in global variables. The symbol table in particular is usually global. We can be sure that the table will always represent the current referencing environment because we control the order in which action routines (including those that modify the environment at the beginnings and ends of scopes) are executed. In a pure attribute grammar we should need to pass symbol table information into and out of productions through inherited and synthesized attributes.

### IN MORE DEPTH

---

We consider attribute space management in more detail on the PLP CD. Using bottom-up and top-down grammars for arithmetic expressions, we illustrate automatic management for both bottom-up and top-down parsers, as well as the ad hoc option for top-down parsers.

---

```

program → stmt_list $$

stmt_list → stmt_list decl | stmt_list stmt | ε

decl → int id | real id

stmt → id := expr | read id | write expr

expr → term | expr add_op term

term → factor | term mult_op factor

factor → (expr) | id | int_const | real_const |
 float (expr) | trunc (expr)

add_op → + | -

mult_op → * | /

```

**Figure 4.10** Context-free grammar for a calculator language with types and declarations.  
The intent is that every identifier be declared before use, and that types not be mixed in computations.

## 4.6 Decorating a Syntax Tree

In our discussion so far we have used attribute grammars solely to decorate parse trees. As we mentioned in the chapter introduction, attribute grammars can also be used to decorate syntax trees. If our compiler uses action routines simply to build a syntax tree, then the bulk of semantic analysis and intermediate code generation will use the syntax tree as base.

Figure 4.10 contains a bottom-up CFG for a calculator language with types and declarations. The grammar differs from that of Example 2.35 (page 81) in three ways: (1) we allow declarations to be intermixed with statements, (2) we differentiate between integer and real constants (presumably the latter contain a decimal point), and (3) we require explicit conversions between integer and real operands. The intended semantics of our language requires that every identifier be declared before it is used, and that types not be mixed in computations. ■

### EXAMPLE 4.12

Bottom-up CFG for calculator language with types

### EXAMPLE 4.13

Syntax tree to average an integer and a real

### EXAMPLE 4.14

Tree grammar for the calculator language with types

Extrapolating from the example in Figure 4.5, it is easy to add semantic functions or action routines to the grammar of Figure 4.10 to construct a syntax tree for the calculator language (Exercise 4.19). The obvious structure for such a tree would represent expressions as we did in Figure 4.7, and would represent a program as a linked list of declarations and statements. As a concrete example, Figure 4.11 contains the syntax tree for a simple program to print the average of an integer and a real. ■

Much as a context-free grammar describes the possible structure of parse trees for a given programming language, we can use a *tree grammar* to represent the possible structure of syntax trees. As in a CFG, each production of a tree grammar represents a possible relationship between a parent and its children in the tree. The parent is the symbol on the left-hand side of the production; the children are

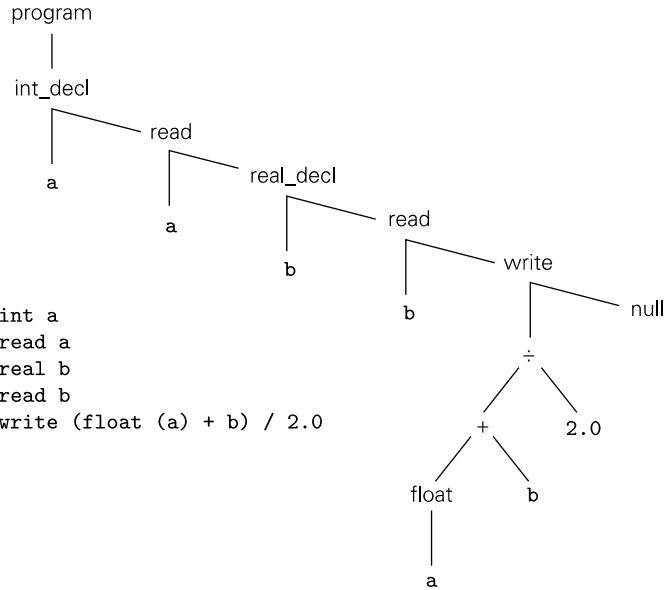


Figure 4.11 Syntax tree for a simple calculator program.

the symbols on the right-hand side. The productions used in Figure 4.11 might look something like this:

```

program —> item
int_decl : item —> id item
read : item —> id item
real_decl : item —> id item
write : item —> expr item
null : item —> ε
‘÷’ : expr —> expr expr
‘+’ : expr —> expr expr
float : expr —> expr
id : expr —> ε
real_const : expr —> ε

```

The notation  $A : B$  on the left-hand side of a production means that  $A$  is one kind of  $B$ , and may appear anywhere a  $B$  is expected on a right-hand side. ■

Tree grammars and context-free grammars differ in important ways. A context-free grammar is meant to define (generate) a language composed of strings of tokens, where each string is the fringe (yield) of a parse tree. Parsing is the process of finding a tree that has a given yield. A tree grammar, as we use it here, is meant to define (or generate) the trees themselves. We have no need for a notion of parsing: we can easily inspect a tree and determine whether (and how) it can

be generated by the grammar. Our purpose in introducing tree grammars is to provide a framework for the decoration of syntax trees. Semantic rules attached to the productions of a tree grammar can be used to define the attribute flow of a syntax tree in exactly the same way that semantic rules attached to the productions of a context-free grammar are used to define the attribute flow of a parse tree. We will use a tree grammar in the remainder of this section to perform static semantic checking. In Chapter 14 we will show how additional semantic rules can be used to generate intermediate code.

**EXAMPLE 4.15**

Tree AG for the calculator language with types

Figure 4.12 contains a complete tree attribute grammar for our calculator language with types. Once decorated, the *program* node at the root of the syntax tree will contain a list, in a synthesized attribute, of all static semantic errors in the program. (The list will be empty if the program is free of such errors.) Each *item* or *expr* node has an inherited attribute *syntab* that contains a list, with types, of all identifiers declared to the left in the tree. Each *item* node also has an inherited attribute *errors\_in* that lists all static semantic errors found to its left in the tree, and a synthesized attribute *errors\_out* to propagate the final error list back to the root. Each *expr* node has one synthesized attribute that indicates its type and another that contains a list of any static semantic errors found inside.

Our handling of semantic errors illustrates a common technique. In order to continue looking for other errors we must provide values for any attributes that would have been set in the absence of an error. To avoid cascading error messages, we choose values for those attributes that will pass quietly through subsequent checks. In our specific example we employ a pseudo-type called *error*, which we associate with any symbol table entry or expression for which we have already generated a message.

In our example grammar we accumulate error messages into a synthesized attribute of the root of the syntax tree. In an ad hoc attribute evaluator we might be tempted to print these messages on the fly as the errors are discovered. In practice, however, particularly in a multipass compiler, it makes sense to buffer the messages so they can be interleaved with messages produced by other phases of the compiler and printed in program order at the end of compilation.

Though it takes a bit of checking to verify the fact, our attribute grammar is noncircular and well defined. No attribute is ever assigned a value more than once. (The helper routines in Figure 4.12 should be thought of as macros rather than semantic functions. For the sake of brevity we have passed them entire tree nodes as arguments. Each macro calculates the values of two different attributes. Under a strict formulation of attribute grammars each macro would be replaced by two separate semantic functions, one per calculated attribute.)

One could convert our attribute grammar into executable code using an automatic attribute evaluator generator. Alternatively, one could create an ad hoc evaluator in the form of mutually recursive subroutines (Exercise 4.18). In the latter case attribute flow would be explicit in the calling sequence of the routines. We could then choose if desired to keep the symbol table in global variables,

```

program → item
 ▷ item.symtab := nil
 ▷ program.errors := item.errors_out
 ▷ item.errors_in := nil

int_decl : item1 → id item2
 ▷ declare_name(id, item1, item2, int)
 ▷ item1.errors_out := item2.errors_out

real_decl : item1 → id item2
 ▷ declare_name(id, item1, item2, real)
 ▷ item1.errors_out := item2.errors_out

read : item1 → id item2
 ▷ item2.symtab := item1.symtab
 ▷ if ⟨id.name, ?⟩ ∈ item1.symtab
 item2.errors_in := item1.errors_in
 else
 item2.errors_in := item1.errors_in + [⟨id.name “undefined at” id.location]]
 ▷ item1.errors_out := item2.errors_out

write : item1 → expr item2
 ▷ expr.symtab := item1.symtab
 ▷ item2.symtab := item1.symtab
 ▷ item2.errors_in := item1.errors_in + expr.errors
 ▷ item1.errors_out := item2.errors_out

‘:=’ : item1 → id expr item2
 ▷ expr.symtab := item1.symtab
 ▷ item2.symtab := item1.symtab
 ▷ if ⟨id.name, A⟩ ∈ item1.symtab -- for some type A
 if A ≠ error and expr.type ≠ error and A ≠ expr.type
 item2.errors_in := item1.errors_in + [“type clash at” item1.location]
 else
 item2.errors_in := item1.errors_in
 else
 item2.errors_in := item1.errors_in + [⟨id.name “undefined at” id.location]]

null : item → ε
 ▷ item.errors_out := item.errors_in

```

**Figure 4.12 Attribute grammar to decorate an abstract syntax tree for the calculator language with types.** We use square brackets to delimit error messages and pointed brackets to delimit symbol table entries. Juxtaposition indicates concatenation within error messages; the ‘+’ and ‘-’ operators indicate insertion and removal in lists. We assume that every node has been initialized by the scanner or by action routines in the parser to contain an indication of the location (line and column) at which the corresponding construct appears in the source (see Exercise 4.20). The ‘?’ symbol is used as a “wild card”; it matches any type. (continued)

```

id : expr → ε
 ▷ if ⟨id.name, A⟩ ∈ expr.symtab -- for some type A
 expr.errors := nil
 expr.type := A
 else
 expr.errors := [id.name “undefined at” id.location]
 expr.type := error

int_const : expr → ε
 ▷ expr.type := int

real_const : expr → ε
 ▷ expr.type := real

‘+’ : expr1 → expr2 expr3
 ▷ expr2.symtab := expr1.symtab
 ▷ expr3.symtab := expr1.symtab
 ▷ check_types(expr1, expr2, expr3)

‘-’ : expr1 → expr2 expr3
 ▷ expr2.symtab := expr1.symtab
 ▷ expr3.symtab := expr1.symtab
 ▷ check_types(expr1, expr2, expr3)

‘×’ : expr1 → expr2 expr3
 ▷ expr2.symtab := expr1.symtab
 ▷ expr3.symtab := expr1.symtab
 ▷ check_types(expr1, expr2, expr3)

‘÷’ : expr1 → expr2 expr3
 ▷ expr2.symtab := expr1.symtab
 ▷ expr3.symtab := expr1.symtab
 ▷ check_types(expr1, expr2, expr3)

float : expr1 → expr2
 ▷ expr2.symtab := expr1.symtab
 ▷ convert_type(expr2, expr1, int, real, “float of non-int”)

trunc : expr1 → expr2
 ▷ expr2.symtab := expr1.symtab
 ▷ convert_type(expr2, expr1, real, int, “trunc of non-real”)

```

**Figure 4.12 (continued on next page)**

rather than passing it from node to node through attributes. Most compilers employ the ad hoc approach.

### CHECK YOUR UNDERSTANDING

10. What is the difference between a semantic function and an action routine?
11. Why can't action routines be placed at arbitrary locations within the right-hand side of productions in an LR CFG?
12. What patterns of attribute flow can be captured easily with action routines?

```

macro declare_name(id, cur_item, next_item : syntax_tree_node; t : type)
 if ⟨id.name, ?⟩ ∈ cur_item.symtab
 next_item.errors_in := cur_item.errors_in + [“redefinition of” id.name “at” cur_item.location]
 next_item.symtab := cur_item.symtab – ⟨id.name, ?⟩ + ⟨id.name, error⟩
 else
 next_item.errors_in := cur_item.errors_in
 next_item.symtab := cur_item.symtab + ⟨id.name, t⟩

macro check_types(result, operand1, operand2)
 if operand1.type = error or operand2.type = error
 result.type := error
 result.errors := operand1.errors + operand2.errors
 else if operand1.type ≠ operand2.type
 result.type := error
 result.errors := operand1.errors + operand2.errors + [“type clash at” result.location]
 else
 result.type := operand1.type
 result.errors := operand1.errors + operand2.errors

macro convert_type(old_expr, new_expr : syntax_tree_node; from_t, to_t : type; msg : string)
 if old_expr.type = from_t or old_expr.type = error
 new_expr.errors := old_expr.errors
 new_expr.type := to_t
 else
 new_expr.errors := old_expr.errors + [msg “at” old_expr.location]
 new_expr.type := error

```

**Figure 4.12 (continued)**

13. Some compilers perform all semantic checks and intermediate code generation in action routines. Others use action routines to build a syntax tree and then perform semantic checks and intermediate code generation in separate traversals of the syntax tree. Discuss the tradeoffs between these two strategies.
14. What sort of information do action routines typically keep in global variables, rather than in attributes?
15. Describe the similarities and differences between context-free grammars and tree grammars.
16. How can a semantic analyzer avoid the generation of cascading error messages?

## 4.7 Summary and Concluding Remarks

This chapter has discussed the task of semantic analysis. We reviewed the sorts of language rules that can be classified as syntax, static semantics, and dynamic se-

mantics, and discussed the issue of whether to generate code to perform dynamic semantic checks. We also considered the role that the semantic analyzer plays in a typical compiler. We noted that both the enforcement of static semantic rules and the generation of intermediate code can be cast in terms of annotation, or *decoration*, of a parse tree or syntax tree. We then presented attribute grammars as a formal framework for this decoration process.

An attribute grammar associates *attributes* with each symbol in a context-free grammar or tree grammar, and *attribute rules* with each production. *Synthesized* attributes are calculated only in productions in which their symbol appears on the left-hand side. The synthesized attributes of tokens are initialized by the scanner. *Inherited* attributes are calculated in productions in which their symbol appears within the right-hand side; they allow calculations internal to a symbol to depend on the context in which the symbol appears. Inherited attributes of the start symbol (goal) can represent the external environment of the compiler. Strictly speaking, attribute grammars allow only *copy rules* (assignments of one attribute to another) and simple calls to *semantic functions*, but we usually relax this restriction to allow more or less arbitrary code fragments in some existing programming language.

Just as context-free grammars can be categorized according to the parsing algorithm(s) that can use them, attribute grammars can be categorized according to the complexity of their pattern of *attribute flow*. S-attributed grammars, in which all attributes are synthesized, can naturally be evaluated in a single bottom-up pass over a parse tree, in precisely the order the tree is discovered by an LR-family parser. L-attributed grammars, in which all attribute flow is depth-first left-to-right, can be evaluated in precisely the order that the parse tree is predicted and matched by an LL-family parser. Attribute grammars with more complex patterns of attribute flow are not commonly used in production compilers but are valuable for syntax-based editors, incremental compilers, and various other tools.

While it is possible to construct automatic tools to analyze attribute flow and decorate parse trees, most compilers rely on *action routines*, which the compiler writer embeds in the right-hand sides of productions to evaluate attribute rules at specific points in a parse. In an LL-family parser, action routines can be embedded at arbitrary points in a production's right-hand side. In an LR-family parser, action routines must follow the production's *left corner*. Space for attributes in a bottom-up compiler is naturally allocated in parallel with the parse stack. Inherited attributes must be “faked” by accessing the synthesized attributes of symbols known to lie below the current production in the stack. Space for attributes in a top-down compiler can be allocated automatically, or managed explicitly by the writer of action routines. The automatic approach has the advantage of regularity, and is easier to maintain; the ad hoc approach is slightly faster and more flexible.

In a *one-pass* compiler, which interleaves scanning, parsing, semantic analysis, and code generation in a single traversal of its input, semantic functions or action routines are responsible for all of semantic analysis and code generation. More

commonly, action routines simply build a syntax tree, which is then decorated during separate traversal(s) in subsequent pass(es).

In subsequent chapters (6–9 in particular) we will consider a wide variety of programming language constructs. Rather than present the actual attribute grammars required to implement these constructs, we will describe their semantics informally, and give examples of the target code. We will return to attribute grammars in Chapter 14, when we consider the generation of intermediate code in more detail.

## 4.8 Exercises

- 4.1 Basic results from automata theory tell us that the language  $L = a^n b^n c^n = \epsilon, abc, aabbcc, aaabbbccc, \dots$  is not context free. It can be captured, however, using an attribute grammar. Give an underlying CFG and a set of attribute rules that associate a Boolean attribute `ok` with the root `R` of each parse tree, such that `R.ok = true` if and only if the string corresponding to the fringe of the tree is in  $L$ .
- 4.2 Modify the grammar of Figure 2.24 so that it accepts only programs that contain at least one `write` statement. Make the same change in the solution to Exercise 2.12. Based on your experience, what do you think of the idea of using the CFG to enforce the rule that every function in C must contain at least one `return` statement?
- 4.3 Give two examples of reasonable semantic rules that *cannot* be checked at reasonable cost, either statically or by compiler-generated code at run time.
- 4.4 Write an S-attributed attribute grammar, based on the CFG of Example 4.6, that accumulates the value of the overall expression into the root of the tree. You will need to use dynamic memory allocation so that individual attributes can hold an arbitrary amount of information.
- 4.5 As we shall learn in Chapter 10, Lisp programs take the form of parenthesized lists. The natural syntax tree for a Lisp program is thus a tree of binary cells (known in Lisp as `cons` cells), where the first child represents the first element of the list and the second child represents the rest of the list. The syntax tree for `(cdr '(a b c))` appears in Figure 4.13. (The notation `'L` is syntactic sugar for `(quote L)`.)

Extend the CFG of Exercise 2.13 to create an attribute grammar that will build such trees. When a parse tree has been fully decorated, the root should have an attribute `v` that refers to the syntax tree. You may assume that each atom has a synthesized attribute `v` that refers to a syntax tree node that holds information from the scanner. In your semantic functions, you may assume the availability of a `cons` function that takes two references as arguments and returns a reference to a new `cons` cell containing those references.

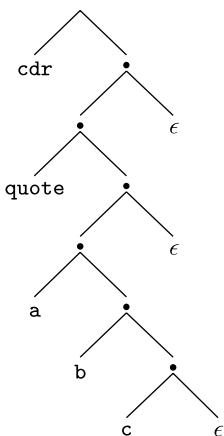


Figure 4.13 Natural syntax tree for the Lisp expression `(cdr '(a b c))`.

- 4.6 Suppose that we want to translate constant expressions into the postfix or “reverse Polish” notation of logician Jan Łukasiewicz. Postfix notation does not require parentheses. It appears in stack-based languages such as Postscript, Forth, and the P-code and Java byte code intermediate forms mentioned in Section 1.4. It also serves as the input language of certain Hewlett-Packard (HP) brand calculators. When given a number, an HP calculator pushes it onto an internal stack. When given an operator, it pops the top two numbers, applies the operator, and pushes the result. The display shows the value at the top of the stack. To compute  $2 \times (5 - 3)/4$  one would enter `2 5 3 - * 4 /`.

Using the underlying CFG of Figure 4.1, write an attribute grammar that will associate with the root of the parse tree a sequence of calculator button pushes, `seq`, that will compute the arithmetic value of the tokens derived from that symbol. You may assume the existence of a function buttons (`c`) that returns a sequence of button pushes (ending with `ENTER` on an HP calculator) for the constant `c`. You may also assume the existence of a concatenation function for sequences of button pushes.

- 4.7 Repeat the previous exercise using the underlying CFG of Figure 4.3.  
 4.8 Consider the following grammar for reverse Polish arithmetic expressions:

$$\begin{aligned} E &\longrightarrow E \; E \; op \mid id \\ op &\longrightarrow + \mid - \mid * \mid / \end{aligned}$$

Assuming that each `id` has a synthesized attribute name of type string, and that each `E` and `op` has an attribute `val` of type string, write an attribute grammar that arranges for the `val` attribute of the root of the parse tree to contain a translation of the expression into conventional infix notation. For example, if the leaves of the tree, left to right, were “A A B − ∗ C /”, then the `val` field of the root would be “( ( A ∗ ( A − B ) ) / C )”. As an

extra challenge, write a version of your attribute grammar that exploits the usual arithmetic precedence and associativity rules to use as few parentheses as possible.

- 4.9 To reduce the likelihood of typographic errors, the digits comprising most credit card numbers are designed to satisfy the so-called *Luhn formula*, standardized by ANSI in the 1960s and named for IBM mathematician Hans Peter Luhn. Starting at the right, we double every other digit (the second-to-last, fourth-to-last, etc.). If the doubled value is 10 or more, we add the resulting digits. We then sum together all the digits. In any valid number the result will be a multiple of 10. For example, 1234 5678 9012 3456 becomes 2264 1658 9022 6416, which sums to 64, so this is not a valid number. If the last digit had been 2, however, the sum would have been 60, so the number would potentially be valid.

Give an attribute grammar for strings of digits that accumulates into the root of the parse tree a Boolean value indicating whether the string is valid according to Luhn's formula. Your grammar should accommodate strings of arbitrary length.

- 4.10 Consider the following CFG for floating-point constants, without exponential notation. (Note that this exercise is somewhat artificial: the language in question is regular, and would be handled by the scanner of a typical compiler.)

$$\begin{aligned} C &\longrightarrow \text{digits} \ . \ \text{digits} \\ \text{digits} &\longrightarrow \text{digit} \ \text{more\_digits} \\ \text{more\_digits} &\longrightarrow \text{digits} \ | \ \epsilon \\ \text{digit} &\longrightarrow 0 \ | \ 1 \ | \ 2 \ | \ 3 \ | \ 4 \ | \ 5 \ | \ 6 \ | \ 7 \ | \ 8 \ | \ 9 \end{aligned}$$

Augment this grammar with attribute rules that will accumulate the value of the constant into a `val` attribute of the root of the parse tree. Your answer should be S-attributed.

- 4.11 One potential criticism of the obvious solution to the previous problem is that the values in internal nodes of the parse tree do not reflect the value, in context, of the fringe below them. Create an alternative solution that addresses this criticism. More specifically, create your grammar in such a way that the `val` of an internal node is the sum of the `vals` of its children. Illustrate your solution by drawing the parse tree and attribute flow for 12.34. (*Hint:* You will probably want a different underlying CFG, and non-L-attributed flow.)

- 4.12 Consider the following attribute grammar for type declarations, based on the CFG of Exercise 2.8.

$$\begin{aligned} \text{decl} &\longrightarrow \text{ID} \ \text{decl\_tail} \\ &\triangleright \text{decl.t} := \text{decl\_tail.t} \\ &\triangleright \text{decl\_tail.in\_tab} := \text{insert}(\text{decl.in\_tab}, \text{ID.n}, \text{decl\_tail.t}) \\ &\triangleright \text{decl.out\_tab} := \text{decl\_tail.out\_tab} \end{aligned}$$

```


$$\begin{array}{l} decl_tail \longrightarrow , \ decl \\ \quad \triangleright \ decl_tail.t := decl.t \\ \quad \triangleright \ decl.in_tab := decl.tail.in_tab \\ \quad \triangleright \ decl.tail.out_tab := decl.out_tab \\ decl_tail \longrightarrow : ID ; \\ \quad \triangleright \ decl_tail.t := ID.n \\ \quad \triangleright \ decl.tail.out_tab := decl.tail.in_tab \end{array}$$


```

Show a parse tree for the string A, B : C;. Then, using arrows and textual description, specify the attribute flow required to fully decorate the tree. (*Hint:* Note that the grammar is *not* L-attributed.)

- 4.13 A CFG-based attribute evaluator capable of handling non-L-attributed attribute flow needs to take a parse tree as input. Explain how to build a parse tree automatically during a top-down or bottom-up parse (i.e., without explicit action routines).
- 4.14 Write an LL(1) grammar with action routines and automatic attribute space management that generates the reverse Polish translation described in Exercise 4.6.
- 4.15 (a) Write a context-free grammar for polynomials in  $x$ . Add semantic functions to produce an attribute grammar that will accumulate the polynomial's derivative (as a string) in a synthesized attribute of the root of the parse tree.  
 (b) Replace your semantic functions with action routines that can be evaluated during parsing.
- 4.16 (a) Write a context-free grammar for `case` or `switch` statements in the style of Pascal or C. Add semantic functions to ensure that the same label does not appear on two different arms of the construct.  
 (b) Replace your semantic functions with action routines that can be evaluated during parsing.
- 4.17 Write an algorithm to determine whether the rules of an arbitrary attribute grammar are noncircular. (Your algorithm will require exponential time in the worst case [JOR75].)
- 4.18 Rewrite the attribute grammar of Figure 4.12 in the form of an ad hoc tree traversal consisting of mutually recursive subroutines in your favorite programming language. Keep the symbol table in a global variable, rather than passing it through arguments.
- 4.19 Write an attribute grammar based on the CFG of Figure 4.10 that will build a syntax tree with the structure described in Figure 4.12.
- 4.20 Augment the attribute grammar of Figure 4.5, Figure 4.6, or Exercise 4.19 to initialize a synthesized attribute in every syntax tree node that indicates the location (line and column) at which the corresponding construct appears in

the source program. You may assume that the scanner initializes the location of every token.

- 4.21 Modify the CFG and attribute grammar of Figures 4.10 and 4.12 to permit mixed integer and real expressions, without the need for `float` and `trunc`. You will want to add an annotation to any node that must be coerced to the opposite type, so that the code generator will know to generate code to do so. Be sure to think carefully about your coercion rules. In the expression `my_int + my_real`, for example, how will you know whether to coerce the integer to be a real or to coerce the real to be an integer?
- 4.22 Explain the need for the  $A : B$  notation on the left-hand sides of productions in a tree grammar. Why isn't similar notation required for context-free grammars?

© 4.23–4.27 In More Depth.

## 4.9 Explorations

- 4.28 One of the most influential applications of attribute grammars was the Cornell Synthesizer Generator [Rep84, RT88], now available commercially from [grammatech.com](http://grammatech.com). Learn how the Generator uses attribute grammars not only for incremental update of semantic information in a program under edit, but also for automatic creation of language based editors from formal language specifications. How general is this technique? What applications might it have beyond syntax-directed editing of computer programs?
- 4.29 The attribute grammars used in this chapter are all quite simple. Most are S- or L-attributed. All are noncircular. Are there any practical uses for more complex attribute grammars? How about automatic attribute evaluators? Using the Bibliographic Notes as a starting point, conduct a survey of attribute evaluation techniques. Where is the line between practical techniques and intellectual curiosities?
- 4.30 The first validated Ada implementation was the Ada/Ed interpreter from New York University [DGAFS<sup>+</sup>80]. The interpreter was written in the set-based language SETL [SDDS86] using a denotational semantics definition of Ada. Learn about the Ada/Ed project, SETL, and denotational semantics. Discuss how the use of a formal definition aided the development process. Also discuss the limitations of Ada/Ed, and expand on the potential role of formal semantics in language design, development, and prototype implementation.
- 4.31 The Scheme language manual [ADH<sup>+</sup>98] includes a formal definition of Scheme in denotational semantics. How long is this definition compared to the more conventional definition in English? How readable is it? What

do the length and the level of readability say about Scheme? About denotational semantics? (For more on denotational semantics, see the texts of Stoy [Sto77] or Gordon [Gor79].)

© 4.32–4.33 In More Depth.

## 4.10 Bibliographic Notes

Much of the early theory of attribute grammars was developed by Knuth [Knu68]. Lewis, Rosenkrantz, and Stearns [LRS74] introduced the notion of an L-attributed grammar. Watt [Wat77] showed how to use marker symbols to emulate inherited attributes in a bottom-up parser. Jazayeri, Ogden, and Rounds [JOR75] showed that exponential time may be required in the worst case to decorate a parse tree with arbitrary attribute flow. Articles by Courcelle [Cou84] and Engelfriet [Eng84] survey the theory and practice of attribute evaluation. The best-known attribute grammar system for language-based editing is the Synthesizer Generator [RT88] (a follow-on to the language-specific Cornell Program Synthesizer [TR81]) of Reps and Teitelbaum. Magpie [SDB84] is an incremental compiler. Action routines to implement many language features can be found in the texts of Fischer and LeBlanc [FL88] or Appel [App97]. Further notes on attribute grammars can be found in the texts of Cooper and Torczon [CT04, pp. 171–188] or Aho, Sethi, and Ullman [ASU86, pp. 340–342].

Marcotty, Ledgard, and Bochmann [MLB76] provide a survey of formal notations for programming language semantics. The seminal paper on axiomatic semantics is by Hoare [Hoa69]. An excellent book on the subject is Gries's *The Science of Programming* [Gri81]. The seminal paper on denotational semantics is by Scott and Strachey [SS71]. Texts on the subject include those of Stoy [Sto77] and Gordon [Gor79].

# 5

## Target Machine Architecture

**As described in Chapter 1**, a compiler is simply a translator. It translates programs written in one language into programs written in another language. This second language can be almost anything—some other high-level language, phototypesetting commands, VLSI (chip) layouts—but most of the time it’s the machine language for some available computer.

Just as there are many different programming languages, there are many different machine languages, though the latter tend to display considerably less diversity than the former. Each machine language corresponds to a different *processor architecture*. Formally, an architecture is the interface between the hardware and the software: the language generated by a compiler, or by a programmer writing for the bare machine. The *implementation* of the processor is a concrete realization of the architecture, generally in hardware. This chapter provides a brief overview of those aspects of processor architecture and implementation of particular importance to compiler writers, and may be worth reviewing even by readers who have seen the material before.

To generate correct code, it suffices for a compiler writer to understand the target architecture. To generate *fast* code, it is generally necessary to understand the implementation as well, because it is the implementation that determines the relative speeds of alternative translations of a given language construct.

Processor implementations change over time, as people invent better ways of doing things, and as technological advances (e.g., increases in the number of transistors that will fit on one chip) make things feasible that were not feasible before. Processor architectures also change, for at least two reasons. Some technological advances can be exploited only by changing the hardware/software interface—for example, by increasing the number of bits that can be added or multiplied in a single instruction. In addition, experience with compilers and applications often suggests that certain new instructions would make programs simpler or faster. Occasionally, technological and intellectual trends converge to produce a revolutionary change in both architecture and implementation. We

will discuss three such changes in Section 5.4: the development of microprogramming in the early 1960s, the development of the microprocessor in the early to mid-1970s, and the development of RISC machines in the early 1980s. As this book goes to press it appears we may be on the cusp of a fourth revolution, as vendors turn to multithreaded and multiprocessor chips in an attempt to increase computational power per watt of heat output.

Most of the discussion in this chapter, and indeed in the rest of the book, will assume that we are compiling for a modern RISC (reduced instruction set computer) architecture. Roughly speaking, a RISC machine is one that sacrifices richness in the instruction set in order to increase the number of instructions that can be executed per second. Where appropriate, we will devote a limited amount of attention to earlier, CISC (complex instruction set computer) architectures. The most popular desktop processor in the world—the x86—is a legacy CISC design, but RISC dominates among newer designs. Modern implementations of the x86 generally run fastest if compilers restrict themselves to a relatively simple subset of the instruction set. Within the processor, a hardware “front end” translates these instructions, on the fly, into a RISC-like internal format.

In the first three sections that follow, we consider the hierarchical organization of memory, the types (formats) of data found in memory, and the instructions used to manipulate those data. The coverage is necessarily somewhat cursory and high-level; much more detail can be found in books on computer architecture (e.g., in Chapter 2 of Hennessy and Patterson’s outstanding text [HP03]).

We consider the interplay between architecture and implementation in Section 5.4. In a supplemental subsection on the PLP CD, we illustrate the differences between CISC and RISC machines using the x86 and MIPS instruction sets as examples. Finally, in Section 5.5, we consider some of the issues that make compiling for modern processors a challenging task.

## 5.1 The Memory Hierarchy

Memory on most machines consists of a numbered sequence of eight-bit bytes. It is not uncommon for modern workstations to contain several gigabytes of memory—much too much to fit on the same chip as the processor. Because memory is off-chip (typically on the other side of a bus), getting at it is much slower than getting at things on-chip. Most computers therefore employ a *memory hierarchy*, in which things that are used more often are kept close at hand. A typical memory hierarchy, with access times and capacities, is shown in Figure 5.1. ■

### EXAMPLE 5.1

Memory hierarchy stats

Only three of the levels of the memory hierarchy—registers, memory, and devices—are a visible part of the hardware/software interface. Compilers manage registers explicitly, loading them from memory when needed and storing them back to memory when done, or when the registers are needed for something else.

|                               | typical access time | typical capacity      |
|-------------------------------|---------------------|-----------------------|
| registers                     | 0.2–0.5ns           | 256–1024 bytes        |
| primary (L1) cache            | 0.4–1ns             | 32K–256K bytes        |
| secondary (L2) cache          | 4–10ns              | 512K–2M bytes         |
| tertiary (off-chip, L3) cache | 10–50ns             | 4–64M bytes           |
| main memory                   | 50–500ns            | 256M–16G bytes        |
| disk                          | 5–15ms              | 80G bytes and up      |
| tape                          | 1–50s               | effectively unlimited |

**Figure 5.1 The memory hierarchy of a workstation-class computer.** Access times and capacities are approximate, based on 2005 technology. Registers must be accessed within a single clock cycle. Primary cache typically responds in 1–2 cycles; off-chip cache in more like 20 cycles. Main memory on a supercomputer can be as fast as off-chip cache; on a workstation it is typically much slower. Disk and tape times are constrained by the movement of physical parts.

Caches are managed by the hardware. Devices are generally accessed only by the operating system.

Registers hold small amounts of data that can be accessed very quickly. A typical RISC machine has two sets of registers, to hold integer and floating-point operands. It also has several special purpose registers, including the *program counter* (PC) and the *processor status register*. The program counter holds the address of the next instruction to be executed. It is incremented automatically when fetching most instructions; branches work by changing it explicitly. The processor status register contains a variety of bits of importance to the operating system (privilege level, interrupt priority level, trap enable bits) and, on some machines, a few bits of importance to the compiler writer. Principal among these are *condition codes*, which indicate whether the most recent arithmetic or logical operation resulted in a zero, a negative value, and/or arithmetic overflow. (We will consider condition codes in more detail in Section 5.3.2.)

Because registers can be accessed every cycle, whereas memory, generally, cannot, good compilers expend a great deal of effort trying to make sure that the data they need most often are in registers, and trying to minimize the amount of time spent moving data back and forth between registers and memory. We will consider algorithms for register management in Section 5.5.2.

Caches are generally smaller but faster than main memory. They are designed to exploit *locality*: the tendency of most computer programs to access the same or nearby locations in memory repeatedly. By automatically moving the contents of these locations into cache, a hierarchical memory system can dramatically improve performance. The idea makes intuitive sense: loops tend to access the same local variables in every iteration, and to walk sequentially through arrays. Instructions, likewise, tend to be loaded from consecutive locations, and code that accesses one element of a structure (or member of a class) is likely to access another.

Primary caches, also known as *level-1 (L1) caches*, are typically located on the same chip as the processor, and usually come in pairs—one for instructions (the

L1 I-cache) and another for data (the L1 D-cache), both of which can be accessed every cycle. Secondary caches are larger and slower, but still faster than main memory. In a modern desktop or laptop system they are typically also on the same chip as the processor. High-end desktop or server-class machines may have an off-chip tertiary (L3) cache as well. Small embedded processors may have a single level of on-chip cache, with or without any off-chip cache. Caches are managed entirely in hardware on most machines, but compilers can increase their effectiveness by generating code with a high degree of locality.

A memory access that finds its data in the cache is said to be a *cache hit*. An access that does not find its data in the cache is said to be a *cache miss*. On a miss, the hardware automatically loads a *line* of the cache with a contiguous block of data containing the requested location, obtained from the next lower level of cache or main memory. (Cache lines vary from as few as 8 to as many as 512 bytes in length.) Assuming that the cache was already full, the load will displace some other line, which is written back to memory if it has been modified.

A final characteristic of memory that is important to the compiler is known as *data alignment*. Most machines are able to manipulate operands of several sizes, typically one, two, four, and eight bytes. Most modern instruction sets refer to these as byte, half-word, word, and double-word operands, respectively; on the x86 they are byte, word, double-word, and quad-word operands. Most recent architectures require  $n$ -byte operands to appear in memory at addresses that are evenly divisible by  $n$ . Integers, for example, which typically occupy four bytes, must appear at a location whose address is evenly divisible by four. This restriction occurs for two reasons. First, buses are designed in such a way that data are delivered to the processor over bit-parallel, aligned communication paths. Loading an integer from an odd address would require that the bits be shifted, adding logic (and time) to the load path. The x86, which for reasons of backward compatibility allows operands to appear at arbitrary addresses, runs faster if those operands are properly aligned. Second, on RISC machines, there are generally not enough bits in an instruction to specify both an operation (e.g., load) and a full address. As we shall see in Section 5.3.1, it is typical to specify an address in terms of an *offset* from some *base location* specified by a register. Requiring that integers be word-aligned allows the offset to be specified in words, rather than

#### DESIGN & IMPLEMENTATION

##### The processor/memory gap

Historically processor speed has increased much faster than memory speed, so the number of processor cycles required to access memory has continued to grow. As a result of this trend, caches have become increasingly critical to performance. To improve the effectiveness of caching, programmers need to choose algorithms whose data access patterns have a high degree of locality. High-quality compilers, likewise, need to consider locality of access when choosing among the many possible translations of a given program.

in bytes, quadrupling the amount of memory that can be accessed using offsets from a given base register.

## 5.2 Data Representation

Data in the memory of most computers are untyped: bits are simply bits. *Operations* are typed, in the sense that different operations *interpret* the bits in memory in different ways. Typical *data formats* include instructions, addresses, binary integers of various lengths, floating-point (real) numbers of various lengths, and characters.

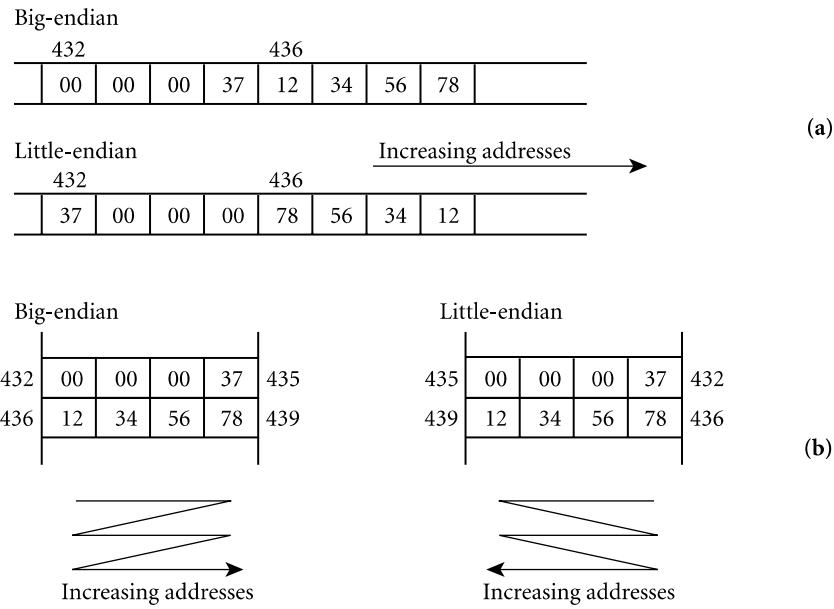
**EXAMPLE 5.2**  
Big- and little-endian

Integers typically come in half-word, word, and (recently) double-word lengths. Floating-point numbers typically come in word and double-word lengths, commonly referred to as *single* and *double precision*. Some machines store the least-significant byte of a multi-word datum at the address of the datum itself, with bytes of increasing numeric significance at higher-numbered addresses. Other machines store the bytes in the opposite order. The first option is called *little-endian*; the second is called *big-endian*. In either case, an  $n$ -byte datum stored at address  $t$  occupies bytes  $t$  through  $t + n - 1$ . The advantage of a little-endian organization is that it is tolerant of variations in operand size. If the value 37 is stored as a word and then a byte is read from the same location, the value 37 will be returned. On a big-endian machine, the value 0 will be returned (the upper eight bits of the number 37, when stored in 32 bits). The problem with the little-endian approach is that it seems to scramble the bytes of integers, when read from left to right (see Figure 5.2a). Little-endian-ness makes a bit more sense if one thinks of memory as a (byte-addressable) array of words (Figure 5.2b). Among CISC machines, the x86 is little-endian, as was the Digital VAX. The IBM 360/370 and the Motorola 680x0 are big-endian. Most of the first-generation RISC machines were also big-endian; most of the current RISC machines can run in either mode.

Support for characters varies widely. Most CISC machines will perform arbitrary arithmetic and logical operations on one-byte quantities. Many CISC machines also provide instructions that perform operations on strings of characters, such as copying, comparing, or searching. Most RISC machines will load and store bytes from or to memory, but operate only on longer quantities in registers.

### 5.2.1 Computer Arithmetic

Binary integers are almost universally represented in two related formats: straightforward binary place-value for unsigned numbers, and *two's complement* for signed numbers. An  $n$ -bit unsigned integer has a value in the range  $0..2^n-1$ , inclusive. An  $n$ -bit two's complement integer has a value in the range



**Figure 5.2** Big-endian and little-endian byte orderings. (a) Two four-byte quantities, the numbers  $37_{16}$  and  $12\ 34\ 56\ 78_{16}$ , stored at addresses 432 and 436, respectively. (b) The same situation with memory visualized as a byte-addressable array of words.

$-2^{n-1} \dots 2^{n-1} - 1$ , inclusive. Most instruction sets provide two forms of most of the arithmetic operators: one for unsigned numbers and one for signed numbers. Even for languages in which integers are always signed, unsigned arithmetic is important for the manipulation of addresses (e.g., pointers).

Floating-point numbers are the computer equivalent of scientific notation: they consist of a *mantissa* or *significand*, *sig*, an *exponent*, *exp*, and (usually) a sign bit, *s*. The value of a floating-point number is then  $-1^s \times \text{sig} \times 2^{\text{exp}}$ . Prior to the mid-1980s, floating-point formats and semantics tended to vary greatly across brands and even models of computers. Different manufacturers made different choices regarding the number of bits in each field, their order, and their internal representation. They also made different choices regarding the behavior of arithmetic operators with respect to rounding, underflow, overflow, invalid operations, and the representation of extremely small quantities. With the completion in 1985 of IEEE standard number 754, however, the situation changed dramatically. Most processors developed in subsequent years conform to the formats and semantics of this standard.

#### IN MORE DEPTH

We consider two's complement and IEEE floating-point arithmetic in more detail on the PLP CD.

## 5.3 Instruction Set Architecture

On a RISC machine, computational instructions operate on values held in registers: a load instruction must be used to bring a value from memory into a register before it can be used as an operand. CISC machines usually allow all or most computational instructions to access operands directly in memory. RISC machines are therefore said to provide a *load-store* or *register-register* architecture; CISC machines are said to provide a *register-memory* architecture.

For binary operations, instructions on RISC machines generally specify three registers: two sources and a destination. Some CISC machines (e.g., the VAX) also provide three-address instructions. Others (e.g., the x86 and the 680x0) provide only two-address instructions; one of the operands is always overwritten by the result. Two-address instructions are more compact, but three-address instructions allow both operands to be reused in subsequent operations. This reuse is crucial on RISC machines: it minimizes the number of artificial restrictions on the ordering of instructions, affording the compiler considerably more freedom in choosing an order that performs well.

### 5.3.1 Addressing Modes

One can imagine many different ways in which a computational instruction might specify the location of its operands. A given operand might be in a register, in memory, or, in the case of read-only constants, in the instruction itself. If the operand is in memory, its address might be found in a register, in memory, or in the instruction, or it might be derived from some combination of values in various locations. Instruction sets differ greatly in the *addressing modes* they provide to capture these various options.

As noted above, most RISC machines require that the operands of computational instructions reside in registers or the instruction. For load and store instructions, which are allowed to access memory, they typically support the *displacement* addressing mode, in which the operand's address is found by adding some small constant (the *displacement*) to the value found in a specified register (the *base*). The displacement is contained in the instruction. Displacement addressing with respect to the frame pointer provides an easy way to access local variables. Displacement addressing with a displacement of zero is sometimes called *register indirect* addressing.

Some RISC machines, including the PowerPC and Sparc, also allow load and store instructions to use an *indexed* addressing mode, in which the operand's address is found by adding the values in two registers. Indexed addressing is useful for arrays: one register (the *base*) contains the address of the array; the second (the *index*) contains the offset of the desired element.

CISC machines typically provide a richer set of addressing modes, and allow them to be used in computational instructions, as well as in loads and stores. On the x86, for example, the address of an operand can be calculated by multiplying the value in one register by a small constant, adding the value found in a second register, and then adding another small constant, all in one instruction.

### 5.3.2 Conditions and Branches

All instruction sets provide a *branching* mechanism to update the program counter under program control. Branches allow compilers to implement conditional statements, subroutines, and loops. Conditional branches are generally controlled in one of two ways. On most CISC machines they use *condition codes*. As mentioned in Section 5.1, condition codes are usually implemented as a set of bits in a special *processor status register*. All or most of the arithmetic, logical, and data-movement instructions update the condition codes as a side effect. The exact number of bits varies from machine to machine, but three and four are common: one bit each to indicate whether the instruction produced a zero value, a negative value, and/or an overflow or carry. To implement the following test, for example,

**EXAMPLE 5.3**

An if statement in x86 assembler

```
A := B + C
if A = 0 then
 body
```

a compiler for the x86<sup>1</sup> might generate

```
movl C, %eax ; move longword C into register eax
addl B, %eax ; add
movl %eax, A ; and store
jne L1 ; branch (jump) if result not equal to zero
body
L1:
```

**EXAMPLE 5.4**

Compare and test instructions

For cases in which the outcome of a branch depends on a value that has not just been computed or moved, most machines provide compare and test instructions. Again on the x86:

---

**I** Readers familiar with the x86 should be warned that this example uses the assembler syntax of the Gnu gcc compiler and its assembler, gas. This syntax differs in several ways from Microsoft and Intel assembler. Most notably, it specifies operands in the opposite order. The instruction `addl B, %eax`, for example, adds the value in B to the value in register `%eax` and leaves the result in `%ebx`: in Gnu assembler the *destination* operand is listed second. In Intel and Microsoft assembler it's the other way around: `addl B, %eax` would add the value in register `%ebx` to the value in B and leave the result in B.

```

if A ≤ B then movl A, %eax ; move long-word A into register eax
 body cmpl B, %eax ; compare to B
 jg L1 ; branch (jump) if greater
 body
L1:
if A > 0 then testl %eax, %eax ; compare %eax (A) to 0
 body jle L2 ; branch if less than or equal
 body
L2:

```

The x86 `cmpl` instruction subtracts its source operand from its destination operand and sets the condition codes according to the result, but it does *not* overwrite the destination operand. The `testl` instruction ands its two operands together and compares the result to zero. Most often, as shown here, the two operands are the same. When they are different, one is typically a *mask* value that allows the programmer or compiler to test individual bits or bits fields in the other operand. ■

Unfortunately, traditional condition codes make it difficult to implement some important performance enhancements. In particular, the fact that they are set by almost every instruction tends to preclude implementations in which logically unrelated instructions might be executed in between (or in parallel with) the instruction that tests a condition and the branch that relies on the outcome of the test. There are several possible ways to address this problem; the handling of conditional branches is one of the areas in which extant RISC machines vary most from one another. The ARM and Sparc architectures make setting of the condition codes optional on an instruction-by-instruction basis. The PowerPC provides eight separate sets of condition codes; compare and branch instructions can specify the set to use. The MIPS has no condition codes (at least not for integer operations); it uses Boolean values in registers instead.

More precisely, where the x86 has 16 different branch instructions based on arithmetic comparisons, the MIPS has only six. Four of these branch if the value in a register is  $<$ ,  $\leq$ ,  $>$ , or  $\geq$  zero. The other two branch if the values in two registers are  $=$  or  $\neq$ . In a convention shared by most RISC machines, register zero is defined to always contain the value zero, so the latter two instructions cover both the remaining comparisons to zero and direct comparisons of registers for equality. More general register-register comparisons (signed and unsigned) require a separate instruction to place a Boolean value in a register that is then named by the branch instruction. Repeating the preceding examples on the MIPS, we get

```

if A ≤ B then lw $3, A ; load word: register 3 := A
 body lw $2, B ; register 2 := B
 slt $2, $2, $3 ; register 2 := (B < A)
 bne $2, $0, L1 ; branch if Boolean true ($\neq 0$)
 body
L1:

```

### **EXAMPLE 5.5**

Conditional branches on the MIPS

```

if A > 0 then blez $3, L2 ; branch if A ≤ 0
 body body
L2:

```

By convention, destination registers are listed first in MIPS assembler (as they are in assignment statements). The `slt` instruction stands for “set less than”; `bne` and `blez` stand for “branch if not equal” and “branch if less than or equal to zero,” respectively. Note that the compiler has used `bne` to compare register 2 to the constant register 0. ■

### CHECK YOUR UNDERSTANDING

1. What is the world’s most popular instruction set architecture (for desktop machines)?
2. What is the difference between big-endian and little-endian addressing?
3. What is the purpose of a cache?
4. Why do many machines have more than one *level* of cache?
5. How many processor cycles does it typically take to access primary (on-chip) cache? How many cycles does it typically take to access main memory?
6. What is data *alignment*? Why do many processors insist upon it?
7. List four common formats (interpretations) for bits in memory.
8. What is IEEE standard number 754? Why is it important?
9. What are the tradeoffs between two-address and three-address instruction formats?
10. Describe at least five different addressing modes. Which of these are commonly supported on RISC machines?
11. What are condition codes? Why do some architectures not provide them? What do they provide instead?

## 5.4 Architecture and Implementation

The typical processor implementation consists of a collection of *functional units*, one (or more) for each logically separable facet of processor activity: instruction fetch, instruction decode, operand fetch from registers, arithmetic computation, memory access, write-back of results to registers, and so on. One could imagine an implementation in which all of the work for a particular instruction is completed before work on the next instruction begins, and in fact this is how many computers used to be constructed. The problem with this organization is

that most of the functional units are idle most of the time. Using ideas originally developed for supercomputers of the 1960s, processor implementations have increasingly moved toward a *pipelined* organization, in which the functional units work like the stations on an assembly line, with different instructions passing through different pipeline stages concurrently. Pipelining is used in even the most inexpensive personal computers today, and in all but the simplest processors for the embedded market. A simple processor may have five or six pipeline stages. The IBM PowerPC G5 has 21; the Intel Pentium 4E has 31.

By allowing (parts of) multiple instructions to execute in parallel, pipelining can dramatically increase the number of instructions that can be completed per second, but it is not a panacea. In particular, a pipeline will *stall* if the same functional unit is needed in two different instructions simultaneously, or if an earlier instruction has not yet produced a result by the time it is needed in a later instruction, or if the outcome of a conditional branch is not known (or guessed) by the time the next instruction needs to be fetched.

We shall see in Section 5.5 that many stalls can be avoided by adding a little extra hardware and then choosing carefully among the various ways of translating a given construct into target code. An important example occurs in the case of floating-point arithmetic, which is typically much slower than integer arithmetic. Rather than stall the entire pipeline while executing a floating-point instruction, we can build a separate functional unit for floating-point math, and arrange for it to operate on a separate set of floating-point registers. In effect, this strategy leads to a *pair* of pipelines—one for integers and one for floating-point—that share their first few stages. The integer branch of the pipeline can continue to execute while the floating-point unit is busy, as long as subsequent instructions do not require the floating-point result. The need to reorder, or *schedule*, instructions so that those that conflict with or depend on one another are separated in time is one of the principal reasons why compiling for modern processors is hard.

### 5.4.1 Microprogramming

As technology advances, there are occasionally times when it becomes feasible to design machines in a very different way. During the 1950s and the early 1960s, the instruction set of a typical computer was implemented by soldering together large numbers of discrete components (transistors, capacitors, etc.) that performed the required operations. To build a faster computer, one generally designed new, more powerful instructions, which required extra hardware. This strategy had the unfortunate effect of requiring assembly language programmers (or compiler writers, though there weren't many of them back then) to learn a new language every time a new and better computer came along.

A fundamental breakthrough occurred in the early 1960s, when IBM hit upon the idea of *microprogramming*. Microprogramming allowed a company to provide the *same* instruction set across a whole line of computers, from

inexpensive slow machines to expensive fast machines. The basic idea was to build a “microengine” in hardware that executed an interpreter program in “firmware.” The interpreter in turn implemented the “machine language” of the computer—in this case, the IBM 360 instruction set. More expensive machines had fancier microengines, with more direct support for the instructions seen by the assembly-level programmer. The top-of-the-line machines had everything in hardware. In effect, the architecture of the machine became an abstract interface behind which hardware designers could hide implementation details, much as the interfaces of modules in modern programming languages allow software designers to limit the information available to users of an abstraction.

In addition to allowing the introduction of computer families, microprogramming made it comparatively easy for architects to extend the instruction set. Numerous studies were published in which researchers identified some sequence of instructions that commonly occurred together (e.g., the instructions that jump to a subroutine and update bookkeeping information in the stack) and then introduced a new instruction to perform the same function as the sequence. The new instruction was usually faster than the sequence it replaced, and almost always shorter (and code size was more important then than now).

### 5.4.2 Microprocessors

A second architectural breakthrough occurred in the mid-1970s, when very large-scale integration (VLSI) chip technology reached the point at which a simple microprogrammed processor could be implemented entirely on one inexpensive chip. The chip boundary is important because it takes much more time and power to drive signals across macroscopic output pins than it does across intra-chip connections, and because the number of pins on a chip is limited by packaging issues. With an entire processor on one chip, it became feasible to build a commercially viable personal computer. Processor architectures of this era include the MOS Technology 6502, used in the Apple II and the Commodore 64, and the Intel 8080 and Zilog Z80, used in the Radio Shack TRS-80 and various CP/M machines. Continued improvements in VLSI technology led, by the mid-1980s, to 32-bit microprogrammed microprocessors such as the Motorola 68000, used in the original Apple Macintosh, and the Intel 80386, used in the first 32-bit IBM PCs.

From an architectural standpoint, the principal impact of the microprocessor revolution was to constrain, temporarily, the number of registers and the size of operands. Where the IBM 360 (*not* a single-chip processor) operated on 32-bit data, with 16 general purpose 32-bit registers, the Intel 8080 operated on 8-bit data, with only seven 8-bit registers and a 16-bit stack pointer. Over time, as VLSI density increased, registers and instruction sets expanded as well. Intel’s 32-bit 80386 was introduced in 1985.

### 5.4.3 RISC

By the early 1980s, several factors converged to make possible a third architectural breakthrough. First, VLSI technology reached the point at which a pipelined 32-bit processor with a sufficiently simple instruction set could be implemented on a single chip, *without* microprogramming. Second, improvements in processor speed were beginning to outstrip improvements in memory speed, increasing the relative penalty for accessing memory, and thereby increasing the pressure to keep things in registers. Third, compiler technology had advanced to the point at which compilers could often match (and sometimes exceed) the quality of code produced by the best assembly language programmers. Taken together, these factors suggested a *reduced instruction set computer* (RISC) architecture with a fast, all-hardware implementation, a comparatively low-level instruction set, a large number of registers, and an optimizing compiler.

The advent of RISC machines ran counter to the ever-more-powerful-instructions trend in processor design but was to a large extent consistent with established trends for supercomputers. Supercomputer instruction sets had always been relatively simple and low-level, in order to facilitate pipelining. Among other things, effective pipelining depends on having most instructions take the same, constant number of cycles to execute, and on minimizing dependences that would prevent a later instruction from starting execution before its predecessors have finished. A major problem with the trend toward more complex instruction sets was that it made it difficult to design high-performance implementations. Instructions on the VAX, for example, could vary in length from one to more than 50 bytes, and in execution time from one to thousands of cycles. Both of these factors tend to lead to pipeline stalls. Variable-length instructions make it difficult to even find the next instruction until the current one has been studied extensively. Variable execution time makes it difficult to keep all the pipeline stages busy. The original VAX (the 11/780) was shipped in 1978, but it wasn't until 1985 that Digital was able to ship a successfully pipelined version, the 8600.<sup>2</sup>

The most basic rule of processor performance holds that total execution time on any machine equals the number of instructions executed times the average number of cycles per instruction times the length in time of a cycle. What we might call the “CISC design philosophy” is to minimize execution time by reducing the number of instructions, letting each instruction do more work. The “RISC philosophy,” by contrast, is to minimize execution time by reducing the length of the cycle and the number of (nonoverlapped) cycles per instruction (CPI).

Recent RISC machines (and RISC-like implementations of the x86) attempt to minimize CPI by executing as many instructions as possible in parallel. The

---

**2** An alternative approach—to maintain microprogramming but pipeline the microengine—was adopted by the 8800 and, more recently, by Intel's Pentium Pro and its successors.

PowerPC G5, for example, can have over 200 instructions simultaneously “in flight.” Some processors have very deep pipelines, allowing the work of an instruction to be divided into very short cycles. Many are *superscalar*: they have multiple parallel pipelines, and start more than one instruction each cycle. (This requires, of course, that the compiler and/or hardware identify instructions that do not depend on one another, so that parallel execution is semantically indistinguishable from sequential execution.) To minimize artificial dependences between instructions (as, for instance, when one instruction must finish using a register as an operand before another instruction overwrites that register with a new value), many machines perform *register renaming*, dynamically assigning logically independent uses of the same architectural register to different locations in a larger set of physical (implementation) registers. High performance processor implementations may actually execute mutually independent instructions *out of order* when they can increase instruction-level parallelism by doing so. These techniques dramatically increase implementation complexity but not architectural complexity; in fact, it is architectural *simplicity* that makes them possible.

#### 5.4.4 Two Example Architectures: The x86 and MIPS

---

**EXAMPLE 5.6**

The x86 ISA

We can illustrate the differences between CISC and RISC machines by examining a representative pair of architectures. The x86 is the most widely used CISC design—in fact, the most widely used processor architecture of any kind (outside the embedded market). The original model, the 8086, was announced in 1978. Major changes were introduced by the 8087, 80286, 80386, Pentium Pro, Pentium/MMX, Pentium III, and Pentium 4. While technically backward compatible, these changes were often out of keeping with the philosophy of the earlier generations. The result is a machine with an enormous number of stylistic inconsistencies and special cases. AMD’s 64-bit extension to the x86, saddled as it was with the need for backward compatibility, is even more complex. Early generations of the x86 were extensively microprogrammed. More recent generations still use microprogramming for the more complex portions of the instruction set, but simpler instructions are translated directly (in hardware) into between one and four microinstructions that are in turn fed to a heavily pipelined, RISC-like computational core. ■

---

**EXAMPLE 5.7**

The MIPS ISA

The MIPS architecture, begun as a commercial spin-off of research at Stanford University, is arguably the simplest of the commercial RISC machines. It too has evolved, through five generations as of 2005, but with one exception—a jump to 64-bit integer operands and addresses in 1991—the changes have been relatively minor. MIPS processors were used by Digital Equipment Corp. for a few years prior to the development of the (now defunct) Alpha architecture, and by Silicon Graphics, Inc. throughout the 1990s. They are now used primarily in embedded applications. MIPS-based tools are also widely used in academia. All

```
f1 := 0
goto L2
L1: f2 := *r1 -- load
 f1 := f1 + f2
 r1 := r1 + 8 -- floating-point numbers are 8 bytes long
 r2 := r2 - 1
L2: if r2 > 0 goto L1
```

**Figure 5.3 Example of pseudo-assembly notation.** The code shown sums the elements of a floating-point vector of length  $n$ . At the beginning, integer register  $r1$  is assumed to point to the vector and register  $r2$  is assumed to contain  $n$ . At the end, floating-point register  $f1$  contains the sum.

models of the MIPS are implemented entirely in hardware; they are not micro-programmed.

#### IN MORE DEPTH

Among the most significant differences between the x86 and MIPS are their memory access mechanisms, their register sets, and the variety of instructions they provide. Like all RISC machines, the MIPS allows only load and store instructions to access memory; all computation is done with values in registers. Like most CISC machines, the x86 allows computational instructions to operate on values in either registers or memory. It also provides a richer set of addressing modes. Like most RISC machines, the MIPS has 32 integer registers and 32 floating-point registers. The x86, by contrast, has only 8 of each, and most of the floating-point instructions treat the floating-point registers as a tiny stack, rather than naming them directly. The MIPS provides many fewer distinct instructions than does the x86, and its instruction set is much more internally consistent; the x86 has a huge number of special cases. All MIPS instructions are exactly 4 bytes long. Instructions on the x86 vary from 1 to 17 bytes.

#### 5.4.5 Pseudo-Assembly Notation

##### EXAMPLE 5.8

##### Pseudo-assembler

At various times throughout the remainder of this book, we will need to consider sequences of machine instructions corresponding to some high-level language construct. Rather than present these sequences in the assembly language of some particular processor architecture, we will (in most cases) rely on a simple notation designed to represent a generic RISC machine. A brief example appears in Figure 5.3.

The notation should in most cases be self-explanatory. It uses “assignment statements” and operators reminiscent of high-level languages, but each line of code corresponds to a single machine instruction, and registers are named explicitly. Control flow is based entirely on gotos and subroutine calls. Conditional

tests assume that the hardware can perform a comparison and branch in a single instruction, where the comparison tests the contents of a register against a small constant or the contents of another register.

 **CHECK YOUR UNDERSTANDING**

12. What is microprogramming? What breakthroughs did its invention make possible?
13. What technological threshold was crossed in the mid-1970s, enabling the introduction of microprocessors? What subsequent threshold, crossed in the early 1980s, made RISC machines possible?
14. What is pipelining?
15. Summarize the difference between the CISC and RISC philosophies in instruction set design.
16. Why do RISC machines allow only load and store instructions to access memory?
17. Name three CISC architectures. Name three RISC architectures. (If you're stumped, see the Summary and Concluding Remarks [Section 5.6].)
18. What three research groups share the credit for inventing RISC? (For this you'll probably need to peek at the Bibliographic Notes [Section 5.9].)
19. How can the designer of a pipelined machine cope with instructions (e.g., floating-point arithmetic) that take much longer than others to compute?

## 5.5 Compiling for Modern Processors

Programming a RISC machine by hand, in assembly language, is a tedious undertaking. Only loads and stores can access memory, and then only with limited addressing modes. Moreover the limited space available in fixed-size instructions means that a nonintuitive two-instruction sequence is required to load a 32-bit constant or to jump to an absolute address. In some sense, complexity that used to be hidden in the microcode of CISC machines has been exported to the compiler.

Fortunately, most of the code for modern processors is generated by compilers, which don't get bored or make careless mistakes, and can easily deal with comparatively primitive instructions. In fact, when compiling for recent implementations of the x86, compilers generally limit themselves to a small, RISC-like subset of the instruction set, which the processor can pipeline effectively. Old programs that make use of more complex instructions still run, but not as fast; they don't take full advantage of the hardware.

**EXAMPLE 5.9**Performance  $\neq$  clock rate

The real difficulty in compiling for modern processors lies not in the need to use primitive instructions, but in the need to keep the pipeline full and to make effective use of registers. A user who trades in a Pentium III PC for one with a Pentium 4 will typically find that while old programs run faster on the new machine, the speed improvement is nowhere near as dramatic as the difference in clock rates would lead one to expect. Improvements will generally be better if one is able to obtain new program versions that have been compiled with the newer processor in mind.

### 5.5.1 Keeping the Pipeline Full

Four main problems may cause a pipelined processor to stall:

*Cache misses.* A load instruction or an instruction fetch may miss in the cache.

*Resource hazards.* Two concurrently executing instructions may need to use the same functional unit at the same time.

*Data hazards.* An instruction may need an operand that has not yet been produced by an earlier but still executing instruction.

*Control hazards.* Until the outcome (and target) of a branch instruction is determined, the processor does not know the location from which to fetch subsequent instructions.

All of these problems are amenable, at least in part, to both hardware and software solutions. On the hardware side, misses can generally be reduced by building larger or more highly associative caches.<sup>3</sup> Resource hazards, likewise, can be addressed by building multiple copies of the various functional units (though most processors don't provide enough to avoid all possible conflicts). Misses, resource hazards, and data hazards can all be addressed by *out-of-order* execution, which allows a processor (at the cost of significant design complexity, chip area, and power consumption) to consider a lengthy "window" of instructions, and make progress on any of them for which operands and hardware resources are available.

Of course, even out-of-order execution works only if the processor is able to fetch instructions, and thus it is control hazards that have the largest potential negative impact on performance. Branches constitute something like 10% of all instructions in typical programs,<sup>4</sup> so even a one-cycle stall on every branch could

**3** The degree of *associativity* of a cache is the number of distinct lines in the cache in which the contents of a given memory location might be found. In a one-way associative (*direct-mapped*) cache, each memory location maps to only one possible line in the cache. If the program uses two locations that map to the same line, the contents of these two locations will keep *evicting* each other, and many misses will result. More highly associative caches are slower but suffer fewer such conflicts.

**4** This is a very rough number. For the SPEC2000 benchmarks, Hennessy and Patterson report percentages varying from 1 to 25 [HP03, pp. 138–139].

be expected to slow down execution by 9% on average. On a deeply pipelined machine one might naively expect to stall for more like five or even ten cycles while waiting for a new program counter to be computed. To avoid such intolerable delays, most workstation-class processors incorporate hardware to *predict* the outcome of each branch, based on past behavior, and to execute speculatively down the predicted path. Assuming that it takes care to avoid any irreversible operations, the processor will suffer stalls only in the case of an incorrect prediction.

On the software side, the compiler has a major role to play in keeping the pipeline full. For any given source program, there is an unbounded number of possible translations into machine code. In general we should prefer shorter translations over longer ones, but we must also consider the extent to which various translations will utilize the pipeline. On an in-order processor (one that always executes instructions in the order they appear in the machine language program), a stall will inevitably occur whenever a load is followed immediately by an instruction that needs the loaded value, because even a first-level cache requires at least one extra cycle to respond. A stall may also occur when the result of a slow-to-complete floating-point operation is needed too soon by another instruction, when two concurrently executing instructions need the same functional unit in the same cycle, or, on a superscalar processor, when an instruction that uses a value is executed concurrently with the instruction that produces it. In all these cases performance may improve significantly if the compiler chooses a translation in which instructions appear in a different order.

The general technique of reordering instructions at compile time so as to maximize processor performance is known as *instruction scheduling*. On an in-order processor the goal is to identify a valid order that will minimize pipeline stalls at run time. To achieve this goal the compiler requires a detailed model of the pipeline. On an out-of-order processor the goal is simply to maximize *instruction-level parallelism* (ILP): the degree to which unrelated instructions lie near one another in the instruction stream (and thus are likely to fall within the processor's instruction window). A compiler for such an out-of-order machine may be able to make do with a less detailed processor model. At the same time, it may need to ensure a higher degree of ILP, since out-of-order execution tends to be found on machines with several pipelines.

Instruction scheduling can have a major impact on resource and data hazards. On machines with so-called *delayed branches* it can also help with control hazards. We will consider the topic of instruction scheduling in some detail in Section ⑩ 15.6. In the remainder of the current subsection we focus on the two cases—loads and branches—where issues of instruction scheduling may actually be embedded in the processor's instruction set. Software techniques to reduce the incidence of cache misses typically require large-scale restructuring of control flow or data layout. Though the better commercial compilers may reorganize loops for better cache locality in scientific programs (a topic we will consider in Section ⑩ 15.7.2), most simply assume that every memory access will hit in the

primary cache. The assumption is generally a good one: most programs on most machines achieve a cache hit rate of well over 90% (often over 99%). The important goal is to make sure that the pipeline can continue to operate during the time that it takes the cache to respond.

### Loads

Consider a load instruction that hits in the primary cache. The number of cycles that must elapse before a subsequent instruction can use the result is known as the *load delay*. Most current machines have a one-cycle load delay. If the instruction immediately after a load attempts to use the loaded value, a one-cycle *load penalty* (a pipeline stall) will occur. Longer pipelines can have load delays of two or even three cycles.

To avoid load penalties (in the absence of out-of-order execution), the compiler may schedule one or more unrelated instructions into the *delay slot(s)* between a load and a subsequent use. In the following code, for example, a simple in-order pipeline will incur a one-cycle penalty between the second and third instructions.

```
r2 := r1 + r2
r3 := A -- load
r3 := r3 + r2
```

If we swap the first two instructions, the penalty goes away:

```
r3 := A -- load
r2 := r1 + r2
r3 := r3 + r2
```

The second instruction gives the first instruction time enough to retrieve A before it is needed in the third instruction. ■

To maintain program correctness, an instruction-scheduling algorithm must respect all *dependences* among instructions. These dependences come in three varieties:

*Flow dependence* (also called *true* or *read-after-write* dependence): a later instruction uses a value produced by an earlier instruction.

*Antidependence* (also called *write-after-read* dependence): a later instruction overwrites a value read by an earlier instruction.

*Output dependence* (also called *write-after-write* dependence): a later instruction overwrites a value written by a previous instruction.

### EXAMPLE 5.11

Renaming registers for scheduling

A compiler can often eliminate anti- and output dependences by *renaming* registers. In the following, for example, antidependences prevent us from moving either the instruction before the load or the one after the add into the delay slot of the load.

```

r3 := r1 + 3 -- immovable ↓
r1 := A -- load
r2 := r1 + r2
r1 := 3 -- immovable ↑

```

If we use a different register as the target of the load, however, then either instruction can be moved:

```

r3 := r1 + 3 -- movable ↓
r5 := A -- load
r2 := r5 + r2
r1 := 3 -- movable ↑

```

The need to rename registers in order to move instructions can increase the number of registers needed by a given stretch of code. To maximize opportunities for concurrent execution, out-of-order processor implementations may perform register renaming dynamically in hardware, as noted in Section 5.4.3. These implementations possess more physical registers than are visible in the instruction set. As instructions are considered for execution, any that use the same architectural register for independent purposes are given separate physical copies on which to do their work. If a processor does not perform hardware register renaming, then the compiler must balance the desire to eliminate pipeline stalls against the desire to minimize the demand for registers (so that they can be used to hold loop indices, local variables, and other comparatively long-lived values).

In order to enforce the flow dependence between a load of a register and its subsequent use, a processor must include so-called *interlock* hardware. To minimize chip area, several of the very early RISC processors provided this hardware only in the case of cache misses. The result was an architecturally visible *delayed load* instruction, in which the value of the loaded register was undefined in the immediately subsequent instruction slot. Filling the delay slot of a delayed load with an unrelated instruction was a matter of correctness, not just of performance. If a compiler was unable to find a suitable “real” instruction, it had to fill the delay slot with a *no-op* (*nop*)—an instruction that has no effect. More recent RISC machines have abandoned delayed loads; their implementations are fully interlocked. Within processor families old binaries continue to work correctly; the (*nop*) instructions are simply redundant.

### **Branches**

Successful pipelining depends on knowing the address of the next instruction before the current instruction has completed or has even been fully decoded. With fixed-size instructions a processor can infer this address for straight-line code but not for the code that follows a branch.<sup>5</sup> In an attempt to minimize the

---

**5** In this context, branches include not only the control flow for conditionals and loops, but also subroutine calls and returns.

impact of branch delays, several early RISC machines defined *delayed branch* instructions similar to the delayed loads just described. In these machines the instruction immediately after the branch is executed regardless of the outcome of the branch. If the branch is not taken, all occurs as one would normally expect. If the branch is taken, however, the order of instructions is the branch itself, the instruction after the branch, and then the instruction at the target of the branch.

Because control may go either of two directions at a branch, finding an instruction to fill a delayed branch slot is slightly trickier than finding one to fill a delayed load slot. The few instructions immediately before the branch are the most obvious candidates to move, provided that they do not contribute to the calculation that controls the branch, and that we don't have to move them past the target of some other branch:

```
B := r2 -- movable ↓
r1 := r2 × r3 -- immovable ♫
if r1 > 0 goto L1
nop
```

(This code sequence assumes that branches are delayed. Unless otherwise noted, we will assume throughout the remainder of the book that they are not.)

To address the problem of unfillable branch delay slots, some more recent RISC machines provide *nullifying* conditional branch instructions. A nullifying branch includes a bit that indicates the direction that the compiler “expects” the branch to go. The hardware executes the instruction in the delay slot only if the branch goes the expected direction. While the branch instruction is making its way down the pipeline, the hardware begins to execute the next instruction. Ideally, by the time it must begin the instruction after that, it will know the outcome of the branch. If the outcome matches the prediction, then the pipeline will proceed without stalling. If the outcome does not match the prediction, then the (not yet completed) instruction in the delay slot will be abandoned, along with any instructions fetched from the target of the branch.

Unfortunately, as architects have moved to more aggressive, deeply pipelined processor implementations, multicycle branch delays have become the norm, and architecturally visible delay slots no longer suffice to hide them. A few processors have been designed with an architecturally visible branch delay of more than one cycle, but this is not generally considered a viable strategy: it is simply too difficult for the compiler to find enough instructions to schedule into the slots. Several processors retain one-slot delayed branches (sometimes with optional nullification) for the sake of backward compatibility and as a means of reducing, but not eliminating, the number of pipeline stalls (the *penalty*) associated with a branch. With or without delayed branches, many processors also employ elaborate hardware mechanisms to predict the outcome and targets of branches early, so that the pipeline can continue anyway. When a prediction turns out to

be incorrect, of course, the hardware must ensure that none of the incorrectly fetched instructions have visible effects. Even when hardware is able to predict the outcome of branches, it can be useful for the compiler to do so also, in order to schedule instructions to minimize load delays in the most likely cross-branch code paths.

### 5.5.2 Register Allocation

The load/store architecture of RISC machines explicitly acknowledges that moving data between registers and memory is expensive. A store instruction costs a minimum of one cycle—more if several stores are executed in succession and the memory system can't keep up. A load instruction costs a minimum of one or two cycles (depending on whether the delay slot can be filled) and can cost scores or even hundreds of cycles in the event of a cache miss. These same costs are present on CISC machines as well, even if they don't stand out as prominently in a casual perusal of assembly code. In order to minimize the use of loads and stores, a good compiler must keep things in registers whenever possible. We saw an example in Chapter 1: the most striking difference between the “optimized” code of Example 1.2 (page 3) and the naive code of Figure 1.5 (page 29) is the absence in the former of most of the loads and stores. As improvements in processor speed continue to outstrip improvements in memory speed, the cost in cycles of a cache miss continues to increase, making good register usage increasingly important.

Register allocation is typically a two-stage process. In the first stage the compiler identifies the portions of the abstract syntax tree that represent *basic blocks*: straight-line sequences of code with no branches in or out. Within each basic block it assigns a “virtual register” to each loaded or computed value. In effect, this assignment amounts to generating code under the assumption that the target machine has an unbounded number of registers. In the second stage, the compiler maps the virtual registers of an entire subroutine onto the architectural (hardware) registers of the machine, using the same architectural register when possible to hold different virtual registers at different times, and *spilling* virtual registers to memory when there aren't enough architectural registers to go around.

We will examine this two-stage process in more detail in Section 15.8. For now, we illustrate the ideas with a simple example. Suppose we are compiling a function that computes the variance  $\sigma^2$  of the contents of an  $n$ -element vector. Mathematically,

$$\sigma^2 = \frac{1}{n} \sum_i (x_i - \bar{x})^2 = \left( \frac{1}{n} \sum_i x_i^2 \right) - \bar{x}^2$$

where  $x_0 \dots x_{n-1}$  are the elements of the vector, and  $\bar{x} = 1/n \sum_i x_i$  is their average. In pseudocode,

---

#### EXAMPLE 5.13

Register allocation for a simple loop

```

1. v1 := &A -- pointer to A[1]
2. v2 := n -- count of elements yet to go
3. w1 := 0.0 -- sum
4. w2 := 0.0 -- squares
5. goto L2
6. L1: w3 := *v1 -- A[i] (floating-point)
7. w1 := w1 + w3 -- accumulate sum
8. w4 := w3 × w3 -- accumulate squares
9. w2 := w2 + w4 -- accumulate squares
10. v1 := v1 + 8 -- 8 bytes per double-word
11. v2 := v2 - 1 -- decrement count
12. L2: if v2 > 0 goto L1
13. w5 := w1 / n -- average
14. w6 := w2 / n -- average of squares
15. w7 := w5 × w5 -- square of average
16. w8 := w6 - w7 -- return value in w8
17. ...

```

Figure 5.4 RISC assembly code for a vector variance computation.

```

double sum := 0
double squares := 0
for int i in 0..n-1
 sum += A[i]
 squares += A[i] × A[i]
double average := sum / n
return (squares / n) - (average × average)

```

After some simple code improvements and the assignment of virtual registers, the assembly language for this function on a RISC machine is likely to look something like Figure 5.4. This code uses two integer virtual registers ( $v1$  and  $v2$ ) and eight floating-point virtual registers ( $w1$ – $w8$ ). For each of these we can compute the range over which the value in the register is useful, or *live*. This range extends from the point at which the value is defined to the last point at which the value is used. For register  $w4$ , for example, the range is only one instruction long, from the assignment at line 8 to the use at line 9. For register  $v1$ , the range is the union of two subranges, one that extends from the assignment at line 1 to the use (and redefinition) at line 10 and another that extends from this redefinition around the loop to the same spot again.

Once we have calculated live ranges for all virtual registers, we can create a mapping onto the architectural registers of the machine. We can use a single architectural register for two virtual registers only if their live ranges do not overlap. If the number of architectural registers required is larger than the number available on the machine (after reserving a few for such special values as the stack pointer), then at various points in the code we shall have to write (spill) some of the virtual registers to memory in order to make room for the others.

```

1. r1 := &A
2. r2 := n
3. f1 := 0.0
4. f2 := 0.0
5. goto L2
6. L1: f3 := *r1 -- no delay
7. f1 := f1 + f3 -- 1-cycle wait for f3
8. f3 := f3 × f3 -- no delay
9. f2 := f2 + f3 -- 4-cycle wait for f3
10. r1 := r1 + 8 -- no delay
11. r2 := r2 - 1 -- no delay
12. L2: if r2 > 0 goto L1 -- no delay
13. f1 := f1 / n
14. f2 := f2 / n
15. f1 := f1 × f1
16. f1 := f2 - f1
17. ... -- return value in f1

```

**Figure 5.5** The vector variance example with physical registers assigned. Also shown in the body of the loop are the number of stalled cycles that can be expected on a simple in-order pipelined machine, assuming a one-cycle penalty for loads, a two-cycle penalty for floating-point adds, and a four-cycle penalty for floating-point multiplies.

In our example program, the live ranges for the two integer registers overlap, so they will have to be assigned to separate physical registers. Among the floating-point registers, w1 overlaps with w2–w4, w2 overlaps with w3–w5, w5 overlaps with w6, and w6 overlaps with w7. There are several possible mappings onto three physical floating-point registers, one of which is shown in Figure 5.5. ■

#### Interaction with Instruction Scheduling

From the point of view of execution speed, the code in Figure 5.5 has at least two problems. First, of the seven instructions in the loop, nearly half are devoted to bookkeeping: updating the pointer, decrementing the loop count, and testing the terminating condition. Second, when run on a pipelined machine, the code is likely to experience a very high number of stalls. Exercise 5.15 explores a first step toward addressing the bookkeeping overhead. We consider the stalls below, and will return to both problems in more detail in Chapter 15.

#### EXAMPLE 5.14

Register allocation and instruction scheduling

We noted in Section 5.5.1 that floating-point instructions commonly employ a separate, longer pipeline. Because they take more cycles to complete, there can be a significant delay before their results are available for use in other instructions. Suppose that floating-point add and multiply instructions must be followed by two and four cycles, respectively, of unrelated computation (these are modest figures; real machines often have longer delays). Also suppose that the result of a load is not available for the usual one-cycle delay. In the context of our vector variance example, these delays imply a total of five stalled cycles in every iteration

```

1. r1 := &A
2. r2 := n
3. f1 := 0.0
4. f2 := 0.0
5. goto L2
6. L1: f3 := *r1
7. r1 := r1 + 8 -- no delay
8. f4 := f3 × f3 -- no delay
9. f1 := f1 + f3 -- no delay
10. r2 := r2 - 1 -- no delay
11. f2 := f2 + f4 -- 1-cycle wait for f4
12. L2: if r2 > 0 goto L1 -- no delay
13. f1 := f1 / n
14. f2 := f2 / n
15. f1 := f1 × f1
16. f1 := f2 - f1
17. ... -- return value in f1

```

**Figure 5.6** The vector variance example after instruction scheduling. All but one cycle of delay has been eliminated. Because we have hoisted the multiply above the first floating-point add, however, we need an extra physical floating-point register.

of the loop, even if the hardware successfully predicts the outcome and target of the branch at the bottom. Added to the seven instructions themselves, this implies a total of 12 cycles per loop iteration (i.e., per vector element).

By rescheduling the instructions in the loop (Figure 5.6) we can eliminate all but one cycle of stall. This brings the total number of cycles per iteration down to only eight, a reduction of 33%. The savings comes at a cost, however: we now execute the multiply instruction before the first floating-point add, and we must use an extra physical register to hold onto the add's second argument. This effect is not unusual: instruction scheduling has a tendency to overlap the live ranges of virtual registers whose ranges were previously disjoint, leading to an increase in the number of architectural registers required. ■

### The Impact of Subroutine Calls

The register allocation scheme outlined above depends implicitly on the compiler being able to see all of the code that will be executed over a given span of time (e.g., an invocation of a subroutine). But what if that code includes calls to other subroutines? If a subroutine were called from only one place in the program, we could allocate registers (and schedule instructions) across both the caller and the callee, effectively treating them as a single unit. Most of the time, however, a subroutine is called from many different places in a program, and the code improvements that we should like to make in the context of one caller will be different from the ones that we should like to make in the context of a different caller. For small, simple subroutines, the compiler may actually choose to expand

a copy of the code at each call site, despite the resulting increase in code size. This *inlining* of subroutines can be an important form of code improvement, particularly for object-oriented languages, which tend to have very large numbers of very small subroutines.

When inlining is not an option, most compilers treat each subroutine as an independent unit. When a body of code for which we are attempting to perform register allocation makes a call to a subroutine, there are several issues to consider:

- Parameters must generally be passed. Ideally, we should like to pass them in registers.
- Any registers that the callee will use internally but that contain useful values in the caller must be spilled to memory and then reread when the callee returns.
- Any variables that the callee might load from memory but that have been kept in a register in the caller must be written back to memory before the call, so that the callee will see the current value.
- Any variables to which the callee might store a value in memory but that have been kept in a register in the caller must be reread from memory when the callee returns, so that the caller will see the current value.

If the caller does not know exactly what the callee might do (this is often the case—the callee might not have been compiled yet), then the compiler must make conservative assumptions. In particular, it must assume that the callee reads and writes *every* variable visible in its scope. The caller must write any such variable back to memory prior to the call if its current value is (only) in a register. If it needs the value of such a variable after the call, it must reread it from memory.

With perfect knowledge of both the caller and the callee, the compiler could arrange across subroutine calls to save and restore precisely those registers that are both in use in the caller and needed (for internal purposes) in the callee. Without this knowledge, we can choose either for the caller to save and restore the registers it is using, before and after the call, or for the callee to save and restore the registers it needs internally, at the top and bottom of the subroutine. In practice it is conventional to choose the latter alternative for at least some static

## DESIGN & IMPLEMENTATION

### In-line subroutines

Subroutine inlining presents, to a large extent, a classic time-space tradeoff. Inlining one instance of a subroutine replaces a relatively short calling sequence with a subroutine body that is typically significantly longer. In return, it avoids the execution overhead of the calling sequence, enables the compiler to perform code improvement across the call without performing interprocedural analysis, and typically improves locality, especially in the L1 instruction cache.

subset of the register set, for two reasons. First, while a subroutine may be called from many locations, there is only one copy of the subroutine itself. Saving and restoring registers in the callee, rather than the caller, can save substantially on code size. Second, because many subroutines (particularly those that are called most frequently) are very small and simple, the set of registers used in the callee tends, on average, to be smaller than the set in use in the caller. We will look at subroutine *calling sequences* in more detail in Chapter 8.

---

 **CHECK YOUR UNDERSTANDING**

---

20. What is a *delayed load* instruction?
  21. What is a *nullifying branch* instruction?
  22. List the four principal causes of pipeline stalls.
  23. What is a pipeline *interlock*?
  24. What is *instruction scheduling*? Why is it important on modern machines?
  25. What is *branch prediction*? Why is it important?
  26. Describe the interaction between instruction scheduling and register allocation.
  27. What is the *live range* of a register?
  28. What is *subroutine inlining*? What benefits does it provide? When is it possible? What is its cost?
  29. Summarize the impact of subroutine calls on register allocation.
- 

## 5.6 Summary and Concluding Remarks

Computer architecture has a major impact on the sort of code that a compiler must generate and the sorts of code improvements it must effect in order to obtain acceptable performance. Since the early 1980s, the trend in processor design has been to equip the compiler with more and more knowledge of the low-level details of processor implementation, so that the generated code can use the implementation to its fullest. This trend has blurred the traditional dividing line between processor architecture and implementation: while a compiler can generate correct code based on an understanding of the architecture alone, it cannot generate fast code unless it understands the implementation as well. In effect, timing issues that were once hidden in the microcode of microprogrammed processors (and that made microprogramming an extremely difficult and arcane craft) have been exported into the compiler.

In the first several sections of this chapter we surveyed the organization of memory and the representation of data (including integer and floating-point

arithmetic), the variety of typical assembly language instructions, and the evolution of modern RISC machines. As examples we compared the x86 and the MIPS. We also introduced a simple notation to be used for assembly language examples in later chapters. In the final section we discussed why compiling for modern machines is hard. The principal tasks include *instruction scheduling*, for load and branch delays and for multiple functional units, and *register allocation*, to minimize memory traffic. We noted that there is often a tension between these tasks, and that both are made more difficult by frequent subroutine calls.

As of 2005 there are four principal commercial RISC architectures: ARM (Intel, Texas Instruments, Motorola, and dozens of others), MIPS (SGI, NEC), Power/PowerPC (IBM, Motorola, Apple), and Sparc (Sun, Texas Instruments, Fujitsu). ARM is the property of ARM Holdings, PLC, an intellectual property firm that relies on licensees for actual fabrication. Though ARM processors are not generally employed in desktop or laptop computers, they power roughly three-quarters of the world's embedded systems, in everything from cell phones and PDAs to remote controls and the dozens of devices in a modern automobile. MIPS processors, likewise, are now principally employed in the embedded market, though they were once common in desktop and high-end machines.

Despite the handicap of a CISC instruction set and the need for backward compatibility, the x86 overwhelmingly dominates the desktop and laptop market, largely due to the marketing prowess of IBM, Intel, and Microsoft, and to the success of Intel and AMD in decoupling the architecture from the implementation. Modern implementations of the x86 incorporate a hardware front-end that translates x86 code, on the fly, into a RISC-like internal format amenable to heavily pipelined execution. Recent processors from Intel and AMD are competitive with the fastest RISC alternatives.

With growing demand for a 64-bit address space, however, a major battle ensued in the x86 world. Intel's IA-64/Itanium processors provide an x86 compatibility mode, but it is implemented in a separate portion of the processor—essentially a Pentium subprocessor embedded in the corner of the chip. Application writers who want speed and address space enhancements were expected to migrate to the (very different) IA-64 instruction set. AMD, by contrast, developed a backward-compatible 64-bit extension to the x86 instruction set; its Opteron processors provide a much smoother upward migration path. In response to market demand, Intel has licensed the Opteron architecture (which it calls EM64T) for use in its 64-bit Pentium processors.

As processor and compiler technology continue to evolve, it is likely that processor implementations will continue to become more complex, and that compilers will take on additional tasks in order to harness that complexity. What is not clear at this point is the form that processor complexity will take. While traditional CISC machines remain popular almost entirely due to the need for backward compatibility, both the CISC and RISC “design philosophies” are still very much alive [SW94]. The “CISC-ish” philosophy says that newly available resources (e.g., increases in chip area) should be used to implement functions that must currently be performed in software, such as vector or graphics operations,

decimal arithmetic, or new addressing modes; the “RISC-ish” philosophy says that resources should be used to improve the speed of existing functions—for example, by increasing cache size, employing faster but larger functional units, or deepening the pipeline and decreasing the cycle time.

Where the first-generation RISC machines from different vendors differed from one another only in minor details, the more recent generations are beginning to diverge, with the ARM and MIPS taking the more RISC-ish approach, the Power/PowerPC family taking the more CISC-ish approach, and the Sparc somewhere in the middle. It is not yet clear which approach will ultimately prove most effective, nor is it even clear that this is the interesting question anymore. Communication latency and heat dissipation are increasingly the limiting factors on both clock speed and the exploitation of instruction-level parallelism. To address these concerns, vendors are increasingly turning to chip-level multiprocessors and other novel architectures, which will almost certainly require new compiler techniques. At perhaps no time in the past 20 years has the future of microarchitecture been in so much flux. However it all turns out, it is clear that processor and compiler technology will continue to evolve together.

## 5.7 Exercises

- 5.1 Modern compilers often find they don’t have enough registers to hold all the things they’d like to hold. At the same time, VLSI technology has reached the point at which there is room on a chip to hold many more registers than are found in the typical ISA. Why are we still using instruction sets with only 32 integer registers? Why don’t we make, say, 64 or 128 of them visible to the programmer?
- 5.2 Some early RISC machines (e.g., the SPARC) provided a “multiply step” instruction that performed one iteration of the standard shift-and-add algorithm for binary integer multiplication. Speculate as to the rationale for this instruction.
- 5.3 Consider sending a message containing a string of integers over the Internet. What problems may occur if the sending and receiving machines have different “endian-ness”? How might you solve these problems?
- 5.4 Why do you think RISC machines standardized on 32-bit instructions? Why not some smaller or larger length? Why not variable lengths?
- 5.5 Consider a machine with three condition codes, N, Z, and O. N indicates whether the most recent arithmetic operation produced a negative result. Z indicates whether it produced a zero result. O indicates whether it produced a result that cannot be represented in the available precision for the numbers being manipulated (i.e., outside the range  $0..2^n$  for unsigned arithmetic,  $-2^{n-1}..2^{n-1}-1$  for signed arithmetic). Suppose we wish to branch on condition A op B, where A and B are unsigned binary numbers, for

$\text{op} \in \{<, \leq, =, \neq, >, \geq\}$ . Suppose we subtract B from A, using two's complement arithmetic. For each of the six conditions, indicate the logical combination of condition-code bits that should be used to trigger the branch. Repeat the exercise on the assumption that A and B are signed, two's complement numbers.

- 5.6 We implied in Section 5.4.1 that if one adds a new instruction to a non-pipelined, microcoded machine, the time required to execute that instruction is (to first approximation) independent of the time required to execute all other instructions. Why is it not strictly independent? What factors could cause overall execution to become slower when a new instruction is introduced?
- 5.7 Suppose that loads constitute 25% of the typical instruction mix on a certain machine. Suppose further that 15% of these loads miss in the on-chip (primary) cache, with a penalty of 40 cycles to reach main memory. What is the contribution of cache misses to the average number of cycles per instruction? You may assume that instruction fetches always hit in the cache. Now suppose that we add an off-chip (secondary) cache that can satisfy 90% of the misses from the primary cache, at a penalty of only 10 cycles. What is the effect on cycles per instruction?
- 5.8 Many recent processors provide a *conditional move* instruction that copies one register into another if and only if the value in a third register is (or is not) equal to zero. Give an example in which the use of conditional moves leads to a shorter program.
- 5.9 The 64-bit AMD Opteron architecture is backward compatible with the x86 instruction set, just as the x86 is backward compatible with the 16-bit 8086 instruction set. Less transparently, the IA-64 Itanium is capable of running legacy x86 applications in “compatibility mode.” But recent members of the ARM and MIPS processor families support *new* 16-bit instructions as an *extension* to the architecture. Why might designers have chosen to introduce these new, less powerful modes of execution?

- 5.10 Consider the following code fragment in pseudo-assembler notation.

```

1. r1 := K
2. r4 := &A
3. r6 := &B
4. r2 := r1 × 4
5. r3 := r4 + r2
6. r3 := *r3 -- load (register indirect)
7. r5 := *(r3 + 12) -- load (displacement)
8. r3 := r6 + r2
9. r3 := *r3 -- load (register indirect)
10. r7 := *(r3 + 12) -- load (displacement)
11. r3 := r5 + r7
12. S := r3 -- store

```

- (a) Give a plausible explanation for this code (what might the corresponding source code be doing?).
- (b) Identify all flow, anti-, and output dependences.
- (c) Schedule the code to minimize load delays on a single-pipeline, in-order processor.
- (d) Can you do better if you rename registers?
- 5.11 With the development of deeper, more complex pipelines, delayed loads and branches have become significantly less appealing as features of a RISC instruction set. Why is it that designers have been able to eliminate delayed loads in more recent machines, but have had to retain delayed branches?
- 5.12 Some processors, including the PowerPC and recent members of the x86 family, require one or more cycles to elapse between a condition-determining instruction and a branch instruction that uses that condition. What options does a scheduler have for filling such delays?
- 5.13 Branch prediction can be performed statically (in the compiler) or dynamically (in hardware). In the static approach, the compiler guesses which way the branch will usually go, encodes this guess in the instruction, and schedules instructions for the expected path. In the dynamic approach, the hardware keeps track of the outcome of recent branches, notices branches or patterns of branches that recur, and predicts that the patterns will continue in the future. Discuss the tradeoffs between these two approaches. What are their comparative advantages and disadvantages?
- 5.14 Consider a machine with a three-cycle penalty for incorrectly predicted branches and a zero-cycle penalty for correctly predicted branches. Suppose that in a typical program 20% of the instructions are conditional branches, which the compiler or hardware manages to predict correctly 75% of the time. What is the impact of incorrect predictions on the average number of cycles per instruction? Suppose the accuracy of branch prediction can be increased to 90%. What is the impact on cycles per instruction?  
Suppose that the number of cycles per instruction would be 1.5 with perfect branch prediction. What is the percentage slowdown caused by mispredicted branches? Now suppose that we have a superscalar processor on which the number of cycles per instruction would be 0.6 with perfect branch prediction. Now what is the percentage slowdown caused by mispredicted branches? What do your answers tell you about the importance of branch prediction on superscalar machines?
- 5.15 Consider the code in Figure 5.6. In an attempt to eliminate the remaining delay and reduce the overhead of the bookkeeping (loop control) instructions, one might consider *unrolling* the loop—that is, creating a new loop in which each iteration performs the work of  $k$  iterations of the original loop. Show the code for  $k = 2$ . You may assume that  $n$  is even and that your target

machine supports displacement addressing. Schedule instructions as tightly as you can. How many cycles does your loop consume per vector element?

© 5.16–5.23 In More Depth.

## 5.8 Explorations

- 5.24 Skip ahead to the sidebar on decimal types on page 314. Write algorithms to convert BCD numbers to binary, and vice versa. Try writing the routines in assembly language for your favorite machine (if your machine has special instructions for this purpose, pretend you're not allowed to use them). How many cycles are required for the conversion?
- 5.25 Is microprogramming an idea that has outlived its usefulness, or are there application domains for which it still makes sense to build a microprogrammed machine? Defend your answer.
- 5.26 If you have access to both CISC and RISC machines, compile a few programs for both machines and compare the size of the target code. Can you generalize about the “space penalty” of RISC code?
- 5.27 Several computers have provided more general versions of the *conditional move* instructions described in Exercise 5.8. Examples include the c. 1965 IBM ACS, the Cray 1, the HP PA-RISC, the ARM, and the Intel IA-64 (Itanium). General purpose conditional execution is sometimes known as *predication*.  
Learn how predication works in ARM or IA-64. Explain how it can sometimes improve performance even when it causes the processor to execute *more* instructions.
- 5.28 If you have access to computers of more than one type, compile a few programs on each machine and time their execution. (If possible, use the same compiler [e.g., gcc] and options on each machine.) Discuss the factors that may contribute to different run times. How closely do the ratios of run times mirror the ratios of clock rates? Why don't they mirror them exactly?
- 5.29 Branch prediction can be characterized as *control speculation*: it makes a guess about the future control flow of the program that saves enough time when it's right to outweigh the cost of cleanup when it's wrong. Some researchers have proposed the complementary notion of *value speculation*, in which the processor would predict the value to be returned by a cache miss, and proceed on the basis of that guess. What do you think of this idea? How might you evaluate its potential?
- 5.30 Can speculation be useful in software? How might you (or a compiler or other tool) be able to improve performance by making guesses that are subject to future verification, with (software) rollback when wrong? (*Hint:*

Think about operations that require communication over slow Internet links.)

- 5.31 Translate the high-level pseudocode for vector variance (Example 5.13) into your favorite programming language, and run it through your favorite compiler. Examine the resulting assembly language. Experiment with different levels of optimization (code improvement). Discuss the quality of the code produced.
  - 5.32 Try to write a code fragment in your favorite programming language that requires so many registers that your favorite compiler is forced to spill some registers to memory (compile with a high level of optimization). How complex does your code have to be?
  - 5.33 If you have access to a compiler that generates code for a machine with architecturally visible load delays, run some programs through it and evaluate the degree of success it has in filling delay slots (an unfilled slot will contain a nop instruction). What percentage of slots is filled? Suppose the machine had interlocked loads. How much space could be saved in typical executable programs if the nops were eliminated?
  - 5.34 Experiment with small subroutines in C++ to see how much time can be saved by expanding them inline.
- © 5.35–5.37 In More Depth.

## 5.9 Bibliographic Notes

The standard reference in computer architecture is the graduate-level text by Patterson and Hennessy [HP03]. More introductory material can be found in the undergraduate computer organization text by the same authors [PH05]. Students without previous assembly language experience may be particularly interested in the text of Bryant and O'Hallaron [BO03], which surveys computer organization from the point of view of the systems programmer, focusing in particular on the correspondence between source-level programs in C and their equivalents in x86 assembler.

The “RISC revolution” of the early 1980s was spearheaded by three separate research groups. The first to start (though last to publish [Rad82]) was the 801 group at IBM’s T. J. Watson Research Center, led by John Cocke. IBM’s Power and PowerPC architectures, though not direct descendants of the 801, take significant inspiration from it. The second group (and the one that coined the term “RISC”) was led by David Patterson [PD80, Pat85] at UC Berkeley. The commercial Sparc architecture is a direct descendant of the Berkeley RISC II design. The third group was led by John Hennessy at Stanford [HJBG81]. The commercial MIPS architecture is a direct descendant of the Stanford design.

Much of the history of pre-1980 processor design can be found in the text by Siewiorek, Bell, and Newell [SBN82]. This classic work contains verbatim

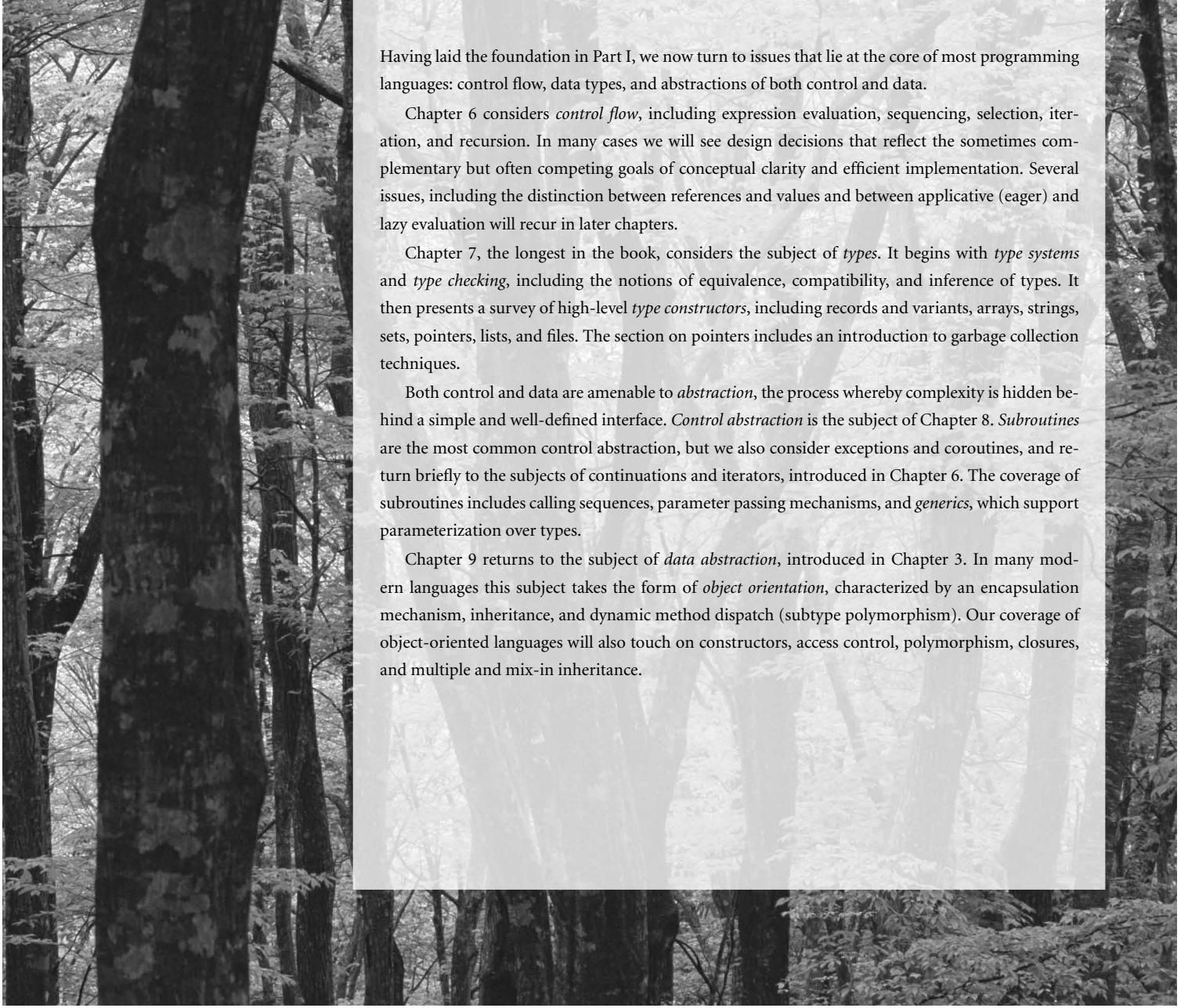
reprints of many important original papers. In the context of RISC processor design, Smith and Weiss [SW94] contrast the more “RISCy” and “CISCy” design philosophies in their comparison of implementations of the PowerPC and Alpha architectures. Appendix C of Hennessy and Patterson’s architecture text (available online at [www.mkp.com/CA3/](http://www.mkp.com/CA3/)) summarizes the similarities and differences among nine different RISC instruction sets. Appendix D describes the x86. Current manuals for all the popular commercial processors are available from their manufacturers.

An excellent treatment of computer arithmetic can be found in Goldberg’s appendix to the Hennessy and Patterson architecture text [Gol03] (available online at [www.mkp.com/CA3/](http://www.mkp.com/CA3/)). The IEEE 754 floating-point standard was printed in *ACM SIGPLAN Notices* in 1985 [IEE87]. The texts of Muchnick [Muc97] and of Cooper and Torczon [CT04] are excellent sources of information on instruction scheduling, register allocation, subroutine optimization, and other aspects of compiling for modern machines.





# Core Issues in Language Design



Having laid the foundation in Part I, we now turn to issues that lie at the core of most programming languages: control flow, data types, and abstractions of both control and data.

Chapter 6 considers *control flow*, including expression evaluation, sequencing, selection, iteration, and recursion. In many cases we will see design decisions that reflect the sometimes complementary but often competing goals of conceptual clarity and efficient implementation. Several issues, including the distinction between references and values and between applicative (eager) and lazy evaluation will recur in later chapters.

Chapter 7, the longest in the book, considers the subject of *types*. It begins with *type systems* and *type checking*, including the notions of equivalence, compatibility, and inference of types. It then presents a survey of high-level *type constructors*, including records and variants, arrays, strings, sets, pointers, lists, and files. The section on pointers includes an introduction to garbage collection techniques.

Both control and data are amenable to *abstraction*, the process whereby complexity is hidden behind a simple and well-defined interface. *Control abstraction* is the subject of Chapter 8. *Subroutines* are the most common control abstraction, but we also consider exceptions and coroutines, and return briefly to the subjects of continuations and iterators, introduced in Chapter 6. The coverage of subroutines includes calling sequences, parameter passing mechanisms, and *generics*, which support parameterization over types.

Chapter 9 returns to the subject of *data abstraction*, introduced in Chapter 3. In many modern languages this subject takes the form of *object orientation*, characterized by an encapsulation mechanism, inheritance, and dynamic method dispatch (subtype polymorphism). Our coverage of object-oriented languages will also touch on constructors, access control, polymorphism, closures, and multiple and mix-in inheritance.



# 6 Control Flow

**Having considered the mechanisms that a compiler uses** to enforce semantic rules (Chapter 4) and the characteristics of the target machines for which compilers must generate code (Chapter 5), we now return to core issues in language design. Specifically, we turn in this chapter to the issue of *control flow* or *ordering* in program execution. Ordering is fundamental to most (though not all) models of computing. It determines what should be done first, what second, and so forth, to accomplish some desired task. We can organize the language mechanisms used to specify ordering into seven principal categories.

1. *sequencing*: Statements are to be executed (or expressions evaluated) in a certain specified order—usually the order in which they appear in the program text.
2. *selection*: Depending on some run-time condition, a *choice* is to be made among two or more statements or expressions. The most common selection constructs are *if* and *case* (*switch*) statements. Selection is also sometimes referred to as *alternation*.
3. *iteration*: A given fragment of code is to be executed repeatedly, either a certain number of times or until a certain run-time condition is true. Iteration constructs include *while*, *do*, and *repeat* loops.
4. *procedural abstraction*: A potentially complex collection of control constructs (*a subroutine*) is encapsulated in a way that allows it to be treated as a single unit, often subject to parameterization.
5. *recursion*: An expression is defined in terms of (simpler versions of) itself, either directly or indirectly; the computational model requires a stack on which to save information about partially evaluated instances of the expression. Recursion is usually defined by means of self-referential subroutines.
6. *concurrency*: Two or more program fragments are to be executed/evaluated “at the same time,” either in parallel on separate processors or interleaved on a single processor in a way that achieves the same effect.

7. *nondeterminacy*: The ordering or choice among statements or expressions is deliberately left unspecified, implying that any alternative will lead to correct results. Some languages require the choice to be random, or fair, in some formal sense of the word.

Though the syntactic and semantic details vary from language to language, these seven principal categories cover all of the control-flow constructs and mechanisms found in most programming languages. A programmer who thinks in terms of these categories, rather than the syntax of some particular language, will find it easy to learn new languages, evaluate the tradeoffs among languages, and design and reason about algorithms in a language-independent way.

Subroutines are the subject of Chapter 8. Concurrency is the subject of Chapter 12. The bulk of this chapter (Sections 6.3 through 6.7) is devoted to a study of the five remaining categories. We begin in Section 6.1 by examining expression evaluation. We consider the syntactic form of expressions, the precedence and associativity of operators, the order of evaluation of operands, and the semantics of the assignment statement. We focus in particular on the distinction between variables that hold a *value* and variables that hold a *reference to* a value; this distinction will play an important role many times in future chapters. In Section 6.2 we consider the difference between *structured* and *unstructured* (goto-based) control flow.

The relative importance of different categories of control flow varies significantly among the different classes of programming languages. Sequencing, for example, is central to imperative (von Neumann and object-oriented) languages, but plays a relatively minor role in functional languages, which emphasize the evaluation of expressions, deemphasizing or eliminating statements (e.g., assignments) that affect program output in any way other than through the return of a value. Similarly, functional languages make heavy use of recursion, whereas imperative languages tend to emphasize iteration. Logic languages tend to deemphasize or hide the issue of control flow entirely: the programmer simply specifies a set of inference rules; the language implementation must find an order in which to apply those rules that will allow it to deduce values that satisfy some desired property.

## 6.1 Expression Evaluation

An expression generally consists of either a simple object (e.g., a literal constant, or a named variable or constant) or an *operator* or function applied to a collection of operands or arguments, each of which in turn is an expression. It is conventional to use the term *operator* for built-in functions that use special, simple syntax, and to use the term *operand* for the argument of an operator. In Algol-family languages, function calls consist of a function name followed by a parenthesized, comma-separated list of arguments, as in

---

**EXAMPLE 6.1**

A typical function call

```
my_func(A, B, C)
```

**EXAMPLE 6.2**

Typical operators

```
a + b
- c
```

Algol-family operators are simpler: they typically take only one or two arguments, and dispense with the parentheses and commas:

In general, a language may specify that function calls (operator invocations) employ prefix, infix, or postfix notation. These terms indicate, respectively, whether the function name appears before, among, or after its several arguments. Most imperative languages use infix notation for binary operators and prefix notation for unary operators and other functions (with parentheses around the arguments). Lisp uses prefix notation for all functions but places the function name *inside* the parentheses, in what is known as *Cambridge Polish*<sup>1</sup> notation:

```
(* (+ 1 3) 2) ; that would be (1 + 3) * 2 in infix
(append a b c my_list)
```

**EXAMPLE 6.3**

Cambridge Polish (prefix) notation

**EXAMPLE 6.4**

Mixfix notation in Smalltalk

A few languages, notably the R scripting language, allow the user to create new infix operators. Smalltalk uses infix notation for *all* functions (which it calls messages), both built-in and user-defined. The following Smalltalk statement sends a “displayOn: at:” message to graphical object `myBox`, with arguments `myScreen` and `100@50` (a pixel location). It corresponds to what other languages would call the invocation of the “displayOn: at:” function with arguments `myBox`, `myScreen`, and `100@50`.

```
myBox displayOn: myScreen at: 100@50
```

**EXAMPLE 6.5**

Conditional expressions

This sort of multiword infix notation occurs occasionally in Algol-family languages as well.<sup>2</sup> In Algol one can say

```
a := if b <> 0 then a/b else 0;
```

Here “`if...then...else`” is a three-operand infix operator. The equivalent operator in C is written “`...? ... : ...`”:

```
a = b != 0 ? a/b : 0;
```

Postfix notation is used for most functions in Postscript, Forth, the input language of certain hand-held calculators, and the intermediate code of some com-

---

**1** Prefix notation was popularized by Polish logicians of the early 20th century; Lisp-like parenthesized syntax was first employed (for noncomputational purposes) by philosopher W. V. Quine of Harvard University (Cambridge, MA).

**2** Most authors use the term “infix” only for binary operators. Multiword operators may be called “mixfix” or left unnamed.

pilers. Postfix appears in a few places in other languages as well. Examples include the pointer dereferencing operator ( $\wedge$ ) of Pascal and the post-increment and -decrement operators ( $++$  and  $--$ ) of C and its descendants.

### 6.1.1 Precedence and Associativity

Most languages provide a rich set of built-in arithmetic and logical operators. When written in infix notation, without parentheses, these operators lead to ambiguity as to what is an operand of what. In Fortran, for example, which uses  $**$  for exponentiation, how should we parse  $a + b * c**d**e/f$ ? Should this group as

$((a + b) * c)**d)**e)/f$

or

$a + ((b * c)**d)**(e/f))$

or

$a + ((b * (c*(d**e))))/f)$

or yet some other option? (In Fortran, the answer is the last of the options shown.)

In any given language, the choice among alternative evaluation orders depends on the *precedence* and *associativity* of operators, concepts we introduced in Section 2.1.3. Issues of precedence and associativity do not arise in prefix or postfix notation.

Precedence rules specify that certain operators, in the absence of parentheses, group “more tightly” than other operators. Associativity rules specify that sequences of operators of equal precedence group to the right or to the left. In most languages multiplication and division group more tightly than addition and subtraction. Other levels of precedence vary widely from one language to another. Figure 6.1 shows the levels of precedence for several well-known languages.

The precedence structure of C (and, with minor variations, of its descendants, C++, Java, and C#) is substantially richer than that of most other languages. It is, in fact, richer than shown in Figure 6.1, because several additional constructs, including type casts, function calls, array subscripting, and record field selection, are classified as operators in C. It is probably fair to say that most C programmers do not remember all of their language’s precedence levels. The intent of the language designers was presumably to ensure that “the right thing” will usually happen when parentheses are not used to force a particular evaluation order. Rather than count on this, however, the wise programmer will consult the manual or add parentheses.

It is also probably fair to say that the relatively flat precedence hierarchy of Pascal is a mistake. In particular, novice Pascal programmers frequently write conditions like

#### EXAMPLE 6.6

A complicated Fortran expression

#### EXAMPLE 6.7

Precedence in four influential languages

#### EXAMPLE 6.8

A “gotcha” in Pascal precedence

| Fortran                                                | Pascal                        | C                                                                                                         | Ada                                 |
|--------------------------------------------------------|-------------------------------|-----------------------------------------------------------------------------------------------------------|-------------------------------------|
|                                                        |                               | ++, -- (post-inc., dec.)                                                                                  |                                     |
| **                                                     | not                           | ++, -- (pre-inc., dec.),<br>+, - (unary),<br>&, * (address, contents of),<br>!, ~ (logical, bit-wise not) | abs (absolute value),<br>not, **    |
| *, /                                                   | *, /,<br>div, mod, and        | * (binary), /,<br>% (modulo division)                                                                     | *, /, mod, rem                      |
| +, - (unary<br>and binary)                             | +,- (unary and<br>binary), or | +,- (binary)                                                                                              | +,- (unary)                         |
|                                                        |                               | <<, >><br>(left and right bit shift)                                                                      | +,- (binary),<br>& (concatenation)  |
| .eq., .ne., .lt.,<br>.le., .gt., .ge.<br>(comparisons) | <, <=, >, >=,<br>=, >>, IN    | <, <=, >, >=<br>(inequality tests)                                                                        | =, /=, <, <=, >, >=                 |
| .not.                                                  |                               | ==, != (equality tests)                                                                                   |                                     |
|                                                        |                               | & (bit-wise and)                                                                                          |                                     |
|                                                        |                               | ^ (bit-wise exclusive or)                                                                                 |                                     |
|                                                        |                               | (bit-wise inclusive or)                                                                                   |                                     |
| .and.                                                  |                               | && (logical and)                                                                                          | and, or, xor<br>(logical operators) |
| .or.                                                   |                               | (logical or)                                                                                              |                                     |
| .eqv., .neqv.<br>(logical comparisons)                 |                               | ? : (if...then...else)                                                                                    |                                     |
|                                                        |                               | =, +=, -=, *=, /=, %=<br>>>=, <<=, &=, ^=,  =<br>(assignment)                                             |                                     |
|                                                        |                               | , (sequencing)                                                                                            |                                     |

**Figure 6.1** Operator precedence levels in Fortran, Pascal, C, and Ada. The operators at the top of the figure group most tightly.

```
if A < B and C < D then (* ouch *)
```

Unless A, B, C, and D are all of type Boolean, which is unlikely, this code will result in a static semantic error, since the rules of precedence cause it to group as A < (B and C) < D. (And even if all four operands are of type Boolean, the result is almost sure to be something other than what the programmer intended.) Most languages avoid this problem by giving arithmetic operators higher precedence than relational (comparison) operators, which in turn have higher prece-

dence than the logical operators. Notable exceptions include APL and Smalltalk, in which all operators are of equal precedence; parentheses *must* be used to specify grouping.

**EXAMPLE 6.9**

Common rules for associativity

Associativity rules are somewhat more uniform across languages, but still display some variety. The basic arithmetic operators almost always associate left-to-right, so  $9 - 3 - 2$  is 4 and not 8. In Fortran, as noted above, the exponentiation operator ( $\star\star$ ) follows standard mathematical convention and associates right-to-left, so  $4\star\star3\star\star2$  is 262144 and not 4096. In Ada, exponentiation does not associate: one must write either  $(4\star\star3)\star\star2$  or  $4\star\star(3\star\star2)$ ; the language syntax does not allow the unparenthesized form. In languages that allow assignments inside expressions (an option we will consider more in Section 6.1.2), assignment associates right-to-left. Thus in C,  $a = b = a + c$  assigns  $a + c$  into  $b$  and then assigns the same value into  $a$ .

Because the rules for precedence and associativity vary so much from one language to another, a programmer who works in several languages is wise to make liberal use of parentheses.

### 6.1.2 Assignments

In a purely functional language, expressions are the building blocks of programs, and computation consists entirely of expression evaluation. The effect of any individual expression on the overall computation is limited to the value that expression provides to its surrounding context. Complex computations employ recursion to generate a potentially unbounded number of values, expressions, and contexts.

In an imperative language, by contrast, computation typically consists of an ordered series of changes to the values of variables in memory. Assignments provide the principal means by which to make the changes. Each assignment takes a pair of arguments: a value and a reference to a variable into which the value should be placed.

In general, a programming language construct is said to have a *side effect* if it influences subsequent computation (and ultimately program output) in any way other than by returning a value for use in the surrounding context. Purely functional languages have no side effects. As a result, the value of an expression in such a language depends only on the referencing environment in which the expression is evaluated, *not* on the time at which the evaluation occurs. If an expression yields a certain value at one point in time, it is guaranteed to yield the same value at any point in time. In fancier terms, expressions in a purely functional language are said to be *referentially transparent*.

By contrast, imperative programming is sometimes described as “computing by means of side effects.” While the evaluation of an assignment may sometimes yield a value, what we really care about is the fact that it changes the value of a variable, thereby affecting the result of any later computation in which the variable appears.

Many (though not all) imperative languages distinguish between *expressions*, which always produce a value, and may or may not have side effects, and *statements*, which are executed *solely* for their side effects, and return no useful value.

### References and Values

On the surface, assignment appears to be a very straightforward operation. Below the surface, however, there are some subtle but important differences in the semantics of assignment in different imperative languages. These differences are often invisible, because they do not affect the behavior of simple programs. They have a major impact, however, on programs that use pointers, and will be explored in further detail in Section 7.7. We provide an introduction to the issues here.

#### **EXAMPLE 6.10**

L-values and r-values

Consider the following assignments in C:

```
d = a;
a = b + c;
```

In the first statement, the right-hand side of the assignment refers to the *value* of *a*, which we wish to place into *d*. In the second statement, the left-hand side refers to the *location* of *a*, where we want to put the sum of *b* and *c*. Both interpretations—value and location—are possible because a variable in C (and in Pascal, Ada, and many other languages) is a named container for a value. We sometimes say that languages like C use a *value model* of variables. Because of their use on the left-hand side of assignment statements, expressions that denote locations are referred to as *l-values*. Expressions that denote values (possibly the value stored in a location) are referred to as *r-values*. Under a value model of variables, a given expression can be either an l-value or an r-value, depending on the context in which it appears. ■

#### **EXAMPLE 6.11**

L-values in C

Of course, not all expressions can be l-values, because not all values have a location, and not all names are variables. In most languages it makes no sense to say  $2 + 3 = a$ , or even  $a = 2 + 3$ , if *a* is the name of a constant. By the same token, not all l-values are simple names; both l-values and r-values can be complicated expressions. In C one may write

```
(f(a)+3)->b[c] = 2;
```

In this expression *f(a)* returns a pointer to some element of an array of structures (records). The assignment places the value 2 into the *c*-th element of field *b* of the third structure after the one to which *f*'s return value points. ■

In C++ it is even possible for a function to return a “reference” to a structure, rather than a pointer to it, allowing one to write

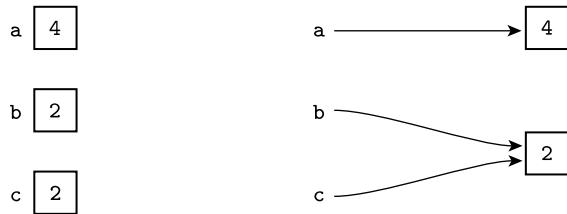
```
g(a).b[c] = 2;
```

We will consider references further in Section 8.3.1.

Several languages make the distinction between l-values and r-values more explicit by employing a *reference model* of variables. In Clu, for example, a variable

#### **EXAMPLE 6.12**

L-values in C++



**Figure 6.2** The value (left) and reference (right) models of variables. Under the reference model, it becomes important to distinguish between variables that refer to the same object and variables that refer to different objects whose values happen (at the moment) to be equal.

### EXAMPLE 6.13

Variables as values and references

```
b := 2;
c := b;
a := b + c;
```

A Pascal programmer might describe this code by saying: “We put the value 2 in `b` and then copy it into `c`. We then read these values, add them together, and place the resulting 4 in `a`.” The Clu programmer would say: “We let `b` refer to 2 and then let `c` refer to it also. We then pass these references to the `+` operator, and let `a` refer to the result, namely 4.”

These two ways of thinking are illustrated in Figure 6.2. With a value model of variables, as in Pascal, any integer variable can contain the value 2. With a reference model of variables, as in Clu, there is (at least conceptually) only one 2—a sort of Platonic Ideal—to which any variable can refer. The practical effect is the same in this example, because integers are *immutable*: the value of 2 never changes, so we can’t tell the difference between two copies of the number 2 and two references to “the” number 2. ■

In a language that uses the reference model, every variable is an l-value. When it appears in a context that expects an r-value, it must be *dereferenced* to obtain the value to which it refers. In most languages with a reference model (including Clu), the dereference is implicit and automatic. In ML, the programmer must

### DESIGN & IMPLEMENTATION

#### Implementing the reference model

It is tempting to assume that the reference model of variables is inherently more expensive than the value model, since a naive implementation would require a level of indirection on every access. As we shall see in Section 7.7.1, however, most compilers for languages with a reference model use multiple copies of immutable objects for the sake of efficiency, achieving exactly the same performance for simple types that they would with a value model.

use an explicit dereference operator, denoted with a prefix exclamation point. We will revisit ML pointers in Section 7.7.1.

The difference between the value and reference models of variables becomes particularly important (specifically, it can affect program output and behavior) if the values to which variables refer can change “in place,” as they do in many programs with linked data structures, or if it is possible for variables to refer to different objects that happen to have the “same” value. In this latter case it becomes important to distinguish between variables that refer to the same object and variables that refer to different objects whose values happen (at the moment) to be equal. (Lisp, as we shall see in Sections 7.10 and 10.3.3, provides more than one notion of equality, to accommodate this distinction.) We will discuss the value and reference models of variables further in Section 7.7. Languages that employ (some variant of) the reference model include Algol 68, Clu, Lisp/Scheme, ML, Haskell, and Smalltalk.

Java uses a value model for built-in types and a reference model for user-defined types (classes). C# and Eiffel allow the programmer to choose between the value and reference models for each individual user-defined type. A C# class is a reference type; a struct is a value type.

### **Boxing**

#### **EXAMPLE 6.14**

#### Wrapper objects in Java 2

A drawback of using a value model for built-in types is that they can't be passed uniformly to methods that expect class typed parameters. Early versions of Java, for example, required the programmer to “wrap” objects of built-in types inside corresponding predefined class types in order to insert them in standard container (collection) classes:

```
import java.util.Hashtable;
...
Hashtable ht = new Hashtable();
...
Integer N = new Integer(13); // Integer is a "wrapper" class
ht.put(N, new Integer(31));
Integer M = (Integer) ht.get(N);
int m = M.intValue();
```

#### **EXAMPLE 6.15**

#### Boxing in Java 5

More recent versions of Java perform automatic *boxing* and *unboxing* operations that avoid the need for wrappers in many cases:

```
ht.put(13, 31);
int m = (Integer) ht.get(13);
```

#### **EXAMPLE 6.16**

#### Boxing in C#

Here the compiler creates hidden Integer objects to hold the values 13 and 31, so they may be passed to put as references. The Integer cast on the return value is still needed, to make sure that the hash table entry for 13 is really an integer and not, say, a floating-point number or string.

C# “boxes” not only the arguments, but the cast as well, eliminating the need for the Integer class entirely. C# also provides so-called *indexers* (Section 9.1,

page 474), which can be used to overload the subscripting ([]) operator, giving the hash table array-like syntax:

```
ht[13] = 31;
int m = (int) ht[13];
```

### Orthogonality

One of the principal design goals of Algol 68 was to make the various features of the language as *orthogonal* as possible. Orthogonality means that features can be used in any combination, the combinations all make sense, and the meaning of a given feature is *consistent*, regardless of the other features with which it is combined. The name is meant to draw an explicit analogy to orthogonal vectors in linear algebra: none of the vectors in an orthogonal set depends on (or can be expressed in terms of) the others, and all are needed in order to describe the vector space as a whole.

Algol 68 was one of the first languages to make orthogonality a principal design goal, and in fact few languages since have given the goal such weight. Among other things, Algol 68 is said to be *expression-oriented*: it has no separate notion of statement. Arbitrary expressions can appear in contexts that would call for a statement in a language like Pascal, and constructs that are considered to be statements in other languages can appear within expressions. The following, for example, is valid in Algol 68:

---

#### EXAMPLE 6.17

Expression orientation in  
Algol 68

```
begin
 a := if b < c then d else e;
 a := begin f(b); g(c) end;
 g(d);
 2 + 3
end
```

Here the value of the `if... then ... else` construct is either the value of its `then` part or the value of its `else` part, depending on the value of the condition. The value of the “statement list” on the right-hand side of the second assignment is the value of its final “statement,” namely the return value of `g(c)`. There is no need to distinguish between procedures and functions, because every subroutine call returns a value. The value returned by `g(d)` is discarded in this example. Finally, the value of the code fragment as a whole is 5, the sum of 2 and 3. ■

C takes an approach intermediate between Pascal and Algol 68. It distinguishes between statements and expressions, but one of the classes of statement is an “expression statement,” which computes the value of an expression and then throws it away. In effect, this allows an expression to appear in any context that would require a statement in most other languages. C also provides special expression forms for selection and sequencing. Algol 60 defines `if... then ... else` as both a statement and an expression.

Both Algol 68 and C allow assignments within expressions. The value of an assignment is simply the value of its right-hand side. Unfortunately, where most

**EXAMPLE 6.18**

A “gotcha” in C conditions

of the descendants of Algol 60 use the `:=` token to represent assignment, C follows Fortran in simply using `=`. It uses `==` to represent a test for equality (Fortran uses `.eq.`). Moreover, C lacks a separate Boolean type. (C99 has a new `_Bool` type, but it's really just a one-bit integer.) In any context that would require a Boolean value in other languages, C accepts an integer (or anything that can be coerced to be an integer). It interprets zero as false; any other value is true. As a result, both of the following constructs are valid—common—in C.

```
if (a == b) {
 /* do the following if a equals b */

if (a = b) {
 /* assign b into a and then do
 the following if the result is nonzero */
```

Programmers who are accustomed to Ada or some other language in which `=` is the equality test frequently write the second form above when the first is what is intended. This sort of bug can be very hard to find. ■

Though it provides a true Boolean type (`bool`), C++ shares the problem of C, because it provides automatic coercions from numeric, pointer, and enumeration types. Java and C# eliminate the problem by disallowing integers in Boolean contexts. The assignment operator is still `=`, and the equality test is still `==`, but the statement `if (a = b) ...` will generate a compile-time type clash error unless `a` and `b` are both `boolean` (Java) or `bool` (C#), which is generally unlikely.

**Combination Assignment Operators****EXAMPLE 6.19**

Updating assignments

Because they rely so heavily on side effects, imperative programs must frequently *update* a variable. It is thus common in many languages to see statements like

```
a = a + 1;
```

or worse,

```
b.c[3].d = b.c[3].d * e;
```

Such statements are not only cumbersome to write and to read (we must examine both sides of the assignment carefully to see if they really are the same), they also result in redundant address calculations (or at least extra work to eliminate the redundancy in the code improvement phase of compilation). ■

If the address calculation has a side effect, then we may need to write a pair of statements instead. Consider the following code in C:

```
void update(int A[], int index_fn(int n)) {
 int i, j;
 /* calculate i */
 ...
 j = index_fn(i);
 A[j] = A[j] + 1;
}
```

**EXAMPLE 6.20**

Side effects and updates

Here we cannot safely write

```
A[index_fn(i)] = A[index_fn(i)] + 1;
```

We have to introduce the temporary variable *j* because we don't know whether *index\_fn* has a side effect or not. If it is being used, for example, to keep a log of elements that have been updated, then we shall want to make sure that update calls it only once.

To eliminate the clutter and compile- or run-time cost of redundant address calculations, and to avoid the issue of repeated side effects, many languages, beginning with Algol 68 and including C and its descendants, provide so-called *assignment operators* to update a variable. Using assignment operators, the statements in Example 6.19 can be written as follows.

```
a += 1;
b.c[3].d *= e;
```

Similarly, the two assignments in the *update* function can be replaced with

```
A[index_fn(i)] += 1;
```

In addition to being aesthetically cleaner, the assignment operator form guarantees that the address calculation is performed only once.

As shown in Figure 6.1, C provides 10 different assignment operators, one for each of its binary arithmetic and bit-wise operators. C also provides prefix and postfix increment and decrement operations. These allow even simpler code in *update*:

```
A[index_fn(i)]++;
```

or

```
++A[index_fn(i)];
```

More significantly, increment and decrement operators provide elegant syntax for code that uses an index or a pointer to traverse an array:

```
A[--i] = b;
*p++ = *q++;
```

When prefixed to an expression, the `++` or `--` operator increments or decrements its operand *before* providing a value to the surrounding context. In the postfix form, `++` or `--` updates its operand *after* providing a value. If *i* is 3 and *p* and *q* point to the initial elements of a pair of arrays, then *b* will be assigned into *A[2]* (not *A[3]*), and the second assignment will copy the initial elements of the arrays (not the second elements).

The prefix forms of `++` and `--` are syntactic sugar for `+=` and `-=`. We could have written

```
A[i -= 1] = b;
```

above. The postfix forms are not syntactic sugar. To obtain an effect similar to the second statement above we would need an auxiliary variable and a lot of

#### EXAMPLE 6.21

##### Assignment operators

#### EXAMPLE 6.22

##### Prefix and postfix inc/dec

#### EXAMPLE 6.23

##### Advantages of postfix inc/dec

extra notation:

`*(t = p, p += 1, t) = *(t = q, q += 1, t);`

Both the assignment operators (`+=`, `-=`) and the increment and decrement operators (`++`, `--`) do “the right thing” when applied to pointers in C. If `p` points to an object that occupies  $n$  bytes in memory (including any bytes required for alignment, as discussed in Section 5.1), then `p += 3` points  $3n$  bytes higher in memory.

### **Multeway Assignment**

#### **EXAMPLE 6.24**

Simple multeway assignment

#### **EXAMPLE 6.25**

Advantages of multeway assignment

`a, b := c, d;`

Here the comma in the right-hand side is *not* the sequencing operator of C. Rather, it serves to define an expression, or *tuple*, consisting of multiple r-values. The comma operator on the left-hand side produces a tuple of l-values. The effect of the assignment is to copy `c` into `a` and `d` into `b`.<sup>3</sup>

While we could just as easily have written

`a := c; b := d;`

the multeway (tuple) assignment allows us to write things like

`a, b := b, a;`

which would otherwise require auxiliary variables. Moreover, multeway assignment allows functions to return tuples, as well as single values:

`a, b, c := foo(d, e, f);`

This notation eliminates the asymmetry (nonorthogonality) of functions in most programming languages, which allow an arbitrary number of arguments but only a single return.

ML generalizes the idea of multeway assignment into a powerful *pattern-matching* mechanism; we will examine this mechanism in more detail in Section 7.2.4.

### **CHECK YOUR UNDERSTANDING**

1. Name seven major categories of control-flow mechanisms.
2. What distinguishes *operators* from other sorts of functions?

---

**3** The syntax shown here is for Clu. Perl, Python, and Ruby follow C in using `=` for assignment. ML requires parentheses around each tuple.

3. Explain the difference between *prefix*, *infix*, and *postfix* notation. What is *Cambridge Polish* notation? Name two programming languages that use postfix notation.
  4. Why don't issues of associativity and precedence arise in Postscript or Forth?
  5. What does it mean for an expression to be *referentially transparent*?
  6. What is the difference between a *value* model of variables and a *reference* model of variables? Why is the distinction important?
  7. What is an *l-value*? An *r-value*?
  8. Why is the distinction between *mutable* and *immutable* values important in the implementation of a language with a reference model of variables?
  9. Define *orthogonality* in the context of programming language design.
  10. What does it mean for a language to be *expression-oriented*?
  11. What are the advantages of updating a variable with an *assignment operator*, rather than with a regular assignment in which the variable appears on both the left- and right-hand sides?
- 

### 6.1.3 Initialization

Because they already provide a construct (the assignment statement) to set the value of a variable, imperative languages do not always provide a means of specifying an initial value for a variable in its declaration. There are at least two reasons, however, why such initial values may be useful:

1. In the case of statically allocated variables (as discussed in Section 3.2), an initial value that is specified in the context of the declaration can be placed into memory by the compiler. If the initial value is set by an assignment statement instead, it will generally incur execution cost at run time.
2. One of the most common programming errors is to use a variable in an expression before giving it a value. One of the easiest ways to prevent such errors (or at least ensure that erroneous behavior is repeatable) is to give every variable a value when it is first declared.

Some languages (e.g., Pascal) have no initialization facility at all; all variables must be given values by explicit assignment statements. To avoid the expense of run-time initialization of statically allocated variables, many Pascal implementations provide initialization as a language extension, generally in the form of a `:= expr` immediately after the name in the declaration. Unfortunately, the extension is usually nonorthogonal, in the sense that it only works for variables of simple, built-in types. A more complete and orthogonal approach to initialization requires a notation for *aggregates*: built-up structured values of user-defined composite types. Aggregates can be found in several languages, including C, Ada,

Fortran 90, and ML; we will discuss them further in Section 7.1.5. It should be emphasized that initialization saves time only for variables that are statically allocated. Variables allocated in the stack or heap at run time must be initialized at run time.<sup>4</sup> It is also worth noting that the problem of using an uninitialized variable occurs not only after elaboration, but also as a result of any operation that destroys a variable's value without providing a new one. Two of the most common such operations are explicit deallocation of an object referenced through a pointer and modification of the *tag* of a variant record. We will consider these operations further in Sections 7.7 and 7.3.4, respectively.

If a variable is not given an initial value explicitly in its declaration, the language may specify a default value. In C, for example, statically allocated variables for which the programmer does not provide an initial value are guaranteed to be represented in memory as if they had been initialized to zero. For most types on most machines, this is a string of zero bits, allowing the language implementation to exploit the fact that most operating systems (for security reasons) fill newly allocated memory with zeros. Zero-initialization applies recursively to the subcomponents of variables of user-defined composite types. The designers of C chose not to incur the run-time cost of automatically zero-filling uninitialized variables that are allocated in the stack or heap. The programmer can specify an initial value if desired; the effect is the same as if an assignment had been placed at the beginning of the code for the variable's scope.

### **Constructors**

Many object-oriented languages allow the programmer to define types for which initialization of dynamically allocated variables occurs automatically, even when no initial value is specified in the declaration. C++ also distinguishes carefully between initialization and assignment. Initialization is interpreted as a call to a *constructor* function for the variable's type, with the initial value as an argument. In the absence of coercion, assignment is interpreted as a call to the type's assignment operator or, if none has been defined, as a simple bit-wise copy of the value on the assignment's right-hand side. The distinction between initialization and assignment is particularly important for user-defined abstract data types that perform their own storage management. A typical example occurs in variable-length character strings. An assignment to such a string must generally deallocate the space consumed by the old value of the string before allocating space for the new value. An initialization of the string must simply allocate space. Initialization with a nontrivial value is generally cheaper than default initialization followed by assignment because it avoids deallocation of the space allocated for the default value. We will return to this issue in Section 9.3.2.

---

**4** For variables that are accessed indirectly (e.g., in languages that employ a reference model of variables), a compiler can often reduce the cost of initializing a stack or heap variable by placing the initial value in static memory, and only creating the pointer to it at elaboration time.

Neither Java nor C# distinguishes between initialization and assignment, or between declaration and definition. Java uses a reference model for all variables of user-defined object types, and provides for automatic storage reclamation, so assignment never copies values. C# allows the programmer to specify a value model when desired (in which case assignment does copy values), but otherwise it mirrors Java. We will return to these issues again in Chapter 9 when we consider object-oriented features in more detail.

### **Definite Assignment**

#### **EXAMPLE 6.26**

Programs outlawed by definite assignment

Java and C# require that a value be “definitely assigned” to a variable before that variable is used in any expression. Both languages provide a precise definition of “definitely assigned,” based on the control flow of the program. Roughly speaking, every possible control path to an expression must assign a value to every variable in that expression. This is a conservative rule; it can sometimes prohibit programs that would never actually use an uninitialized variable. In Java:

```
int i;
final static int j = 3;
...
if (j > 0) {
 i = 2;
}
...
if (j > 0) {
 System.out.println(i);
 // error: "i might not have been initialized"
}
```

### **DESIGN & IMPLEMENTATION**

#### **Safety v. performance**

A recurring theme in any comparison between C++ and Java is the latter’s willingness to accept additional run-time cost in order to obtain cleaner semantics or increased reliability. Definite assignment is one example: it may force the programmer to perform “unnecessary” initializations on certain code paths, but in so doing it avoids the many subtle errors that can arise from missing initialization in other languages. Similarly, the Java specification mandates automatic garbage collection, and its reference model of user-defined types forces most objects to be allocated in the heap. As we shall see in Chapters 7 and 9, Java also requires both dynamic binding of all method invocations and run-time checks for out-of-bounds array references, type clashes, and other dynamic semantic errors. Clever compilers can reduce or eliminate the cost of these requirements in certain common cases, but for the most part the Java design reflects an evolutionary shift away from performance as *the* overriding design goal.

While a human being might reason that `i` will only be used when it has previously been given a value, it is uncomputable to make such determinations in the general case, and the compiler does not attempt it.

### Dynamic Checks

Instead of giving every uninitialized variable a default value, a language or implementation can choose to define the use of an uninitialized variable as a dynamic semantic error, and can catch these errors at run time. The advantage of the semantic checks is that they will often identify a program bug that is masked or made more subtle by the presence of a default value. With appropriate hardware support, uninitialized variable checks can even be as cheap as default values, at least for certain types. In particular, a compiler that relies on the IEEE standard for floating-point arithmetic can fill uninitialized floating-point numbers with a *signaling NaN* value, as discussed in Section ⑩ 5.2.1. Any attempt to use such a value in a computation will result in a hardware interrupt, which the language implementation may catch (with a little help from the operating system), and use to trigger a semantic error message.

For most types on most machines, unfortunately, the costs of catching all uses of an uninitialized variable at run time are considerably higher. If every possible bit pattern of the variable's representation in memory designates some legitimate value (and this is often the case), then extra space must be allocated somewhere to hold an initialized/uninitialized flag. This flag must be set to "uninitialized" at elaboration time and to "initialized" at assignment time. It must also be checked (by extra code) at every use—or at least at every use that the code improver is unable to prove is redundant. Dynamic semantic checks for uninitialized variables are common in interpreted languages, which already incur significant overhead on every variable access. Because of their cost, however, the checks are usually not performed in languages that are compiled.

#### 6.1.4 Ordering Within Expressions

While precedence and associativity rules define the order in which binary infix operators are applied within an expression, they do not specify the order in which the operands of a given operator are evaluated. For example, in the expression

---

**EXAMPLE 6.27**

Indeterminate ordering

$$a - f(b) - c * d$$

we know from associativity that `f(b)` will be subtracted from `a` before performing the second subtraction, and we know from precedence that the right operand of that second subtraction will be the result of `c * d`, rather than merely `c`, but without additional information we do not know whether `a - f(b)` will be evaluated before or after `c * d`. Similarly, in a subroutine call with multiple arguments

$$f(a, g(b), c)$$

we do not know the order in which the arguments will be evaluated.

There are two main reasons why the order can be important:

**EXAMPLE 6.28**

A value that depends on ordering

1. *Side effects:* If  $f(b)$  may modify  $d$ , then the value of  $a - f(b) - c * d$  will depend on whether the first subtraction or the multiplication is performed first. Similarly, if  $g(b)$  may modify  $a$  and/or  $c$ , then the values passed to  $f(a, g(b), c)$  will depend on the order in which the arguments are evaluated.

**EXAMPLE 6.29**

An optimization that depends on ordering

2. *Code improvement:* The order of evaluation of subexpressions has an impact on both register allocation and instruction scheduling. In the expression  $a * b + f(c)$ , it is probably desirable to call  $f$  before evaluating  $a * b$ , because the product, if calculated first, would need to be saved during the call to  $f$ , and  $f$  might want to use all the registers in which it might easily be saved. In a similar vein, consider the sequence

```
a := B[i];
c := a * 2 + d * 3;
```

Here it is probably desirable to evaluate  $d * 3$  before evaluating  $a * 2$ , because the previous statement,  $a := B[i]$ , will need to load a value from memory. Because loads are slow, if the processor attempts to use the value of  $a$  in the next instruction (or even the next few instructions on many machines), it will have to wait. If it does something unrelated instead (i.e., evaluate  $d * 3$ ), then the load can proceed in parallel with other computation.

Because of the importance of code improvement, most language manuals say that the order of evaluation of operands and arguments is undefined. (Java and C# are unusual in this regard: they require left-to-right evaluation.) In the absence of an enforced order, the compiler can choose whatever order results in faster code.

### Applying Mathematical Identities

Some language implementations (e.g., for dialects of Fortran) allow the compiler to *rearrange* expressions involving operators whose mathematical abstractions are commutative, associative, and/or distributive, in order to generate faster code. Consider the following Fortran fragment.

**EXAMPLE 6.30**

Optimization and mathematical “laws”

```
a = b + c
d = c + e + b
```

Some compilers will rearrange this as

```
a = b + c
d = b + c + e
```

They can then recognize the *common subexpression* in the first and second statements, and generate code equivalent to

```
a = b + c
d = a + e
```

Similarly,

```
a = b/c/d
e = f/d/c
```

may be rearranged as

```
t = c * d
a = b/t
e = f/t
```

Unfortunately, while mathematical arithmetic obeys a variety of commutative, associative, and distributive laws, computer arithmetic is not as orderly. The problem is that numbers in a computer are of limited precision. With 32-bit arithmetic, the expression  $b - c + d$  can be evaluated safely left to right if  $a$ ,  $b$ , and  $c$  are all integers between two billion and three billion ( $2^{32}$  is a little less than 4.3 billion). If the compiler attempts to reorganize this expression as  $b + d - c$ , however (e.g., in order to delay its use of  $c$ ), then arithmetic overflow will occur.

Many languages, including Pascal and most of its descendants, provide dynamic semantic checks to detect arithmetic overflow. In some implementations these checks can be disabled to eliminate their run-time overhead. In C and C++, the effect of arithmetic overflow is implementation-dependent. In Java, it is well defined: the language definition specifies the size of all numeric types, and requires two's complement integer and IEEE floating-point arithmetic. In C#, the programmer can explicitly request the presence or absence of checks by tagging an expression or statement with the `checked` or `unchecked` keyword. In a completely different vein, Scheme, Common Lisp, and several scripting languages place no *a priori* limit on the size of numbers; space is allocated to hold extra-large values on demand.

Even in the absence of overflow, the limited precision of floating-point arithmetic can cause different arrangements of the “same” expression to produce sig-

## DESIGN & IMPLEMENTATION

### Evaluation order

Expression evaluation represents a difficult tradeoff between semantics and implementation. To limit surprises, most language definitions require the compiler, if it ever reorders expressions, to respect any ordering imposed by parentheses. The programmer can therefore use parentheses to prevent the application of arithmetic “identities” when desired. No similar guarantee exists with respect to the order of evaluation of operands and arguments. It is therefore unwise to write expressions in which a side effect of evaluating one operand or argument can affect the value of another. As we shall see in Section 6.3, some languages, notably Euclid and Turing, outlaw such side effects.

**EXAMPLE 6.31**

Reordering and numerical stability

nificantly different results, invisibly. Single-precision IEEE floating-point numbers devote 1 bit to the sign, 8 bits to the exponent (power of 2), and 23 bits to the mantissa. Under this representation,  $a + b$  is guaranteed to result in a loss of information if  $|\log_2(a/b)| > 23$ . Thus if  $b = -c$ , then  $a + b + c$  may appear to be zero, instead of  $a$ , if the magnitude of  $a$  is small, while the magnitude of  $b$  and  $c$  is large. In a similar vein, a number like 0.1 cannot be represented precisely, because its binary representation is a “repeating decimal”: 0.0001001001.... For certain values of  $x$ ,  $(0.1 + x) * 10.0$  and  $1.0 + (x * 10.0)$  can differ by as much as 25%, even when 0.1 and  $x$  are of the same magnitude. ■

### 6.1.5 Short-Circuit Evaluation

**EXAMPLE 6.32**

Short-circuited expressions

Boolean expressions provide a special and important opportunity for code improvement and increased readability. Consider the expression  $(a < b)$  and  $(b < c)$ . If  $a$  is greater than  $b$ , there is really no point in checking to see whether  $b$  is less than  $c$ ; we know the overall expression must be false. Similarly, in the expression  $(a > b)$  or  $(b > c)$ , if  $a$  is indeed greater than  $b$  there is no point in checking to see whether  $b$  is greater than  $c$ ; we know the overall expression must be true. A compiler that performs *short-circuit evaluation* of Boolean expressions will generate code that skips the second half of both of these computations when the overall value can be determined from the first half. ■

**EXAMPLE 6.33**

Saving time with short-circuiting

Short-circuit evaluation can save significant amounts of time in certain situations:

```
if (very_unlikely_condition && very_expensive_function()) ...
```

**EXAMPLE 6.34**

Short-circuit pointer chasing

But time is not the only consideration, or even the most important one. Short-circuiting changes the *semantics* of Boolean expressions. In C, for example, one can use the following code to search for an element in a list.

```
p = my_list;
while (p && p->key != val)
 p = p->next;
```

C short-circuits its `&&` and `||` operators, and uses zero for both nil and false, so `p->key` will be accessed if and only if `p` is non-nil. The syntactically similar code in Pascal does not work, because Pascal does not short-circuit `and` and `or`:

```
p := my_list;
while (p <> nil) and (p^.key <> val) do (* ouch! *)
 p := p^.next;
```

Here both of the `<>` relations will be evaluated before and-ing their results together. At the end of an unsuccessful search, `p` will be `nil`, and the attempt to access `p^.key` will be a run-time (dynamic semantic) error, which the compiler may or may not have generated code to catch. To avoid this situation, the Pascal programmer must introduce an auxiliary Boolean variable and an extra level of nesting:

```

1. function tally(word : string) : integer;
2. (* Look up word in hash table. If found, increment tally; If not
3. found, enter with a tally of 1. In either case, return tally. *)
4. ...
5. function misspelled(word : string) : Boolean;
6. (* Check to see if word is mis-spelled and return appropriate
7. indication. If yes, increment global count of mis-spellings. *)
8. ...
9. while not eof(doc_file) do begin
10. w := get_word(doc_file);
11. if (tally(w) = 10) and misspelled(w) then
12. writeln(w);
13. end;
14. writeln(total_misspellings);

```

**Figure 6.3** Pascal code that counts on the evaluation of Boolean operands.

```

p := my_list;
still_searching := true;
while still_searching do
 if p = nil then
 still_searching := false
 else if p^.key = val then
 still_searching := false
 else
 p := p^.next;

```

#### **EXAMPLE 6.35**

Short-circuiting and other errors

Short-circuit evaluation can also be used to avoid out-of-bound subscripts:

```

const MAX = 10;
int A[MAX]; /* indices from 0 to 9 */
...
if (i >= 0 && i < MAX && A[i] > foo) ...

```

division by zero:

```
if (d <> 0 && n/d > threshold) ...
```

and various other errors.

Short-circuiting is not necessarily as attractive for situations in which a Boolean subexpression can cause a side effect. Suppose we wish to count occurrences of words in a document, and print a list of all misspelled words that appear ten or more times, together with a count of the total number of misspellings. Pascal code for this task appears in Figure 6.3. Here the *if* statement at line 9 tests the conjunction of two subexpressions, both of which have important side effects. If short-circuit evaluation is used, the program will not compute the right result. The code can be rewritten to eliminate the need for non-short-circuit evaluation, but one might argue that the result is more awkward than the version shown.

#### **EXAMPLE 6.36**

When not to use short-circuiting

**EXAMPLE 6.37**

Optional short-circuiting

So now we have seen situations in which short-circuiting is highly desirable, and others in which at least some programmers would find it undesirable. A few languages, among them Clu, Ada, and C, provide both regular *and* short-circuit Boolean operators. (Similar flexibility can be achieved with *if...then...else* in an expression-oriented language such as Algol 68; see Exercise 6.10.) In Clu, the regular Boolean operators are *and* and *or*; the short-circuit operators are *cand* and *cor* (for *conditional and* and *or*):

```
if d ~= 0 cand n/d > threshold then ...
```

In Ada, the regular operators are also *and* and *or*; the short-circuit operators are the two-word operators *and then* and *or else*:

```
found_it := p /= null and then p.key = val;
```

(Clu and Ada use  $\sim=$  and  $/=$ , respectively, for “not equal.”) C’s logical *&&* and *||* operators short-circuit; the bit-wise *&* and *|* operators can be used as non-short-circuiting alternatives when their arguments are logical (zero or one) values. ■

When used to determine the flow of control in a selection or iteration construct, short-circuit Boolean expressions do not really have to calculate a Boolean value; they simply have to ensure that control takes the proper path in any given situation. We will look more closely at the generation of code for short-circuit expressions in Section 6.4.1.

 **CHECK YOUR UNDERSTANDING**


---

12. Given the ability to assign a value into a variable, why is it useful to be able to specify an *initial* value?
  13. What are *aggregates*? Why are they useful?
  14. Explain the notion of *definite assignment* in Java and C#.
  15. Why is it generally expensive to catch all uses of uninitialized variables at run time?
  16. Why is it impossible to catch all uses of uninitialized variables at compile time?
  17. Why do most languages leave unspecified the order in which the arguments of an operator or function are evaluated?
  18. What is *short-circuit* Boolean evaluation? Why is it useful?
- 

**6.2****Structured and Unstructured Flow****EXAMPLE 6.38**Control flow with *gotos* in Fortran

Control flow in assembly languages is achieved by means of conditional and unconditional jumps (branches). Early versions of Fortran mimicked the low-level

approach by relying heavily on `goto` statements for most nonprocedural control flow:

```
if A .lt. B goto 10 ! ".lt." means "<"
...
10
```

The 10 on the bottom line is a *statement label*. ■

`Goto` statements also feature prominently in other early imperative languages. In Cobol and PL/I they provide the only means of writing logically controlled (`while`-style) loops. Algol 60 and its successors provide a wealth of non-`goto`-based constructs, but until recently most Algol-family languages still provided `goto` as an option.

Throughout the late 1960s and much of the 1970s, language designers debated hotly the merits and evils of `gotos`. It seems fair to say the detractors won. Ada and C# allow `gotos` only in limited contexts. Modula (1, 2, and 3), Clu, Eiffel, and Java do not allow them at all. Fortran 90 and C++ allow them primarily for compatibility with their predecessor languages. (Java reserves the token `goto` as a keyword, to make it easier for a Java compiler to produce good error messages when a programmer uses a C++ `goto` by mistake.)

The abandonment of `gotos` was part of a larger “revolution” in software engineering known as *structured programming*. Structured programming was the “hot trend” of the 1970s, in much the same way that object-oriented programming was the trend of the 1990s. Structured programming emphasizes top-down design (i.e., progressive refinement), modularization of code, structured types (records, sets, pointers, multidimensional arrays), descriptive variable and constant names, and extensive commenting conventions. The developers of structured programming were able to demonstrate that within a subroutine, almost any well-designed imperative algorithm can be elegantly expressed with only sequencing, selection, and iteration. Instead of labels, structured languages rely on the boundaries of lexically nested constructs as the targets of branching control.

Many of the structured control-flow constructs familiar to modern programmers were pioneered by Algol 60. These include the `if...then...else` construct and both enumeration (`for`) and logically (`while`) controlled loops. The `case` statement was introduced by Wirth and Hoare in Algol W [WH66] as an alternative to the more unstructured computed `goto` and `switch` constructs of Fortran and Algol 60, respectively. `Case` statements were adopted in limited form by Algol 68, and more completely by Pascal, Modula, C, Ada, and a host of modern languages.

### 6.2.1 Structured Alternatives to `goto`

Once the principal structured constructs had been defined, most of the controversy surrounding `gotos` revolved around a small number of special cases, each of which was eventually addressed in structured ways.

**EXAMPLE 6.39**

Leaving the middle of a loop

```
while not eof do begin
 readln(line);
 if all_blanks(line) then goto 100;
 consume_line(line)
end;
100:
```

Less commonly, one would also see a label *inside* the end of a loop, to serve as the target of a goto that would terminate a given iteration early. As we shall see in Section 6.5.5, mid-loop exits are supported by special “one-and-a half” loop constructs in languages like Modula, C, and Ada. Some languages also provide a statement to skip the remainder of the current loop iteration: *continue* in C; *cycle* in Fortran 90; *next* in Perl.

**EXAMPLE 6.40**

Returning from the middle of a subroutine

*Early returns from subroutines:* Gotos were used fairly often in Pascal to terminate the current subroutine:

```
procedure consume_line(var line: string);
...
begin
...
 if line[i] = '%' then goto 100;
 (* rest of line is a comment *)
...
100:
end;
```

At a minimum, this goto statement avoids putting the remainder of the procedure in an *else* clause. If the terminating condition is discovered within a deeply nested *if...then...else*, it may avoid introducing an auxiliary variable that must be tested repeatedly in the remainder of the procedure (*if not comment\_line then ...*). ■

The obvious alternative to this use of *goto* is an explicit *return* statement. Algol 60 does not have one, and neither does Pascal, but Fortran always has, and most modern Algol descendants have adopted it.

*Multilevel returns:* Returns and (local) *gos* allow control to return from the current subroutine. On occasion it may make sense to return from a *surrounding* routine. Imagine, for example, that we are searching for an item matching some desired pattern with a collection of files. The search routine might invoke several nested routines, or a single routine multiple times, once for each place in which to search. In such a situation certain historic languages, including Algol 60, PL/I, and Pascal, permit a *goto* to branch to a lexically visible label *outside* the current subroutine:

**EXAMPLE 6.41**

Escaping a nested subroutine

```

function search(key : string) : string;
var rtn : string;
...
procedure search_file(fname : string);
...
begin
...
for ... (* iterate over lines *)
...
if found(key, line) then begin
 rtn := line;
 goto 100;
end;
...
end;

begin (* search *)
...
for ... (* iterate over files *)
...
search_file(fname);
...
100: return rtn;
end;

```

In the event of a nonlocal `goto`, the language implementation must guarantee to repair the run-time stack of subroutine call information. This repair operation is known as *unwinding*. It requires not only that the implementation deallocate the stack frames of any subroutines from which we have escaped, but also that it perform any bookkeeping operations, such as restoration of register contents, that would have been performed when returning from those routines.

As a more structured alternative to the nonlocal `goto`, Common Lisp provides a `return-from` statement that names the lexically surrounding function or block from which to return, and also supplies a return value (eliminating the need for the artificial `rtn` variable in Example 6.41).

But what if `search_file` were not nested inside `search`? We might, for example, wish to call it from routines that search files in different orders. In this case the `goto` of Pascal does not suffice. Algol 60 and PL/I allow labels to be passed as parameters, so a dynamically nested subroutine can perform a `goto` to a caller-defined location. PL/I also allows labels to be stored in variables. If a nested routine needs to return a value it can assign it to some variable in a scope that surrounds all calls. Alternatively, we can pass a reference parameter into every call, into which the result should be written.

Common Lisp again provides a more structured alternative, also available in Ruby. In either language an expression can be surrounded with a `catch`

---

**EXAMPLE 6.42**

Structured nonlocal transfers

block, whose value can be provided by any dynamically nested routine that executes a matching `throw`. In Ruby we might write

```
def searchFile(fname, pattern)
 file = File.open(fname)
 file.each { |line|
 throw :found, line if line =~ /#{pattern}/
 }
end

match = catch :found do
 searchFile("f1", key)
 searchFile("f2", key)
 searchFile("f3", key)
 "not found\n" # default value for catch,
end # if control gets this far
print match
```

Here the `throw` expression specifies a *tag*, which must appear in a matching `catch`, together with a value (`line`) to be returned as the value of the `catch`. (The `if` clause attached to the `throw` performs a regular-expression pattern match, looking for `pattern` within `line`. We will consider pattern matching in more detail in Section 13.4.2.)

*Errors and other exceptions:* The notion of a multilevel return assumes that the callee knows what the caller expects, and can return an appropriate value. In a related and arguably more common situation, a deeply nested block or subroutine may discover that it is unable to proceed with its usual function and, moreover, lacks the contextual information it would need to recover in any graceful way. The only recourse in such a situation is to “back out” of the nested context to some point in the program that is able to recover. Conditions that require a program to “back out” are usually called *exceptions*. We saw an example in Section ⑩ 2.3.4, where we considered phrase-level recovery from syntax errors in a recursive-descent parser.

The most straightforward but generally least satisfactory way to cope with exceptions is to use auxiliary Boolean variables within a subroutine (`if still_ok then ...`) and to return status codes from calls:

```
status := my_proc(args);
if status = ok then ...
```

The auxiliary Booleans can be eliminated by using a nonlocal `goto` or multi-level return, but the caller to which we return must still inspect status codes explicitly. As a structured alternative, many modern languages provide an *exception handling* mechanism for convenient, nonlocal recovery from exceptions. We will discuss exception handling in more detail in Section 8.5. Typically the programmer appends a block of code called a *handler* to any computation in which an exception may arise. The job of the handler is to take

#### EXAMPLE 6.43

Error-checking with status codes

whatever remedial action is required to recover from the exception. If the protected computation completes in the normal fashion, execution of the handler is skipped.

Multilevel returns and structured exceptions have strong similarities. Both involve a control transfer from some inner, nested context back to an outer context, unwinding the stack on the way. The distinction lies in where the computing occurs. In a multilevel return the inner context has all the information it needs. It completes its computation, generating a return value if appropriate, and transfers to the outer context in a way that requires no post-processing. At an exception, by contrast, the inner context cannot complete its work. It performs an “abnormal” return, triggering execution of the handler.

Common Lisp and Ruby provide mechanisms for both multilevel returns and exceptions, but this dual support is relatively rare. Most languages support only exceptions; programmers implement multilevel returns by writing a trivial handler. In an unfortunate overloading of terminology, the names `catch` and `throw`, which Common Lisp and Ruby use for multilevel returns, are used for exceptions in several other languages.

### 6.2.2 Continuations

The notion of nonlocal `gosub`s that unwind the stack can be generalized by defining what are known as *continuations*. In low-level terms, a continuation consists of a code address and a referencing environment to be restored when jumping to that address. In higher-level terms, a continuation is an abstraction that captures a *context* in which execution might continue. Continuations are fundamental to denotational semantics. They also appear as first-class values in certain languages (notably Scheme and Ruby), allowing the programmer to define new control-flow constructs.

Continuation support in Scheme takes the form of a general purpose function called `call-with-current-continuation`, sometimes abbreviated `call/cc`.

#### DESIGN & IMPLEMENTATION

##### Cleaning up continuations

The implementation of continuations in Scheme and Ruby is surprisingly straightforward. Because local variables have unlimited extent in both languages, activation records must in general be allocated on the heap. As a result, explicit deallocation is neither required nor appropriate when jumping through a continuation; frames that are no longer accessible will eventually be reclaimed by a general purpose *garbage collector* (to be discussed in Section 7.7.3). Restoration of state (e.g., saved registers) from escaped routines is not required either: the continuation closure holds everything required to resume the captured context.

This function takes a single argument,  $f$ , which is itself a function. It calls  $f$ , passing as argument a continuation  $c$  that captures the current program counter and referencing environment. The continuation is represented by a closure, indistinguishable from the closures used to represent subroutines passed as parameters. At any point in the future,  $f$  can call  $c$  to reestablish the captured context. If nested calls have been made, control pops out of them, as it does with exceptions. More generally, however,  $c$  can be saved in variables, returned explicitly by subroutines, or called repeatedly, even after control has returned from  $f$  (recall that closures in Scheme have unlimited extent; see Section 3.5). `Call/cc` suffices to build a wide variety of control abstractions, including `gotos`, mid-loop `exits`, multilevel returns, exceptions, iterators (Section 6.5.3), call-by-name parameters (Section 8.3.1), and coroutines (Section 8.6). It even subsumes the notion of returning from a subroutine, though it seldom replaces it in practice.

First-class continuations are an extremely powerful facility. They can be very useful if applied in well-structured ways (i.e., to define new control-flow constructs). Unfortunately, they also allow the undisciplined programmer to construct completely inscrutable programs.

## 6.3 Sequencing

Like assignment, sequencing is central to imperative programming. It is the principal means of controlling the order in which side effects (e.g., assignments) occur: when one statement follows another in the program text, the first statement executes before the second. In most imperative languages, lists of statements can be enclosed with `begin...end` or `{...}` delimiters and then used in any context in which a single statement is expected. Such a delimited list is usually called a *compound statement*. A compound statement preceded by a set of declarations is sometimes called a *block*.

In languages like Algol 68 and C, which blur or eliminate the distinction between statements and expressions, the value of a statement (expression) list is the value of its final element. In Common Lisp, the programmer can choose to return the value of the first element, the second, or the last. Of course, sequencing is a useless operation unless the subexpressions that do not play a part in the return value have side effects. The various sequencing constructs in Lisp are used only in program fragments that do not conform to a purely functional programming model.

Even in imperative languages, there is debate as to the value of certain kinds of side effects. In Euclid and Turing, for example, functions (that is, subroutines that return values, and that therefore can appear within expressions) are not permitted to have side effects. Among other things, side-effect freedom ensures that a Euclid or Turing function, like its counterpart in mathematics, is always *idempotent*: if called repeatedly with the same set of arguments, it will always return the same value, and the number of consecutive calls (after the first) will not affect

the results of subsequent execution. In addition, side-effect freedom for functions means that the value of a subexpression will never depend on whether that subexpression is evaluated before or after calling a function in some other subexpression. These properties make it easier for a programmer or theorem-proving system to reason about program behavior. They also simplify code improvement, for example by permitting the safe rearrangement of expressions.

Unfortunately, there are some situations in which side effects in functions are highly desirable. We saw one example in the `gen_new_name` function of Figure 3.6 (page 125). Another arises in the typical interface to a pseudo-random number generator.

```
procedure srand(seed : integer)
 -- Initialize internal tables.
 -- The pseudo-random generator will return a different
 -- sequence of values for each different value of seed.

function rand() : integer
 -- No arguments; returns a new "random" number.
```

Obviously `rand` needs to have a side effect, so that it will return a different value each time it is called. One could always recast it as a procedure with a reference parameter:

```
procedure rand(var n : integer)
```

but most programmers would find this less appealing. Ada strikes a compromise: it allows side effects in functions in the form of changes to static or global variables, but does not allow a function to modify its parameters. ■

## 6.4 Selection

### EXAMPLE 6.45

Selection in Algol 60

Selection statements in most imperative languages employ some variant of the `if...then...else` notation introduced in Algol 60:

```
if condition then statement
else if condition then statement
else if condition then statement
...
else statement
```

As we saw in Section 2.3.2, languages differ in the details of the syntax. In Algol 60 and Pascal both the `then` clause and the `else` clause are defined to contain a single statement (this can of course be a `begin...end` compound statement). To avoid grammatical ambiguity, Algol 60 requires that the statement after the `then` begin with something other than `if` (`begin` is fine). Pascal eliminates this restriction in favor of a “disambiguating rule” that associates an `else` with the closest unmatched `then`. Algol 68, Fortran 77, and more modern languages avoid

the ambiguity by allowing a statement *list* to follow either `then` or `else`, with a terminating keyword at the end of the construct.

**EXAMPLE 6.46**

Elsif/elif

To keep terminators from piling up at the end of nested `if` statements, most languages with terminators provide a special `elsif` or `elif` keyword. In Modula-2, one writes

```
IF a = b THEN ...
ELSIF a = c THEN ...
ELSIF a = d THEN ...
ELSE ...
END
```

**EXAMPLE 6.47**

Cond in Lisp

In Lisp, the equivalent construct is

```
(cond
 ((= A B)
 (...))
 ((= A C)
 (...))
 ((= A D)
 (...))
 (T
 (...)))
```

Here `cond` takes as arguments a sequence of pairs. In each pair the first element is a condition; the second is an expression to be returned as the value of the overall construct if the condition evaluates to `T` (`T` means “true” in most Lisp dialects).

### 6.4.1 Short-Circuited Conditions

While the condition in an `if...then...else` statement is a Boolean expression, there is usually no need for evaluation of that expression to result in a Boolean value in a register. Most machines provide conditional branch instructions that capture simple comparisons. Put another way, the purpose of the Boolean expression in a selection statement is not to compute a value to be stored, but to cause control to branch to various locations. This observation allows us to generate particularly efficient code (called *jump code*) for expressions that are amenable to the short-circuit evaluation of Section 6.1.5. Jump code is applicable not only to selection statements such as `if...then...else`, but to logically controlled loops as well; we will consider the latter in Section 6.5.5.

In the usual process of code generation, either via an attribute grammar or via ad hoc syntax tree decoration, a synthesized attribute of the root of an expression subtree acquires the name of a register into which the value of the expression will be computed at run time. The surrounding context then uses this register name when generating code that uses the expression. In jump code, *inherited* attributes of the root inform it of the addresses to which control should branch if the ex-

**EXAMPLE 6.48**

Code generation for a Boolean condition

pression is true or false respectively. Jump code can be generated quite elegantly by an attribute grammar, particularly one that is *not* L-attributed (Exercise 6.9).

Suppose, for example, that we are generating code for the following source.

```
if ((A > B) and (C > D)) or (E ≠ F) then
 then_clause
else
 else_clause
```

In Pascal, which does not use short-circuit evaluation, the output code would look something like this.

```
r1 := A -- load
r2 := B
r1 := r1 > r2
r2 := C
r3 := D
r2 := r2 > r3
r1 := r1 & r2
r2 := E
r3 := F
r2 := r2 ≠ r3
r1 := r1 | r2
if r1 = 0 goto L2
L1: then_clause -- (label not actually used)
 goto L3
L2: else_clause
L3:
```

**EXAMPLE 6.49**

Code generation for short-circuiting

The root of the subtree for  $((A > B) \text{ and } (C > D)) \text{ or } (E \neq F)$  would name  $r1$  as the register containing the expression value. ■

In jump code, by contrast, the inherited attributes of the condition's root would indicate that control should “fall through” to  $L1$  if the condition is true, or branch to  $L2$  if the condition is false. Output code would then look something like this:

```
r1 := A
r2 := B
if r1 <= r2 goto L4
r1 := C
r2 := D
if r1 > r2 goto L1
L4: r1 := E
r2 := F
if r1 = r2 goto L2
L1: then_clause
 goto L3
L2: else_clause
L3:
```

Here the value of the Boolean condition is never explicitly placed into a register. Rather it is implicit in the flow of control. Moreover for most values of A, B, C, D, and E, the execution path through the jump code is shorter and therefore faster (assuming good branch prediction) than the straight-line code that calculates the value of every subexpression. ■

**EXAMPLE 6.50**

Short-circuit creation of a Boolean value

```
found_it := p /= null and then p.key = val;
```

is equivalent to

```
if p /= null and then p.key = val then
 found_it := true;
else
 found_it := false;
end if;
```

and can be translated as

```
r1 := p
if r1 = 0 goto L1
r2 := r1→key
if r2 ≠ val goto L1
r1 := 1
goto L2
L1: r1 := 0
L2: found_it := r1
```

The astute reader will notice that the first goto L1 can be replaced by goto L2, since r1 already contains a zero in this case. The code improvement phase of the compiler will notice this also, and make the change. It is easier to fix this sort of thing in the code improver than it is to generate the better version of the code in the first place. The code improver has to be able to recognize jumps to redundant instructions for other reasons anyway; there is no point in building special cases into the short-circuit evaluation routines. ■

**DESIGN & IMPLEMENTATION****Short-circuit evaluation**

Short-circuit evaluation is one of those happy cases in programming language design where a clever language feature yields both more useful semantics *and* a faster implementation than existing alternatives. Other at least arguable examples include case statements, local scopes for for loop indices (Section 6.5.1), with statements in Pascal (Section 7.3.3), and parameter modes in Ada (Section 8.3.1).

### 6.4.2 Case/Switch Statements

**EXAMPLE 6.51**

Case statements and nested ifs

The case statements of Algol W and its descendants provide alternative syntax for a special case of nested if...then...else. When each condition compares the same integer expression to a different compile-time constant, then the following code (written here in Modula-2)

```
i := ... (* potentially complicated expression *)
IF i = 1 THEN
 clause_A
ELSIF i IN 2, 7 THEN
 clause_B
ELSIF i IN 3..5 THEN
 clause_C
ELSIF (i = 10) THEN
 clause_D
ELSE
 clause_E
END
```

can be rewritten as

```
CASE ... (* potentially complicated expression *) OF
 1: clause_A
 | 2, 7: clause_B
 | 3..5: clause_C
 | 10: clause_D
 ELSE clause_E
END
```

The elided code fragments (*clause\_A*, *clause\_B*, etc.) after the colons and the ELSE are called the *arms* of the CASE statement. The lists of constants in front of the colons are CASE statement *labels*. The constants in the label lists must be disjoint, and must be of a type compatible with the tested expression. Most languages allow this type to be anything whose values are discrete: integers, characters, enumerations, and subranges of the same. C# allows strings as well. ■

The CASE statement version of the code above is certainly less verbose than the IF...THEN...ELSE version, but syntactic elegance is not the principal motivation for providing a CASE statement in a programming language. The principal motivation is to facilitate the generation of efficient target code. The IF...THEN...ELSE statement is most naturally translated as follows.

```
r1 := ... -- calculate tested expression
if r1 ≠ 1 goto L1
clause_A
goto L6
L1: if r1 = 2 goto L2
 if r1 ≠ 7 goto L3
L2: clause_B
 goto L6
```

**EXAMPLE 6.52**

Translation of nested ifs

```

 goto L6 -- jump to code to compute address
L1: clause_A
 goto L7
L2: clause_B
 goto L7
L3: clause_C
 goto L7
 ...
L4: clause_D
 goto L7
L5: clause_E
 goto L7

L6: r1 := ... -- computed target of branch
 goto *r1
L7:

```

**Figure 6.4** General form of target code generated for a five-arm case statement. One could eliminate the initial goto L6 and the final goto L7 by computing the target of the branch at the top of the generated code, but it may be cumbersome to do so, particularly in a one-pass compiler. The form shown adds only a single jump to the control flow in most cases, and allows the code for all of the arms of the `case` statement to be generated as encountered, before the code to determine the target of the branch can be deduced.

```

L3: if r1 < 3 goto L4
 if r1 > 5 goto L4
 clause_C
 goto L6
L4: if r1 ≠ 10 goto L5
 clause_D
 goto L6
L5: clause_E
L6:

```

Rather than test its expression sequentially against a series of possible values, the `case` statement is meant to *compute* an address to which it jumps in a single instruction. The general form of the target code generated from a `case` statement appears in Figure 6.4. The code at label L6 can take any of several forms. The most common of these simply indexes into an array:

```

T: &L1 -- tested expression = 1
 &L2
 &L3
 &L3
 &L3
 &L5
 &L2
 &L5
 &L5
 &L4 -- tested expression = 10

```

#### EXAMPLE 6.53

##### Jump tables

```

L6: r1 := ... -- calculate tested expression
 if r1 < 1 goto L5
 if r1 > 10 goto L5 -- L5 is the "else" arm
 r1 -= 1 -- subtract off lower bound
 r2 := T[r1]
 goto *r2
L7:

```

Here the “code” at label *T* is actually a table of addresses, known as a *jump table*. It contains one entry for each integer between the lowest and highest values, inclusive, found among the `case` statement labels. The code at *L6* checks to make sure that the tested expression is within the bounds of the array (if not, we should execute the `else` arm of the `case` statement). It then fetches the corresponding entry from the table and branches to it. ■

#### **Alternative Implementations**

A linear jump table is fast. It is also space-efficient when the overall set of `case` statement labels is dense and does not contain large ranges. It can consume an extraordinarily large amount of space, however, if the set of labels is nondense or includes large value ranges. Alternative methods to compute the address to which to branch include sequential testing, hashing, and binary search. Sequential testing (as in an `if...then...else` statement) is the method of choice if the total number of `case` statement labels is small. It runs in time  $O(n)$ , where  $n$  is the number of labels. A hash table is attractive if the range of label values is large but has many missing values and no large ranges. With an appropriate hash function it will run in time  $O(1)$ . Unfortunately, a hash table requires a separate entry for each possible value of the tested expression, making it unsuitable for statements with large value ranges. Binary search can accommodate ranges easily. It runs in time  $O(\log n)$ , with a relatively low constant factor.

To generate good code for all possible `case` statements, a compiler needs to be prepared to use a variety of strategies. During compilation it can generate code for the various arms of the `case` statement as it finds them, while simultaneously building up an internal data structure to describe the label set. Once it has seen all the arms, it can decide which form of target code to generate. For the sake of simplicity, most compilers employ only some of the possible implementations. Many use binary search in lieu of hashing. Some generate only indexed jump tables; others only that plus sequential testing. Users of less sophisticated compilers may need to restructure their `case` statements if the generated code turns out to be unexpectedly large or slow.

#### **Syntax and Label Semantics**

As with `if...then...else` statements, the syntactic details of `case` statements vary from language to language. In keeping with the style of its other structured statements, Pascal defines each arm of a `case` statement to contain a single statement; `begin...end` delimiters are required to bracket statement lists. Modula, Ada, Fortran 90, and many other languages expect arms to contain statement

lists by default. Modula uses `|` to separate an arm from the following label. Ada brackets labels with `when` and `=>`.

Standard Pascal does not include a default clause: all values on which to take action must appear explicitly in label lists. It is a dynamic semantic error for the expression to evaluate to a value that does not appear. Most Pascal compilers permit the programmer to add a default clause, labeled either `else` or `otherwise`, as a language extension. Modula allows an optional `else` clause. If one does not appear in a given `case` statement, then it is a dynamic semantic error for the tested expression to evaluate to a missing value. Ada requires arm labels to cover *all* possible values in the domain of the type of the tested expression. If the type of tested expression has a very large number of values, then this coverage must be accomplished using ranges or an `others` clause. In some languages, notably C and Fortran 90, it is *not* an error for the tested expression to evaluate to a missing value. Rather, the entire construct has no effect when the value is missing.

### **The C switch Statement**

C's syntax for `case` (`switch`) statements (retained by C++ and Java) is unusual in other respects.

```
switch (... /* tested expression */) {
 case 1: clause_A
 break;
 case 2:
 case 7: clause_B
 break;
 case 3:

 case 4:
 case 5: clause_C
 break;
 case 10: clause_D
 break;
 default: clause_E
 break;
}
```

## DESIGN & IMPLEMENTATION

### Case statements

Case statements are one of the clearest examples of language design driven by implementation. Their primary reason for existence is to facilitate the generation of jump tables. Ranges in label lists (not permitted in Pascal or C) may reduce efficiency slightly, but binary search is still dramatically faster than the equivalent series of `ifs`.

Here each possible value for the tested expression must have its own label within the `switch`; ranges are not allowed. In fact, lists of labels are not allowed, but the effect of lists can be achieved by allowing a label (such as 2, 3, and 4 above) to have an *empty* arm that simply “falls through” into the code for the subsequent label. Because of the provision for fall-through, an explicit `break` statement must be used to get out of the `switch` at the end of an arm, rather than falling through into the next. There are rare circumstances in which the ability to fall through is convenient:

```
letter_case = lower;
switch (c) {
 ...
 case 'A' :
 letter_case = upper;
 /* FALL THROUGH! */
 case 'a' :
 ...
 break;
 ...
}
```

Most of the time, however, the need to insert a `break` at the end of each arm—and the compiler’s willingness to accept arms without breaks, silently—is a recipe for unexpected and difficult-to-diagnose bugs. C# retains the familiar C syntax, including multiple consecutive labels, but requires every nonempty arm to end with a `break`, `goto`, `continue`, or `return`.

### ***Historical Origins***

#### **EXAMPLE 6.55**

Fortran computed goto

Modern `case` statements are a descendant of the `computed goto` statement of Fortran and the `switch` construct of Algol 60. In early versions of Fortran, one could specify multiway branching based on an integer value as follows.

```
goto (15, 100, 150, 200), I
```

If `I` is one, control jumps to the statement labeled 15. If `I` is two, control jumps to the statement labeled 100. If `I` is outside the range 1...4, the statement has no effect. Any integer-valued expression could be used in place of `I`. Computed `gos`tos are still allowed in Fortran 90 but are identified by the language manual as a *deprecated* feature, retained to facilitate compilation of old programs.

In Algol 60, a `switch` is essentially an array of labels:

```
switch S := L15, L100, L150, L200;
...
goto S[I];
```

Algol 68 eliminates the `gos`tos by, in essence, indexing into an array of statements, but the syntax is rather cumbersome.

 **CHECK YOUR UNDERSTANDING**

19. List the principal uses of `goto`, and the structured alternatives to each.
20. Explain the distinction between exceptions and multilevel returns.
21. What are *continuations*? What other language features do they subsume?
22. Why is sequencing a comparatively unimportant form of control flow in Lisp?
23. Explain why it may sometimes be useful for a function to have side effects.
24. Describe the *jump code* implementation of short-circuit Boolean evaluation.
25. Why do imperative languages commonly provide a `case` statement in addition to `if... then... else`?
26. Describe three different search strategies that might be employed in the implementation of a `case` statement, and the circumstances in which each would be desirable.

## 6.5 Iteration

Iteration and recursion are the two mechanisms that allow a computer to perform similar operations repeatedly. Without at least one of these mechanisms, the running time of a program (and hence the amount of work it can do and the amount of space it can use) is a linear function of the size of the program text, and the computational power of the language is no greater than that of a finite automaton. In a very real sense, it is iteration and recursion that make computers useful. In this section we focus on iteration. Recursion is the subject of Section 6.6.

Programmers in imperative languages tend to use iteration more than they use recursion (recursion is more common in functional languages). In most languages, iteration takes the form of *loops*. Like the statements in a sequence, the iterations of a loop are generally executed for their side effects: their modifications of variables. Loops come in two principal varieties; these differ in the mechanisms used to determine how many times they iterate. An *enumeration-controlled* loop is executed once for every value in a given finite set. The number of iterations is therefore known before the first iteration begins. A *logically controlled* loop is executed until some Boolean condition (which must necessarily depend on values altered in the loop) changes value. The two forms of loops share a single construct in Algol 60. They are distinct in most later languages, with the notable exception of Common Lisp, whose `loop` macro provides an astonishing array of options for initialization, index modification, termination detection, conditional execution, and value accumulation.

### 6.5.1 Enumeration-Controlled Loops

**EXAMPLE 6.57**

Early Fortran do loop

```
do 10 i = 1, 10, 2
 ...
10 continue
```

The number after the `do` is a label that must appear on some statement later in the current subroutine; the statement it labels is the last one in the *body* of the loop: the code that is to be executed multiple times. `Continue` is a “no-op”: a statement that has no effect. Using a `continue` for the final statement of the loop makes it easier to modify code later: additional “real” statements can be added to the bottom of the loop without moving the label.<sup>5</sup>

The variable name after the label is the *index* of the loop. The comma-separated values after the equals sign indicate the initial value of the index, the maximum value it is permitted to take, and the amount by which it is to increase in each iteration (this is called the *step size*). A bit more precisely, the loop above is equivalent to

```
i = 1
10 ...
 i = i + 2
 if i <= 10 goto 10
```

Index variable `i` in this example will take on the values 1, 3, 5, 7, and 9 in successive loop iterations. Compilers can translate this loop into very simple, fast code for most machines.

In practice, unfortunately, this early form of loop proved to have several problems. Some of these problems were comparatively minor. The loop bounds and step size (1, 10, and 2 in our example) were required to be positive integer constants or variables: no expressions were allowed. Fortran 77 removed this restriction, allowing arbitrary positive and negative integer and real expressions. Also, as we saw in Section 2.16 (page 57), trivial lexical errors can cause a Fortran IV compiler to misinterpret the code as an ordinary sequence of statements beginning with an assignment. Fortran 77 makes such misinterpretation less likely by allowing an extra comma after the label in the `do` loop header. Fortran 90 takes back (makes “obsolete”) the ability to use real numbers for loop bounds and step sizes. The problem with reals is that limited precision can cause comparisons (e.g., between the index and the upper bound) to produce unexpected or even implementation-dependent results when the values are close to one another.

---

**5** The `continue` statement of C probably takes its name from this typical use of the no-op in Fortran, but its semantics are very different: the C `continue` starts the next iteration of the loop even when the current one has not finished.

The more serious problems with the Fortran IV do loop are a bit more subtle:

- If statements in the body of the loop (or in subroutines called from the body of the loop) change the value of *i*, then the loop may execute a different number of times than one would assume based on the bounds in its header. If the effect is accidental, the bug is hard to find. If the effect is intentional, the code is hard to read.
- Goto statements may jump into or out of the loop. Code that jumps out and (optionally) back in again is expressly allowed (if difficult to understand). On the other hand, code that simply jumps in, without properly initializing *i*, almost certainly represents a programming error, but will not be caught by the compiler.
- If control leaves a do loop via a goto, the value of *i* is the one most recently assigned. If the loop terminates normally, however, the value of *i* is implementation-dependent. Based on Example 6.58, one might expect the final value to be the first one outside the loop bounds:  $L + (\lfloor (U-L)/S \rfloor + 1) \times S$ , where *L*, *U*, and *S* are the lower and upper bounds of the loop and the step size, respectively. Unfortunately, if the upper bound is close to the largest value that can be represented given the precision of integers on the target machine, then the increment at the bottom of the final iteration of the loop may cause arithmetic overflow. On most machines this overflow will result in an apparently negative value, which will prevent the loop from terminating correctly. On some it will cause a run-time exception that requires the intervention of the operating system in order to continue execution. To ensure correct termination and/or avoid the cost of an exception, a compiler must generate more complex (and slower) code when it is unable to rule out overflow at compile time. In this event, the index may contain its final value (not the “next” value) after normal termination of the loop.
- Because the test against the upper bound appears at the bottom of the loop, the body will always be executed at least once, even if the “low” bound is larger than the “high” bound.

## DESIGN & IMPLEMENTATION

### Numerical imprecision

The writers of numerical software know that the results of arithmetic computations are often approximations. A comparison between values that are approximately equal “may go either way.” The Fortran 90 designers appear to have decided that such comparisons should be explicit. Fortran 90 do loops, like the for loops of most other languages, reflect the precision of discrete types. The programmer who wants to control iteration with floating-point values must use an explicit comparison in a pre-test or post-test loop (Section 6.5.5).

**EXAMPLE 6.59****Modula-2 for loop**

These problems arise in a larger context than merely Fortran IV. They must be addressed in the design of enumeration-controlled loops in any language. Consider the arguably more friendly syntax of Modula-2:

```
FOR i := first TO last BY step DO
 ...
END
```

where `first`, `last`, and `step` can be arbitrarily complex expressions of an integer, enumeration, or subrange type. Based on the preceding discussion, one might ask several questions.

1. Can `i`, `first`, and/or `last` be modified in the loop? If so, what is the effect on control?
2. What happens if `first` is larger than `last` (or smaller, in the case of a negative `step`)?
3. What is the value of `i` when the loop is finished?
4. Can control jump into the loop from outside?

We address these questions in the paragraphs below. ■

#### ***Changes to Loop Indices or Bounds***

Most languages, including Algol 68, Pascal, Ada, Fortran 77 and 90, and Modula-3, prohibit changes to the loop index within the body of an enumeration-controlled loop. They also guarantee to evaluate the bounds of the loop exactly once, before the first iteration, so any changes to variables on which those bounds depend will not have any effect on the number of iterations executed. Modula-2 is vague; the manual says that the index “should not be changed” by the body of the loop [Wir85b, Sec. 9.8]. ISO Pascal goes to considerable lengths to prohibit modification. Paraphrasing slightly, it says [Int90, Sec. 6.8.3.9] that the index variable must be declared in the closest enclosing block, and that neither the body of the `for` statement itself nor any statement contained in a subroutine local to the block can “threaten” the index variable. A statement is said to threaten a variable if it

- Assigns to it
- Passes it to a subroutine by reference
- Reads it from a file
- Is a structured statement containing a simpler statement that threatens it

The prohibition against threats in local subroutines is made because a local variable will be accessible to those subroutines, and one of them, if called from within the loop, might change the value of the variable even if it is not passed to it by reference.

### Empty Bounds

Modern languages refrain from executing an enumeration-controlled loop if the bounds are empty. In other words, they test the terminating condition *before* the first iteration. The initial test requires a few extra instructions but leads to much more intuitive behavior. The loop

#### EXAMPLE 6.60

Obvious translation of a  
for loop

```
FOR i := first TO last BY step DO
 ...
END
```

can be translated as

```
r1 := first
r2 := step
r3 := last
L1: if r1 > r3 goto L2
 ...
 -- loop body; use r1 for i
 r1 := r1 + r2
 goto L1
L2:
```

#### EXAMPLE 6.61

For loop translation with  
test at the bottom

```
r1 := first
r2 := step
r3 := last
goto L2
L1: ...
 -- loop body; use r1 for i
 r1 := r1 + r2
L2: if r1 ≤ r3 goto L1
```

The advantage of this second version is that each iteration of the loop contains a single conditional branch, rather than a conditional branch at the top and an unconditional branch at the bottom. (We will consider yet another version in Exercise ⑩ 15.4.)

The translations shown above work only if  $\text{first} + (\lfloor(\text{last} - \text{first})/\text{step}\rfloor + 1) \times \text{step}$  does not exceed the largest representable integer. If the compiler cannot verify this property at compile time, then it will have to generate more cautious code (to be discussed in Example 6.63).

#### EXAMPLE 6.62

Reverse direction for loop

**Loop Direction** The astute reader may also have noticed that the code shown here implicitly assumes that *step* is positive. If *step* is negative, the test for termination must “go the other direction.” If *step* is not a compile-time constant, then the compiler cannot tell which form of test to use. Some languages, including Pascal and Ada, require the programmer to predict the sign of the step. In Pascal, one must say

```
for i := 10 downto 1 do ...
```

In Ada, one must say

```
for i in reverse 1..10 do ...
```

Modula-2 and Modula-3 do not require special syntax for “backward” loops, but insist that `step` be a compile-time constant so the compiler can tell the difference (Modula (1) has no `for` loop). ■

**EXAMPLE 6.63**

For loop translation with iteration count

```
r1 := first
r2 := step
r3 := max(└(last - first + step)/step┘, 0) -- iteration count
-- NB: this calculation may require several instructions.
-- It is guaranteed to result in a value within the precision
of the machine,
-- but we have to be careful to avoid overflow during its calculation.
if r3 ≤ 0 goto L2
L1: ... -- loop body; use r1 for i
 r1 := r1 + r2
 r3 := r3 - 1
 if r3 > 0 goto L1
 i := r1
L2:
```

The use of the iteration count avoids the need to test the sign of `step` within the loop. It also avoids problems with overflow when testing the terminating condition (assuming that we have been suitably careful in calculating the iteration count). Some processors, including the PowerPC, PA-RISC, and most CISC machines, can decrement the iteration count, test it against zero, and conditionally branch, all in a single instruction. In simple cases, the code improvement phase of the compiler may be able to use a technique known as *induction variable elimination* to eliminate the need to maintain both `r1` and `r3`. ■

**Access to the Index Outside the Loop**

Several languages, including Fortran IV and Pascal, leave the value of the loop index undefined after termination of the loop. Others, such as Fortran 77 and Algol 60, guarantee that the value is the one “most recently assigned.” For “normal” termination of the loop, this is the first value that exceeds the upper bound. It is not clear what happens if this value exceeds the largest value representable on the machine (or the smallest value in the case of a negative step size). A similar question arises in Pascal, in which the type of an index can be a subrange or enumeration. In this case the first value “after” the upper bound can often be invalid.

**EXAMPLE 6.64**

Index value after loop

```
var c : 'a'...'z';
...
for c := 'a' to 'z' do begin
 ...
end;
(* what comes after 'z'? *)
```

**EXAMPLE 6.65**

Preserving the final index value

Examples like this illustrate the rationale for leaving the final value of the index undefined in Pascal. The alternative—defining the value to be the last one that was valid—would force the compiler to generate slower code for every loop, with two branches in each iteration instead of one:

```
r1 := 'a'
r2 := 'z'
if r1 > r2 goto L3 -- Code improver may remove this test,
 -- since 'a' and 'z' are constants.
L1: ... -- loop body; use r1 for i
 if r1 = r2 goto L2
 r1 := r1 + 1
 -- NB: Pascal step size is always 1 (or -1 if downto)
 goto L1
L2: i := r1
L3:
```

Note that the compiler must generate this sort of code in any event (or use an iteration count) if arithmetic overflow may interfere with testing the terminating condition.

Several languages, including Algol W, Algol 68, Ada, Modula-3, and C++, avoid the issue of the value held by the index outside the loop by making the index a local variable of the loop. The header of the loop is considered to contain a *declaration* of the index. Its type is inferred from the bounds of the loop, and its scope is the loop's body. Because the index is not visible outside the loop, its value is not an issue. Since it is not visible even to local subroutines, much of the concept of “threatening” in Pascal becomes unnecessary. Finally, there is no chance that a value held in the index variable before the loop, and needed after, will inadvertently be destroyed. (Of course, the programmer must not give the index the same name as any variable that must be accessed within the loop, but this is a strictly local issue: it has no ramifications outside the loop.)

**DESIGN & IMPLEMENTATION****For loops**

Modern `for` loops reflect the impact of both semantic and implementation challenges. As suggested by the subheadings of Section 6.5.1, the semantic challenges include changes to loop indices or bounds from within the loop, the scope of the index variable (and its value, if any, outside the loop), and `goto`s that enter or leave the loop. Implementation challenges include the imprecision of floating-point values (discussed in the sidebar on page 272), the direction of the bottom-of-loop test, and overflow at the end of the iteration range. The “combination loops” of C (to be discussed in Section 6.5.2) move responsibility for these challenges out of the compiler and into the application program.

**Jumps**

Algol 60, Fortran 77, and most of their successors place restrictions on the use of the `goto` statement that prevent it from entering a loop from outside. `Gotos` can be used to *exit* a loop prematurely, but this is a comparatively clean operation; questions of uninitialized indices and bounds do not arise. As we shall see in Section 6.5.5, many languages provide an `exit` statement as a semistructured alternative to a loop-escaping `goto`.

**6.5.2 Combination Loops****EXAMPLE 6.66****Algol 60 for loop**

Algol 60, as mentioned above, provides a single loop construct that subsumes the properties of more modern enumeration- and logically controlled loops. The general form is given by

```

for_stmt → for id := for_list do stmt
for_list → enumerator (, enumerator)*
enumerator → expr
→ expr step expr until expr
→ expr while condition

```

Here the index variable takes on values specified by a sequence of enumerators, each of which can be a single value, a range of values similar to that of modern enumeration-controlled loops, or an expression with a terminating condition. Each expression in the current enumerator is reevaluated at the top of the loop. This reevaluation is what makes the `while` form of enumerator useful: its condition typically depends on the current value of the index variable. All of the following are equivalent.

```

for i := 1, 3, 5, 7, 9 do ...
for i := 1 step 2 until 10 do ...
for i := 1, i + 2 while i < 10 do ...

```

In practice the generality of the Algol 60 `for` loop turns out to be overkill. The repeated reevaluation of bounds, in particular, can lead to loops that are very hard to understand. Some of the power of the Algol 60 loop is retained in a cleaner form in the `for` loop of C. A substantially *more* powerful version (not described here) is found in Common Lisp.

C's `for` loop is, strictly speaking, logically controlled. Any enumeration-controlled loop, however, can be rewritten in a logically controlled form (this is of course what the compiler does when it translates into assembler), and C's `for` loop is deliberately designed to facilitate writing the logically controlled equivalent of a Pascal or Algol-style `for` loop. Our Modula-2 example

```

FOR i := first TO last BY step DO
...
END

```

**EXAMPLE 6.67****Combination (for) loop  
in C**

would usually be written in C as

```
for (i = first; i <= last; i += step) {
 ...
}
```

C defines this to be roughly equivalent to

```
i = first;
while (i <= last) {
 ...
 i += step;
}
```



This definition means that it is the programmer's responsibility to worry about the effect of overflow on testing of the terminating condition. It also means that both the index and any variables contained in the terminating condition can be modified by the body of the loop, or by subroutines it calls, and these changes *will* affect the loop control. This, too, is the programmer's responsibility.

Any of the three substatements in the `for` loop header can be null (the condition is considered true if missing). Alternatively, a substatement can consist of a sequence of comma-separated expressions. The advantage of the C `for` loop over its `while` loop equivalent is compactness and clarity. In particular, all of the code affecting the flow of control is localized within the header. In the `while` loop, one must read both the top and the bottom of the loop to know what is going on.

### 6.5.3 Iterators

In all of the examples we have seen so far (with the possible exception of the combination loops of Algol 60, Common Lisp, or C), a `for` loop iterates over the elements of an arithmetic sequence. In general, however, we may wish to iterate over the elements of any well-defined set (what are often called *containers* or *collections* in object-oriented code). Clu introduced an elegant *iterator* mechanism (also found in Python, Ruby, and C#) to do precisely that. Euclid and several more recent languages, notably C++ and Java, define a standard interface for *iterator objects* (sometimes called *enumerators*) that are equally easy to use but not as easy to write. Icon, conversely, provides a generalization of iterators, known as *generators*, that combines enumeration with backtracking search.<sup>6</sup>

#### **True Iterators**

Clu, Python, Ruby, and C# allow any container abstraction to provide an *iterator* that enumerates its items. The iterator resembles a subroutine that is permitted to

---

**6** Unfortunately, terminology is not consistent across languages. Euclid uses the term "generator" for what are called "iterator objects" here. Python uses it for what are called "true iterators" here.

**EXAMPLE 6.68**

Simple iterator in Clu

contain `yield` statements, each of which produces a loop index value. For loops are then designed to incorporate a call to an iterator. The Modula-2 fragment

```
FOR i := first TO last BY step DO
 ...
END
```

would be written as follows in Clu.

```
for i in int$from_to_by(first, last, step) do
 ...
end
```

Here `from_to_by` is a built-in iterator that yields the integers from `first` to `first + ⌊(last - first)/step⌋ × step` in increments of `step`. ■

When called, the iterator calculates the first index value of the loop, which it returns to the main program by executing a `yield` statement. The `yield` behaves like `return`, except that when control transfers back to the iterator after completion of the first iteration of the loop, the iterator continues where it last left off—not at the beginning of its code. When the iterator has no more elements to yield it simply returns (without a value), thereby terminating the loop.

In effect, an iterator is a separate thread of control, with its own program counter, whose execution is interleaved with that of the `for` loop to which it supplies index values.<sup>7</sup> The iteration mechanism serves to “decouple” the algorithm required to enumerate elements from the code that uses those elements.

As an illustrative example, consider the pre-order enumeration of nodes from a binary tree. A Clu iterator for this task appears in Figure 6.5. Invoked from the header of a `for` loop, it takes the root of a tree as argument. It yields the root node for the first iteration and then calls itself recursively, twice, to enumerate the nodes of the left and right subtrees. ■

**Iterator Objects**

As realized in most imperative languages, iteration involves both a special form of `for` loop and a mechanism to enumerate values for the loop. These concepts can be separated. Euclid, C++, and Java all provide enumeration-controlled loops reminiscent of those of Clu. They have no `yield` statement, however, and no separate thread-like context to enumerate values; rather, an iterator is an ordinary object (in the object-oriented sense of the word) that provides methods for initialization, generation of the next index value, and testing for completion. Between calls, the state of the iterator must be kept in the object’s data members.

Figure 6.6 contains the Java equivalent of the code in Figure 6.5. The `for` loop at the bottom is syntactic sugar for

**EXAMPLE 6.69**

Clu iterator for tree enumeration

**EXAMPLE 6.70**

Java iterator for tree enumeration

<sup>7</sup> Because iterators are interleaved with loops in a very regular way, they can be implemented more easily (and cheaply) than fully general threads. We will consider implementation options further in Section 8.6.3.

```

bin_tree = cluster is ..., pre_order, ... % export list
node = record [left, right: bin_tree, val: int]
rep = variant [some: node, empty: null]
...
pre_order = iter(t: cvt) yields(bin_tree)
tagcase t
 tag empty: return
 tag some(n: node):
 yield(n.val)
 for i: int in pre_order(n.left) do
 yield(i)
 end
 for i: int in pre_order(n.right) do
 yield(i)
 end
 end
end pre_order
...
end bin_tree
...
for i: int in bin_tree$pre_order(e) do
 stream$putl(output, int$unparse(i))
end

```

**Figure 6.5** Clu iterator for pre-order enumeration of the nodes of a binary tree. In this (simplistic) example we have assumed that the datum in a tree node is simply an `int`. Within the `bin_tree` cluster, the `rep` (representation) declaration indicates that a binary tree is either a `node` or empty. The `cvt` (convert) in the header of `pre_order` indicates that parameter `t` is a `bin_tree` whose internal structure (`rep`) should be visible to the code of `pre_order` itself but not to the caller. In the `for` loop at the bottom, `int$unparse` produces the character string equivalent of a given `int`, and `stream$putl` prints a line to the specified stream.

```

for (Iterator<Integer> it = myTree.iterator(); it.hasNext();) {
 Integer i = it.next();
 System.out.println(i);
}

```

#### DESIGN & IMPLEMENTATION

##### “True” iterators and iterator objects

While the `iterator` library mechanisms of C++ and Java are highly useful, it is worth emphasizing that they are *not* the functional equivalents of “true” iterators, as found in Clu, Python, Ruby, and C#. Their key limitation is the need to maintain all intermediate state in the form of explicit data structures, rather than in the program counter and local variables of a resumable execution context.

```

class TreeNode<T> implements Iterable<T> {
 TreeNode<T> left;
 TreeNode<T> right;
 T val;
 ...
 public Iterator<T> iterator() {
 return new TreeIterator(this);
 }
 private class TreeIterator implements Iterator<T> {
 private Stack<TreeNode<T>> s = new Stack<TreeNode<T>>();
 TreeIterator(TreeNode<T> n) {
 s.push(n);
 }
 public boolean hasNext() {
 return !s.empty();
 }
 public T next() {
 if (!hasNext())
 throw new NoSuchElementException();
 }
 TreeNode<T> n = s.pop();
 if (n.right != null) {
 s.push(n.right);
 }
 if (n.left != null) {
 s.push(n.left);
 }
 return n.val;
 }
 public void remove() {
 throw new UnsupportedOperationException();
 }
 }
 ...
}
...
TreeNode<Integer> myTree = ...
...
for (Integer i : myTree) {
 System.out.println(i);
}

```

**Figure 6.6** Java code for pre-order enumeration of the nodes of a binary tree. The nested `TreeIterator` class uses an explicit `Stack` object (borrowed from the standard library) to keep track of subtrees whose nodes have yet to be enumerated. Java generics, specified as `<T>` type arguments for `TreeNode`, `Stack`, `Iterator`, and `Iterable`, allow `next` to return an object of the appropriate type (here `Integer`), rather than the undifferentiated `Object`. The `remove` method is part of the `Iterator` interface and must therefore be provided, if only as a placeholder.

The expression following the colon in the concise version of the loop header must support the standard `Iterable` interface, which includes an `iterator()` method that returns an `Iterator` object.

**EXAMPLE 6.71**
**Iterator objects in C++**

C++ takes a different tack. Rather than propose a special version of the `for` loop that would interface with iterator objects, the designers of the C++ standard library used the language's unusually flexible overloading and reference mechanisms (Sections 3.6.2 and 8.3.1) to redefine comparison (`!=`), increment (`++`), dereference (`*`), and so on, in a way that makes iterating over the elements of a set look very much like using pointer arithmetic (Section 7.7.1) to traverse a conventional array:

```
tree_node<int> *my_tree = ...
...
for (tree_node<int>::iterator n = my_tree->begin();
 n != my_tree->end(); ++n) {
 cout << *n << "\n";
}
```

C++ encourages programmers to think of iterators as if they were pointers. Iterator `n` in this example encapsulates all the state encapsulated by iterator `it` in the (no syntactic sugar) Java code of Example 6.70. To obtain the next element of the set, however, the C++ programmer “dereferences” `n`, using the `*` or `->` operators. To advance to the following element, the programmer uses the increment (`++`) operator. The `end` method returns a reference to a special iterator that “points beyond the end” of the set. The increment (`++`) operator must return a reference that tests equal to this special iterator when the set has been exhausted.

We leave the code of the C++ tree iterator to Exercise 6.15. The details are somewhat messier than Figure 6.6, due to operator overloading, the value model of variables (which requires explicit references and pointers), and the lack of garbage collection. Also, because C++ lacks a common `Object` base class, its container classes are always type-specific. Where generics can minimize the need for type casts in Java and C#, they serve a more fundamental role in C++: without them one cannot write safe, general purpose container code.

***Iterating with First-Class Functions***

In functional languages, the ability to specify a function “inline” facilitates a programming idiom in which the body of a loop is written as a function, with the loop index as an argument. This function is then passed as the final argument to an iterator. In Scheme we might write

```
(define upto
 (lambda (low high step f)
 (if (<= low high)
 (begin
 (f low)
 (upto (+ low step) high step f))
 '())))
```

**EXAMPLE 6.72**
**Passing the “loop body” to an iterator in Scheme**

We could then sum the first 50 odd numbers as follows.

```
(let ((sum 0))
 (uptoby 1 100 2
 (lambda (i)
 (set! sum (+ sum i))))
 sum) ==> 2500
```

Here the body of the loop, `(set! sum (+ sum i))`, is an assignment. The  $\Rightarrow$  symbol (not a part of Scheme) is used here to mean “evaluates to.”

Smalltalk, which we consider in Section 9.6.1, provides mechanisms that support a similar idiom:

```
sum <- 0.
1 to: 100 by: 2 do:
 [:i | sum <- sum + i]
```

Like a `lambda` expression in Scheme, a square-bracketed *block* in Smalltalk creates a first-class function, which we then pass as argument to the `to: by: do:` iterator. The iterator calls the function repeatedly, passing successive values of the index variable `i` as argument. Iterators in Ruby employ a similar but somewhat less general mechanism: where a Smalltalk method can take an arbitrary number of blocks as argument, a Ruby method can take only one. Continuations (Section 6.2.2) and lazy evaluation (Section 6.6.2) also allow the Scheme/Lisp programmer to create iterator objects and more traditional style true iterators; we consider these options in Exercises 6.30 and 6.31.

### **Iterating without Iterators**

---

#### **EXAMPLE 6.74**

Imitating iterators in C

In a language with neither true iterators nor iterator objects, one can still decouple set enumeration from element use through programming conventions. In C, for example, one might define a `tree_iter` type and associated functions that could be used in a loop as follows.

```
tree_node *my_tree;
tree_iter ti;
...
for (ti_create(my_tree, &ti); !ti_done(ti); ti_next(&ti)) {
 tree_node *n = ti_val(ti);
 ...
}
ti_delete(&ti);
```

There are two principal differences between this code and the more structured alternatives: (1) the syntax of the loop is a good bit less elegant (and arguably more prone to accidental errors), and (2) the code for the iterator is simply a type and some associated functions; C provides no abstraction mechanism to group them together as a module or a class. By providing a standard interface for iterator abstractions, object-oriented languages like C++, Python, Ruby, Java, and C# facilitate the design of higher-order mechanisms that manipulate whole

containers: sorting them, merging them, finding their intersection or difference, and so on. We leave the C code for `tree_iter` and the various `ti_` functions to Exercise 6.16. ■

#### 6.5.4 Generators in Icon

Icon generalizes the concept of iterators, providing a *generator* mechanism that causes any expression in which it is embedded to enumerate multiple values on demand.

##### IN MORE DEPTH

Icon's enumeration-controlled loop, the `every` loop, can contain not only a generator, but any expression that *contains* a generator. Generators can also be used in constructs like `if` statements, which will execute their nested code if *any* generated value makes the condition true, automatically searching through all the possibilities. When generators are nested, Icon explores all possible combinations of generated values, and will even *backtrack* where necessary to undo unsuccessful control-flow branches or assignments.

#### 6.5.5 Logically Controlled Loops

In comparison to enumeration-controlled loops, logically controlled loops have many fewer semantic subtleties. The only real question to be answered is where within the body of the loop the terminating condition is tested. By far the most common approach is to test the condition before each iteration. The familiar `while` loop syntax to do this was introduced in Algol-W and retained in Pascal:

```
while condition do statement
```

As with selection statements, most Pascal successors use an explicit terminating keyword, so that the body of the loop can be a statement list. ■

Neither (pre-90) Fortran nor Algol 60 really provides a `while` loop construct; their loops were designed to be controlled by enumeration. To obtain the effect of a `while` loop in Fortran 77, one must resort to `gotos`:

```
10 if negated_condition goto 20
 ...
 goto 10
20
```

##### Post-test Loops

Occasionally it is handy to be able to test the terminating condition at the bottom of a loop. Pascal introduced special syntax for this case, which was retained in Modula but dropped in Ada. A *post-test loop* allows us, for example, to write

##### EXAMPLE 6.75

While loop in Pascal

##### EXAMPLE 6.76

Imitating while loops in  
Fortran 77

##### EXAMPLE 6.77

Post-test loop in Pascal  
and Modula

```
repeat
 readln(line)
until line[1] = '$';
```

instead of

```
readln(line);
while line[1] <> '$' do
 readln(line);
```

The difference between these constructs is particularly important when the body of the loop is longer. Note that the body of a post-test loop is always executed at least once. ■

#### **EXAMPLE 6.78**

Post-test loop in C

#### **EXAMPLE 6.79**

Midtest loop in Modula

C provides a post-test loop whose condition works “the other direction” (i.e., “while” instead of “until”):

```
do {
 line = read_line(stdin);
} while line[0] != '$';
```

#### **Midtest Loops**

Finally, as we saw in Section 6.2, it is sometimes appropriate to test the terminating condition in the middle of a loop. This “midtest” can be accomplished with an if and a goto in most languages, but a more structured alternative is preferable. Modula (1) introduced a *midtest*, or *one-and-a-half loop* that allows a terminating condition to be tested as many times as desired within the loop:

```
loop
 statement_list
when condition exit
 statement_list
when condition exit
 ...
end
```

Using this notation, the Pascal construct

```
while true do begin
 readln(line);
 if all_blanks(line) then goto 100;
 consume_line(line)
end;
100:
```

can be written as follows in Modula (1).

```
loop
 line := ReadLine;
when AllBlanks(line) exit;
 ConsumeLine(line)
end;
```

The `when` clause here is syntactically part of the `loop` construct. The syntax ensures that an `exit` can occur only within a `loop`, but it has the unfortunate side effect of preventing an exit from within a nested construct. ■

**EXAMPLE 6.80**

Exit as a separate statement

Modula-2 abandoned the `when` clause in favor of a simpler `EXIT` statement, which is typically placed inside an `IF` statement:

```
LOOP
 line := ReadLine;
 IF AllBlanks(line) THEN EXIT END;
 ConsumeLine(line)
END;
```

Because `EXIT` is no longer part of the `LOOP` construct syntax, the semantic analysis phase of compilation must ensure that `EXITS` appear only inside `LOOPs`. There may still be an arbitrary number of them inside a given `LOOP`. Modula-3 allows an `EXIT` to leave a `WHILE`, `REPEAT`, or `FOR` loop, as well as a plain `LOOP`. ■

**EXAMPLE 6.81**

Break statement in C

The C `break` statement, which we have already seen in the context of `switch` statements, can be used in a similar manner:

```
for (;;) {
 line = read_line(stdin);
 if (all_blanks(line)) break;
 consume_line(line);
}
```

Here the missing condition in the `for` loop header is assumed to always be true; for some reason, C programmers have traditionally considered this syntax to be stylistically preferable to the equivalent `while (1)`. ■

**EXAMPLE 6.82**

Exiting a nested loop

In Ada an `exit` statement takes an optional loop-name argument that allows control to escape a nested loop:

```
outer: loop
 get_line(line, length);
 for i in 1..length loop
 exit outer when line(i) = '$';
 consume_char(line(i));
 end loop;
end loop outer;
```

Java extends the C/C++ `break` statement in a similar fashion: Java loops can be labeled as in Ada, and the `break` statement takes an optional loop name as parameter. ■

 **CHECK YOUR UNDERSTANDING**

27. Describe three subtleties in the implementation of enumeration-controlled loops.
28. Why do most languages not allow the bounds or increment of an enumeration-controlled loop to be floating-point numbers?

29. Why do many languages require the step size of an enumeration-controlled loop to be a compile-time constant?
  30. Describe the “iteration count” loop implementation. What problem(s) does it solve?
  31. What are the advantages of making an index variable local to the loop it controls?
  32. What is a *container* (a *collection*)?
  33. Explain the difference between true iterators and iterator objects.
  34. Cite two advantages of iterator objects over the use of programming conventions in a language like C.
  35. Describe the approach to iteration typically employed in languages with first-class functions.
  36. Give an example in which a *midtest* loop results in more elegant code than does a pretest or post-test loop.
  37. Does C have enumeration-controlled loops? Explain.
- 

## 6.6 Recursion

Unlike the control-flow mechanisms discussed so far, recursion requires no special syntax. In any language that provides subroutines (particularly functions), all that is required is to permit functions to call themselves, or to call other functions that then call them back in turn. Most programmers learn in a data structures class that recursion and (logically controlled) iteration provide equally powerful means of computing functions: any iterative algorithm can be rewritten, automatically, as a recursive algorithm, and vice versa. We will compare iteration and recursion in more detail in the first subsection below. In the subsection after that we will consider the possibility of passing *unevaluated* expressions into a function. While usually inadvisable, due to implementation cost, this technique will sometimes allow us to write elegant code for functions that are only defined on a subset of the possible inputs, or that explore logically infinite data structures.

### 6.6.1 Iteration and Recursion

As we noted in Section 3.2, Fortran 77 and certain other languages do not permit recursion. A few functional languages do not permit iteration. Most modern languages, however, provide both mechanisms. Iteration is in some sense the more “natural” of the two in imperative languages, because it is based on the repeated modification of variables. Recursion is the more natural of the two in functional

**EXAMPLE 6.83**

A “naturally iterative” problem

$$\sum_{1 \leq i \leq 10} f(i)$$

it seems natural to use iteration. In C one would say

```
typedef int (*int_func) (int);
int summation(int_func f, int low, int high) {
 /* assume low <= high */
 int total = 0;
 int i;
 for (i = low; i <= high; i++) {
 total += f(i);
 }
 return total;
}
```

**EXAMPLE 6.84**

A “naturally recursive” problem

$$\text{gcd}(a, b) \equiv \begin{cases} a & \text{if } a = b \\ \text{gcd}(a - b, b) & \text{if } a > b \\ \text{gcd}(a, b - a) & \text{if } b > a \end{cases}$$

recursion may seem more natural:

```
int gcd(int a, int b) {
 /* assume a, b > 0 */
 if (a == b) return a;
 else if (a > b) return gcd(a-b, b);
 else return gcd(a, b-a);
}
```

**EXAMPLE 6.85**

Implementing problems “the other way”

In both these cases, the choice could go the other way:

```
typedef int (*int_func) (int);
int summation(int_func f, int low, int high) {
 /* assume low <= high */
 if (low == high) return f(low);
 else return f(low) + summation(f, low+1, high);
}

int gcd(int a, int b) {
 /* assume a, b > 0 */
 while (a != b) {
 if (a > b) a = a-b;
 else b = b-a;
 }
 return a;
}
```

**Tail Recursion**

It is often argued that iteration is more efficient than recursion. It is more accurate to say that *naive implementation* of iteration is usually more efficient than naive implementation of recursion. In the preceding examples, the iterative implementations of summation and greatest divisors will be more efficient than the recursive implementations if the latter make real subroutine calls that allocate space on a run-time stack for local variables and bookkeeping information. An “optimizing” compiler, however, particularly one designed for a functional language, will often be able to generate excellent code for recursive functions. It is particularly likely to do so for *tail-recursive* functions such as gcd above. A tail-recursive function is one in which additional computation never follows a recursive call: the return value is simply whatever the recursive call returns. For such functions, dynamically allocated stack space is unnecessary: the compiler can *reuse* the space belonging to the current iteration when it makes the recursive call. In effect, a good compiler will recast our recursive gcd function as

**EXAMPLE 6.86**

Implementation of tail recursion

```
int gcd(int a, int b) {
 /* assume a, b > 0 */
 start:
 if (a == b) return a;
 else if (a > b) {
 a = a-b; goto start;
 } else {
 b = b-a; goto start;
 }
}
```

Even for functions that are not tail-recursive, automatic, often simple transformations can produce tail-recursive code. The general case of the transformation employs conversion to what is known as *continuation-passing style* [FWH01, Chaps. 7–8]. In effect, a recursive function can always avoid doing any work after returning from a recursive call by passing that work into the recursive call, in the form of a continuation.

Some specific transformations (not based on continuation-passing) are often employed by skilled users of functional languages. Consider, for example, the recursive summation function of Example 6.85, written here in Scheme:

```
(define summation (lambda (f low high)
 (if (= low high)
 (f low) ; then part
 (+ (f low) (summation f (+ low 1) high)))))) ; else part
```

Recall that Scheme, like all Lisp dialects, uses Cambridge Polish notation for expressions. The `lambda` keyword is used to introduce a function. As recursive calls return, our code calculates the sum from “right to left”: from `high` down to `low`. If the programmer (or compiler) recognizes that addition is associative, we can rewrite the code in a tail-recursive form:

**EXAMPLE 6.87**

By-hand creation of tail-recursive code

```
(define summation (lambda (f low high subtotal)
 (if (= low high)
 (+ subtotal (f low))
 (summation f (+ low 1) high (+ subtotal (f low))))))
```

Here the `subtotal` parameter accumulates the sum from left to right, passing it into the recursive calls. Because it is tail-recursive, this function can be translated into machine code that does not allocate stack space for recursive calls. Of course, the programmer won't want to pass an explicit `subtotal` parameter to the initial call, so we hide it (the parameter) in an auxiliary, "helper" function:

```
(define summation (lambda (f low high)
 (letrec ((sum-helper (lambda (low subtotal)
 (let ((new_subtotal (+ subtotal (f low))))
 (if (= low high)
 new_subtotal
 (sum-helper (+ low 1) new_subtotal))))))
 (sum-helper low 0))))
```

The `let` construct in Scheme serves to introduce a nested scope in which local names (e.g., `new_subtotal`) can be defined. The `letrec` construct permits the definition of recursive functions (e.g., `sum-helper`). ■

### **Thinking Recursively**

#### **EXAMPLE 6.88**

Naive recursive Fibonacci function

$$F_n \equiv \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

The naive way to implement this recurrence in Scheme is

```
(define fib (lambda (n)
 (cond ((= n 0) 1)
 ((= n 1) 1)
 (#t (+ (fib (- n 1)) (fib (- n 2)))))))
; #t means 'true' in Scheme
```

#### **EXAMPLE 6.89**

Efficient iterative Fibonacci function

Unfortunately, this algorithm takes exponential time, when linear time is possible. In C, one might write

```
int fib(int n) {
 int f1 = 1; int f2 = 1;
 int i;
 for (i = 2; i <= n; i++) {
 int temp = f1 + f2;
 f1 = f2; f2 = temp;
 }
 return f2;
```

**EXAMPLE 6.90**

Efficient tail-recursive Fibonacci function

One can write this iterative algorithm in Scheme: Scheme includes (nonfunctional) iterative features. It is probably better, however, to draw inspiration from the tail-recursive summation function of Example 6.87 and write the following  $O(n)$  recursive function.

```
(define fib (lambda (n)
 (letrec ((fib-helper (lambda (f1 f2 i)
 (if (= i n)
 f2
 (fib-helper f2 (+ f1 f2) (+ i 1)))))))
 (fib-helper 0 1 0))))
```

For a programmer accustomed to writing in a functional style, this code is perfectly natural. One might argue that it isn't "really" recursive; it simply casts an iterative algorithm in a tail-recursive form, and this argument has some merit. Despite the algorithmic similarity, however, there is an important difference between the iterative algorithm in C and the tail-recursive algorithm in Scheme: the latter has no side effects. Each recursive call of the `fib-helper` function creates a new scope, containing new variables. The language implementation may be able to reuse the space occupied by previous instances of the same scope, but it guarantees that this optimization will never introduce bugs. ■

We have already noted that many primarily functional languages, including Common Lisp, Scheme, and ML, provide certain nonfunctional features, including iterative constructs that are executed for their side effects. It is also possible to define an iterative construct as syntactic sugar for tail recursion, by arranging for successive iterations of a loop to introduce new scopes. The only tricky part is to make values from a previous iteration available in the next, when all local names have been reused for different variables. The dataflow language Val [McG82] and its successor, Sisal, provide this capability through a special keyword, `old`. The newer pH language, a parallel dialect of Haskell, provides the inverse keyword, `next`. Figure 6.7 contains side-effect-free iterative code for our Fibonacci function in Sisal. We will mention Sisal and pH again in Sections 10.7 and 12.3.6. ■

**EXAMPLE 6.91**

Tail-recursive Fibonacci function in Sisal

### 6.6.2 Applicative- and Normal-Order Evaluation

Throughout the discussion so far we have assumed implicitly that arguments are evaluated before passing them to a subroutine. This need not be the case. It is possible to pass a representation of the *unevaluated* arguments to the subroutine instead, and to evaluate them only when (if) the value is actually needed. The former option (evaluating before the call) is known as *applicative-order evaluation*; the latter (evaluating only when the value is actually needed) is known as *normal-order evaluation*. Normal-order evaluation is what naturally occurs in macros. It also occurs in short-circuit Boolean evaluation, *call-by-name* parameters (to be discussed in Section 8.3.1), and certain functional languages (to be discussed in Section 10.4).

```

function fib(n : integer returns integer)
for initial
 f1 := 0;
 f2 := 1;
 i := 0;
while i < n repeat
 i := old i + 1;
 f1 := old f2;
 f2 := old f1 + old f2;
returns value of f2
end for
end function

```

**Figure 6.7** Fibonacci function in Sisal. Each iteration of the `while` loop defines a new scope, with new variables named `i`, `f1`, and `f2`. The previous instances of these variables are available in each iteration as `old i`, `old f1`, and `old f2`. The entire `for` construct is an *expression*; it can appear in any context in which a value is expected.

#### EXAMPLE 6.92

##### Divisibility macro in C

Historically, C has relied heavily on macros for small, nonrecursive “functions” that need to execute quickly. To determine whether one integer divides another evenly, the C programmer might write

```
#define DIVIDES(a,n) (!((n) % (a)))
/* true iff n has zero remainder modulo a */
```

In every location in which the programmer uses `DIVIDES`, the compiler (actually a preprocessor that runs before the compiler) will substitute the right-hand side of the macro definition, textually, with parameters substituted as appropriate: `DIVIDES(y + z, x)` becomes `(!((x) % (y+z)))`.

#### DESIGN & IMPLEMENTATION

##### Inline as a hint

Formally, the `inline` keyword is a *hint* in C++ and C99, rather than a *directive*: it suggests but does not require that the compiler actually expand the subroutine inline. The compiler is free to use a conventional implementation when `inline` has been specified, or to use an in-line implementation when `inline` has *not* been specified, if it has reason to believe that this will result in better code. In effect, the inclusion of the `inline` keyword in the language is an acknowledgment on the part of the language designers that compiler technology is not (yet) at the point where it can always make a better decision with respect to inlining than can an expert programmer. The choice to make `inline` a hint is an acknowledgment that compilers sometimes *are* able to make a better decision, and that their ability to do so is likely to improve over time.

**EXAMPLE 6.93****“Gotchas” in C macros**

Macros suffer from several limitations. In the code above, for example, the parentheses around *a* and *n* in the right-hand side of the definition are essential. Without them, DIVIDES(*y* + *z*, *x*) would be replaced by !(*x* % *y* + *z*), which is the same as !(*(x* % *y*) + *z*), according to the rules of precedence. More importantly, in a definition like

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

the expression MAX(*x*++, *y*++) may behave unexpectedly, since the increment side effects will happen more than once. In general, normal-order evaluation is safe only if arguments cause no side effects when evaluated. Finally, because macros are purely textual abbreviations, they cannot be incorporated naturally into high-level naming and scope rules. Given the following definition, for example,

```
#define SWAP(a,b) {int t = (a); (a) = (b); (b) = t;}
```

problems will arise if the programmer writes SWAP(*x*, *t*). In C, a macro that “returns” a value must be an expression. Since C is not a completely expression-oriented language like Algol 68, many constructs (e.g., loops) cannot occur within an expression (see Exercise 6.28). ■

All of these problems can be avoided in C by using real functions instead of macros. In most C implementations, however, the macros are much more efficient. They avoid the overhead of the subroutine call mechanism (including register saves and restores), and the code they generate can be integrated into any code improvements that the compiler is able to effect in the code surrounding the call. In C++ and C99, the programmer can obtain the best of both worlds by prefacing a function definition with a special `inline` keyword. This keyword instructs the compiler to expand the definition of the function at the point of call, if possible. The resulting code is then generally as efficient as a macro, but has the semantics of a function call.

Algol 60 uses normal-order evaluation by default (applicative order is also available). This choice was presumably made to mimic the behavior of macros. Most programmers in 1960 wrote mainly in assembler, and were accustomed to macro facilities. Because the parameter-passing mechanisms of Algol 60 are part of the language, rather than textual abbreviations, problems like misinterpreted precedence or naming conflicts do not arise. Side effects, however, are still very much an issue. We will discuss Algol 60 parameters in more detail in Section 8.3.1.

### **Lazy Evaluation**

From the points of view of clarity and efficiency, applicative-order evaluation is generally preferable to normal-order evaluation. It is therefore natural for it to be employed in most languages. In some circumstances, however, normal-order evaluation can actually lead to faster code, or to code that works when applicative-order evaluation would lead to a run-time error. In both cases, what

matters is that normal-order evaluation will sometimes not evaluate an argument at all, if its value is never actually needed. Scheme provides for optional normal-order evaluation in the form of built-in functions called `delay` and `force`.<sup>8</sup> These functions provide an implementation of *lazy evaluation*. In the absence of side effects, lazy evaluation has the same semantics as normal-order evaluation, but the implementation keeps track of which expressions have already been evaluated, so it can reuse their values if they are needed more than once in a given referencing environment. A delayed expression is sometimes called a *promise*. The mechanism used to keep track of which promises have already been evaluated is sometimes called *memoization*.<sup>9</sup> Because applicative-order evaluation is the default in Scheme, the programmer must use special syntax not only to pass an unevaluated argument, but also to use it. In Algol 60, subroutine headers indicate which arguments are to be passed which way; the point of call and the uses of parameters within subroutines look the same in either case.

**EXAMPLE 6.94**

Lazy evaluation of an infinite data structure

A common use of lazy evaluation is to create so-called *infinite* or *lazy data structures* that are “fleshed out” on demand. The following example, adapted from the Scheme manual [ADH<sup>+</sup>98, p. 28], creates a “list” of all the natural numbers.

```
(define naturals
 (letrec ((next (lambda (n) (cons n (delay (next (+ n 1)))))))
 (next 1)))
(define head car)
(define tail (lambda (stream) (force (cdr stream))))
```

**DESIGN & IMPLEMENTATION****Normal-order evaluation**

Normal-order evaluation is one of many examples we have seen where arguably desirable semantics have been dismissed by language designers because of fear of implementation cost. Other examples in this chapter include side-effect freedom (which allows normal order to be implemented via lazy evaluation), iterators (Section 6.5.3), sidebar and nondeterminacy (Section 6.7). As noted in the sidebar on page 248, however, there has been a tendency over time to trade a bit of speed for cleaner semantics and increased reliability. Within the functional programming community, Miranda and its successor Haskell are entirely side-effect free, and use normal-order (lazy) evaluation for all parameters.

- 
- 8** More precisely, `delay` is a *special form*, rather than a function. Its argument is passed to it unevaluated.
  - 9** Within the functional programming community, the term *lazy evaluation* is often used for any implementation that declines to evaluate unneeded function parameters; this includes both naive implementations of normal-order evaluation and the memoizing mechanism described here.

Here `cons` can be thought of, roughly, as a concatenation operator. `Car` returns the head of a list; `cdr` returns everything but the head. Given these definitions, we can access as many natural numbers as we want:

|                               |                 |
|-------------------------------|-----------------|
| (head naturals)               | $\Rightarrow 1$ |
| (head (tail naturals))        | $\Rightarrow 2$ |
| (head (tail (tail naturals))) | $\Rightarrow 3$ |

The list will occupy only as much space as we have actually explored. More elaborate lazy data structures (e.g., trees) can be valuable in combinatorial search problems, in which a clever algorithm may explore only the “interesting” parts of a potentially enormous search space. ■

## 6.7 Nondeterminacy

Our final category of control flow is nondeterminacy. A nondeterministic construct is one in which the choice between alternatives (i.e., between control paths) is deliberately unspecified. We have already seen examples of nondeterminacy in the evaluation of expressions (Section 6.1.4): in most languages, operator or subroutine arguments may be evaluated in any order. Some languages, notably Algol 68 and various concurrent languages, provide more extensive nondeterministic mechanisms, which cover statements as well.

### IN MORE DEPTH

Absent a nondeterministic construct, the author of a code fragment in which order does not matter must choose some arbitrary (artificial) order. Such a choice can make it more difficult to construct a formal correctness proof. Some language designers have also argued that it is inelegant. The most compelling uses for nondeterminacy arise in concurrent programs, where imposing an arbitrary choice on the order in which a thread interacts with its peers may cause the system as a whole to deadlock. For such programs one may need to ensure that the choice among nondeterministic alternatives is *fair* in some formal sense.

### CHECK YOUR UNDERSTANDING

38. What is a *tail-recursive* function? Why is tail recursion important?
39. Explain the difference between *applicative* and *normal-order* evaluation of expressions. Under what circumstances is each desirable?
40. Describe three common pitfalls associated with the use of macros.
41. What is *lazy evaluation*? What are *promises*? What is *memoization*?
42. Give two reasons why lazy evaluation may be desirable.

43. Name a language in which parameters are always evaluated lazily.
  44. Give two reasons why a programmer might sometimes want control flow to be *nondeterministic*.
- 

## 6.8 Summary and Concluding Remarks

In this chapter we introduced the principal forms of control flow found in programming languages: sequencing, selection, iteration, procedural abstraction, recursion, concurrency, and nondeterminacy. Sequencing specifies that certain operations are to occur in order, one after the other. Selection expresses a choice among two or more control-flow alternatives. Iteration and recursion are the two ways to execute operations repeatedly. Recursion defines an operation in terms of simpler instances of itself; it depends on procedural abstraction. Iteration repeats an operation for its side effect(s). Sequencing and iteration are fundamental to imperative (especially von Neumann) programming. Recursion is fundamental to functional programming. Nondeterminacy allows the programmer to leave certain aspects of control flow deliberately unspecified. We touched on concurrency only briefly; it will be the subject of Chapter 12. Procedural abstractions (subroutines) are the subject of Chapter 8.

Our survey of control-flow mechanisms was preceded by a discussion of expression evaluation. We considered the distinction between l-values and r-values, and between the value model of variables, in which a variable is a named container for data, and the reference model of variables, in which a variable is a reference to a data object. We considered issues of precedence, associativity, and ordering within expressions. We examined short-circuit Boolean evaluation and its implementation via jump code, both as a semantic issue that affects the correctness of expressions whose subparts are not always well defined, and as an implementation issue that affects the time required to evaluate complex Boolean expressions.

In our survey we encountered many examples of control-flow constructs whose syntax and semantics have evolved considerably over time. Particularly noteworthy has been the phasing out of goto-based control flow and the emergence of a consensus on structured alternatives. While convenience and readability are difficult to quantify, most programmers would agree that the control-flow constructs of a language like Ada are a dramatic improvement over those of, say, Fortran IV. Examples of features in Ada that are specifically designed to rectify control-flow problems in earlier languages include explicit terminators (`end if`, `end loop`, etc.) for structured constructs; `elsif` clauses; label ranges and `others` clauses in `case` statements; implicit declaration of `for` loop indices as read-only local variables; explicit `return` statements; multi-level loop `exit` statements; and exceptions.

The evolution of constructs has been driven by many goals, including ease of programming, semantic elegance, ease of implementation, and run-time efficiency. In some cases these goals have proven complementary. We have seen for example that short-circuit evaluation leads both to faster code and (in many cases) to cleaner semantics. In a similar vein, the introduction of a new local scope for the index variable of an enumeration-controlled loop avoids both the semantic problem of the value of the index after the loop and (to some extent) the implementation problem of potential overflow.

In other cases improvements in language semantics have been considered worth a small cost in run-time efficiency. We saw this in the addition of a pretest to the Fortran do loop and in the introduction of midtest loops (which almost always require at least two branch instructions). Iterators provide another example: like many forms of abstraction, they add a modest amount of run-time cost in many cases (e.g., in comparison to explicitly embedding the implementation of the enumerated set in the control flow of the loop), but with a large pay-back in modularity, clarity, and opportunities for code reuse. Sisal's developers would argue that even if Fortran does enjoy a performance edge in some cases, functional programming provides a more important benefit: facilitating the construction of correct, maintainable code. The developers of Java would argue that for many applications the portability and safety provided by extensive semantic checking, standard-format numeric types, and so on are far more important than speed.

The ability of Sisal to compete with Fortran (it does very well with numeric code) is due to advances in compiler technology, and to advances in automatic code improvement in particular. We have seen several other examples of cases in which advances in compiler technology or in the simple willingness of designers to build more complex compilers have made it possible to incorporate features once considered too expensive. Label ranges in Ada case statements require that the compiler be prepared to generate code employing binary search. In-line functions in C++ eliminate the need to choose between the inefficiency of tiny functions and the messy semantics of macros. Exceptions (as we shall see in Section 8.5.4) can be implemented in such a way that they incur no cost in the common case (when they do not occur), but the implementation is quite tricky. Iterators, boxing, generics (Section 8.4), and first-class functions are likewise rather tricky, but are increasingly found in mainstream imperative languages.

Some implementation techniques (e.g., rearranging expressions to uncover common subexpressions, or avoiding the evaluation of guards in a nondeterministic construct once an acceptable choice has been found) are sufficiently important to justify a modest burden on the programmer (e.g., adding parentheses where necessary to avoid overflow or ensure numeric stability, or ensuring that expressions in guards are side-effect-free). Other semantically useful mechanisms (e.g., lazy evaluation, continuations, or truly random nondeterminacy) are usually considered complex or expensive enough to be worthwhile only in special circumstances (if at all).

In comparatively primitive languages, we can often obtain some of the benefits of missing features through programming conventions. In early dialects of

Fortran, for example, we can limit the use of *gotos* to patterns that mimic the control flow of more modern languages. In languages without short-circuit evaluation, we can write nested selection statements. In languages without iterators, we can write sets of subroutines that provide equivalent functionality.

## 6.9 Exercises

- 6.1 We noted in Section 6.1.1 that most binary arithmetic operators are left-associative in most programming languages. In Section 6.1.4, however, we also noted that most compilers are free to evaluate the operands of a binary operator in either order. Are these statements contradictory? Why or why not?
- 6.2 As noted in Figure 6.1, Fortran and Pascal give unary and binary minus the same level of precedence. Is this likely to lead to nonintuitive evaluations of certain expressions? Why or why not?
- 6.3 Translate the following expression into postfix and prefix notation:

$$[-b + \sqrt{b \times b - 4 \times a \times c}] / (2 \times a)$$

Do you need a special symbol for unary negation?

- 6.4 In Lisp, most of the arithmetic operators are defined to take two or more arguments, rather than strictly two. Thus `(* 2 3 4 5)` evaluates to 120, and `(- 16 9 4)` evaluates to 3. Show that parentheses are necessary to disambiguate arithmetic expressions in Lisp (in other words, give an example of an expression whose meaning is unclear when parentheses are removed).

In Section 6.1.1 we claimed that issues of precedence and associativity do not arise with prefix or postfix notation. Reword this claim to make explicit the hidden assumption.

- 6.5 Example 6.31 claims that “For certain values of  $x$ ,  $(0.1 + x) * 10.0$  and  $1.0 + (x * 10.0)$  can differ by as much as 25%, even when  $0.1$  and  $x$  are of the same magnitude.” Verify this claim. (Warning: If you’re using an x86 processor, be aware that floating-point calculations [even on single precision variables] are performed internally with 80 bits of precision. Roundoff errors will appear only when intermediate results are stored out to memory [with limited precision] and read back in again.)
- 6.6 Languages that employ a reference model of variables also tend to employ automatic garbage collection. Is this more than a coincidence? Explain.
- 6.7 In Section 6.1.2 we noted that C uses `=` for assignment and `==` for equality testing. The language designers state “Since assignment is about twice as frequent as equality testing in typical C programs, it’s appropriate that the operator be half as long” [KR88, p. 17]. What do you think of this rationale?

- 6.8** Consider a language implementation in which we wish to catch every use of an uninitialized variable. In Section 6.1.3 we noted that for types in which every possible bit pattern represents a valid value, extra space must be used to hold an initialized/uninitialized flag. Dynamic checks in such a system can be expensive, largely because of the address calculations needed to access the flags. We can reduce the cost in the common case by having the compiler generate code to automatically initialize every variable with a distinguished *sentinel* value. If at some point we find that a variable's value is different from the sentinel, then that variable must have been initialized. If its value *is* the sentinel, we must double-check the flag. Describe a plausible allocation strategy for initialization flags, and show the assembly language sequences that would be required for dynamic checks, with and without the use of sentinels.
- 6.9** Write an attribute grammar, based on the following context-free grammar, that accumulates jump code for Boolean expressions (with short-circuiting) into a synthesized attribute of *condition*, and then uses this attribute to generate code for *if* statements.

```

stmt → if condition then stmt else stmt
 → other_stmt
condition → c_term | condition or c_term
c_term → relation | c_term and relation
relation → c_fact | c_fact comparator c_fact
c_fact → identifier | not c_fact | (condition)
comparator → < | <= | = | > | >=

```

(*Hint:* Your task will be easier if you do *not* attempt to make the grammar L-attributed. For further details see Fischer and LeBlanc's compiler book [FL88, Sec. 14.1.4].)

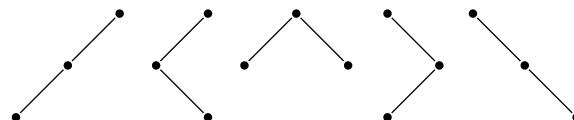
- 6.10** Neither Algol 60 nor Algol 68 employs short-circuit evaluation for Boolean expressions. In both languages, however, an *if...then...else* construct can be used as an expression. Show how to use *if...then...else* to achieve the effect of short-circuit evaluation.
- 6.11** Consider the following expression in C:  $a/b > 0 \&& b/a > 0$ . What will be the result of evaluating this expression when  $a$  is zero? What will be the result when  $b$  is zero? Would it make sense to try to design a language in which this expression is guaranteed to evaluate to *false* when either  $a$  or  $b$  (but not both) is zero? Explain your answer.
- 6.12** As noted in Section 6.4.2, languages vary in how they handle the situation in which the tested expression in a *case* statement does not appear among the labels on the arms. C and Fortran 90 say the statement has no effect. Pascal and Modula say it results in a dynamic semantic error. Ada says that the labels must *cover* all possible values for the type of the expression, so

the question of a missing value can never arise at run time. What are the tradeoffs among these alternatives? Which do you prefer? Why?

- 6.13 Write the equivalent of Figure 6.5 in C# 2.0, Python, or Ruby. Write a second version that performs an in-order enumeration rather than pre-order.
- 6.14 Revise the algorithm of Figure 6.6 so that it performs an in-order enumeration, rather than pre-order.
- 6.15 Write a C++ pre-order iterator to supply tree nodes to the loop in Example 6.71. You will need to know (or learn) how to use pointers, references, inner classes, and operator overloading in C++. For the sake of (relative) simplicity, you may assume that the datum in a tree node is always an `int`; this will save you the need to use generics. You may want to use the `stack` abstraction from the C++ standard library.
- 6.16 Write code for the `tree_iter` type (`struct`) and the `ti_create`, `ti_done`, `ti_next`, `ti_val`, and `ti_delete` functions employed in Example 6.74.
- 6.17 Write, in C#, Python, or Ruby, an iterator that yields
  - (a) all permutations of the integers  $1 \dots n$
  - (b) all combinations of  $k$  integers from the range  $1 \dots n$  ( $0 \leq k \leq n$ ).

You may represent your permutations and combinations using either a list or an array.

- 6.18 Use iterators to construct a program that outputs (in some order) all *structurally distinct* binary trees of  $n$  nodes. Two trees are considered structurally distinct if they have different numbers of nodes or if their left or right subtrees are structurally distinct. There are, for example, 5 structurally distinct trees of 3 nodes:



These are most easily output in “dotted parenthesized form”:

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

(*Hint:* Think recursively! If you need help, see Section 2.2 of the text by Finkel [Fin96].)

- 6.19 Build true iterators in Java using threads. (This requires knowledge of material in Chapter 12.) Make your solution as clean and as general as possible. In particular, you should provide the standard `Iterator` or `IEnumerable`

interface for use with extended `for` or `foreach` loops, but the programmer should not have to write these. Instead, he or she should write a class with an `Iterate` method, which should in turn be able to call a `Yield` method, which you should also provide. Evaluate the cost of your solution. How much more expensive is it than standard Java iterator objects?

- 6.20** In an expression-oriented language such as Algol 68 or Lisp, a `while` loop (a `do` loop in Lisp) has a value as an expression. How do you think this value should be determined? (How is it determined in Algol 68 and Lisp?) Is the value a useless artifact of expression orientation, or are there reasonable programs in which it might actually be used? What do you think should happen if the condition on the loop is such that the body is never executed?
- 6.21** Recall the “blank line” loop of Example 6.80, here written in Modula-2.

```
LOOP
 line := ReadLine;
 IF AllBlanks(line) THEN EXIT END;
 ConsumeLine(line)
END;
```

Show how you might accomplish the same task using a `while` or `repeat` loop, if midtest loops were not available. (*Hint:* One alternative duplicates part of the code; another introduces a Boolean flag variable.) How do these alternatives compare to the midtest version?

- 6.22** Rubin [Rub87] used the following example (rewritten here in C) to argue in favor of a `goto` statement.

```
int first_zero_row = -1; /* none */
int i, j;
for (i = 0; i < n; i++) {
 for (j = 0; j < n; j++) {
 if (A[i][j]) goto next;
 }
 first_zero_row = i;
 break;
next: ;
}
```

The intent of the code is to find the first all-zero row, if any, of an  $n \times n$  matrix. Do you find the example convincing? Is there a good structured alternative in C? In any language?

- 6.23** Bentley [Ben86, Chap. 4] provides the following informal description of binary search.

We are to determine whether the sorted array  $X[1..N]$  contains the element  $T$ . . . . Binary search solves the problem by keeping track of a range within the array in which  $T$  must be if it is anywhere in the array. Initially, the range is the entire array. The range is shrunk by comparing its middle

element to T and discarding half the range. The process continues until T is discovered in the array or until the range in which it must lie is known to be empty.

Write code for binary search in your favorite imperative programming language. What loop construct(s) did you find to be most useful? (NB: When he asked more than a hundred professional programmers to solve this problem, Bentley found that only about 10% got it right the first time, without testing.)

- 6.24 A *loop invariant* is a condition that is guaranteed to be true at a given point within the body of a loop on every iteration. Loop invariants play a major role in *axiomatic semantics*, a formal reasoning system used to prove properties of programs. In a less formal way, programmers who identify (and write down!) the invariants for their loops are more likely to write correct code. Show the loop invariant(s) for your solution to the preceding exercise. (*Hint:* You will find the distinction between  $<$  and  $\leq$  [or between  $>$  and  $\geq$ ] to be crucial.)
- 6.25 If you have taken a course in automata theory or recursive function theory, explain why `while` loops are strictly more powerful than `for` loops. (If you haven't had such a course, skip this question!) Note that we're referring here to Pascal-style `for` loops, not C-style.
- 6.26 Show how to calculate the number of iterations of a general Fortran 90-style `do` loop. Your code should be written in an assembler-like notation, and should be guaranteed to work for all valid bounds and step sizes. Be careful of overflow! (*Hint:* While the bounds and step size of the loop can be either positive or negative, you can safely use an unsigned integer for the iteration count.)
- 6.27 Write a tail-recursive function in Scheme or ML to compute  $n$  factorial ( $n! = \prod_{1 \leq i \leq n} i = 1 \times 2 \times \dots \times n$ ). (*Hint:* You will probably want to define a "helper" function, as discussed in Section 6.6.1.)
- 6.28 Can you write a macro in standard C that "returns" the greatest common divisor of a pair of arguments, without calling a subroutine? Why or why not?
- 6.29 Give an example in C in which an in-line subroutine may be significantly faster than a functionally equivalent macro. Give another example in which the macro is likely to be faster. (*Hint:* Think about applicative versus normal-order evaluation of arguments.)
- 6.30 Use lazy evaluation (`delay` and `force`) to implement iterator objects in Scheme. More specifically, let an iterator be either the null list or a pair consisting of an element and a promise that when forced will return an iterator. Give code for an `uptoby` function that returns an iterator, and a `for-iter` function that accepts as arguments a one-argument function and an iterator. These should allow you to evaluate such expressions as

```
(for-iter (lambda (e) (display e) (newline)) (uptoby 10 50 3))
```

Note that unlike the standard Scheme `for-each`, `for-iter` should not require the existence of a list containing the elements over which to iterate; the intrinsic space required for `(for-iter f (uptoby 1 n 1))` should be only  $O(1)$ , rather than  $O(n)$ .

- 6.31** (Difficult) Use `call-with-current-continuation` (`call/cc`) to implement the following structured nonlocal control transfers in Scheme. (This requires knowledge of material in Chapter 10.) You will probably want to consult a Scheme manual for documentation not only on `call/cc`, but on `define-syntax` and `dynamic-wind` as well.
- (a) Multilevel returns. Model your syntax after the `catch` and `throw` of Common Lisp.
  - (b) True iterators. In a style reminiscent of Exercise 6.30, let an iterator be a function which when `call/cc`-ed will return either a null list or a pair consisting of an element and an iterator. As in that previous exercise, your implementation should support expressions like

```
(for-iter (lambda (e) (display e) (newline)) (uptoby 10 50 3))
```

Where the implementation of `uptoby` in Exercise 6.30 required the use of `delay` and `force`, however, you should provide an `iterator` macro (a Scheme *special form*) and a `yield` function that allows `uptoby` to look like an ordinary tail-recursive function with an embedded `yield`:

```
(define upto
 (iterator (low high step)
 (letrec ((helper
 (lambda (next)
 (if (> next high) '()
 (begin ; else clause
 (yield next)
 (helper (+ next step)))))))
 (helper low))))
```

- 6.32** Explain why the following guarded commands in SR are *not* equivalent.

|                                                                                                                                        |                                                                                                                                        |
|----------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <code>if a &lt; b -&gt; c := a</code><br><code>[] b &lt; c -&gt; c := b</code><br><code>[] else -&gt; c := d</code><br><code>fi</code> | <code>if a &lt; b -&gt; c := a</code><br><code>[] b &lt; c -&gt; c := b</code><br><code>[] true -&gt; c := d</code><br><code>fi</code> |
|----------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|

© 6.33–6.35 In More Depth.

## 6.10 Explorations

- 6.36** Consider again the idea of *loop unrolling*, introduced in Exercise 5.15. Loop unrolling is traditionally implemented by the code improvement phase of a compiler. It can be implemented at source level, however, if we are faced with the prospect of “hand optimizing” time-critical code on a system whose compiler is not up to the task. Unfortunately, if we replicate the body of a loop  $k$  times, we must deal with the possibility that the original number of loop iterations,  $n$ , may not be a multiple of  $k$ . Writing in C, and letting  $k = 4$ , we might transform the main loop of Exercise 5.15 from

```
i = 0;
do {
 sum += A[i]; squares += A[i] * A[i]; i++;
} while (i < N);

to

i = 0; j = N/4;
do {
 sum += A[i]; squares += A[i] * A[i]; i++;
 sum += A[i]; squares += A[i] * A[i]; i++;
 sum += A[i]; squares += A[i] * A[i]; i++;
 sum += A[i]; squares += A[i] * A[i]; i++;
} while (--j > 0);
do {
 sum += A[i]; squares += A[i] * A[i]; i++;
} while (i < N);
```

In 1983, Tom Duff of Lucasfilm realized that code of this sort can be “simplified” in C by interleaving a `switch` statement and a loop. The result is rather startling, but perfectly valid C. It’s known in programming folklore as “Duff’s device.”

```
i = 0; j = (N+3)/4;
switch (N%4)
 case 0: do{ sum += A[i]; squares += A[i] * A[i]; i++;
 case 3: sum += A[i]; squares += A[i] * A[i]; i++;
 case 2: sum += A[i]; squares += A[i] * A[i]; i++;
 case 1: sum += A[i]; squares += A[i] * A[i]; i++;
 } while (--j > 0);
}
```

Duff announced his discovery with “a combination of pride and revulsion.” He noted that “Many people ... have said that the worst feature of C is that switches don’t break automatically before each `case` label. This code forms some sort of argument in that debate, but I’m not sure whether it’s

for or against.” What do you think? Is it reasonable to interleave a loop and a `switch` in this way? Should a programming language permit it? Is automatic fall-through ever a good idea?

- 6.37 Using your favorite language and compiler, investigate the order of evaluation of subroutine parameters. Are they usually evaluated left to right or right to left? Are they ever evaluated in the other order? (Can you be sure?) Write a program in which the order makes a difference in the results of the computation.
- 6.38 Consider the different approaches to arithmetic overflow adopted by Pascal, C, Java, C#, and Common Lisp, as described in Section 6.1.4. Speculate as to the differences in language design goals that might have caused the designers to adopt the approaches they did.
- 6.39 Learn more about container classes and the *design patterns* (structured programming idioms) they support. Explore the similarities and differences among the standard container libraries of C++, Java, and C#. Which of these libraries do you find the most appealing? Why?

© 6.40–6.43 In More Depth.

## 6.11 Bibliographic Notes

Many of the issues discussed in this chapter feature prominently in papers on the history of programming languages. Pointers to several such papers can be found in the Bibliographic Notes for Chapter 1. Fifteen papers comparing Ada, C, and Pascal can be found in the collection edited by Feuer and Gehani [FG84]. References for individual languages can be found in Appendix A.

Niklaus Wirth has been responsible for a series of influential languages over a 30-year period, including Pascal [Wir71], its predecessor Algol W [WH66], and the successors Modula [Wir77b], Modula-2 [Wir85b], and Oberon [Wir88b]. The `case` statement of Algol W is due to Hoare [Hoa81]. Bernstein [Ber85] considers a variety of alternative implementations for `case`, including multilevel versions appropriate for label sets consisting of several dense “clusters” of values. Guarded commands are due to Dijkstra [Dij75]. Duff’s device was originally posted to netnews, the predecessor of Usenet news, in May 1984. The original posting appears to have been lost, but Duff’s commentary on it can be found at many Internet sites, including [www.lysator.liu.se/c/duffs-device.html](http://www.lysator.liu.se/c/duffs-device.html).

Debate over the supposed merits or evils of the `goto` statement dates from at least the early 1960s, but became a good bit more heated in the wake of a 1968 article by Dijkstra (“Go To Statement Considered Harmful” [Dij68b]). The structured programming movement of the 1970s took its name from the text of Dahl, Dijkstra, and Hoare [DDH72]. A dissenting letter by Rubin in 1987 (“‘GOTO Considered Harmful’ Considered Harmful” [Rub87]; Exercise 6.22) elicited a flurry of responses.

What has been called the “reference model of variables” in this chapter is called the “object model” in Clu; Liskov and Guttag describe it in Sections 2.3 and 2.4.2 of their text on abstraction and specification [LG86]. Clu iterators are described in an article by Liskov et al. [LSAS77] and in Chapter 6 of the Liskov and Guttag text. Icon generators are discussed in Chapters 11 and 14 of the text by Griswold and Griswold [GG96]. The tree-enumeration algorithm of Exercise 6.18 was originally presented (without iterators) by Solomon and Finkel [SF80].

Several texts discuss the use of invariants (Exercise 6.24) as a tool for writing correct programs. Particularly noteworthy are the works of Dijkstra [Dij76] and Gries [Gri81]. Kernighan and Plauger provide a more informal discussion of the art of writing good programs [KP78].

The Blizzard [SFL<sup>+</sup>94] and Shasta [SG96] systems for software distributed shared memory (S-DSM) make use of sentinels (Exercise 6.8). We will discuss S-DSM in Section 12.2.1.

Michaelson [Mic89, Chap. 8] provides an accessible formal treatment of applicative-order, normal-order, and lazy evaluation. Friedman, Wand, and Haynes provide an excellent discussion of continuation-passing style [FWH01, Chaps. 7–8].

# 7 Data Types

**Most programming languages include a notion of type** for expressions and/or objects.<sup>1</sup> Types serve two principal purposes:

## EXAMPLE 7.1

Operations that leverage type information

## EXAMPLE 7.2

Errors captured by type information

1. Types provide implicit context for many operations, so the programmer does not have to specify that context explicitly. In Pascal, for instance, the expression `a + b` will use integer addition if `a` and `b` are of `integer` type; it will use floating-point addition if `a` and `b` are of `real` type. Similarly, the operation `new p`, where `p` is a pointer, will allocate a block of storage from the heap that is the right size to hold an object of the type pointed to by `p`; the programmer does not have to specify (or even know) this size. In C++, Java, and C#, the operation `new my_type()` not only allocates (and returns a pointer to) a block of storage sized for an object of type `my_type`, it also automatically calls any user-defined initialization (*constructor*) function that has been associated with that type.
2. Types limit the set of operations that may be performed in a semantically valid program. They prevent the programmer from adding a character and a record, for example, or from taking the arctangent of a set, or passing a file as a parameter to a subroutine that expects an integer. While no type system can promise to catch every nonsensical operation that a programmer might put into a program by mistake, good type systems catch enough mistakes to be highly valuable in practice.

Section 7.1 looks more closely at the meaning and purpose of types, and presents some basic definitions. Section 7.2 addresses questions of *type equivalence* and *type compatibility*: when can we say that two types are the same, and when can we use a value of a given type in a given context? Sections 7.3–7.9 con-

---

**I** Recall that unless otherwise noted we are using the term *object* informally to refer to anything that might have a name. Object-oriented languages, which we will study in Chapter 9, assign a different, more formal meaning to the term.

sider syntactic, semantic, and pragmatic issues for some of the most important *composite* types: records, arrays, strings, sets, pointers, lists, and files. The section on pointers includes a more detailed discussion of the value and reference models of variables introduced in Section 6.1.2, and of the heap management issues introduced in Section 3.2. The section on files (mostly on the PLP CD) includes a discussion of input and output. Section 7.10 considers what it means to compare two complex objects for equality, or to assign one into the other.

## 7.1 Type Systems

In Section 5.2 we noted that computer hardware is capable of interpreting bits in memory in several different ways. The various functional units of a processor may interpret bits as, among other things, instructions, addresses, characters, and integer and floating-point numbers of various lengths. The bits themselves, however, are untyped; the hardware on most machines makes no attempt to keep track of which interpretations correspond to which locations in memory. Assembly languages reflect this lack of typing: operations of any kind can be applied to values in arbitrary locations. High-level languages, on the other hand, almost always associate types with values, to provide the contextual information and error-checking just mentioned.

Informally, a *type system* consists of (1) a mechanism to define types and associate them with certain language constructs and (2) a set of rules for *type equivalence*, *type compatibility*, and *type inference*. The constructs that must have types are precisely those that have values, or that can refer to objects that have values. These constructs include named constants, variables, record fields, parameters, and sometimes subroutines; explicit (manifest) constants (e.g., 17, 3.14, "foo"); and more complicated expressions containing these. Type equivalence rules determine when the types of two values are the same. Type compatibility rules determine when a value of a given type can be used in a given context. Type inference rules define the type of an expression based on the types of its constituent parts or (sometimes) the surrounding context.

The distinction between the type of an expression (e.g., a name) and the type of the object to which it refers is important in a language with polymorphic variables or parameters, since a given name may refer to objects of different types at different times. In a language without polymorphism, the distinction doesn't matter.

Subroutines are considered to have types in some languages, but not in others. Subroutines need to have types if they are first- or second-class values (i.e., if they can be passed as parameters, returned by functions, or stored in variables). In each of these cases there is a construct in the language whose value is a dynamically determined subroutine; type information allows the language to limit the set of acceptable values to those that provide a particular subroutine interface (i.e., particular numbers and types of parameters). In a statically scoped language that

never creates references to subroutines dynamically (one in which subroutines are always third-class values), the compiler can always identify the subroutine to which a name refers, and can ensure that the routine is called correctly without necessarily employing a formal notion of subroutine types.

### 7.1.1 Type Checking

*Type checking* is the process of ensuring that a program obeys the language's type compatibility rules. A violation of the rules is known as a *type clash*. A language is said to be *strongly typed* if it prohibits, in a way that the language implementation can enforce, the application of any operation to any object that is not intended to support that operation. A language is said to be *statically typed* if it is strongly typed and type checking can be performed at compile time. In the strictest sense of the term, few languages are statically typed. In practice, the term is often applied to languages in which most type checking can be performed at compile time, and the rest can be performed at run time.

A few examples: Ada is strongly typed and, for the most part, statically typed (certain type constraints must be checked at run time). A Pascal implementation can also do most of its type checking at compile time, though the language is not quite strongly typed: untagged variant records (to be discussed in Section 7.3) are its only loophole. C89 is significantly more strongly typed than its predecessor dialects, but still significantly less strongly typed than Pascal. Its loopholes include unions, subroutines with variable numbers of parameters, and the interoperability of pointers and arrays (to be discussed in Section 7.7.1). Implementations of C rarely check anything at run time. A few high-level languages (e.g., Bliss [WRH71]) are completely untyped, like assembly languages.

Dynamic (run-time) type checking is a form of late binding and tends to be found in languages that delay other issues until run time as well. Lisp, Smalltalk, and most scripting languages are dynamically (though strongly) typed. Languages with dynamic scoping are generally dynamically typed (or not typed at all): if the compiler can't identify the object to which a name refers, it usually can't determine the type of the object either.

### 7.1.2 Polymorphism

*Polymorphism* (Section 3.6.3) allows a single body of code to work with objects of multiple types. It may or may not imply the need for run-time type checking. As implemented in Lisp, Smalltalk, and the various scripting languages, fully dynamic typing allows the programmer to apply arbitrary operations to arbitrary objects. Only at run time does the language implementation check to see that the objects actually implement the requested operations. Because the types of objects can be thought of as implied (unspecified) parameters, dynamic typing is said to support *implicit parametric polymorphism*.

Unfortunately, while powerful and straightforward, dynamic typing incurs significant run-time cost. It also delays the reporting of errors. ML and its descendants employ a sophisticated system of *type inference* to support implicit parametric polymorphism in conjunction with static typing. The ML compiler infers for every object and expression a (possibly unique) type that captures precisely those properties that the object or expression must have to be used in the context(s) in which it appears. With rare exceptions, the programmer need not specify the types of objects explicitly. The task of the compiler is to determine whether there exists a consistent assignment of types to expressions that guarantees, statically, that no operation will ever be applied to a value of an inappropriate type at run time. This job can be formalized as the problem of *unification*; we discuss it further in Section 7.2.4.

In object-oriented languages, *subtype polymorphism* allows a variable  $X$  of type  $T$  to refer to an object of any type derived from  $T$ . Since derived types are required to support all of the operations of the base type, the compiler can be sure that any operation acceptable for an object of type  $T$  will be acceptable for any object referred to by  $X$ . Given a straightforward model of inheritance, type checking for subtype polymorphism can be implemented entirely at compile time. Most languages that envision such an implementation, including C++, Eiffel, Java, and C#, also provide *explicit parametric polymorphism (generics)*, which allow the programmer to define classes with type parameters. Generics are particularly useful for *container (collection)* classes: “list of  $T$ ” ( $\text{List} < T >$ ), “stack of  $T$ ” ( $\text{Stack} < T >$ ), and so on, where  $T$  is left unspecified. Like subtype polymorphism, generics can usually be type-checked at compile time, though Java sometimes performs redundant checks at run time for the sake of interoperability with preexisting non-generic code. Smalltalk, Objective-C, Python, and Ruby use a single mechanism

## DESIGN & IMPLEMENTATION

### Dynamic typing

The growing popularity of scripting languages has led a number of prominent software developers to publicly question the value of static typing. They ask: given that we can't check everything at compile time, how much pain is it worth to check the things we can? As a general rule, it is easier to write type-correct code than to prove that we have done so, and static typing requires such proofs. As type systems become more complex (due to object orientation, generics, etc.), the complexity of static typing increases correspondingly. Anyone who has written extensively in Ada or C++ on the one hand, and in Python or Scheme on the other, cannot help but be struck at how much easier it is to write code without complex type declarations. Dynamic checking incurs some run-time cost, of course, and delays the reporting of errors, but this is increasingly seen as insignificant in comparison to the potential increase in human productivity. The choice between static and dynamic typing promises to provide one of the most interesting language debates of the coming decade.

for both parametric and subtype polymorphism, with checking delayed until run time. We will consider generics further in Section 8.4, and derived types in Chapter 9.

### 7.1.3 The Definition of Types

Some early high-level languages (e.g., Fortran 77, Algol 60, and Basic) provide a small, built-in, and nonextensible set of types. As we saw in Section 3.3.1, Fortran does not require variables to be declared; it incorporates default rules to determine the type of undefined variables based on the spelling of their names (Basic has similar rules). As noted in the previous subsection, a few languages (e.g., Bliss) dispense with types, while others keep track of them automatically at compile time (as in ML, Miranda, or Haskell) or at run time (as in Lisp/Scheme or Smalltalk). In most languages, however, users must explicitly declare the type of every object, together with the characteristics of every type that is not built-in.

There are at least three ways to think about types, which we may call the *denotational*, *constructive*, and *abstraction-based* points of view. From the denotational point of view, a type is simply a set of values. A value has a given type if it belongs to the set; an object has a given type if its value is guaranteed to be in the set. From the constructive point of view, a type is either one of a small collection of *built-in* types (integer, character, Boolean, real, etc.; also called *primitive* or *predefined* types) or a *composite* type created by applying a type *constructor* (record, array, set, etc.) to one or more simpler types. (This use of the term *constructor* is unrelated to the initialization functions of object-oriented languages. It also differs in a more subtle way from the use of the term in ML.) From the abstraction-based point of view, a type is an *interface* consisting of a set of operations with well-defined and mutually consistent semantics. For most programmers (and language designers), types usually reflect a mixture of these viewpoints.

In denotational semantics (one of the leading ways to formalize the meaning of programs), a set of values is known as a *domain*. Types are domains. The meaning of an expression in denotational semantics is a value from the domain that represents the expression's type. (Domains are in some sense a generalization of types. The meaning of any language construct is a value from a domain. The meaning of an assignment statement, for example, is a value from a domain whose elements are functions. Each function maps a *store*—a mapping from names to values that represents the current contents of memory—to another store, which represents the contents of memory after the assignment.) One of the nice things about the denotational view of types is that it allows us in many cases to describe user-defined composite types (records, arrays, etc.) in terms of mathematical operations on sets. We will allude to these operations again in Section 7.1.4.

Because it is based on mathematical objects, the denotational view of types usually ignores such implementation issues as limited precision and word length.

This limitation is less serious than it might at first appear: checks for such errors as arithmetic overflow are usually implemented outside of the type system of a language anyway: they result in a run-time error, but this error is not called a type clash.

When a programmer defines an enumerated type (e.g., `enum hue {red, green, blue}` in C), he or she certainly thinks of this type as a set of values. For most other varieties of user-defined type, however, one typically does not think in terms of sets of values. Rather, one usually thinks in terms of the way the type is built from simpler types, or in terms of its meaning or purpose. These ways of thinking reflect the constructive and abstraction-based points of view. The constructive point of view was pioneered by Algol W and Algol 68, and is characteristic of most languages designed in the 1970s and 1980s. The abstraction-based point of view was pioneered by Simula-67 and Smalltalk, and is characteristic of modern object-oriented languages. It can also be adopted as a matter of programming discipline in non-object-oriented languages. We will discuss the abstraction-based point of view in more detail in Chapter 9. The remainder of this chapter focuses on the constructive point of view.

#### 7.1.4 The Classification of Types

The terminology for types varies some from one language to another. This subsection presents definitions for the most common terms. Most languages provide built-in types similar to those supported in hardware by most processors: integers, characters, Booleans, and real (floating-point) numbers.

Booleans (sometimes called *logicals*) are typically implemented as one-byte quantities, with 1 representing `true` and 0 representing `false`. As noted in Section 6.1.2, C is unusual in its lack of a Boolean type: where most languages would expect a Boolean value, C expects an integer; zero means `false`, and anything else means `true`. As noted in Section 6.5.4, Icon replaces Booleans with a more general notion of *success* and *failure*.

Characters have traditionally been implemented as one-byte quantities as well, typically (but not always) using the ASCII encoding. More recent languages (e.g., Java and C#) use a two-byte representation designed to accommodate the *Unicode* character set. Unicode is an international standard designed to capture the characters of a wide variety of languages (see sidebar on page 313). The first 128 characters of Unicode (`\u0000` through `\u007f`) are identical to ASCII. C++ provides both regular and “wide” characters, though for wide characters both the encoding and the actual width are implementation-dependent. Fortran 2003 supports four-byte Unicode characters.

##### Numeric Types

A few languages (e.g., C and Fortran) distinguish between different lengths of integers and real numbers; most do not, and leave the choice of precision to the

implementation. Unfortunately, differences in precision across language implementations lead to a lack of portability: programs that run correctly on one system may produce run-time errors or erroneous results on another. Java and C# are unusual in providing several lengths of numeric types, with a specified precision for each.

A few languages, including C, C++, C#, and Modula-2, provide both signed and unsigned integers (Modula-2 calls unsigned integers *cardinals*). A few languages (e.g., Fortran, C99, Common Lisp, and Scheme) provide a built-in complex type, usually implemented as a pair of floating-point numbers that represent the real and imaginary Cartesian coordinates. Other languages (e.g., C++)

## DESIGN & IMPLEMENTATION

### Multilingual character sets

The ISO 10646 international standard defines a *Universal Character Set (UCS)* intended to include all characters of all known human languages. (It also sets aside a “private use area” for such artificial [constructed] languages as Klingon, Tengwar, and Cirth [Tolkein Elvish]. Allocation of this private space is coordinated by a volunteer organization known as the ConScript Unicode Registry.) All natural languages currently employ codes in the 16-bit *Basic Multilingual Plane (BMP)*: 0x0000 through 0xffffd.

*Unicode* is an expanded version of ISO 10646, maintained by an international consortium of software manufacturers. In addition to mapping tables, it covers such topics as rendering algorithms, directionality of text, and sorting and comparison conventions.

While recent languages have moved toward 16- or 32-bit internal character representations, these cannot be used for external storage—text files—without causing severe problems with backward compatibility. To accommodate Unicode without breaking existing tools, Ken Thompson in 1992 proposed a multibyte “expanding” code known as *UTF-8* (UCS/Unicode Transformation Format, 8-bit) and codified as a formal annex (appendix) to ISO 10646. UTF-8 characters occupy a maximum of 6 bytes—3 if they lie in the BMP, and only 1 if they are ordinary ASCII. The trick is to observe that ASCII is a 7-bit code; in any legacy text file the most significant bit of every byte is 0. In UTF-8 a most significant bit of 1 indicates a multibyte character. Two-byte codes begin with the bits 110. Three-byte codes begin with 1110. Second and subsequent bytes of multibyte characters always begin with 10.

On some systems one also finds files encoded in one of ten variants of the older 8-bit ISO 8859 standard, but these are inconsistently rendered across platforms. On the web, non-ASCII characters are typically encoded with *numeric character references*, which bracket a Unicode value, written in decimal or hex, with an ampersand and a semicolon. The copyright symbol (©), for example, is &169#;. Many characters also have symbolic *entity names*, (e.g., &copy;) but not all browsers support these.

support complex numbers in a standard library. A few languages (e.g., Scheme and Common Lisp) provide a built-in rational type, usually implemented as a pair of integers that represents the numerator and denominator. Common Lisp and most dialects of Scheme support integers (and rationals) of arbitrary precision; the implementation uses multiple words of memory where appropriate.

Ada supports *fixed-point* types, which are represented internally by integers but have an implied decimal point at a programmer-specified position among the digits. Fixed-point numbers provide a compact representation of nonintegral values (e.g., dollars and cents) within a restricted range. For example, 32-bit hardware integers can represent fixed-point numbers with two (decimal) digits to the right of the decimal point in the range of roughly negative 20 million to positive 20 million. Double-precision (64-bit) numbers would be required to capture the same range in floating-point, since single-precision IEEE floating-point numbers have only 23 bits of significand (Section 5.2.1). Addition and subtraction of fixed-point numbers (with the same number of decimal places) can use ordinary integer operations. Multiplication and division are slightly more complicated, as are operations on values with different numbers of digits to the right of the decimal point (Exercise 7.4).

Integers, Booleans, and characters are all examples of *discrete* types (also called *ordinal* types): the domains to which they correspond are countable, and have a well-defined notion of predecessor and successor for each element other than the first and the last. (In most implementations the number of possible integers is finite, but this is usually not reflected in the type system.) Two varieties of user-

## DESIGN & IMPLEMENTATION

### Decimal types

A few languages, notably Cobol and PL/I, provide a *decimal* type for fixed-point representation of integers in *binary-coded decimal* (BCD) format. BCD devotes one *nibble* (four bits—half a byte) to each decimal digit. Machines that support BCD in hardware can perform arithmetic directly on the BCD representation of a number, without converting it to and from binary form. This capability is particularly useful in business and financial applications, which treat their data as both numbers and character strings: converting a string of ASCII digits to or from BCD is significantly cheaper than converting it to or from binary. BCD format can be found on many (though by no means all) CISC machines, and on at least one RISC machine: the HP PA-RISC.

C# also provides a *decimal* type, but its representation is closer to that of Ada's fixed point types than to the *decimal* types of Cobol and PL/I. Specifically, a C# *decimal* variable is a 128-bit datum that includes 96 *binary* bits of precision, a sign, and a decimal *scaling factor* that can vary between  $10^{-28}$  and  $10^{28}$ . Values of *decimal* type have greater precision but smaller range than double-precision floating-point values. Within their range they are ideal for financial calculations, because they represent decimal fractions precisely.

defined types, enumerations and subranges, are also discrete. Discrete, rational, real, and complex types together constitute the *scalar* types. Scalar types are also sometimes called *simple* types.

### **Enumeration Types**

Enumerations were introduced by Wirth in the design of Pascal. They facilitate the creation of readable programs, and allow the compiler to catch certain kinds of programming errors. An enumeration type consists of a set of named elements. In Pascal, one can write

```
type weekday = (sun, mon, tue, wed, thu, fri, sat);
```

The values of an enumeration type are ordered, so comparisons are generally valid (`mon < tue`), and there is usually a mechanism to determine the predecessor or successor of an enumeration value (in Pascal, `tomorrow := succ(today)`). The ordered nature of enumerations facilitates the writing of enumeration-controlled loops:

```
for today := mon to fri do begin ...
```

It also allows enumerations to be used to index arrays:

```
var daily_attendance : array [weekday] of integer;
```

An alternative to enumerations, of course, is simply to declare a collection of constants:

```
const sun = 0; mon = 1; tue = 2; wed = 3; thu = 4; fri = 5; sat = 6;
```

In C, the difference between the two approaches is purely syntactic. The declaration

```
enum weekday {sun, mon, tue, wed, thu, fri, sat};
```

is essentially equivalent to

```
typedef int weekday;
const weekday sun = 0, mon = 1, tue = 2,
 wed = 3, thu = 4, fri = 5, sat = 6;
```

In Pascal and most of its descendants, however, the difference between an enumeration and a set of integer constants is much more significant: the enumeration is a full-fledged type, incompatible with integers. Using an integer or an enumeration value in a context expecting the other will result in a type clash error at compile time.

Values of an enumeration type are typically represented by small integers, usually a consecutive range of small integers starting at zero. In many languages these *ordinal values* are semantically significant, because built-in functions can be used to convert an enumeration value to its ordinal value, and sometimes vice versa. In Pascal, the built-in function `ord` takes an argument of any enumeration type (including `char` and `Boolean`, which are considered built-in enumerations) and returns the argument's ordinal value. The built-in function `chr` takes an argu-

### **EXAMPLE 7.3**

#### Enumerations in Pascal

### **EXAMPLE 7.4**

#### Enumerations as constants

### **EXAMPLE 7.5**

#### Converting to and from enumeration type

ment  $i$  of type integer and returns the character whose ordinal value is  $i$  (or generates a run-time error if there is no such character). In Ada, `weekday'pos(mon) = 1` and `weekday'val(1) = mon`.

**EXAMPLE 7.6**

Distinguished values for enums

```
enum mips_special_regs {gp = 28, fp = 30, sp = 29, ra = 31};
```

(The intuition behind these values is explained in Section 5.4.4.)

In Ada this declaration would be written

```
type mips_special_regs is (gp, sp, fp, ra); -- must be sorted
for mips_special_regs use (gp => 28, sp => 29, fp => 30, ra => 31);
```

In recent versions of Java one can obtain a similar effect by giving values an extra field (here named `register`):

```
enum mips_special_regs { gp(28), fp(30), sp(29), ra(31);
 private final int register;
 mips_special_regs(int r) register = r;
 public int reg() return register;
}
...
int n = mips_special_regs.fp.reg();
```

As noted in Section 3.6.2, Pascal and C do not allow the same element name to be used in more than one enumeration type in the same scope. Java and C# do, but the programmer must identify elements using fully qualified names: `mips_special_regs.fp`. Ada relaxes this requirement by saying that element names are overloaded; the type prefix can be omitted whenever the compiler can infer it from context.

**Subrange Types**

Like enumerations, subranges were first introduced in Pascal, and are found in many subsequent Algol-family languages. A subrange is a type whose values compose a contiguous subset of the values of some discrete *base* type (also called the *parent* type). In Pascal and most of its descendants, one can declare subranges of integers, characters, enumerations, and even other subranges. In Pascal, subranges look like this:

```
type test_score = 0..100;
workday = mon..fri;
```

In Ada one would write

```
type test_score is new integer range 0..100;
subtype workday is weekday range mon..fri;
```

The `range...` portion of the definition in Ada is called a type *constraint*. In this example `test_score` is a *derived* type, incompatible with integers. The `workday`

**EXAMPLE 7.8**

Subranges in Pascal

**EXAMPLE 7.9**

Subranges in Ada

type, on the other hand, is a *constrained subtype*; workdays and weekdays can be more or less freely intermixed. The distinction between derived types and subtypes is a valuable feature of Ada; we will discuss it further in Section 7.2.1. ■

One could of course use integers to represent test scores, or a weekday to represent a workday. Using an explicit subrange has several advantages. For one thing, it helps to document the program. A comment could also serve as documentation, but comments have a bad habit of growing out of date as programs change, or of being omitted in the first place. Because the compiler analyzes a subrange declaration, it knows the expected range of subrange values, and can generate code to perform dynamic semantic checks to ensure that no subrange variable is ever assigned an invalid value. These checks can be valuable debugging tools. In addition, since the compiler knows the number of values in the subrange, it can sometimes use fewer bits to represent subrange values than it would need to use to represent arbitrary integers. In the example above, `test_score` values can be stored in a single byte.

#### EXAMPLE 7.10

Space requirements of subrange type

```
type water_temperature = 273..373; (* degrees Kelvin *)
```

would be stored in at least two bytes. While there are only 101 distinct values in the type, the largest (373) is too large to fit in a single byte in its natural encoding. (An unsigned byte can hold values in the range 0..255; a signed byte can hold values in the range -128..127.) ■

#### Composite Types

Nonscalar types are usually called *composite*, or *constructed* types. They are generally created by applying a *type constructor* to one or more simpler types. Common composite types include records (structures), variant records (unions), ar-

#### DESIGN & IMPLEMENTATION

##### Multiple sizes of integers

The space savings possible with (small-valued) subrange types in Pascal and Ada is achieved in several other languages by providing more than one size of built-in integer type. C and C++, for example, support integer arithmetic on signed and unsigned variants of `char`, `short`, `int`, `long`, and (in C99) `long long` types, with monotonically nondecreasing sizes.<sup>2</sup>

---

<sup>2</sup> More specifically, the C99 standard requires ranges for these types corresponding to lengths of at least 1, 2, 2, 4, and 8 bytes, respectively. In practice, one finds implementations in which plain `ints` are 2, 4, or 8 bytes long, including some in which they are the same size as `shorts` but shorter than `longs`, and some in which they are the same size as `longs`, but longer than `shorts`.

rays, sets, pointers, lists, and files. All but pointers and lists are easily described in terms of mathematical set operations (pointers and lists can be described mathematically as well, but the description is less intuitive).

*Records* were introduced by Cobol, and have been supported by most languages since the 1960s. A record consists of a collection of *fields*, each of which belongs to a (potentially different) simpler type. Records are akin to mathematical *tuples*; a record type corresponds to the Cartesian product of the types of the fields.

*Variant records* differ from “normal” records in that only one of a variant record’s fields (or collections of fields) is valid at any given time. A variant record type is the *union* of its field types, rather than their Cartesian product.

*Arrays* are the most commonly used composite types. An array can be thought of as a function that maps members of an *index* type to members of a *component* type. Arrays of characters are often referred to as *strings*, and are often supported by special purpose operations not available for other arrays.

*Sets*, like enumerations and subranges, were introduced by Pascal. A set type is the mathematical powerset of its base type, which must usually be discrete. A variable of a set type contains a collection of distinct elements of the base type.

*Pointers* are l-values. A pointer value is a *reference* to an object of the pointer’s base type. Pointers are often but not always implemented as addresses. They are most often used to implement *recursive* data types. A type  $T$  is recursive if an object of type  $T$  may contain one or more references to other objects of type  $T$ .

*Lists*, like arrays, contain a sequence of elements, but there is no notion of mapping or indexing. Rather, a list is defined recursively as either an empty list or a pair consisting of a head element and a reference to a sublist. While the length of an array must be specified at elaboration time in most (though not all) languages, lists are always of variable length. To find a given element of a list, a program must examine all previous elements, recursively or iteratively, starting at the head. Because of their recursive definition, lists are fundamental to programming in most functional languages.

*Files* are intended to represent data on mass storage devices, outside the memory in which other program objects reside. Like arrays, most files can be conceptualized as a function that maps members of an index type (generally integer) to members of a component type. Unlike arrays, files usually have a notion of *current position*, which allows the index to be implied implicitly in consecutive operations. Files often display idiosyncrasies inherited from physical input/output devices. In particular, the elements of some files must be accessed in sequential order.

We will examine composite types in more detail in Sections 7.3 through 7.9.

### 7.1.5 Orthogonality

In Section 6.1.2 we discussed the importance of orthogonality in the design of expressions, statements, and control-flow constructs. Orthogonality is equally important in the design of type systems. Languages vary greatly in the degree of orthogonality they display. A language with a high degree of orthogonality tends to be easier to understand, to use, and to reason about in a formal way. We have noted that languages like Algol 68 and C enhance orthogonality by eliminating (or at least blurring) the distinction between statements and expressions. To characterize a statement that is executed for its side effect(s) and has no useful value, some languages provide an “empty” type. In C and Algol, for example, a subroutine that is meant to be used as a procedure is generally declared with a “return” type of `void`. In ML, the empty type is called `unit`. If the programmer wishes to call a subroutine that does return a value, but the value is not needed in this particular case (all that matters is the side effect[s]), then the return value in C can be *cast* to `void` (casts will be discussed in Section 7.2.1):

```
foo_index = insert_in_symbol_table(foo);
...
(void) insert_in_symbol_table(bar); /* don't care where it went */
/* cast is optional; implied if omitted */
```

#### EXAMPLE 7.11

`Void` (empty) type

#### EXAMPLE 7.12

Making do without `void`

In a language (e.g., Pascal) without an empty type, the latter of these two calls would need to use a dummy variable:

```
var dummy : symbol_table_index;
...
dummy := insert_in_symbol_table(bar);
```

The type system of Pascal is more orthogonal than that of (pre-Fortran 90) Fortran. Among other things, it allows arrays to be constructed from any discrete index type and any component type; pre-Fortran 90 arrays are always indexed by integers and have scalar components. At the same time, Pascal displays several nonorthogonal wrinkles. As we shall see in Section 7.3, it requires that variant fields of a record follow all other fields. It limits function return values to scalar and pointer types. It requires the bounds of each array to be specified at compile time except when the array is a formal parameter of a subroutine. Perhaps most important, while it allows subroutines to be passed as parameters, it does not give them first-class status: a subroutine cannot be returned by a function or stored in a variable. By contrast, the type system of ML, which we examine in Section ⑩ 7.2.4, is almost completely orthogonal.

One particularly useful aspect of type orthogonality is the ability to specify literal values of arbitrary composite types. Several languages provide this capability, but many others do not. Pascal and Modula provide notation for literal character strings and sets, but not for arrays, records, or recursive data structures. The lack of notation for most user-defined composite types means that many Pascal and Modula programs must devote time in every program run to initializing data structures full of compile-time constants.

**EXAMPLE 7.13**

## Aggregates in Ada

Composite values in Ada are specified using *aggregates*:

```
type person is record
 name : string (1..10);
 age : integer;
end record;
p, q : person;
A, B : array (1..10) of integer;
...
p := ("Jane Doe ", 37);
q := (age => 36, name => "John Doe ");
A := (1, 0, 3, 0, 3, 0, 3, 0, 0, 0);
B := (1 => 1, 3 | 5 | 7 => 3, others => 0);
```

Here the aggregates assigned into p and A are *positional*; the aggregates assigned into q and B name their elements explicitly. The aggregate for B uses a short-hand notation to assign the same value (3) into array elements 3, 5, and 7, and to assign a 0 into all unnamed fields. Several languages, including C, Fortran 90, and Lisp, provide similar capabilities. ML provides a very general facility for composite expressions, based on the use of *constructors* (discussed in Section 7.2.4). ■

 **CHECK YOUR UNDERSTANDING**


---

1. What purpose(s) do types serve in a programming language?
  2. What does it mean for a language to be *strongly typed*? *Statically typed*? What prevents, say, C from being strongly typed?
  3. Name two important programming languages that are strongly but dynamically typed.
  4. What is a *type clash*?
  5. Discuss the differences between the *denotational*, *constructive*, and *abstraction-based* views of types.
  6. What is the difference between *discrete* and *scalar* types?
  7. Give two examples of languages that lack a Boolean type. What do they use instead?
  8. In what ways may an enumeration type be preferable to a collection of named constants? In what ways may a subrange type be preferable to its base type? In what ways may a string be preferable to an array of characters?
  9. What does it mean for a set of language features (e.g., a type system) to be *orthogonal*?
  10. What are *aggregates*?
-

## 7.2 Type Checking

In most statically typed languages, every definition of an object (constant, variable, subroutine, etc.) must specify the object's type. Moreover, many of the contexts in which an object might appear are also typed, in the sense that the rules of the language constrain the types that an object in that context may validly possess. In the following subsections we will consider the topics of *type equivalence*, *type compatibility*, and *type inference*. Of the three, type compatibility is the one of most concern to programmers. It determines when an object of a certain type can be used in a certain context. At a minimum, the object can be used if its type and the type expected by the context are equivalent (i.e., the same). In many languages, however, compatibility is a looser relationship than equivalence: objects and contexts are often compatible even when their types are different. Our discussion of type compatibility will touch on the subjects of type *conversion* (also called *casting*), which changes a value of one type into a value of another; type *coercion*, which performs a conversion automatically in certain contexts; and *non-converting* type casts, which are sometimes used in systems programming to interpret the bits of a value of one type as if they represented a value of some other type.

Whenever an expression is constructed from simpler subexpressions, the question arises: given the types of the subexpressions (and possibly the type expected by the surrounding context), what is the type of the expression as a whole? This question is answered by type inference. Type inference is often trivial: the sum of two integers is still an integer, for example. In other cases (e.g., when dealing with sets) it is a good bit trickier. Type inference plays a particularly important role in ML, Miranda, and Haskell, in which all type information is inferred.

### 7.2.1 Type Equivalence

In a language in which the user can define new types, there are two principal ways of defining type equivalence. *Structural equivalence* is based on the *content* of type definitions: roughly speaking, two types are the same if they consist of the same components, put together in the same way. *Name equivalence* is based on the *lexical occurrence* of type definitions: roughly speaking, each definition introduces a new type. Structural equivalence is used in Algol-68, Modula-3, and (with various wrinkles) C and ML. It was also used in many early implementations of Pascal. Name equivalence is the more popular approach in recent languages. It is used in Java, C#, standard Pascal, and most Pascal descendants, including Ada.

The exact definition of structural equivalence varies from one language to another. It requires that one decide which potential differences between types are important, and which may be considered unimportant. Most people would probably agree that the format of a declaration should not matter: in a Pascal-like language with structural equivalence,

---

**EXAMPLE 7.14**

Trivial differences in type

```
type foo = record a, b : integer end;
```

should be considered the same as

```
type foo = record
 a, b : integer
end;
```

These definitions should probably also be considered the same as

```
type foo = record
 a : integer;
 b : integer
end;
```

But what about

```
type foo = record
 b : integer;
 a : integer
end;
```

Should the reversal of the order of the fields change the type? Here the answer is not as clear: ML says no; most languages say yes.

#### **EXAMPLE 7.15**

Other minor differences in type

In a similar vein, the definition of structural equivalence should probably “factor out” different representations of constants: again in a Pascal-like notation,

```
type str = array [1..10] of char;
```

should be considered the same as

```
type str = array [1..2*5] of char;
```

On the other hand, these should probably be considered different from

```
type str = array [0..9] of char;
```

Here the length of the array has not changed, but the index values are different.

To determine if two types are structurally equivalent, a compiler can expand their definitions by replacing any embedded type names with their respective definitions, recursively, until nothing is left but a long string of type constructors, field names, and built-in types. If these expanded strings are the same, then the types are equivalent, and conversely. Recursive and pointer-based types complicate matters, since their expansion does not terminate, but the problem is not insurmountable; we consider a solution in Exercise 7.23.

#### **EXAMPLE 7.16**

The problem with structural equivalence

Structural equivalence is a straightforward but somewhat low-level, implementation-oriented way to think about types. Its principal problem is an inability to distinguish between types that the programmer may think of as distinct, but which happen by coincidence to have the same internal structure:

```

1. type student = record
2. name, address : string
3. age : integer
4. type school = record
5. name, address : string
6. age : integer
7. x : student;
8. y : school;
9. ...
10. x := y; -- is this an error?

```

Most programmers would probably want to be informed if they accidentally assigned a value of type school into a variable of type student, but a compiler whose type checking is based on structural equivalence will blithely accept such an assignment.

Name equivalence is based on the assumption that if the programmer takes the effort to write two type definitions, then those definitions are probably meant to represent different types. In our example code, variables *x* and *y* will be considered to have different types under name equivalence: *x* uses the type declared at line 1; *y* uses the type declared at line 4. ■

#### ***Variants of Name Equivalence***

##### **EXAMPLE 7.17**

Semantically equivalent alias types

```
TYPE new_type = old_type;
```

Should *new\_type* and *old\_type* be considered the same or different? The answer may depend on how the types are used. One possible use is the following.

```

TYPE stack_element = INTEGER; (* or whatever type the user prefers *)
MODULE stack;
IMPORT stack_element;
EXPORT push, pop;
...
PROCEDURE push(elem : stack_element);
...
PROCEDURE pop() : stack_element;
...

```

Here the *stack* module is meant to serve as an abstraction that allows the programmer, via textual inclusion, to create a stack of any desired type (in this case integer). If aliased types are not considered equivalent, then the stack is no longer reusable; it cannot be used for objects whose type has a name of the programmer's choosing. ■

##### **EXAMPLE 7.18**

Semantically distinct alias types

Unfortunately, there are other times, even in Modula-2, when aliased types should probably not be the same.

```

TYPE celsius_temp = REAL;
 fahrenheit_temp = REAL;
VAR c : celsius_temp;
 f : fahrenheit_temp;
...
f := c; (* this should probably be an error *)

```

**EXAMPLE 7.19**

Derived types and subtypes in Ada

A language in which aliased types are considered distinct is said to have *strict name equivalence*. A language in which aliased types are considered equivalent is said to have *loose name equivalence*. Most Pascal-family languages (including Modula-2) use loose name equivalence. Ada achieves the best of both worlds by allowing the programmer to indicate whether an alias represents a *derived type* or a *subtype*. A subtype is compatible with its base (parent) type; a derived type is incompatible. (Subtypes of the same base type are also compatible with each other.) The types of Examples 7.17 and 7.18 would be written as follows.

```

subtype stack_element is integer;
...
type celsius_temp is new integer;
type fahrenheit_temp is new integer;

```

Modula-3, which relies on structural type equivalence, achieves some of the effect of derived types through use of a *branding* mechanism. A BRANDED type is distinct from all other types, regardless of structure. Branding is permitted only for pointers and abstract objects (in the object-oriented sense of the word). Its principal purpose is not to distinguish among types like `celsius_temp` and `fahrenheit_temp` above, but rather to prevent the programmer from using structural equivalence, deliberately or accidentally, to look inside an abstraction that is supposed to be opaque.

One way to think about the difference between strict and loose name equivalence is to remember the distinction between declarations and definitions (Section 3.3.3). Under strict name equivalence, a declaration `type A = B` is considered a definition. Under loose name equivalence it is merely a declaration; `A` shares the definition of `B`.

**EXAMPLE 7.20**

Name v. structural equivalence

Consider the following example.

1. type cell = ... -- whatever
2. type alink = pointer to cell
3. type blink = alink
4. p, q : pointer to cell
5. r : alink
6. s : blink
7. t : pointer to cell
8. u : alink

Here the declaration at line 3 is an alias; it defines `blink` to be “the same as” `alink`. Under strict name equivalence, line 3 is both a declaration and a definition, and `blink` is a new type, distinct from `alink`. Under loose name equivalence, line 3 is just a declaration; it uses the definition at line 2.

Under strict name equivalence, *p* and *q* have the same type, because they both use the *anonymous* (unnamed) type definition on the right-hand side of line 4, and *r* and *u* have the same type, because they both use the definition at line 2. Under loose name equivalence, *r*, *s*, and *u* all have the same type, as do *p* and *q*. Under structural equivalence, all six of the variables shown have the same type, namely pointer to whatever cell is. ■

Both structural and name equivalence can be tricky to implement in the presence of separate compilation. We will return to this issue in Section 14.6.

### Type Conversion and Casts

#### EXAMPLE 7.21

Contexts that expect a given type

In a language with static typing, there are many contexts in which values of a specific type are expected. In the statement

*a* := *expression*

we expect the right-hand side to have the same type as *a*. In the expression

*a* + *b*

the overloaded + symbol designates either integer or floating-point addition; we therefore expect either that *a* and *b* will both be integers or that they will both be reals. In a call to a subroutine,

foo(arg1, arg2, ..., argN)

we expect the types of the arguments to match those of the formal parameters, as declared in the subroutine's header. ■

Suppose for the moment that we require in each of these cases that the types (expected and provided) be exactly the same. Then if the programmer wishes to use a value of one type in a context that expects another, he or she will need to specify an explicit *type conversion* (also sometimes called a *type cast*). Depending on the types involved, the conversion may or may not require code to be executed at run time. There are three principal cases:

1. The types would be considered structurally equivalent, but the language uses name equivalence. In this case the types employ the same low-level representation, and have the same set of values. The conversion is therefore a purely conceptual operation; no code will need to be executed at run time.
2. The types have different sets of values, but the intersecting values are represented in the same way. One type may be a subrange of the other, for example, or one may consist of two's complement signed integers, while the other is unsigned. If the provided type has some values that the expected type does not, then code must be executed at run time to ensure that the current value is among those that are valid in the expected type. If the check fails, then a dynamic semantic error results. If the check succeeds, then the underlying representation of the value can be used, unchanged. Some language implementations may allow the check to be disabled, resulting in faster but potentially unsafe code.

3. The types have different low-level representations, but we can nonetheless define some sort of correspondence among their values. A 32-bit integer, for example, can be converted to a double-precision IEEE floating-point number with no loss of precision. Most processors provide a machine instruction to effect this conversion. A floating-point number can be converted to an integer by rounding or truncating, but fractional digits will be lost, and the conversion will overflow for many exponent values. Again, most processors provide a machine instruction to effect this conversion. Conversions between different lengths of integers can be effected by discarding or sign-extending high-order bytes.

**EXAMPLE 7.22**

## Type conversions in Ada

```

n : integer; -- assume 32 bits
r : real; -- assume IEEE double-precision
t : test_score; -- as in Example 7.9
c : celsius_temp; -- as in Example 7.19
...
t := test_score(n); -- run-time semantic check required
n := integer(t); -- no check req.; every test_score is an int
r := real(n); -- requires run-time conversion
n := integer(r); -- requires run-time conversion and check
n := integer(c); -- no run-time code required
c := celsius_temp(n); -- no run-time code required

```

In each of these last six lines, the name of a type is used as a pseudo-function that performs a type conversion. The first conversion requires a run-time check to ensure that the value of *n* is within the bounds of a *test\_score*. The second conversion requires no code, since every possible value of *t* is acceptable for *n*. The third and fourth conversions require code to change the low-level representation of values. The fourth conversion also requires a semantic check. It is generally understood that converting from a floating-point value to an integer results in the loss of fractional digits; this loss is not an error. If the conversion results in integer overflow, however, an error needs to result. The final two conversions require no run-time code; the *integer* and *celsius\_temp* types (at least as we have defined them) have the same sets of values and the same underlying representation. A purist might say that *celsius\_temp* should be defined as *new integer range -273..integer'last*, in which case a run-time semantic check would be required on the final conversion. ■

**Nonconverting Type Casts** Occasionally, particularly in systems programs, one needs to change the type of a value *without* changing the underlying implementation—in other words, to interpret the bits of a value of one type as if they were another type. One common example occurs in memory allocation algorithms, which use a large array of characters or integers to represent a heap, but then reinterpret portions of that array as pointers and integers (for bookkeeping purposes), or as various user-allocated data structures. Another common exam-

ple occurs in high-performance numeric software, which may need to reinterpret a floating-point number as an integer or a record in order to extract the exponent, significand, and sign fields. These fields can be used to implement special purpose algorithms for square root, trigonometric functions, and so on.

A change of type that does not alter the underlying bits is called a *nonconverting type cast*. It should not be confused with use of the term *cast* for conversions in languages like C. In Ada, nonconverting casts can be effected using instances of a built-in generic subroutine called `unchecked_conversion`:

```
-- assume 'float' has been declared to match IEEE single-precision
function cast_float_to_int is
 new unchecked_conversion(float, integer);
function cast_int_to_float is
 new unchecked_conversion(integer, float);
...
f := cast_int_to_float(n);
n := cast_float_to_int(f);
```

**EXAMPLE 7.23**

Unchecked conversions in Ada

**EXAMPLE 7.24**

Type conversions in C

A type conversion in C (i.e., what C calls a type cast) is specified by using the name of the desired type, in parentheses, as a prefix operator:

```
r = (float) n; /* generates code for run-time conversion */
n = (int) r; /* also run-time conversion, with no overflow check */
```

C and its descendants do not by default perform run-time checks for arithmetic overflow on any operation, though such checks can be enabled if desired in C#.

C++ inherits the casting mechanism of C but also provides a family of semantically cleaner alternatives. Specifically, `static_cast` performs a type conversion, `reinterpret_cast` performs a nonconverting type cast, and `dynamic_cast` allows programs that manipulate pointers of polymorphic types to perform assignments whose validity cannot be guaranteed statically, but can be checked at run time (more on this in Chapter 9). There is also a `const_cast` that can be used to add or remove read-only qualification. C-style type casts in C++ are defined in terms of `const_cast`, `static_cast`, and `reinterpret_cast`; the precise behavior depends on the source and target types.

Any nonconverting type cast constitutes a dangerous subversion of the language's type system. In a language with a weak type system such subversions can be difficult to find. In a language with a strong type system, the use of explicit nonconverting type casts at least labels the dangerous points in the code, facilitating debugging if problems arise.

### 7.2.2 Type Compatibility

Most languages do not require equivalence of types in every context. Instead, they merely say that a value's type must be compatible with that of the context in which it appears. In an assignment statement, the type of the right-hand side

must be compatible with that of the left-hand side. The types of the operands of `+` must either both be compatible with the built-in integer type, or both be compatible with the built-in floating-point type. In a subroutine call, the types of any arguments passed into the subroutine must be compatible with the types of the corresponding formal parameters, and the types of any formal parameters passed back to the caller must be compatible with the types of the corresponding arguments.

The definition of type compatibility varies greatly from language to language. Ada takes a relatively restrictive approach: an Ada type `S` is compatible with an expected type `T` if and only if (1) `S` and `T` are equivalent, (2) one is a subtype of the other (or both are subtypes of the same base type), or (3) both are arrays, with the same numbers and types of elements in each dimension. Pascal is only slightly more lenient: in addition to allowing the intermixing of base and subrange types, it allows an integer to be used in a context where a real is expected.

### ***Coercion***

Whenever a language allows a value of one type to be used in a context that expects another, the language implementation must perform an automatic, implicit

#### **DESIGN & IMPLEMENTATION**

##### **Nonconverting casts**

C programmers sometimes attempt a nonconverting type cast by taking the address of an object, converting the type of the resulting pointer, and then dereferencing:

```
r = *((float *) &n);
```

This arcane bit of hackery usually works, because most (but not all!) implementations use the same representation for pointers to integers and pointers to floating-point values—namely, an address. The ampersand operator (`&`) means “address of,” or “pointer to.” The parenthesized `(float *)` is the type name for “pointer to float” (`float` is a built-in floating-point type). The prefix `*` operator is a pointer dereference. The cast produces no run-time code; it merely causes the compiler to interpret the bits of `n` as if it were a `float`. The reinterpretation will fail if `n` is not an l-value (has no address), or if `ints` and `floats` have different sizes (again, this second condition is often but not always true in C). If `n` does not have an address then the compiler will announce a static semantic error. If `int` and `float` do not occupy the same number of bytes, then the effect of the cast may depend on a variety of factors, including the relative size of the objects, the alignment and “endian-ness” of memory (Section 5.2), and the choices the compiler has made regarding what to place in adjacent locations in memory. Safer and more portable nonconverting casts can be achieved in C by means of unions (variant records); we consider this option in Exercise 7.9.

**EXAMPLE 7.25**

## Coercion in Ada

conversion to the expected type. This conversion is called a *type coercion*. Like an explicit conversion, a coercion may require run-time code to perform a dynamic semantic check or to convert between low level representations. Ada coercions sometimes need the former, though never the latter:

```
d : weekday; -- as in Example 7.3
k : workday; -- as in Example 7.9
type calendar_column is new weekday;
c : calendar_column;
...
k := d; -- run-time check required
d := k; -- no check required; every workday is a weekday
c := d; -- static semantic error;
 -- weekdays and calendar_columns are not compatible
```

To perform this third assignment in Ada we would have to use an explicit conversion:

```
c := calendar_column(d);
```

**EXAMPLE 7.26**

## Coercion in C

Coercions are a controversial subject in language design. Because they allow types to be mixed without an explicit indication of intent on the part of the programmer, they represent a significant weakening of type security. Fortran and C, which have relatively weak type systems, perform quite a bit of coercion. They allow values of most numeric types to be intermixed in expressions, and will coerce types back and forth “as necessary.” Here are some examples in C.

```
short int s;
unsigned long int l;
char c; /* may be signed or unsigned -- implementation-dependent */
float f; /* usually IEEE single-precision */
double d; /* usually IEEE double-precision */
...
s = l; /* l's low-order bits are interpreted as a signed number. */
l = s; /* s is sign-extended to the longer length, then
 its bits are interpreted as an unsigned number. */
s = c; /* c is either sign-extended or zero-extended to s's length;
 the result is then interpreted as a signed number. */
f = l; /* l is converted to floating-point. Since f has fewer
 significant bits, some precision may be lost. */

d = f; /* f is converted to the longer format; no precision lost. */
f = d; /* d is converted to the shorter format; precision may be lost.
 If d's value cannot be represented in single-precision, the
 result is undefined, but NOT a dynamic semantic error. */
```

Fortran 90 allows arrays and records to be intermixed if their types have the same *shape*. Two arrays have the same shape if they have the same number of dimensions, each dimension has the same size, and the individual elements have

the same shape. These rules are roughly equivalent to the compatibility rules for arrays in Ada, but Fortran 90 allows arrays to be used in many more contexts. In particular, it allows its full set of arithmetic operations to be applied, element-by-element, to array-valued operands.

Two Fortran 90 records have the same shape if they have the same number of fields, and corresponding fields, in order, have the same shape. Field *names* do not matter, nor do the actual high and low bounds of array dimensions. C does not allow records (structures) to be intermixed unless they are structurally equivalent, with identical field names. C provides no operations that take an entire array as an operand. C does, however, allow arrays and *pointers* to be intermixed in many cases; we will discuss this unusual form of type compatibility further in Section 7.7.1.

Most modern languages reflect a trend toward static typing and away from type coercion. Some language designers have argued, however, that coercions are a natural way in which to support abstraction and program extensibility, by making it easier to use new types in conjunction with existing ones. C++ in particular provides an extremely rich, *programmer-extensible* set of coercion rules. When defining a new type (a *class* in C++), the programmer can define coercion operations to convert values of the new type to and from existing types. These rules interact in complicated ways with the rules for resolving overloading (Section 3.6.2); they add significant flexibility to the language, but are one of the most difficult C++ features to understand and use correctly.

### **Overloading and Coercion**

We have noted (in Section 3.6.3) that overloading and coercion (as well as various forms of polymorphism) can sometimes be used to similar effect. It is worth repeating some of the distinctions here. An overloaded name can refer to more than one object; the ambiguity must be resolved by context. In the expression  $a + b$ , for example,  $+$  may refer to either the integer or the floating-point addition operation. In a language without coercion,  $a$  and  $b$  must either both be integer or both be real; the compiler chooses the appropriate interpretation of  $+$  depending on their type. In a language with coercion,  $+$  refers to the floating-point addition operation if either  $a$  or  $b$  is real; otherwise it refers to the integer addition operation. If only one of  $a$  and  $b$  is real, the other is coerced to match. One could imagine a language in which  $+$  was not overloaded, but rather referred to floating-point addition in all cases. Coercion could still allow  $+$  to take integer arguments, but they would always be converted to real. The problem with this approach is that conversions from integer to floating-point format take a non-negligible amount of time, especially on machines without hardware conversion instructions, and floating-point addition is significantly more expensive than integer addition.

In most languages literal (manifest) constants (e.g., numbers, character strings, the empty set `[]` or the null pointer `[nil]`) can be intermixed in expressions with values of many types. One might say that constants are over-

loaded: `nil` for example might be thought of as referring to the null pointer value for whatever type is needed in the surrounding context. More commonly, however, constants are simply treated as a special case in the language's type-checking rules. Internally, the compiler considers a constant to have one of a small number of built-in "constant types" (`int const`, `real const`, `string`, `nil`), which it then coerces to some more appropriate type as necessary, even if coercions are not supported elsewhere in the language. Ada formalizes this notion of "constant type" for numeric quantities: an integer constant (one without a decimal point) is said to have type `universal_integer`; a floating-point constant (one with an embedded decimal point and/or an exponent) is said to have type `universal_real`. The `universal_integer` type is compatible with any type derived from `integer`; `universal_real` is compatible with any type derived from `real`.

### **Generic Reference Types**

For systems programming, or to facilitate the writing of general purpose *container (collection)* objects (lists, stacks, queues, sets, etc.) that hold references to other objects, several languages provide a "generic reference" type. In C and C++, this type is called `void *`. In Clu it is called `any`; in Modula-2, `address`; in Modula-3, `refany`; in Java, `Object`; in C#, `object`. Arbitrary l-values can be assigned into an object of generic reference type, with no concern about type safety: because the type of the object referred to by a generic reference is unknown, the compiler will not allow any operations to be performed on that object. Assignments back into objects of a particular reference type (e.g., a pointer to a programmer-specified record type) are a bit trickier, if type safety is to be maintained. We would not want a generic reference to a floating-point number, for example, to be assigned into a variable that is supposed to hold a reference to an integer, because subsequent operations on the "integer" would interpret the bits of the object incorrectly. In object-oriented languages, the question of how to ensure the validity of a generic to specific assignment generalizes to the question of how to ensure the validity of any assignment in which the type of the object on left-hand side supports operations that the object on the right-hand side may not.

One way to ensure the safety of generic to specific assignments (or, in general, less specific to more specific assignments) is to make objects self-descriptive—that is, to include in the representation of each object an indication of its type. This approach is common in object-oriented languages: it is taken in Java, C#, Eiffel, Modula-3, and C++. (Smalltalk objects are self-descriptive, but Smalltalk variables are not typed.) Type tags in objects can consume a nontrivial amount of space, but allow the implementation to prevent the assignment of an object of one type into a variable of another. In Java and C#, a generic to specific assignment requires a type cast, but will generate an exception if the generic reference does not refer to an object of the casted type. In Eiffel, the equivalent operation uses a special assignment operator (`?=` instead of `:=`); in C++ it uses a `dynamic_cast` operation.

**EXAMPLE 7.27**Java container of `Object`

Java and C# programmers frequently create container classes that hold objects of the generic reference class (`Object` or `object`, respectively). When an object is removed from a container, it must be assigned (with a type cast) into a variable of an appropriate class before anything interesting can be done with it:<sup>3</sup>

```
import java.util.*; // library containing Stack container class
...
Stack myStack = new Stack();
String s = "Hi, Mom";
Foo f = new Foo(); // f is of user-defined class type Foo
...
myStack.push(s);
myStack.push(f); // we can push any kind of object on a stack
...
s = (String) myStack.pop();
// type cast is required, and will generate an exception at run
// time if element at top-of-stack is not a string
```

In a language without type tags, the assignment of a generic reference into an object of a specific reference type cannot be checked, because objects are not self-descriptive: there is no way to identify their type at run time. The programmer must therefore resort to an (unchecked) type conversion. C++ minimizes the overhead of type tags by permitting `dynamic_cast` operations only on objects of polymorphic types. A thorough explanation of this restriction requires an understanding of virtual methods and their implementation, something we defer to Sections 9.4.1 and 9.4.2.

### 7.2.3 Type Inference

We have seen how type checking ensures that the components of an expression (e.g., the arguments of a binary operator) have appropriate types. But what determines the type of the overall expression? In most cases, the answer is easy. The result of an arithmetic operator usually has the same type as the operands. The result of a comparison is usually Boolean. The result of a function call has the type declared in the function's header. The result of an assignment (in languages in which assignments are expressions) has the same type as the left-hand side. In a few cases, however, the answer is not obvious. In particular, operations on subranges and on composite objects do not necessarily preserve the types of the operands. We examine these cases in the remainder of this subsection. We then consider (on the PLP CD) a more elaborate form of type inference found in ML, Miranda, and Haskell.

---

**3** If the programmer knows that a container will be used to hold objects of only one type, then it may be possible to eliminate the type cast and, ideally, its run-time cost by using generics (Section 8.4).

### **Subranges**

**EXAMPLE 7.28**

Inference of subrange types

```
type Atype = 0..20;
 Btype = 10..20;
var a : Atype;
 b : Btype;
```

what is the type of  $a + b$ ? Certainly it is neither *Atype* nor *Btype*, since the possible values range from 10 to 40. One could imagine it being a new anonymous subrange type with 10 and 40 as bounds. The usual answer in Pascal and its descendants is to say that the result of any arithmetic operation on a subrange has the subrange's base type, in this case integer. ■

In Ada, the type of an arithmetic expression assumes special significance in the header of a *for* loop (Section 6.5.1) because it determines the type of the index variable. For the sake of uniformity, Ada says that the index of a *for* loop always has the base type of the loop bounds, whether they are built-up expressions or simple variables or constants.

If the result of an arithmetic operation is assigned into a variable of a subrange type, then a dynamic semantic check may be required. To avoid the expense of some unnecessary checks, a compiler may keep track at compile time of the largest and smallest possible values of each expression, in essence computing the anonymous  $10\dots 40$  type. Appropriate bounds for the result of an arithmetic operator can always be calculated from the values for the operands. In addition, for example,

```
result.min := operand1.min + operand2.min
result.max := operand1.max + operand2.max
```

For subtraction,

```
result.min := operand1.min - operand2.max
result.max := operand1.max - operand2.min
```

The rules for other operators are analogous. ■

When an expression is assigned to a subrange variable or passed as a subrange parameter, the compiler can decide on the need for checks based on the bounds of the expected type and on the minimum and maximum values maintained for the expression. If the minimum possible value of the expression is smaller than the lower bound of the expected type, or if the maximum possible value of the expression is larger than the upper bound of the expected type, a run-time check is required. At the same time, if the minimum possible value of the expression is larger than the upper bound of the expected type, or the maximum possible value of the expression is smaller than the lower bound of the expected type, then the compiler can issue a semantic error message at compile time.

**EXAMPLE 7.29**

Using inference to avoid run-time checks

**EXAMPLE 7.30**

Heuristic nature of subrange inference

It should be noted that this bounds-tracking technique will not eliminate all unnecessary checks. In the following Ada code, for example, a compiler that aimed to do a perfect job of predicting the need for dynamic semantic checks would need to predict the possible return values of a programmer-specified function.

```
a : integer range 0..20;
b : integer range 10..20;
function foo(i : integer) return integer is ...
...
a := b - foo(10); -- does this require a dynamic semantic check?
```

If `foo(10)` is guaranteed to lie between 0 and 10, then no dynamic check is required; the assignment is sure to be ok. If `foo(10)` is guaranteed to be greater than 20 or less than  $-10$ , then again no check is required; an error can be announced at compile time. Unfortunately, the value of `foo` may depend on values read at run time. Even if it does not, basic results in complexity theory imply that no compiler will be able to predict the behavior of all user-specified functions. Because of these limitations, the compiler must inevitably generate some unnecessary run-time checks; straightforward tracking of the minimum and maximum values for expressions is only a heuristic that allows us to eliminate some unnecessary checks in practice. More sophisticated techniques can be used to eliminate many checks in loops; we will consider these in Section 15.5.2. ■

### Composite Types

Most built-in operators in most languages take operands of built-in types. Some operators, however, can be applied to values of composite types, including aggregates. Type inference becomes an issue when an operation on composites yields a result of a different type than the operands.

**EXAMPLE 7.31**

Type inference on string operations

Character strings provide a simple example. In Pascal, the literal string '`abc`' has type `array [1..3] of char`. In Ada, the analogous string (denoted "`abc`") is considered to have an incompletely specified type that is compatible with any three-element array of characters. In the Ada expression "`abc`" & "`defg`", "`abc`" is a three-character array, "`defg`" is a four-character array, and the result is a seven-character array formed by concatenating the two. For all three, the size of the array is known, but the bounds and the index type are not; they must be inferred from context. The seven-character result of the concatenation could be assigned into an array of type `array (1..7) of character` or into an array of type `array (weekday) of character`, or into any other seven-element character array. ■

**EXAMPLE 7.32**

Type inference for sets

Operations on composite values also occur when manipulating sets in Pascal and Modula. As with string concatenation, operations on sets do not necessarily produce a result of the same type as the operands. Consider the following example in Pascal.

```

var A : set of 1..10;
 B : set of 10..20;
 C : set of 1..15;
 i : 1..30;
 ...
C := A + B * [1..5, i];

```

Pascal provides three operations on sets: union (+), intersection (\*), and difference (-). Set operands are said to have compatible types if their elements have the same base type T. The result of a set operation is then of type `set of T`. In the example above, A, B, and the constructed set `[1..5, i]` all have the same base type—namely integer. The type of the right-hand side of the assignment is therefore `set of integer`. When an expression is assigned to a set variable or passed as a set parameter, a dynamic semantic check may be required. In the example, the assignment will require a check to ensure that none of the possible values between 16 and 20 actually occur in the set. ■

As with subranges, a compiler can avoid the need for checks in certain cases by keeping track of the minimum and maximum possible members of the set expression. Because a set may have many members, some of which may be known at compile time, it can be useful to track not only the largest and smallest values that may be in a set, but also the values that are known to be in the set (see Exercise 7.7).

In Section 7.2.2 we noted that Fortran 90 allows all of its built-in arithmetic operations to be applied to arrays. The result of an array operation has the same shape as the operands. Each of its elements is the result of applying the operation to the corresponding elements of the operand arrays. Since shape is preserved, type inference is not an issue.

#### 7.2.4 The ML Type System

The most sophisticated form of type inference occurs in certain functional languages—notably ML, Miranda, and Haskell. Programmers have the option of declaring the types of objects in these languages, in which case the compiler behaves much like that of a more traditional statically typed language. As we noted near the beginning of Section 7.1, however, programmers may also choose not to declare certain types, in which case the compiler will infer them, based on the known types of manifest constants, the explicitly declared types of any objects that have them, and the syntactic structure of the program. ML-style type inference is the invention of the language’s creator, Robin Milner.<sup>4</sup>

---

**4** Robin Milner (1934–), of Cambridge University’s Computer Laboratory, is responsible not only for the development of ML and its type system, but for the Logic of Computable Functions, which provides a formal basis for machine-assisted proof construction, and the Calculus of Communicating Systems, which provides a general theory of concurrency. He received the ACM Turing Award in 1991.

 IN MORE DEPTH

The key to type inference in ML and its descendants is to *unify* the (partial) type information available for two expressions whenever the rules of the type system say that their types must be the same. Information known about each is then known about the other as well. Any discovered inconsistencies are identified as static semantic errors. Any expression whose type remains incompletely specified after inference is automatically polymorphic; this is the *implicit parametric polymorphism* referred to in Section 3.6.3. ML family languages also incorporate a powerful run-time pattern-matching facility and several unconventional structured types, including ordered tuples, (unordered) records, lists, and a datatype mechanism that subsumes unions and recursive types.

 CHECK YOUR UNDERSTANDING

11. What is the difference between *type equivalence* and *type compatibility*?
12. Discuss the comparative advantages of *structural* and *name* equivalence for types. Name three languages that use each approach.
13. Explain the difference between *strict* and *loose* name equivalence.
14. Explain the distinction between *derived* types and *subtypes* in Ada.
15. Explain the difference between *type conversion*, *type coercion*, and *nonconverting type casts*.
16. Summarize the arguments for and against coercion.
17. Under what circumstances does a type conversion require a run-time check?
18. What purpose is served by “generic reference” types?
19. What is *type inference*? Describe three contexts in which it occurs.

## 7.3 Records (Structures) and Variants (Unions)

As we have seen, record types allow related data of heterogeneous types to be stored and manipulated together. Some languages (notably Algol 68, C, C++, and Common Lisp) use the term *structure* (declared with the keyword `struct`) instead of *record*. Fortran 90 simply calls its records “types”: they are the only form of programmer-defined type other than arrays, which have their own special syntax. Structures in C++ are defined as a special form of *class* (one in which members are globally visible by default). Java has no distinguished notion of `struct`; its programmers use classes in all cases. C# uses a reference model for variables of `class` types, and a value model for variables of `struct` types. C# `structs` do not support inheritance.

### 7.3.1 Syntax and Operations

#### EXAMPLE 7.33

A Pascal record

```
type two_chars = packed array [1..2] of char;
(* Packed arrays will be explained in Example 7.39.
 Packed arrays of char are compatible with quoted strings. *)
type element = record
 name : two_chars;
 atomic_number : integer;
 atomic_weight : real;
 metallic : Boolean
end;
```

#### EXAMPLE 7.34

A C struct

```
struct element {
 char name[2];
 int atomic_number;
 double atomic_weight;
 _Bool metallic;
};
```

In C, the corresponding declaration would be  
 Each of the record components is known as a *field*. To refer to a given field of a record, most languages use “dot” notation. In Pascal:

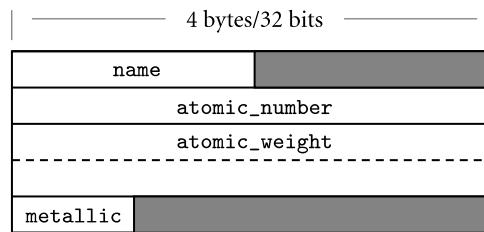
```
var copper : element;
const AN = 6.022e23; (* Avogadro's number *)
...
copper.name := 'Cu';
atoms := mass / copper.atomic_weight * AN;
```

The C notation is similar to that of Pascal; in Fortran 90 one would say copper%name and copper%atomic\_weight. Cobol and Algol 68 reverse the order of the field and record names: name of copper and atomic\_weight of copper. ML’s notation is also “reversed,” but uses a prefix #: #name copper and #atomic\_weight copper. (Fields of an ML record can also be extracted using patterns.) In Common Lisp, one would say (element-name copper) and (element-atomic\_weight copper).

Most languages allow record definitions to be nested. Again in Pascal:

```
type short_string = packed array [1..30] of char;
type ore = record
 name : short_string;
 element_yielded : record
 name : two_chars;
 atomic_number : integer;
 atomic_weight : real;
 metallic : Boolean
 end
end;
```

Alternatively, one could say



**Figure 7.1** Likely layout in memory for objects of type `element` on a 32-bit machine. Alignment restrictions lead to the shaded “holes.”

```
type ore = record
 name : short_string;
 element_yielded : element
end;
```

In Fortran 90 and Common Lisp, only the second alternative is permitted: record fields can have record types, but the declarations cannot be lexically nested. Naming for nested records is straightforward: `malachite.element_yielded.atomic_number` in Pascal or C; `atomic_number` of `element_yielded` of `malachite` in Cobol; `#atomic_number #element_yielded malachite` in ML; `(element-atomic_number (ore-element_yielded malachite))` in Common Lisp. ■

#### EXAMPLE 7.37

ML records and tuples

As noted in Example 7.14, ML differs from most languages in specifying that the order of record fields is insignificant. The ML record value `{name = "Cu", atomic_number = 29, atomic_weight = 63.546, metallic = true}` is the same as the value `{atomic_number = 29, name = "Cu", atomic_weight = 63.546, metallic = true}` (they will test true for equality). ML tuples are defined as abbreviations for records whose field names are small integers. The values `("Cu", 29)`, `{1 = "Cu", 2 = 29}`, and `{2 = 29, 1 = "Cu"}` will all test true for equality. ■

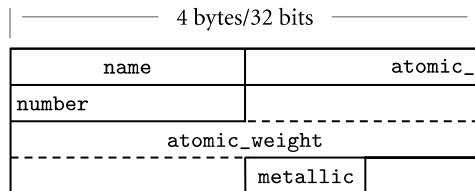
### 7.3.2 Memory Layout and Its Impact

The fields of a record are usually stored in adjacent locations in memory. In its symbol table, the compiler keeps track of the offset of each field within each record type. When it needs to access a field, the compiler typically generates a load or store instruction with displacement addressing. For a local object, the base register is the frame pointer; for a global object, the base register is the globals pointer. In either case, the displacement is the sum of the record’s offset from the register and the field’s offset within the record.

#### EXAMPLE 7.38

Memory layout for a record type

A likely layout for our `element` type on a 32-bit machine appears in Figure 7.1. Because the `name` field is only two characters long, it occupies two bytes in memory. Since `atomic_number` is an integer, and must (on most machines) be longword-aligned, there is a two-byte “hole” between the end of `name` and the be-



**Figure 7.2** Likely memory layout for packed element records. The `atomic_number` and `atomic_weight` fields are nonaligned, and can only be read or written (on most machines) via multi-instruction sequences.

ginning of `atomic_number`. Similarly, since Boolean variables (in most language implementations) occupy a single byte, there are three bytes of empty space between the end of the `metallic` field and the next aligned location. In an array of elements, most compilers would devote 20 bytes to every member of the array.

Pascal allows the programmer to specify that a record type (or an array, set, or file type) should be *packed*:

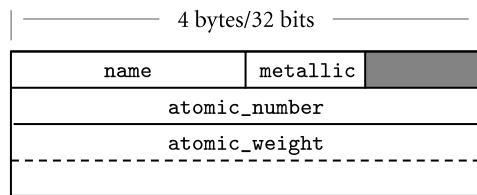
```
type element = packed record
 name : two_chars;
 atomic_number : integer;
 atomic_weight : real;
 metallic : Boolean
end;
```

The keyword `packed` indicates that the compiler should optimize for space instead of speed. In most implementations a compiler will implement a packed record without holes, by simply “pushing the fields together.” To access a non-aligned field, however, it will have to issue a multi-instruction sequence that retrieves the pieces of the field from memory and then reassembles them in a register. A likely packed layout for our `element` type (again for a 32-bit machine) appears in Figure 7.2. It is 15 bytes in length. An array of packed `element` records would probably devote 16 bytes to each member of the array—that is, it would align each element. A packed array of packed records would probably devote only 15 bytes to each; only every fourth element would be aligned. Ada, Modula-3, and C provide more elaborate packing mechanisms, which allow the programmer to specify precisely how many bits are to be devoted to each field.

Most languages allow a value to be assigned to an entire record in a single operation:

```
my_element := copper;
```

Ada also allows records to be compared for equality (`if my_element = copper then ...`), but most other languages (including Pascal, Modula, C, and C++) do not, though C++ allows the programmer to define equality tests for individual record types.



**Figure 7.3 Rearranging record fields to minimize holes.** By sorting fields according to the size of their alignment constraint, a compiler can minimize the space devoted to holes, while keeping the fields aligned.

For small records, both copies and comparisons can be performed in-line on a field-by-field basis. For longer records, we can save significantly on code space by deferring to a library routine. A `block_copy` routine can take source address, destination address, and length as arguments, but the analogous `block_compare` routine would fail on records with different (garbage) data in the holes. One solution is to arrange for all holes to contain some predictable value (e.g., zero), but this requires code at every elaboration point. Another is to have the compiler generate a customized field-by-field comparison routine for every record type. Different routines would be called to compare records of different types. Languages like Pascal and C avoid the whole issue by simply outlawing full-record comparisons.

#### EXAMPLE 7.41

Minimizing holes by sorting fields

In addition to complicating comparisons, holes in records waste space. Packing eliminates holes, but at potentially heavy cost in access time. A compromise, adopted by some compilers, is to sort a record's fields according to the size of their alignment constraints. All byte-aligned fields come first, followed by any half-word aligned fields, word-aligned fields, and (if the hardware requires) double-word-aligned fields. For our `element` type, the resulting rearrangement is shown in Figure 7.3. ■

In most cases, reordering of fields is purely an implementation issue: the programmer need not be aware of it, as long as all instances of a record type are reordered in the same way. The exception occurs in systems programs, which sometimes “look inside” the implementation of a data type with the expectation

#### DESIGN & IMPLEMENTATION

##### The order of record fields

Issues of record field order are intimately tied to implementation tradeoffs: Holes in records waste space, but alignment makes for faster access. If holes contain garbage, we can't compare records by looping over words or bytes, but zero-ing out the holes would incur costs in time and code space. Predictable layout is important for mirroring hardware structures in “systems” languages, but reorganization may be advantageous in large records if we can group frequently accessed fields together, so they lie in the same cache line.

that it will be mapped to memory in a particular way. A kernel programmer, for example, may count on a particular layout strategy in order to define a record that mimics the organization of memory-mapped control registers for a particular Ethernet device. C and C++, which are designed in large part for systems programs, guarantee that the fields of a `struct` will be allocated in the order declared. The first field is guaranteed to have the coarsest alignment required by the hardware for any type (generally a four- or eight-byte boundary). Subsequent fields have the natural alignment for their type. To accommodate systems programs, Ada and C++ allow the programmer to specify nonstandard alignment for the fields of specific record types.

### 7.3.3 With Statements

#### EXAMPLE 7.42

Pascal `with` statement

In programs with complicated data structures, manipulating the fields of a deeply nested record can be awkward:

```
ruby.chemical_composition.elements[1].name := 'Al';
ruby.chemical_composition.elements[1].atomic_number := 13;
ruby.chemical_composition.elements[1].atomic_weight := 26.98154;
ruby.chemical_composition.elements[1].metallic := true;
```

Pascal provides a `with` statement to simplify such constructions:

```
with ruby.chemical_composition.elements[1] do begin
 name := 'Al';
 atomic_number := 13;
 atomic_weight := 26.98154;
 metallic := true
end;
```

#### IN MORE DEPTH

Pascal `with` statements are generally considered an improvement on the earlier *elliptical references* of Cobol and PL/I. They still suffer from several limitations, however, most of which are addressed in Modula-3. Similar functionality can be achieved with nested scopes in languages like Lisp and ML, which use a reference model of variables, and in languages like C and C++, which allow the programmer to create pointers or references to arbitrary objects.

### 7.3.4 Variant Records

#### EXAMPLE 7.43

Variant record in Pascal

A variant record provides two or more alternative fields or collections of fields, only one of which is valid at any given time. In Pascal, we might augment our `element` type as follows.

```

type long_string = packed array [1..200] of char;
type string_ptr = ^long_string;
type element = record
 name : two_chars;
 atomic_number : integer;
 atomic_weight : real;
 metallic : Boolean;
 case naturally_occuring : Boolean of
 true : (
 source : string_ptr;
 (* textual description of principal commercial source *)
 prevalence : real;
 (* fraction, by weight, of Earth's crust *)
);
 false : (
 lifetime : real;
 (* half-life in seconds of the most stable known isotope *)
)
 end;

```

Here the `naturally_occuring` field of the record is known as its *tag*, or *discriminant*. A true tag indicates that the element has at least one naturally occurring stable isotope; in this case the record contains two additional fields—`source` and `prevalence`—that describe how the element may be obtained and how commonly it occurs. A false tag indicates that the element results only from atomic collisions or the decay of heavier elements; in this case, the record contains an additional field—`lifetime`—that indicates how long atoms so created tend to survive before undergoing radioactive decay. Each of the parenthesized field lists (one containing `source` and `prevalence`, the other containing `lifetime`) is known as a *variant*. Either the first or the second variant may be useful, but never both at once. From an implementation point of view, these nonoverlapping uses mean that the variants may share space (see Figure 7.4).

Variant records have their roots in the equivalence statement of Fortran I and in the union types of Algol 68. The Fortran syntax looks like this:

```

integer i
real r
logical b
equivalence (i, r, b)

```

The equivalence statement informs the compiler that `i`, `r`, and `b` will never be used at the same time, and should share the same space in memory.

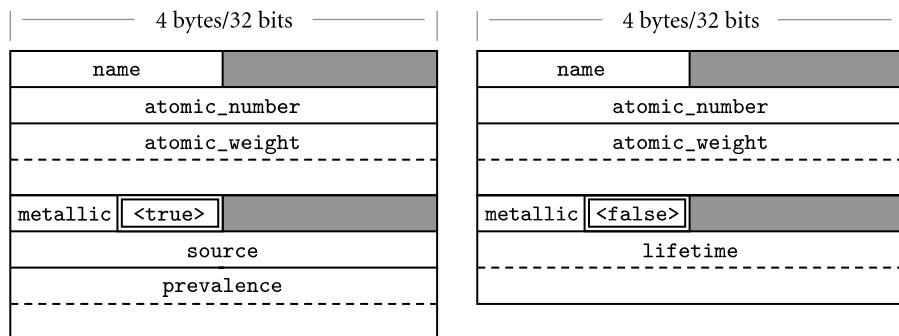
Pascal's principal contribution to union types (retained by Modula and Ada) was to integrate them with records. This was an important contribution, because the need for alternative types seldom arises anywhere else. In our running example, we use the same field-name syntax to access both the `atomic_weight` and `lifetime` fields of an `element`, despite the fact that the former is present

#### EXAMPLE 7.44

Fortran equivalence statement

#### EXAMPLE 7.45

Mixing structs and unions in C



**Figure 7.4** Likely memory layouts for element variants. The value of the `naturally_occuring` field (shown here with a double border) determines which of the interpretations of the remaining space is valid. Type `string_ptr` is assumed to be represented by a (four-byte) pointer to dynamically allocated storage.

in every `element`, while the latter is present only in those that are not naturally occurring. Without the integration of records and unions, the notation is less convenient. Here's what it looks like in C:

```
struct element {
 char name[2];
 int atomic_number;
 double atomic_weight;
 _Bool metallic;
 _Bool naturally_occuring;
 union {
 struct {
 char *source;
 double prevalence;
 } natural_info;
 double lifetime;
 } extra_fields;
} copper;
```

Because the `union` is not a part of the `struct`, we have to introduce two extra levels of naming. The third field is still `copper.atomic_weight`, but the `source` field must be accessed as `copper.extra_fields.natural_info.source`. A similar situation occurs in ML, in which datatypes can be used for unions, but the notation is not integrated with records (Exercise (C) 7.33). ■

### Safety

#### EXAMPLE 7.46

Breaking type safety with equivalence

One of the principal problems with equivalence statements is that they provide no built-in means of determining which of the equivalence-ed objects is currently valid: the program must keep track. Mistakes in which the programmer writes to one object and then reads from the other are relatively common:

```
r = 3.0
...
print '(I10)', i
```

Here the print statement, which attempts to output i as a 10-digit integer, will (in most implementations) take its bits from the floating-point representation of 3.0. This is almost certainly a mistake, but one that the language implementation will not catch. ■

**EXAMPLE 7.47**

Union conformity in Algol 68

Fortran equivalence statements introduce an extreme case of aliases: not only are there two names for the “same thing” (in this case the same block of storage), but the types associated with those names are different. To address this potential source of bugs, the Algol 68 designers required that the language implementation track union-ed types at run time:

```
union (int, real, bool) uirb
 # uirb can be an integer, a floating-point number, or a Boolean #
...
uirb := 1 # uirb is now an integer #
...
uirb := 3.14 # uirb is now a floating-point number #
```

To use the value stored inside a union, the programmer must employ a special form of case statement (called a *conformity clause* in Algol 68) that determines which type is currently valid:

```
case uirb in
 (int i) : print(i),
 (real r) : print(r),
 (bool b) : print(b)
esac
```

The labels on the arms of the case statement provide names for the “deunified” values. A similar tagcase construct can be found in Clu. ■

To enforce correct usage of union types in Algol 68, the language implementation must maintain a hidden variable for every union object that indicates which type is currently valid. When an object of a union type is assigned a value, the hidden variable is also set, to indicate the type of the value just assigned. When execution encounters a conformity clause, the hidden field is inspected to determine which arm to execute.

In effect, the tag field of a Pascal variant record is an explicit representation of the hidden variable required in an Algol 68 union. Our integer/floating-point/Boolean example could be written as follows in Pascal.

```
type tag = (is_int, is_real, is_bool);
var uirb : record
 case which : tag of
 is_int : (i : integer);
 is_real : (r : real);
 is_bool : (b : Boolean)
 end;
```

**EXAMPLE 7.48**

Tagged variant record in Pascal

**EXAMPLE 7.49**

Breaking type safety with variant records

Unfortunately, while the hidden tag of an Algol 68 union can only be changed implicitly, by assigning a value of a different type to the union as a whole, the tag of a Pascal variant record can be changed by an ordinary assignment statement. The compiler can generate code to verify that a field in variant  $v$  is never accessed unless the value of the tag indicates that  $v$  is currently valid, but this is not enough to guarantee type safety. It can catch errors of the form

```
uirb.which := is_real;
uirb.r := 3.0;
...
writeln(uirb.i); (* dynamic semantic error *)
```

but it cannot catch the following.

```
uirb.which := is_real;
uirb.r := 3.0;
uirb.which := is_int;
...
writeln(uirb.i); (* no intervening assignment to i *)
writeln(uirb.i); (* ouch! *)
```

Any Pascal implementation will accept this code, but the output is likely to be erroneous, just as it was in Fortran. ■

Semantically speaking, changing the tag of a Pascal variant record should make the remaining fields of the variant *uninitialized*. It is possible, by adding hidden fields, to flag them as such and generate a semantic error message on any subsequent access, but the code to do so is expensive [FL80], and outlaws programs that, while arguably erroneous, are permitted by the language definition (Exercise 7.12).

The situation in Pascal is actually worse than our example so far might imply. Additional insecurity stems from the fact that Pascal's tag fields are *optional*. We could eliminate the `which` field of our `uirb` record:

```
var uirb : record
 case tag of
 is_int : (i : integer);
 is_real : (r : real);
 is_bool : (b : Boolean)
 end;

 ...
uirb.r := 3.0;
writeln(uirb.i); (* no intervening assignment to i *)
writeln(uirb.i); (* ouch! *)
```

Now the language implementation is not required to devote any space to either an explicit or hidden tag, but even the limited form of checking (make sure the tag has an appropriate value when a field of a variant is accessed) is no longer possible (but see Exercise 7.13). Variant records with tags (explicit or hidden) are known as *discriminated unions*. Variant records without tags are known as *nondiscriminated unions*. ■

**EXAMPLE 7.50**

Untagged variants in Pascal

The degree of type safety provided is arguably the most important dimension of variation among the variant records and union types of modern languages. Though designed after Algol 68 (and borrowing its union terminology), the union types of C are semantically closer to Fortran's equivalence statements. Their fields share space, but nothing prevents the programmer from using them in inappropriate ways. By contrast, the variant records of Ada are syntactically similar to those of Pascal, but are as type-safe as the unions of Algol 68. Concerned at the lack of type safety in Pascal and Modula-2, and reluctant to introduce the complexity of Ada's rules, the designers of Modula-3 chose to eliminate variant records from the language entirely. They note [Har92, p. 110] that much of the same effect can be obtained via object types and subtypes. The designers of Java and C#, likewise, dropped the unions of C and C++.

### **Variants in Ada**

#### **EXAMPLE 7.51**

Ada variants and tags  
(discriminants)

Ada variant records must always have a tag (called the *discriminant* in Ada). Moreover, the tag can never be changed without simultaneously assigning values to all of the fields of the corresponding variant. The assignment can occur either via whole-record assignment (e.g., A := B, where A and B are variant records) or via assignment of an aggregate (e.g., A := {which => is\_real, r => pi});). In addition to appearing as a field within the record, the discriminant of a variant record in Ada must also appear in the header of the record's declaration:

```
type element (naturally_occuring : Boolean := true) is record
 name : string (1..2);
 atomic_number : integer;
 atomic_weight : real;
 metallic : Boolean;
 case naturally_occuring is
 when true =>
 source : string_ptr;
 prevalence : real;
 when false =>
 lifetime : real;
 end case;
end record;
```

Here we have not only declared the discriminant of the record in its header, we have also specified a default value for it. A declaration of a variable of type `element` has the option of accepting this default value:

```
copper : element;
```

or overriding it:

```
plutonium : element (false);
neptunium : element (naturally_occuring => false);
-- alternative syntax
```

If the type declaration for `element` did not specify a default value for `naturally_occuring`, then all variables of type `element` would have to pro-

vide a value. These rules guarantee that the tag field of a variant record is never uninitialized.

An Ada record variable whose declaration specifies a value for the discriminant is said to be *constrained*. Its tag field can never be changed by a subsequent assignment. This immutability means that the compiler can allocate just enough space to hold the specified variant; this space may in some cases be significantly smaller than would be required for other variants. A variable whose declaration does not provide an initial value for the discriminant is said to be *unconstrained*. Its tag will be initialized to the value in the type declaration, but may be changed by later (whole-record) assignments, so the space that the record occupies must be large enough to hold any possible variant.

An Ada subtype definition can also constrain the discriminant(s) of its parent type:

```
subtype natural_element is element (true);
```

Variables of type `natural_element` will all be constrained; their `naturally_occuring` field cannot be changed. Because `natural_element` is a subtype, rather than a derived type, values of type `element` and `natural_element` are compatible with each other, though a run-time semantic check will usually be required to assign the former into the latter.

Ada uses record discriminants not only for variant tags, but in general for any value that affects the size of a record. Here is an example that uses a discriminant to specify the length of an array:

#### EXAMPLE 7.52

A discriminated subtype in Ada

#### EXAMPLE 7.53

Discriminated array in Ada

### DESIGN & IMPLEMENTATION

#### The placement of variant fields

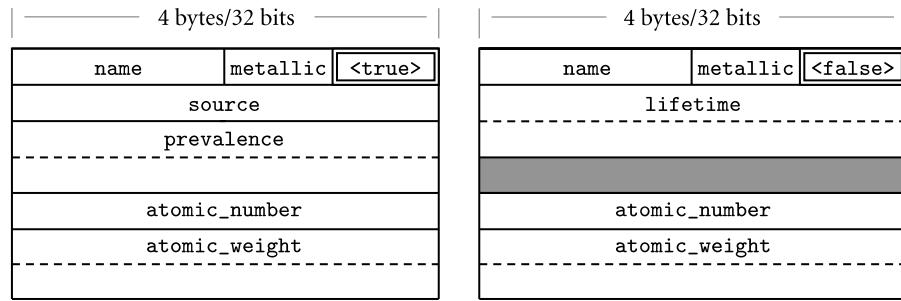
To facilitate space-saving in constrained variant records, Ada requires that all variant parts of a record appear at the end. This rule ensures that every field has a constant offset from the beginning of the record, with no holes (in any variant) other than those required for alignment. When a constrained variant record is elaborated, the Ada run-time system need only allocate sufficient space to hold the specified variant, which is never allowed to change. Pascal has a similar rule, designed for a similar purpose. When a variant record is allocated from the heap in Pascal (via the built-in `new` operator), the programmer has the option of specifying case labels for the variant portions of the record. A record so allocated is never allowed to change to a different variant, so the implementation can allocate precisely the right amount of space.

Modula-2, which does not provide `new` as a built-in operation, eliminates the ordering restriction on variants. All variables of a variant record type must be large enough to hold any variant. The usual implementation assigns a fixed offset to every field, with holes following small internal variants as necessary (see Figure 7.5 and Exercise 7.14).

```

TYPE element = RECORD
 name : ARRAY [1..2] OF CHAR;
 metallic : BOOLEAN;
 CASE naturally_occuring : BOOLEAN OF
 TRUE :
 source : string_ptr;
 prevalence : REAL;
 | FALSE :
 lifetime : REAL;
 END;
 atomic_number : INTEGER;
 atomic_weight : REAL;
END;

```



**Figure 7.5** Likely memory layout for a variant record in Modula-2. Here the variant portion of the record is not required to lie at the end. Every field has a fixed offset from the beginning of the record, with internal holes as necessary following small-size variants.

```

type element_array is array (integer range <>) of element;
type alloy (num_components : integer) is record
 name : string (1..30);
 components : element_array (1..num_components);
 tensile_strength : real;
end record;

```

The `<>` notation in the initial definition of `element_array` indicates that the bounds are not statically known. We will have more to say about dynamic arrays in Section 7.4.2. As with discriminants used for variant tags, the programmer must either specify a default value for the discriminant in the type declaration (we did not do so above) or else every declaration of a variable of the type must specify a value for the discriminant (in which case the variable is constrained, and the discriminant cannot be changed). ■

#### CHECK YOUR UNDERSTANDING

20. Discuss the significance of “holes” in records. Why do they arise? What problems do they cause?

21. What is *packing*? What are its advantages and disadvantages?
  22. Why might a compiler reorder the fields of a record? What problems might this cause?
  23. Why is it useful to integrate variants (unions) with records (structs)? Why not leave them as separate mechanisms, as they are in Algol 68 and C?
  24. Discuss the type safety problems that arise with variant records. How can these problems be addressed?
  25. What is a *tag (discriminant)*? How does it differ from an ordinary field?
  26. Summarize the rules that prevent access to inappropriate fields of a variant record in Ada.
  27. Why might one wish to *constrain* a variable, so that it can hold only one variant of a type?
- 

## 7.4 Arrays

Arrays are the most common and important composite data types. They have been a fundamental part of almost every high-level language, beginning with Fortran I. Unlike records, which group related fields of disparate types, arrays are usually homogeneous. Semantically, they can be thought of as a mapping from an index type to a component or element type. Some languages (e.g., Fortran) require that the index type be integer; many languages allow it to be any discrete type. Some languages (e.g., Fortran 77) require that the element type of an array be scalar. Most (including Fortran 90) allow any element type.

Some languages (notably scripting languages) allow nondiscrete index types. The resulting associative arrays must generally be implemented with hash tables, rather than with the more efficient contiguous allocation to be described in Section 7.4.3. Associative arrays in C++ are known as maps; they are supported by a standard library template. Java and C# have similar library classes. For the purposes of this chapter, we will assume that array indices are discrete (but see Section 13.4.3).

### 7.4.1 Syntax and Operations

Most languages refer to an element of an array by appending a subscript—delimited by parentheses or square brackets—to the name of the array. In Fortran and Ada, one says `A(3)`; in Pascal and C, one says `A[3]`. Since parentheses are generally used to delimit the arguments to a subroutine call, square bracket subscript notation has the advantage of distinguishing between the two. The difference in notation makes a program easier to compile and, arguably, easier to

read. Fortran's use of parentheses for arrays stems from the absence of square bracket characters on IBM keypunch machines, which at one time were widely used to enter Fortran programs. Ada's use of parentheses represents a deliberate decision on the part of the language designers to embrace notational ambiguity for functions and arrays. If we think of an array as a mapping from the index type to the element type, it makes perfectly good sense to use the same notation used for functions. In some cases, a programmer may even choose to change from an array to a function-based implementation of a mapping, or vice versa (Exercise 7.15).

### **Declarations**

#### **EXAMPLE 7.54**

##### Array declarations

```
char upper[26];
```

In Fortran:

```
character, dimension (1:26) :: upper
character (26) upper ! shorthand notation
```

In C, the lower bound of an index range is always zero: the indices of an  $n$ -element array are  $0 \dots n - 1$ . In Fortran, the lower bound of the index range is one by default. Fortran 90 allows a different lower bound to be specified if desired, using the notation shown in the first of the two declarations above.

In other languages, arrays are declared with an array constructor. In Pascal:

```
var upper : array ['a'...'z'] of char;
```

In Ada:

```
upper : array (character range 'a'...'z') of character;
```

#### **EXAMPLE 7.55**

##### Multidimensional arrays

Most languages make it easy to declare multidimensional arrays:

```
matrix : array (1..10, 1..10) of real; -- Ada
```

```
real, dimension (10,10) :: matrix ! Fortran
```

In some languages (e.g., Pascal, Ada, and Modula-3), one can also declare a multidimensional array by using the array constructor more than once in the same declaration. In Modula-3,

```
VAR matrix : ARRAY [1..10], [1..10] OF REAL;
```

is syntactic sugar for

```
VAR matrix : ARRAY [1..10] OF ARRAY [1..10] OF REAL;
```

and `matrix[3, 4]` is syntactic sugar for `matrix[3][4]`. Similar equivalences hold in Pascal.

**EXAMPLE 7.56**

Multidimensional v. built-up arrays

In Ada, by contrast,

```
matrix : array (1..10, 1..10) of real;
```

is not the same as

```
matrix : array (1..10) of array (1..10) of real;
```

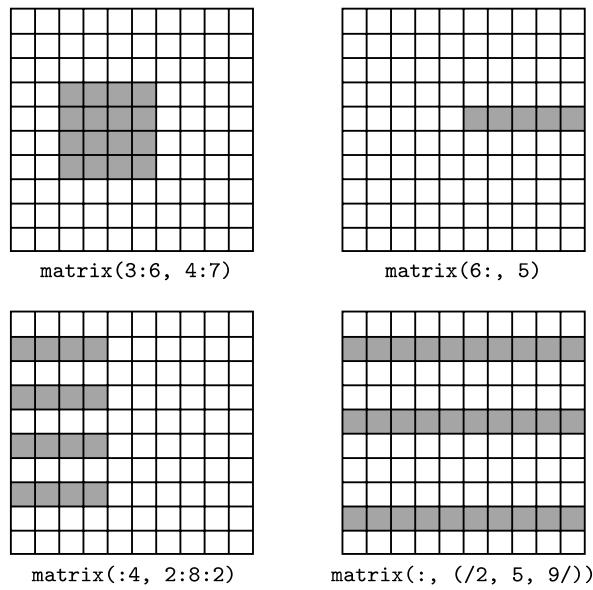
The former is a two-dimensional array, while the latter is an array of one-dimensional arrays. With the former declaration, we can access individual real numbers as `matrix(3, 4)`; with the latter we must say `matrix(3)(4)`. The two-dimensional array is arguably more elegant, but the array of arrays supports additional operations: it allows us to name the rows of `matrix` individually

**DESIGN & IMPLEMENTATION****Is [] an operator?**

The definition of associative arrays in C++ leverages the ability to overload square brackets (`[]`), which C++ treats as an operator. C#, like C++, provides extensive facilities for operator overloading, but it does not use these facilities to support associative arrays. Instead, the language provides a special *indexer* mechanism, with its own unique syntax:

```
class directory {
 Hashtable table; // from standard library
 ...
 public directory() { // constructor
 table = new Hashtable();
 }
 ...
 public string this[string name] { // indexer method
 get {
 return (string) table.get_Item(name);
 }
 set {
 table.Add(name, value); // value is implicitly
 } // a parameter of set
 }
 ...
 directory d = new directory();
 ...
 d["Jane Doe"] = "234-5678";
 Console.WriteLine(d["Jane Doe"]);
}
```

Why the difference? In C++, operator `[]` can return a reference (an explicit lvalue—see Section 8.3.1), which can be used on either side of an assignment. C# has no comparable notion of reference, so it needs separate methods to `get` and `set` the value of `d["Jane Doe"]`.



**Figure 7.6** Array slices (sections) in Fortran 90. Much like the values in the header of an enumeration-controlled loop (Section 6.5.1),  $a:b:c$  in a subscript indicates positions  $a, a+c, a+2c, \dots$  through  $b$ . If  $a$  or  $b$  is omitted, the corresponding bound of the array is assumed. If  $c$  is omitted, 1 is assumed. It is even possible to use negative values of  $c$  in order to select positions in reverse order. The slashes in the second subscript of the lower-right example delimit an explicit list of positions.

(`matrix(3)` is a 10-element, single-dimensional array), and it allows us to take *slices*, as discussed below. ■

In C, one must also declare an array of arrays, and use two-subscript notation, but C's integration of pointers and arrays (to be discussed in Section 7.7.1) means that slices are not supported.

```
double matrix[10][10];
```

Given this definition, `matrix[3][4]` denotes an individual element of the array, but `matrix[3]` denotes a *reference*, either to the third row of the array or to the first element of that row, depending on context. ■

### Slices and Array Operations

---

**EXAMPLE 7.58**

Array slice operations

A *slice* or *section* is a rectangular portion of an array. Fortran 90 provides extensive facilities for slicing, as do many scripting languages, including Perl, Python, Ruby, and R. Figure 7.6 illustrates some of the possibilities in Fortran 90, using the declaration of `matrix` shown above. Ada provides more limited support: a slice is simply a contiguous range of elements in a one-dimensional array. ■

In most languages, the only operations permitted on an array are selection of an element (which can then be used for whatever operations are valid on its type), and assignment. A few languages (e.g., Ada and Fortran 90) allow arrays

to be compared for equality. Ada allows one-dimensional arrays whose elements are discrete to be compared for *lexicographic ordering*:  $A < B$  if the first element of  $A$  that is not equal to the corresponding element of  $B$  is less than that corresponding element. Ada also allows the built-in logical operators (`or`, `and`, `xor`) to be applied to Boolean arrays.

Fortran 90 has a very rich set of *array operations*: built-in operations that take entire arrays as arguments. Because Fortran uses structural type equivalence, the operands of an array operator need only have the same element type and shape. In particular, slices of the same shape can be intermixed in array operations, even if the arrays from which they were sliced have very different shapes. Any of the built-in arithmetic operators will take arrays as operands; the result is an array, of the same shape as the operands, whose elements are the result of applying the operator to corresponding elements. As a simple example,  $A + B$  is an array each of whose elements is the sum of the corresponding elements of  $A$  and  $B$ . Fortran 90 also provides a huge collection of *intrinsic*, or built-in functions. More than 60 of these (including logic and bit manipulation, trigonometry, logs and exponents, type conversion, and string manipulation) are defined on scalars but will also perform their operation element-wise if passed arrays as arguments. The function `tan(A)`, for example, returns an array consisting of the tangents of the elements of  $A$ . Many additional intrinsic functions are defined solely on arrays. These include searching and summarization, transposition, and reshaping and subscript permutation.

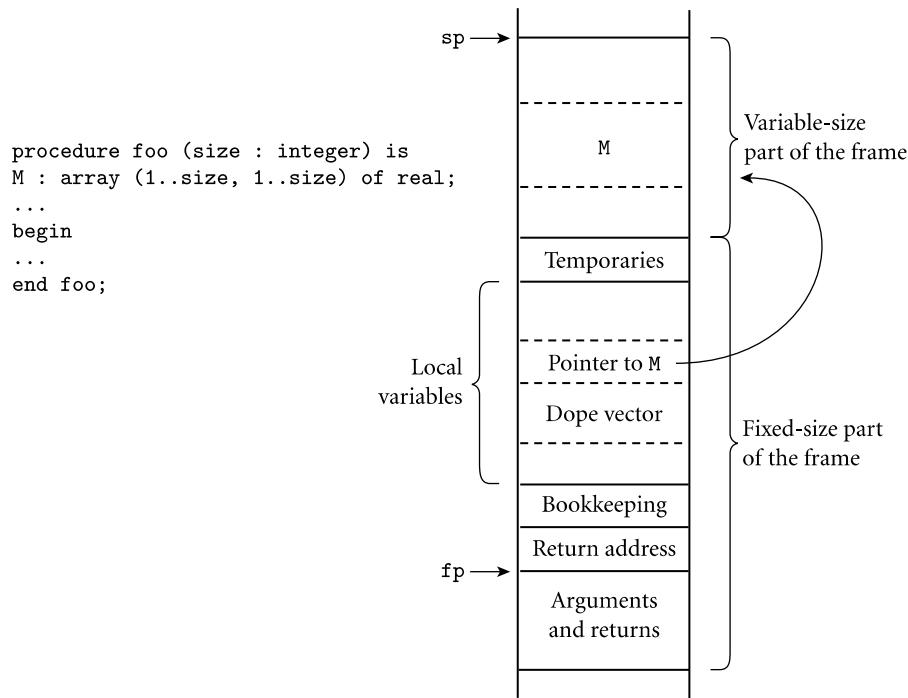
An equally rich set of array operations can be found in APL, an array manipulation language developed by Iverson and others in the early to mid-1960s.<sup>5</sup> APL was designed primarily as a terse mathematical notation for array manipulations. It employs an enormous character set that makes it difficult to use with conventional keyboards. Its variables are all arrays, and many of the special characters denote array operations. APL implementations are designed for interpreted, interactive use. They are best suited to “quick and dirty” solutions of mathematical problems. The combination of very powerful operators with very terse notation makes APL programs notoriously difficult to read and understand. The J notation, a successor to APL, uses a conventional character set.

### 7.4.2 Dimensions, Bounds, and Allocation

In all of the examples in the previous subsection, the number of dimensions and bounds of each array (what Fortran calls its *shape*) were specified in the declaration. This need not be the case. And even when the shape of an array is specified, it may depend in some languages on values that are not known at compile time.

---

**5** Kenneth Iverson (1920–2004), a Canadian mathematician, joined the faculty at Harvard University in 1954, where he conceived APL as a notation for describing mathematical algorithms. He moved to IBM in 1960, where he helped develop the notation into a practical programming language. He was named an IBM Fellow in 1970, and received the ACM Turing Award in 1979.



**Figure 7.7** Allocation in Ada of local arrays whose shape is bound at elaboration time. Here *M* is a square two-dimensional array whose width is determined by a parameter passed to *foo* at run time. The compiler arranges for a pointer to *M* to reside at a static offset from the frame pointer. *M* cannot be placed among the other local variables because it would prevent those higher in the frame from having static offsets. Additional variable-size arrays are easily accommodated. The purpose of the *dope vector* field is explained in Section 7.4.3.

The time at which the shape of an array is bound has a major impact on how storage for the array is managed. At least five cases arise.

*global lifetime, static shape:* If the shape of an array is known at compile time, and if the array can exist throughout the execution of the program, then the compiler can allocate space for the array in static global memory.

*local lifetime, static shape:* If the shape of the array is known at compile time, but the array should not exist throughout the execution of the program (generally because it is a local variable of a potentially recursive subroutine), then space can be allocated in the subroutine's stack frame at run time.

*local lifetime, shape bound at elaboration time:* In some languages (e.g., Ada and C99), the shape of an array may not be known until elaboration time. In this case it is still possible to place the space for the array in the stack frame of its subroutine, but an extra level of indirection is required (see Figure 7.7).

In order to ensure that every local object can be found using a known offset from the frame pointer, we divide the stack frame into a *fixed-size part* and a *variable-size part*. An object whose size is statically known goes in the fixed-

#### EXAMPLE 7.59

Stack allocation of elaborated arrays

size part. An object whose size is not known until elaboration time goes in the variable-size part, and a pointer to it goes in the fixed-size part. (We shall see in Section 7.4.3 that the pointer must be augmented with a descriptor, or *dope vector*, that specifies any bounds that were not known at compile time.) If the elaboration of the array is buried in a nested block, the compiler delays allocating space (i.e., changing the stack pointer) until the block is entered. It still allocates space for the pointer among the local variables when the subroutine itself is entered.

**EXAMPLE 7.60**

Stack allocation of new arrays

*arbitrary lifetime, shape bound at elaboration time:* In Java and C#, every array variable is a reference to an object in the object-oriented sense of the word. The declaration `int [] A` does not allocate space; it simply creates a reference. To make the reference refer to something, the programmer must either explicitly allocate a new object from the heap (`A = new int [size]`) or assign a reference from another array (`A = B`), which already holds a reference to an object in the heap. In either case, the size of an array, once allocated, never changes.

*arbitrary lifetime, dynamic shape:* If the size of an array can change as the result of executable statements, then allocation in the stack frame will not suffice, because the space at both ends of an array might be in use for something else when the array needs to grow. To allow the size to change, an array must generally be allocated from the heap. (A pointer to the array still resides in the fixed-size portion of the stack frame.) In most cases, increasing the size will require that we allocate a larger block, copy any data that is to be retained from the old block to the new, and then deallocate the old.

Arrays of static shape are heavily used by many kinds of programs. Arrays whose shape is not known until elaboration time are also very common, particularly in numerical software. Many scientific programs rely on numerical libraries for linear algebra and the manipulation of systems of equations. Since different programs use arrays of different shapes, the subroutines in these libraries need to be able to take arguments whose size is not known at compile time.

**Conformant Arrays****EXAMPLE 7.61**

Conformant array parameters

Early versions of Pascal required the shape of all arrays to be specified statically. Standard Pascal relaxes this requirement by allowing array parameters to have bounds that are symbolic names rather than constants. It calls these parameters *conformant arrays*:

```
function DotProduct(A, B : array [lower..upper : integer] of real)
 : real;
var i : integer;
 rtn : real;
begin
 rtn := 0;
 for i := lower to upper do rtn := rtn + A[i] * B[i];
 DotProduct := rtn
end;
```

Here `lower` and `upper` are initialized at the time of call, providing `DotProduct` with the information it needs to understand the shape of `A` and `B`. In effect, `lower` and `upper` are extra parameters of `DotProduct`. Conformant arrays can be passed either by value or by reference. C also supports dynamic-size array parameters, as a natural consequence of its merger of arrays and pointers (to be discussed in Section 7.7.1). C arrays are always passed by reference. ■

**EXAMPLE 7.62**

Local arrays of dynamic shape

Pascal does not allow a local variable to be an array of dynamic shape. Ada and C99 do. Among other things, local arrays can be declared to match the shape of dynamic-size array parameters, facilitating the implementation of algorithms that require “scratch space.” Figure 7.8 contains an Ada example. The program shown plays Conway’s game of Life [Gar70]. (Life is a *cellular automaton* meant to model biological populations. The patterns it produces can be amazingly complex and beautiful. Type “Conway Game of Life” into any search engine for a wealth of online examples.) The main routine allocates a local array the same size as the game board, which it uses to calculate successive generations. Note that much more efficient algorithms exist; we present this one because it is brief and clear.

The `<>` notation in the definition of `lifeboard` indicates that the bounds of the array are not statically known. Ada actually defines an array type to *have* no bounds. The type of any array with bounds is a *constrained subtype* of an array type with the same number of dimensions but unknown bounds. Bounds of a dynamic array can be obtained at run-time through use of the *array attributes* `'first` and `'last`. `A'first(1)` is the low bound of `A`’s first dimension; `A'last(2)` is the upper bound of its second dimension. The expression `A'range` is short for `A'first..A'last`. ■

**Dynamic Arrays****EXAMPLE 7.63**

Dynamic strings in Java and C#

Several languages, including Snobol, Icon, and Perl, allow strings—arrays of characters—to change size after elaboration time. Java and C# provide a similar capability (with a similar implementation), but describe the semantics differently: string variables in Java and C# are references to immutable string objects:

```
String s = "short";
...
s = s + " but sweet"; // + is the concatenation operator
```

Here the declaration `String s` introduces a string variable, which we initialize with a reference to the constant string `"short"`. In the subsequent assignment, `+` creates a new string containing the concatenation of the old `s` and the constant `" but sweet"`; `s` is then set to refer to this new string, rather than the old. Java and C# strings, by the way, are *not* the same as arrays of characters: strings are immutable, but elements of an array can be changed in place. ■

**EXAMPLE 7.64**

Elaborated arrays in Fortran 90

Dynamically resizable arrays (other than strings) appear in APL, Perl, and Common Lisp. They are also supported by the `vector`, `Vector`, and `ArrayList` classes of the C++, Java, and C# libraries, respectively. Fortran 90 allows specifi-

```

type presence is integer range 0..1;
type lifeboard is array (integer range <>, integer range <>) of presence;
 -- cell is 1 if occupied; 0 otherwise
 -- border row around the edge is permanently empty
unexpected : exception;
procedure life(B : in out lifeboard;
 generations : in integer) is
T : lifeboard(B'range(1), B'range(2));
 -- mimic the bounds of B
begin
 for i in 1..generations loop
 T := B; -- copy board, including empty borders
 for i in B'first(1)+1..B'last(1)-1 loop
 for j in B'first(2)+1..B'last(2)-1 loop
 case T(i-1, j-1) + T(i-1, j) + T(i-1, j+1)
 + T(i, j-1) + T(i, j+1)
 + T(i+1, j-1) + T(i+1, j) + T(i+1, j+1) is
 when 0 | 1 => B(i, j) := 0;
 -- die of loneliness
 when 2 => B(i, j) := T(i, j);
 -- no-op; survive if present
 when 3 => B(i, j) := 1;
 -- reproduce
 when 4..8 => B(i, j) := 0;
 -- die of overcrowding
 when others =>
 raise unexpected;
 end case;
 end loop;
 end loop;
 end loop;
end life;

```

**Figure 7.8** Dynamic local arrays in Ada.

cation of the bounds of an array to be delayed until after elaboration, but it does not allow those bounds to change once they have been defined:

```

real, dimension (:,:), allocatable :: mat
 ! mat is two-dimensional, but with unspecified bounds
...
allocate (mat (a:b, 0:m-1))
 ! first dimension has bounds a..b; second has bounds 0..m-1
...
deallocate (mat)
 ! implementation is now free to reclaim mat's space

```

A similar effect can be obtained in some languages through the use of pointers (see Exercise 7.18). ■

### 7.4.3 Memory Layout

Arrays in most language implementations are stored in contiguous locations in memory. In a one-dimensional array, the second element of the array is stored immediately after the first (subject to alignment constraints); the third is stored immediately after the second, and so forth. For arrays of records, it is common for each subsequent element to be aligned at an address appropriate for any type; small holes between consecutive records may result. On some machines, an implementation may even place holes between elements of built-in types. Some languages (e.g., Pascal) allow the programmer to specify that an array be packed. A packed array generally has no holes between elements, but access to its elements may be slow. A packed array of records may have holes *within* the records, unless they too are packed.

For multidimensional arrays, it still makes sense to put the first element of the array in the array's first memory location. But which element comes next? There are two reasonable answers, called *row-major* and *column-major* order. In row-major order, consecutive locations in memory hold elements that differ by one in the final subscript (except at the ends of rows).  $A[2, 4]$ , for example, is followed by  $A[2, 5]$ . In column-major order, consecutive locations hold elements that differ by one in the *initial* subscript:  $A[2, 4]$  is followed by  $A[3, 4]$ . These options are illustrated for two-dimensional arrays in Figure 7.9. The layouts for three or more dimensions are analogous. Fortran uses column-major order; most other languages use row-major order.<sup>6</sup> The advantage of row-major order is that it makes it easy to define a multidimensional array as an array of subarrays, as described in Section 7.4.1. With column-major order, the elements of the subarray would not be contiguous in memory. ■

The difference between row- and column-major layout can be important for programs that use nested loops to access all the elements of a large, multidimensional array. On modern machines the speed of such loops is often limited by memory system performance, which depends heavily on the effectiveness of caching (Section 5.1). Figure 7.9 shows the orientation of cache lines for row- and column-major layout of arrays. If a small array is accessed frequently, all or most of its elements are likely to remain in the cache, and the orientation of cache lines will not matter. For a large array, however, many of the accesses that occur during a full-array traversal are likely to result in cache misses, because the corresponding lines have been evicted from the cache (to make room for other things) since the last traversal. If array elements are accessed in order of consecutive addresses, then each miss will bring into the cache not only the desired element, but the next several elements as well. If elements are accessed across cache lines instead (i.e., along the rows of a Fortran array, or the columns

#### EXAMPLE 7.65

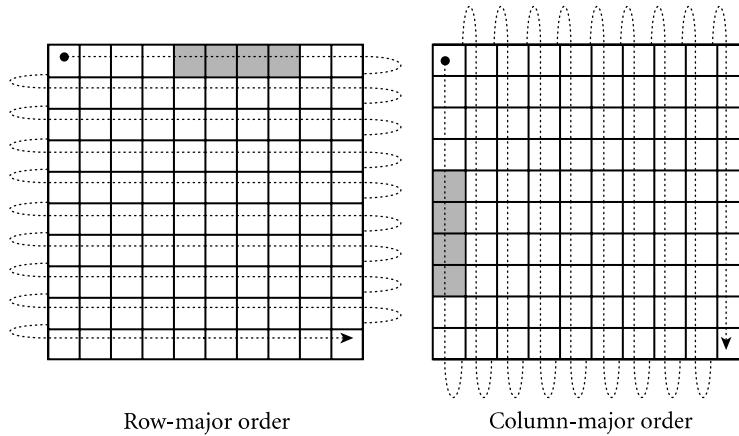
Row-major v.  
column-major array layout

#### EXAMPLE 7.66

Array layout and cache  
performance

---

<sup>6</sup> Correspondence with Frances Allen, an IBM Fellow and Fortran pioneer, suggests that column-major order was originally adopted in order to accommodate idiosyncrasies of the console debugger and instruction set of the IBM model 704 computer, on which the language was first implemented.



**Figure 7.9** Row- and column-major memory layout for two-dimensional arrays. In row-major order, the elements of a row are contiguous in memory; in column-major order, the elements of a column are contiguous. The second cache line of each array is shaded, on the assumption that each element is an eight-byte floating-point number, that cache lines are 32 bytes long (a common size), and that the array begins at a cache line boundary. If the array is indexed from  $A[0,0]$  to  $A[9,9]$ , then in the row-major case elements  $A[0,4]$  through  $A[0,7]$  share a cache line; in the column-major case elements  $A[4,0]$  through  $A[7,0]$  share a cache line.

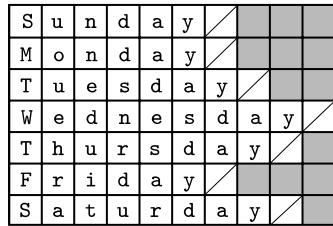
of an array in most other languages), then there is a good chance that almost every access will result in a cache miss, dramatically reducing the performance of the code. ■

### Row-Pointer Layout

Some languages employ an alternative to contiguous allocation for some arrays. Rather than require the rows of an array to be adjacent, they allow them to lie anywhere in memory, and create an auxiliary array of pointers to the rows. If the array has more than two dimensions, it may be allocated as an array of pointers to arrays of pointers to.... This *row-pointer* memory layout requires more space in most cases but has three potential advantages. First, it sometimes allows individual elements of the array to be accessed more quickly, especially on CISC machines with slow multiplication instructions (see the discussion of address calculations below). Second, it allows the rows to have different lengths, without devoting space to holes at the ends of the rows; the lack of holes may sometimes offset the increased space for pointers. Third, it allows a program to construct an array from preexisting rows (possibly scattered throughout memory) without copying. C, C++, and C# provide both contiguous and row-pointer organizations for multidimensional arrays. Technically speaking, the contiguous layout is a true multidimensional array, while the row-pointer layout is an array of pointers to arrays. Java uses the row-pointer layout for all arrays.

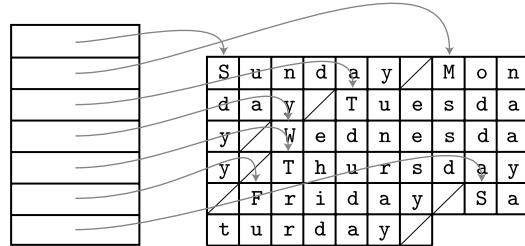
```
char days[] [10] = {
 "Sunday", "Monday", "Tuesday",
 "Wednesday", "Thursday",
 "Friday", "Saturday"
};

...
days[2][3] == 's'; /* in Tuesday */
```



```
char *days[] = {
 "Sunday", "Monday", "Tuesday",
 "Wednesday", "Thursday",
 "Friday", "Saturday"
};

...
days[2][3] == 's'; /* in Tuesday */
```



**Figure 7.10** Contiguous array allocation v. row pointers in C. The declaration on the left is a true two-dimensional array. The slashed boxes are NUL bytes; the shaded areas are holes. The declaration on the right is an array of pointers to arrays of characters. In both cases, we have omitted bounds in the declaration that can be deduced from the size of the initializer (aggregate). Both data structures permit individual characters to be accessed using double subscripts, but the memory layout (and corresponding address arithmetic) is quite different.

#### EXAMPLE 7.67

##### Contiguous v. row-pointer array layout

By far the most common use of the row-pointer layout in C is to represent arrays of strings. A typical example appears in Figure 7.10. In this example (representing the days of the week), the row-pointer memory layout consumes 57 bytes for the characters themselves (including a NUL byte at the end of each string), plus 28 bytes for pointers (assuming a 32-bit architecture), for a total of 85 bytes. The contiguous layout alternative devotes ten bytes to each day (room enough for Wednesday and its NUL byte), for a total of 70 bytes. The additional space required

#### DESIGN & IMPLEMENTATION

##### Array layout

The layout of arrays in memory, like the ordering of record fields, is intimately tied to tradeoffs in design and implementation. While column-major layout appears to offer no advantages on modern machines, its continued use in Fortran means that programmers must be aware of the underlying implementation in order to achieve good locality in nested loops. Row-pointer layout, likewise, has no performance advantage on modern machines (and a likely performance *penalty*, at least for numeric code), but it is a more natural fit for the “reference to object” data organization of languages like Java. Its impacts on space consumption and locality may be positive or negative, depending on the details of individual applications.

for the row-pointer organization comes to 21%. In other cases, row pointers may actually save space. A Java compiler written in C, for example, would probably use row pointers to store the character-string representations of the 51 Java keywords and wordlike literals. This data structure would use  $51 \times 4 = 204$  bytes for the pointers, plus 343 bytes for the keywords, for a total of 547 bytes (548 when aligned). Since the longest keyword (`synchronized`) requires 13 bytes (including space for the terminating NUL), a contiguous two-dimensional array would consume  $51 \times 13 = 663$  bytes (664 when aligned). In this case, row pointers save a little over 21%. ■

### Address Calculations

#### EXAMPLE 7.68

Indexing a contiguous array

For the usual contiguous layout of arrays, calculating the address of a particular element is somewhat complicated, but straightforward. Suppose a compiler is given the following declaration for a three-dimensional array.

$A : \text{array } [L_1 \dots U_1] \text{ of array } [L_2 \dots U_2] \text{ of array } [L_3 \dots U_3] \text{ of elem\_type;}$

Let us define constants for the sizes of the three dimensions:

$$S_3 = \text{size of elem\_type}$$

$$S_2 = (U_3 - L_3 + 1) \times S_3$$

$$S_1 = (U_2 - L_2 + 1) \times S_2$$

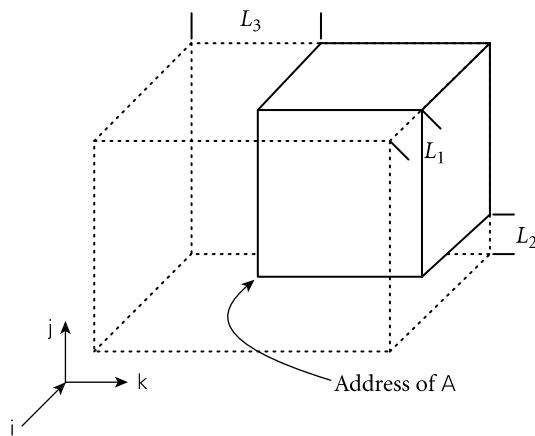
Here the size of a row ( $S_2$ ) is the size of an individual element ( $S_3$ ) times the number of elements in a row (assuming row-major layout). The size of a plane ( $S_1$ ) is the size of a row ( $S_2$ ) times the number of rows in a plane. The address of  $A[i, j, k]$  is then

$$\begin{aligned} &\text{address of } A \\ &+ (i - L_1) \times S_1 \\ &+ (j - L_2) \times S_2 \\ &+ (k - L_3) \times S_3 \end{aligned}$$

As written, this computation involves three multiplications and six additions/subtractions. We could compute the entire expression at run time, but in most cases a little rearrangement reveals that much of the computation can be performed at compile time. In particular, if the bounds of the array are known at compile time, then  $S_1$ ,  $S_2$ , and  $S_3$  are compile-time constants, and the subtractions of lower bounds can be distributed out of the parentheses:

$$\begin{aligned} &(i \times S_1) + (j \times S_2) + (k \times S_3) + \text{address of } A \\ &- [(L_1 \times S_1) + (L_2 \times S_2) + (L_3 \times S_3)] \end{aligned}$$

The bracketed expression in this formula is a compile-time constant (assuming the bounds of  $A$  are statically known). If  $A$  is a global variable, then the address of  $A$  is statically known as well, and can be incorporated in the bracketed expression.



**Figure 7.11** Virtual location of an array with nonzero lower bounds. By computing the constant portions of an array index at compile time, we effectively index into an array whose starting address is offset in memory but whose lower bounds are all zero.

If  $A$  is a local variable of a subroutine (with static shape), then the address of  $A$  can be decomposed into a static offset (included in the bracketed expression) plus the contents of the frame pointer at run time. We can think of the address of  $A$  plus the bracketed expression as calculating the location of an imaginary array whose  $[i, j, k]$ th element coincides with that of  $A$ , but whose lower bound in each dimension is zero. This imaginary array is illustrated in Figure 7.11. ■

If  $A$ 's elements are integers, and are allocated contiguously in memory, then the instruction sequence to load  $A[i, j, k]$  into a register looks something like this:

```
-- assume i is in r1, j is in r2, and k is in r3
1. r4 := r1 × S1
2. r5 := r2 × S2
3. r6 := &A - L1 × S1 - L2 × S2 - L3 × 4 -- one or two instructions
4. r6 := r6 + r4
5. r6 := r6 + r5
6. r7 := *r6[r3] -- load
```

We have assumed that the hardware provides an indexed addressing mode, and that it scales its indexing by the size of the quantity loaded (in this case a four-byte integer). ■

#### EXAMPLE 7.70

Static and dynamic portions of an array index

If  $i$ ,  $j$ , and/or  $k$  is known at compile time, then additional portions of the calculation of the address of  $A[i, j, k]$  will move from the dynamic to the static part of the formula shown above. If all of the subscripts are known, then the entire address can be calculated statically. Conversely, if any of the bounds of the array are not known at compile time, then portions of the calculation will move from the static to the dynamic part of the formula. For example, if  $L_1$  is not known until run time, but  $k$  is known to be 3 at compile time, then the

calculation becomes

$$(i \times S_1) + (j \times S_2) - (L_1 \times S_1) + \text{address of } A - [(L_2 \times S_2) + (L_3 \times S_3) - (3 \times S_3)]$$

Again, the bracketed part can be computed at compile time. If lower bounds are always restricted to zero, as they are in C, then they never contribute to run-time cost.

In all our examples, we have ignored the issue of dynamic semantic checks for out-of-bound subscripts. We explore the code for these in Exercise 7.22. In Section 15.5.2 we will consider code improvement techniques that can be used to eliminate many checks statically, particularly in enumeration-controlled loops.

The notion of “static part” and “dynamic part” of an address computation generalizes to more than just arrays. Suppose, for example, that  $V$  is a messy local array of records containing a nested, two-dimensional array in field  $M$ . The address of  $V[i].M[3, j]$  could be calculated as

$$\begin{aligned} & i \times S_1^V \\ & - L_1^V \times S_1^V \\ & + M\text{'s offset as a field} \\ & + (3 - L_1^V) \times S_1^V \\ & + j \times S_2^V \\ & - L_2^V \times S_2^V \\ & + \text{fp} \\ & + \text{offset of } V \text{ in frame} \end{aligned}$$

### EXAMPLE 7.71

#### Indexing complex structures

### DESIGN & IMPLEMENTATION

#### Lower bounds on array indices

In C, the lower bound of every array dimension is always zero. It is often assumed that the language designers adopted this convention in order to avoid subtracting lower bounds from indices at run time, thereby avoiding a potential source of inefficiency. As our discussion has shown, however, the compiler can avoid any run-time cost by translating to a virtual starting location. (The one exception to this statement occurs when the lower bound has a very large absolute value: if any index (scaled by element size) exceeds the maximum offset available with displacement mode addressing [typically  $2^{15}$  bytes on RISC machines], then subtraction may still be required at run time.)

A more likely explanation lies in the interoperability of arrays and pointers in C (Section 7.7.1): C's conventions allow the compiler to generate code for an index operation on a pointer without worrying about the lower bound of the array into which the pointer points. Interestingly, Fortran array dimensions have a default lower bound of 1; unless the programmer explicitly specifies a lower bound of 0, the compiler must always translate to a virtual starting location.

Here the calculations on the left must be performed at run time; the calculations on the right can be performed at compile time. (The notation for bounds and size places the name of the variable in a superscript and the dimension in a subscript:  $L_2^M$  is the lower bound of the second dimension of M.) ■

**EXAMPLE 7.72**

Pseudo-assembler for  
row-pointer array indexing

Address calculation for arrays that use row pointers is comparatively straightforward. Using our three-dimensional array A as an example, the expression  $A[i, j, k]$  is equivalent to  $(*(*A[i])[j])[k]$  or, in more Pascal-like notation,  $A[i]^j^k$ . The instruction sequence to load  $A[i, j, k]$  into a register looks something like this:

```
-- assume i is in r1, j is in r2, and k is in r3
1. r4 := &A -- one or two instructions
2. r4 := *r4[r1]
3. r4 := *r4[r2]
4. r7 := r4[r3]
```

Assuming that the loads at lines 2 and 3 hit in the cache, this code will be comparable in cost to the instruction sequence for contiguous allocation shown above (given load delays). If the intermediate loads miss in the cache, it will be slower. On a 1970s CISC machine, the balance would probably tip in favor of the row-pointer code: multiplies would be slower, and memory accesses would be faster. In any event (contiguous or row-pointer allocation, old or new machine), important code improvements will often be possible when several array references use the same subscript expression, or when array references are embedded in loops. ■

**Dope Vectors**

For every array, a compiler maintains dimension, bounds, and size information in the symbol table. For every record, it maintains the offset of every field. When the bounds and size of array dimensions are statically known, the compiler can look them up in the symbol table in order to compute the address of elements of the array. When the bounds and size are not statically known, the compiler must arrange for them to be available when the compiled program needs to compute an address at run time. The usual mechanism employs a *run-time descriptor*, or *dope vector* for the array.<sup>7</sup> Typically, a dope vector for an array of dynamic shape will contain the lower bound of each dimension and the size of

---

<sup>7</sup> The name “dope vector” presumably derives from the notion of “having the dope on (something),” a colloquial expression that originated in horse racing: advance knowledge that a horse has been drugged (“doped”) is of significant, if unethical, use in placing bets.

every dimension except the last (which will always be statically known). If the language implementation performs dynamic semantic checks for out-of-bounds subscripts in array references, then the dope vector will need to contain upper bounds as well. Given upper and lower bounds, the size information is redundant, but it is usually included anyway, to avoid computing it repeatedly at run time.

If some of the dimension bounds or sizes for an array are known at compile time, then they may be omitted from the dope vector. One might imagine, then, that the size of a dope vector would depend on the number of statically unknown quantities. More commonly, it depends only on the number of dimensions: the modest loss in space is offset by the comparative simplicity of always being able to find a given bound or size at the same offset within the dope vector for any array of the appropriate number of dimensions.

The dope vector for an array of dynamic shape is generally placed next to the pointer to the array in the fixed-size part of the stack frame. The contents of the dope vector are initialized at elaboration time, or whenever the array changes shape. If one fully dynamic array is assigned into a second and the two are of different shapes and sizes, then run-time code will not only need to deallocate the old heap space of the target array, allocate new space, and copy the data into it, but it will also need to copy information from the dope vector of the source array into the dope vector of the target.

In some languages a record may contain an array of dynamic shape. In order to arrange for every field to have a static offset from the beginning of the record, a compiler can treat the record much like the fixed-size portion of the stack frame, with a pointer to the array at a fixed offset in the record, and the data in the variable-size part of the current stack frame. The problem with this approach is that it abandons contiguous allocation for records. Among other things, a block copy routine can no longer be used to assign one record into another. An arguably preferable approach is to abandon fixed field offsets and create dope vectors for dynamic-size records, just as we do for dynamic-size arrays. The dope vector for a record lists the offsets of the record's fields. All of the actual data then go in the variable-size part of the stack frame or the heap (depending on whether sizes are known at elaboration time).

#### **CHECK YOUR UNDERSTANDING**

---

28. What is an array *slice*? For what purposes are slices useful?
29. Is there any significant difference between a two-dimensional array and an array of one-dimensional arrays?
30. What is the *shape* of an array?
31. Under what circumstances can an array declared within a subroutine be allocated in the stack? Under what circumstances must it be allocated in the heap?

32. What is a *conformant* array?
  33. Discuss the comparative advantages of *contiguous* and *row-pointer* layout for arrays.
  34. Explain the difference between *row-major* and *column-major* layout for contiguously allocated arrays. Why does a programmer need to know which layout the compiler uses? Why do most language designers consider row-major layout to be better?
  35. How much of the work of computing the address of an element of an array can be performed at compile time? How much must be performed at run time?
  36. What is a *dope vector*? What purpose does it serve?
- 

## 7.5 Strings

In many languages, a string is simply an array of characters. In other languages, strings have special status, with operations that are not available for arrays of other sorts. Particularly powerful string facilities are found in Snobol, Icon, and the various scripting languages.

As we saw in Section 6.5.3, mechanisms to search for patterns within strings are a key part of Icon's distinctive generator-based control flow. Icon has dozens of built-in string operators, functions, and generators, including sophisticated pattern-matching facilities based on regular expressions. Perl, Python, Ruby, and other scripting languages provide similar functionality, though none includes the full power of Icon's backtracking search. We will consider the string and pattern-matching facilities of scripting languages in more detail in Section 13.4.2. In the remainder of this section we focus on the role of strings in more traditional languages.

Almost all programming languages allow literal strings to be specified as a sequence of characters, usually enclosed in single or double quote marks. Many languages, including C and its descendants, distinguish between literal characters (usually delimited with single quotes) and literal strings (usually delimited with double quotes). Other languages (e.g., Pascal) make no distinction: a character is just a string of length one. Most languages also provide *escape sequences* that allow nonprinting characters and quote marks to appear inside of strings. In Pascal, for example, a quote mark is included in a string by doubling it: 'ab' 'cde' is a six-character string whose third character is a quote mark.

C99 and C++ provide a very rich set of escape sequences. An arbitrary character can be represented by a backslash followed by (a) 1–3 octal (base 8) digits, (b) an x and one or more hexadecimal (base 16) digits, (c) a u and exactly four

### EXAMPLE 7.73

Character escapes in C and C++

hexadecimal digits, or (d) a U and exactly eight hexadecimal digits. The \u notation is meant to capture the two-byte (16-bit) Unicode character set. The \U notation is for 32-bit “extended” characters. Many of the most common control characters also have single-character escape sequences, many of which have been adopted by other languages as well. For example, \n is a line feed; \t is a tab; \r is a carriage return; \\ is a backslash. C# omits the octal sequences of C99 and C++; Java also omits the 32-bit extended sequences.

The set of operations provided for strings is strongly tied to the implementation envisioned by the language designer(s). Several languages that do not in general allow arrays to change size dynamically do provide this flexibility for strings. The rationale is twofold. First, manipulation of variable-length strings is fundamental to a huge number of computer applications, and in some sense “deserves” special treatment. Second, the fact that strings are one-dimensional, have one-byte elements, and never contain references to anything else makes dynamic-size strings easier to implement than general dynamic arrays.

Some languages require that the length of a string-valued variable be bound no later than elaboration time, allowing the variable to be implemented as a contiguous array of characters in the current stack frame. Languages in this category include C, Pascal, and Ada. Pascal and Ada support a few string operations, including assignment and comparison for lexicographic ordering. C, on the other hand, provides only the ability to create a pointer to a string literal. Because of C’s unification of arrays and pointers, even assignment is not supported. Given the declaration `char *s`, the statement `s = "abc"` makes `s` point to the constant `"abc"` in static storage. If `s` is declared as an array, rather than a pointer (`char s[4]`), then the statement will trigger an error message from the compiler. To assign one array into another in C, the program must copy the elements individually.

Other languages allow the length of a string-valued variable to change over its lifetime, requiring that the variable be implemented as a block or chain of blocks in the heap. Languages in this category include Lisp, Icon, ML, Java, and C#. ML and Lisp provide strings as a built-in type. C++, Java, and C# provide them as predefined classes of object, in the formal, object-oriented sense. In all these languages a string variable is a *reference* to a string. Assigning a new value to such a variable makes it refer to a different object. Concatenation and other string operators implicitly create new objects. The space used by objects that are no longer reachable from any variable is reclaimed automatically.

## 7.6 Sets

A programming language set is an unordered collection of an arbitrary number of distinct values of a common type. Sets were introduced by Pascal, and are found in many more recent languages as well. They are a useful form of composite type for many applications. Pascal supports sets of any discrete type, and provides union, intersection, and difference operations:

### EXAMPLE 7.75

Set types

```

var A, B, C : set of char;
D, E : set of weekday;
...
A := B + C; (* union; A := {x | x is in B or x is in C} *)
A := B * C; (* intersection; A := {x | x is in B and x is in C} *)
A := B - C; (* difference; A := {x | x is in B and x is not in C} *)

```

The type from which elements of a set are drawn is known as the *base* or *universe* type. Icon supports sets of characters (called *csets*) but not sets of any other base type. As illustrated in Section 6.5.4, *csets* play an important role in Icon's search facilities. Ada does not provide a set constructor for types, but its generic facility can be used to define a set package (module) with functionality comparable to the sets of Pascal [IBFW91, pp. 242–244].

There are many ways to implement sets, including arrays, hash tables, and various forms of trees. The most common implementation employs a bit vector whose length (in bits) is the number of distinct values of the base type. A set of characters, for example (in a language that uses ASCII) would be 128 bits—16 bytes—in length. A one in the  $k$ th position in the bit vector indicates that the  $k$ th element of the base type is a member of the set; a zero indicates that it is not. Operations on bit-vector sets can make use of fast logical instructions on most machines. Union is bit-wise or; intersection is bit-wise and; difference is bit-wise not, followed by bit-wise and.

## DESIGN & IMPLEMENTATION

### Representing sets

Unfortunately, bit vectors do not work well for large base types: a set of integers, represented as a bit vector, would consume some 500 megabytes on a 32-bit machine. With 64-bit integers, a bit-vector set would consume more memory than is currently contained on all the computers in the world. Because of this problem, many languages (including early versions of Pascal, but not the ISO standard) limit sets to base types of fewer than some fixed number of members. Both 128 and 256 are common limits; they suffice to cover ASCII characters. A few languages (e.g., early versions of Modula-2) limit base types to the number of elements that can be represented by a one-word bit vector, but there is really no excuse for such a severe restriction. A language that permits sets with very large base types must employ an alternative implementation (e.g., a hash table). It will still be expensive to represent sets with enormous numbers of elements, but reasonably easy to represent sets with a modest number of elements drawn from a very large universe.

## 7.7 Pointers and Recursive Types

A recursive type is one whose objects may contain one or more references to other objects of the type. Most recursive types are records, since they need to contain something in addition to the reference, implying the existence of heterogeneous fields. Recursive types are used to build a wide variety of “linked” data structures, including lists and trees.

In languages like Lisp, ML, Clu, or Java, which use a reference model of variables, it is easy for a record of type `foo` to include a reference to another record of type `foo`: every variable (and hence every record field) *is* a reference anyway. In languages like C, Pascal, or Ada, which use a value model of variables, recursive types require the notion of a *pointer*: a variable (or field) whose value is a reference to some object. Pointers were first introduced in PL/I.

In some languages (e.g., Pascal, Ada 83, and Modula-3), pointers are restricted to point only to objects in the heap. The only way to create a new pointer value (without using variant records or casts to bypass the type system) is to call a built-in function that allocates a new object in the heap and returns a pointer to it. In other languages (e.g., PL/I, Algol 68, C, C++, and Ada 95), one can create a pointer to a nonheap object by using an “address of” operator. We will examine pointer operations and the ramifications of the reference and value models in more detail in the following subsection.

In any language that permits new objects to be allocated from the heap, the question arises: how and when is storage reclaimed for objects that are no longer needed? In short-lived programs it may be acceptable simply to leave the storage unused, but in most cases unused space must be reclaimed, to make room for other things. A program that fails to reclaim the space for objects that are no longer needed is said to “leak memory.” If such a program runs for an extended period of time, it may run out of space and crash.

Many languages, including C, C++, Pascal, and Modula-2, require the programmer to reclaim space explicitly. Other languages, including Modula-3, Java, C#, and all the functional and scripting languages, require the language imple-

### DESIGN & IMPLEMENTATION

#### Implementation of pointers

It is common for programmers (and even textbook writers) to equate pointers with addresses, but this is a mistake. A pointer is a high level concept: a reference to an object. An address is a low-level concept: the location of a word in memory. Pointers are often implemented as addresses, but not always. On a machine with a *segmented* memory architecture, a pointer may consist of a segment id and an offset within the segment. In a language that attempts to catch uses of dangling references, a pointer may contain both an address and an access key.

mentation to reclaim unused objects automatically. Explicit storage reclamation simplifies the language implementation, but raises the possibility that the programmer will forget to reclaim objects that are no longer live (thereby leaking memory) or will accidentally reclaim objects that are still in use (thereby creating *dangling references*). Automatic storage reclamation (otherwise known as *garbage collection*) dramatically simplifies the programmer's task, but raises the question of how the language implementation is to distinguish garbage from active objects. We will discuss these issues further in Sections 7.7.2 and 7.7.3.

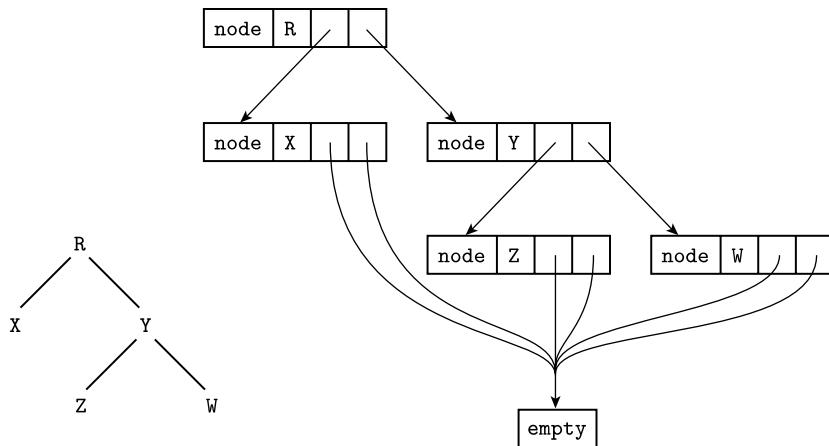
### 7.7.1 Syntax and Operations

Operations on pointers include allocation and deallocation of objects in the heap, dereferencing of pointers to access the objects to which they point, and assignment of one pointer into another. The behavior of these operations depends heavily on whether the language is functional or imperative, and on whether it employs a reference or value model for variables/names.

Functional languages generally employ a reference model for names (a purely functional language has no variables or assignments). Objects in a functional language tend to be allocated automatically as needed, with a structure determined by the language implementation. Most implementations of Lisp, for example, build lists out of two-pointer blocks called *cons* cells. Lisp's imperative features allow the programmer to modify *cons* cells explicitly, but this ability must be used with care: because of the reference model, a *cons* cell is commonly part of the object to which more than one variable refers; a change made through one variable will often change other variables as well.

Variables in an imperative language may use either a value or a reference model, or some combination of the two. In C, Pascal, or Ada, which employ a value model, the assignment  $A := B$  puts the value of  $B$  into  $A$ . If we want  $B$  to refer to an object, and we want  $A := B$  to make  $A$  refer to the object to which  $B$  refers, then  $A$  and  $B$  must be pointers. In Clu and Smalltalk, which employ a reference model, the assignment  $A := B$  always makes  $A$  refer to the same object to which  $B$  refers. A straightforward implementation would represent every variable as an address, but this would lead to very inefficient code for built-in types. A better and more common approach is to use addresses for variables that refer to *mutable* objects such as tree nodes, whose value can change, but to use actual values for variables that refer to *immutable* objects such as integers, real numbers, and characters. In other words, while every variable is semantically a reference, it does not matter whether a reference to the number 3 is implemented as the address of a 3 in memory or as the value 3 itself: since the value of “*the 3*” never changes, the two are indistinguishable.

Java charts an intermediate course, in which the usual implementation of the reference model is made explicit in the language semantics. Variables of built-in Java types (integers, floating-point numbers, characters, and Booleans) employ a value model; variables of user-defined types (strings, arrays, and other objects in



**Figure 7.12** Implementation of a tree in ML. The abstract (conceptual) tree is shown at the lower left.

the object-oriented sense of the word) employ a reference model. The assignment  $A := B$  in Java places the value of  $B$  into  $A$  if  $A$  and  $B$  are of built-in type; it makes  $A$  refer to the object to which  $B$  refers if  $A$  and  $B$  are of user-defined type. C# mirrors Java by default, but additional language features, explicitly labeled “unsafe,” allow systems programmers to use pointers when desired.

#### Reference Model

---

**EXAMPLE 7.76**  
Tree type in ML

Section ⑩ 7.2.4 explains how ML datatypes can be used to declare recursive types:

```
datatype chr_tree = empty | node of char * chr_tree * chr_tree;
```

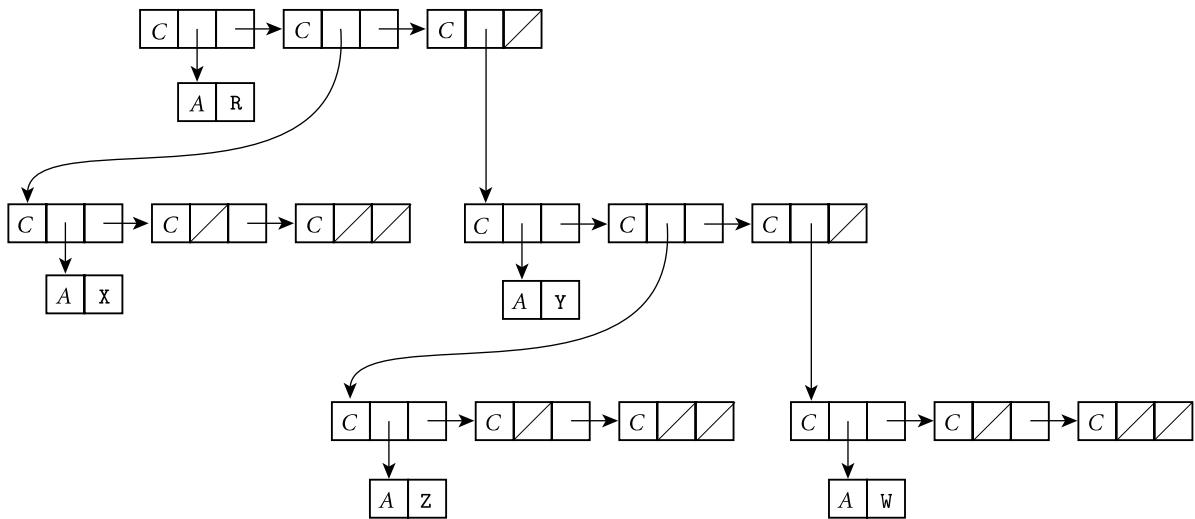
The `node` constructor of a `chr_tree` builds tuples containing a reference to a character and two references to `chr_trees`.

It is natural in ML to include a `chr_tree` within a `chr_tree` because every variable is a reference. The tree `node ("R", node ("X", empty, empty), node ("Y", node ("Z", empty, empty), node ("W", empty, empty)))` would most likely be represented in memory as shown in Figure 7.12. Each individual rectangle in the right-hand portion of this figure represents a block of storage allocated from the heap. In effect, the tree is a tuple (record) tagged to indicate that it is a `node`. This tuple in turn refers to two other tuples that are also tagged as `nodes`. At the fringe of the tree are tuples that are tagged as `empty`; these contain no further references. Because all `empty` tuples are the same, the implementation is free to use just one, and to have every reference point to it. ■

---

**EXAMPLE 7.77**  
Tree type in Lisp

In Lisp, which uses a reference model of variables but is not statically typed, our tree could be specified textually as `'(#\R (#\X ()()) (#\Y (#\Z ()())) (#\W ()()))`, and would be represented in memory as shown in Figure 7.13. The parentheses denote a *list*, which in Lisp consists of two references: one to



**Figure 7.13** Implementation of a tree in Lisp. A diagonal slash through a box indicates a `nil` pointer. The `C` and `A` tags serve to distinguish the two kinds of memory blocks: `cons` cells and blocks containing atoms.

the head of the list and one to the remainder of the list (which is itself a list). The prefix `#\` notation serves the same purpose as surrounding quotes in other languages. As we noted in Section 7.7.1, a Lisp list is almost always represented in memory by a `cons` cell containing two pointers. A binary tree can be represented as a three-element (three `cons` cell) list. The first cell represents the root; the second and third cells represent the left and right subtrees. Each heap block is tagged to indicate whether it is a `cons` cell or an *atom*. An atom is anything other than a `cons` cell—that is, an object of a built-in type (integer, real, character, string, etc.), or a user-defined structure (record) or array. The uniformity of Lisp lists (everything is a `cons` cell or an atom) makes it easy to write polymorphic functions, though without the static type checking of ML. ■

If one programs in a purely functional style in ML or in Lisp, the data structures created with recursive types turn out to be acyclic. New objects refer to old ones, but old ones never change, and thus never point to new ones. Circular structures can be defined only by using the imperative features of the languages. In ML, these features include an explicit notion of pointer, discussed briefly under “Value Model” below.

#### EXAMPLE 7.78

Mutually recursive types in ML

Even when writing in a functional style, one often finds a need for *types* that are *mutually recursive*. In a compiler, for example, it is likely that symbol table records and syntax tree nodes will need to refer to each other. A syntax tree node that represents a subroutine call will need to refer to the symbol table record that represents the subroutine. The symbol table record, for its part, will need to refer to the syntax tree node at the root of the subtree that represents the subroutine’s code. If types are declared one at a time, and if names must be declared before they can be used, then whichever mutually recursive type is declared first will be

unable to refer to the other. ML addresses this problem by allowing types to be declared together in a group:

```
datatype sym_tab_rec = variable of ...
| type of ...
| ...
| subroutine of {code : syn_tree_node, ...}
and syn_tree_node = expression of ...
| loop of ...
| ...
| subr_call of {subr : sym_tab_rec, ...};
```

Mutually recursive types of this sort are trivial in Lisp, since it is dynamically typed. (Common Lisp includes a notion of structures, but field types are not declared. In simpler Lisp dialects programmers use nested lists in which fields are merely positional conventions.)

### **Value Model**

#### **EXAMPLE 7.79**

Tree types in Pascal, Ada, and C

```
type chr_tree_ptr = ^chr_tree;
chr_tree = record
 left, right : chr_tree_ptr;
 val : char
end;
```

The Ada declaration is similar:

```
type chr_tree;
type chr_tree_ptr is access chr_tree;
type chr_tree is record
 left, right : chr_tree_ptr;
 val : character;
end record;
```

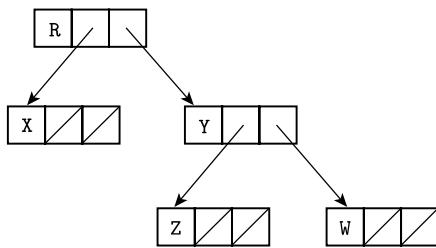
In C, the equivalent declaration<sup>8</sup> is as follows.

```
struct chr_tree {
 struct chr_tree *left, *right;
 char val;
};
```

As mentioned in Section 3.3.3, Pascal permits forward references in the declaration of pointer types, to support recursive types. Ada and C use incomplete type declarations instead.

---

**8** One of the peculiarities of the C type system is that `struct` tags are not exactly type names. In this example, the name of the type is the two-word phrase `struct chr_tree`. To obtain a one-word name, one can say `typedef struct chr_tree chr_tree_type`, or even `typedef struct chr_tree chr_tree: struct`. `tags` and `typedef` names have separate name spaces, so the same name can be used in each.



**Figure 7.14** Typical implementation of a tree in a language with explicit pointers. As in Figure 7.13, a diagonal slash through a box indicates a nil pointer.

#### EXAMPLE 7.80

Allocating heap nodes

No aggregate syntax is available for linked data structures in Pascal, Ada, or C; a tree must be constructed node by node. To allocate a new node from the heap, the programmer calls a built-in function. In Pascal:

```
new(my_ptr);
```

In Ada:

```
my_ptr := new chr_tree;
```

In C:

```
my_ptr = (struct chr_tree *) malloc(sizeof(struct chr_tree));
```

C's `malloc` is defined as a library function, not a built-in part of the language (though some compilers recognize and optimize it as a special case); hence the need to specify the size of the allocated object, and to cast the return value to the appropriate type. C++, Java, and C# replace `malloc` with a built-in `new`:

```
my_ptr = new chr_tree(arg_list);
```

In addition to “knowing” the size of the requested type, the C++/Java/C# `new` will automatically call any user-specified *constructor* (initialization) function, passing the specified argument list. In a similar but less flexible vein, Ada's `new` may specify an initial value for the allocated object:

```
my_ptr := new chr_tree'(null, null, 'X');
```

After we have allocated and linked together appropriate nodes in C, Pascal, or Ada, our tree example is likely to be implemented as shown in Figure 7.14. As in Lisp, a leaf is distinguished from an internal node simply by the fact that its two pointer fields are nil.

To access the object referred to by a pointer, most languages use an explicit dereferencing operator. In Pascal and Modula this operator takes the form of a postfix “up-arrow”:

```
my_ptr^.val := 'X';
```

In C it is a prefix star:

```
(*my_ptr).val = 'X';
```

#### EXAMPLE 7.81

Object-oriented allocation

of heap nodes

#### EXAMPLE 7.82

Pointer-based tree

#### EXAMPLE 7.83

Pointer dereferencing

Because pointers so often refer to records (structs), for which the prefix notation is awkward, C also provides a postfix “right-arrow” operator that plays the role of the “up-arrow dot” combination in Pascal:

```
my_ptr->val = 'X';
```

**EXAMPLE 7.84**

Implicit dereferencing in Ada

On the assumption that pointers almost always refer to records, Ada dispenses with dereferencing altogether. The same dot-based syntax can be used to access either a field of the record `foo` or a field of the record *pointed to* by `foo`, depending on the type of `foo`:

```
T : chr_tree;
P : chr_tree_ptr;
...
T.val := 'X';
P.val := 'Y';
```

In those cases in which one actually wants to name the *entire* object referred to by a pointer, Ada provides a special “pseudofield” called `all`:

```
T := P.all;
```

In essence, pointers in Ada are automatically dereferenced when needed. A more ambitious (and unfortunately rather confusing) form of automatic dereferencing can be found in Algol 68.

The imperative features of ML include an assignment statement, but this statement requires that the left-hand side be a pointer: its effect is to make the pointer refer to the object on the right-hand side. To access the object referred to by a pointer, one uses an exclamation point as a prefix dereferencing operator:

```
val p = ref 2; (* p is a pointer to 2 *)
...
p := 3; (* p now points to 3 *)
...
let val n = !p in ...
(* n is simply 3 *)
```

ML thus makes the distinction between l-values and r-values very explicit. Most Algol-family languages blur the distinction by implicitly dereferencing variables on the right-hand side of every assignment statement. Algol 68 and Ada blur the distinction further by dereferencing pointers automatically in certain circumstances.

The imperative features of Lisp do not include a dereferencing operator. Since every object has a self-evident type, and assignment is performed using a small set of built-in operators, there is never any ambiguity as to what is intended. Assignment in Common Lisp employs the `setf` operator (Scheme uses `set!`), rather than the more common `:=`. For example, if `foo` refers to a list, then `(cdr foo)` is the right-hand (“rest of list”) pointer of `foo`’s cons cell, and the assignment `(setf (cdr foo) foo)` makes this pointer refer back to `foo`, creating a one-cons-cell circular list.

**EXAMPLE 7.86**

Assignment in Lisp

**Pointers and Arrays in C****EXAMPLE 7.87**

Array names and pointers  
in C

```
int n;
int *a; /* pointer to integer */
int b[10]; /* array of 10 integers */
```

Now all of the following are valid.

1. `a = b;` /\* make a point to the initial element of b \*/
2. `n = a[3];`
3. `n = *(a+3);` /\* equivalent to previous line \*/
4. `n = b[3];`
5. `n = *(b+3);` /\* equivalent to previous line \*/

In most contexts, an unsubscripted array name in C is automatically converted to a pointer to the array's first element (the one with index zero), as shown here in line 1. (Line 5 embodies the same conversion.) Lines 3 and 5 illustrate *pointer arithmetic*: given a pointer to an element of an array, the addition of an integer  $k$  produces a pointer to the element  $k$  positions later in the array (earlier if  $k$  is negative). The prefix `*` is a pointer dereference operator. Pointer arithmetic is valid only within the bounds of a single array, but C compilers are not required to check this.

Remarkably, the subscript operator `[ ]` in C is actually defined in terms of pointer arithmetic: lines 2 and 4 are syntactic sugar for lines 3 and 5, respectively. More precisely,  $E1[E2]$ , for any expressions  $E1$  and  $E2$ , is defined to be  $(*(E1)+(E2))$ , which is of course the same as  $(*(E2)+(E1))$ . (Extra parentheses have been used in this definition to avoid any questions of precedence if  $E1$  and  $E2$  are complicated expressions.) Correctness requires only that one operand of `[ ]` have an array type and the other have an integral type. Thus `A[3]` is equivalent to `3[A]`, something that comes as a surprise to most programmers. ■

**EXAMPLE 7.88**

Pointer comparison and  
subtraction in C

In addition to allowing an integer to be added to a pointer, C allows pointers to be subtracted from one another or compared for ordering, provided that they refer to elements of the same array. The comparison  $p < q$ , for example, tests to see if  $p$  refers to an element closer to the beginning of the array than the one referred to by  $q$ . The expression  $p - q$  returns the number of array positions that separate the elements to which  $p$  and  $q$  refer. All arithmetic operations on pointers "scale" their results as appropriate, based on the size of the referenced objects. For multidimensional arrays with row-pointer layout,  $a[i][j]$  is equivalent to  $(*(a+i))[j]$  or  $*(a[i]+j)$  or  $*(*(a+i)+j)$ . ■

**EXAMPLE 7.89**

Pointer and array  
declarations in C

Despite the interoperability of pointers and arrays in C, programmers need to be aware that the two are not the same, particularly in the context of variable declarations, which need to allocate space when elaborated. The declaration of a pointer variable allocates space to hold a pointer, while the declaration of an array variable allocates space to hold the whole array. In the case of an array the declaration must specify a size for each dimension. Thus `int *a[n]`, when elab-

orated, will allocate space for  $n$  row pointers; `int a[n][m]` will allocate space for a two-dimensional array with contiguous layout.<sup>9</sup>

**EXAMPLE 7.90**

Arrays as parameters in C

When an array is included in the argument list of a function call, C passes a pointer to the first element of the array, not the array itself. For a one-dimensional array of integers, the corresponding formal parameter may be declared as `int a[]` or `int *a`. For a two-dimensional array of integers with row-pointer layout, the formal parameter may be declared as `int *a[]` or `int **a`. For a two-dimensional array with contiguous layout, the formal parameter may be declared as `int a[][]` or `int (*a)[m]`. The size of the first dimension is irrelevant; all that is passed is a pointer, and C performs no dynamic checks to ensure that references are within the bounds of the array.

In all cases, a declaration must allow the compiler (or human reader) to determine the size of the *elements* of an array or, equivalently, the size of the objects referred to by a pointer. Thus neither `int a[][]` nor `int (*a)[]` is a valid declaration: neither provides the compiler with the size information it needs to generate code for `a + i` or `a[i]`. (An exception: a variable declaration that includes initialization to an aggregate can omit size information if that information can be inferred from the contents of the aggregate.)

The built-in `sizeof` operator returns the size in bytes of an object or type. When given an array as argument it returns the size of the entire array. When given a pointer as argument it returns the size of the pointer itself. If `a` is an

**EXAMPLE 7.91**

Sizeof in C

**DESIGN & IMPLEMENTATION****Pointers and arrays**

Many C programs use pointers instead of subscripts to iterate over the elements of arrays. Before the development of modern optimizing compilers, pointer-based array traversal often served to eliminate redundant address calculations, thereby leading to faster code. With modern compilers, however, the opposite may be true: redundant address calculations can be identified as common subexpressions, and certain other code improvements are easier for indices than they are for pointers. In particular, as we shall see in Chapter 15, pointers make it significantly more difficult for the code improver to determine when two l-values may be *aliases* for one another.

Today the use of pointer arithmetic is mainly a matter of personal taste: some C programmers consider pointer-based algorithms to be more elegant than their array-based counterparts. Certainly the fact that arrays are passed as pointers makes it natural to write subroutines in the pointer style.

---

**9** To read declarations in C, it is helpful to follow the following rule: start at the name of the variable and work right as far as possible, subject to parentheses; then work left as far as possible; then jump out a level of parentheses and repeat. Thus `int *a[n]` means that `a` is an  $n$ -element array of pointers to integers, while `int (*a)[n]` means that `a` is a pointer to an  $n$ -element array of integers.

array, `sizeof(a) / sizeof(a[0])` returns the number of elements in the array. Similarly, if pointers occupy 4 bytes and double-precision floating point numbers occupy 8 bytes, then given

```
double *a; /* pointer to double */
double (*b)[10]; /* pointer to array of 10 doubles */
```

we have `sizeof(a) = sizeof(b) = 4`, `sizeof(*a) = sizeof(*b[0]) = 8`, and `sizeof(*b) = 80`. In most cases, `sizeof` can be evaluated at compile time; the principal exception occurs for variable-length arrays, whose shape is not known until elaboration time.

#### EXAMPLE 7.92

Multidimensional array parameters in C

Variable-length arrays are particularly useful in numeric code, where we can write general purpose library routines that manipulate arrays of arbitrary size:

```
double determinant(int rows, int cols, double M[rows][cols]) {
 ...
 val = M[i][j]; /* normal syntax */
```

It is possible but awkward to write functionally equivalent code in earlier versions of C:

```
double determinant(int rows, int cols, double *M) {
 ...
 val = *(M + (i * cols) + j); /* M[i][j] */
```

#### CHECK YOUR UNDERSTANDING

37. Name three languages that provide particularly extensive support for character strings.
38. Why might a language permit operations on strings that it does not provide for arrays?
39. What are the strengths and weaknesses of the bit-vector representation for sets? How else might sets be implemented?
40. Discuss the tradeoffs between pointers and the recursive types that arise naturally in a language with a reference model of variables.
41. Summarize the ways in which one dereferences a pointer in various programming languages.
42. What is the difference between a *pointer* and an *address*?
43. Discuss the advantages and disadvantages of the interoperability of pointers and arrays in C.
44. Under what circumstances must the bounds of a C array be specified in its declaration?

### 7.7.2 Dangling References

In Section 3.2 we described three *storage classes* for objects: static, stack, and heap. Static objects remain live for the duration of the program. Stack objects are live for the duration of the subroutine in which they are declared. Heap objects have a less well-defined lifetime.

When an object is no longer live, a long-running program needs to reclaim the object's space. Stack objects are reclaimed automatically as part of the subroutine calling sequence. How are heap objects reclaimed? There are two alternatives. Languages like Pascal, C, and C++ require the programmer to reclaim an object explicitly. In Pascal:

```
dispose(my_ptr);
```

In C:

```
free(my_ptr);
```

In C++:

```
delete my_ptr;
```

C++ provides additional functionality: prior to reclaiming the space, it automatically calls any user-provided *destructor* function for the object. A destructor can reclaim space for subsidiary objects, remove the object from indices or tables, print messages, or perform any other operation appropriate at the end of the object's lifetime. ■

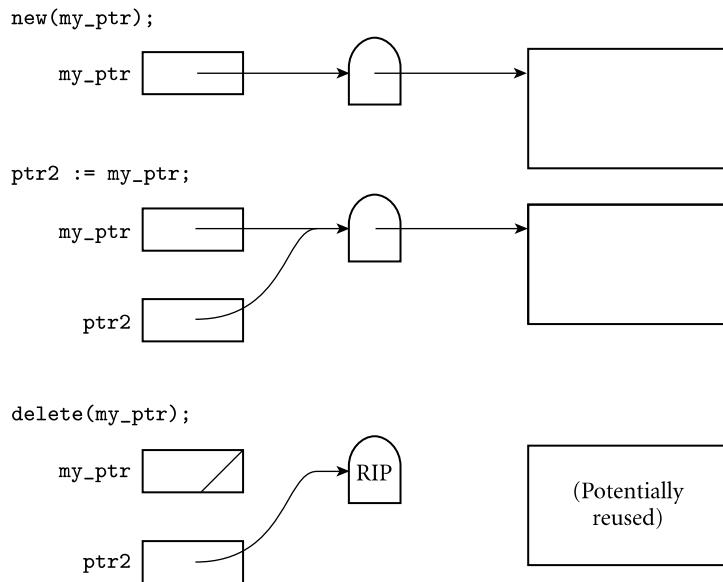
A *dangling reference* is a live pointer that no longer points to a valid object. In languages like Algol 68 or C, which allow the programmer to create pointers to stack objects, a dangling reference may be created when a subroutine returns while some pointer in a wider scope still refers to a local object of that subroutine. In a language with explicit reclamation of heap objects, a dangling reference is created whenever the programmer reclaims an object to which pointers still refer. (Note that while the `dispose` and `delete` operators of Pascal and C++ change their pointer argument to `nil`, this does not solve the problem, because *other* pointers may still refer to the same object.) Because a language implementation may reuse the space of reclaimed stack and heap objects, a program that uses a dangling reference may read or write bits in memory that are now part of some other object. It may even modify bits that are now part of the implementation's bookkeeping information, corrupting the structure of the stack or heap.

Algol 68 addresses the problem of dangling references to stack objects by forbidding a pointer from pointing to any object whose lifetime is briefer than that of the pointer itself. Unfortunately, this rule is difficult to enforce. Among other things, since both pointers and objects to which pointers might refer can be passed as arguments to subroutines, dynamic semantic checks are possible only if reference parameters are accompanied by a hidden indication of lifetime. Ada 95 has a more restrictive rule that is easier to enforce: it forbids a pointer

---

**EXAMPLE 7.93**

Explicit storage  
reclamation



**Figure 7.15** Tombstones. A valid pointer refers to a tombstone that in turn refers to an object. A dangling reference refers to an “expired” tombstone.

from pointing to any object whose lifetime is briefer than that of the pointer’s *type*.

### Tombstones

*Tombstones* [Lom75, Lom85] are a mechanism by which a language implementation can catch all dangling references, to objects in both the stack and the heap. The idea is simple: rather than have a pointer refer to an object directly, we introduce an extra level of indirection (Figure 7.15). When an object is allocated in the heap (or when a pointer is created to an object in the stack), the language run-time system allocates a tombstone. The pointer contains the address of the tombstone; the tombstone contains the address of the object. When the object is reclaimed, the tombstone is modified to contain a value (typically zero) that cannot be a valid address. To avoid special cases in the generated code, tombstones are also created for pointers to static objects. ■

For heap objects, it is easy to invalidate a tombstone when the program calls the deallocation operation. For stack objects, the language implementation must be able to find all tombstones associated with objects in the current stack frame when returning from a subroutine. One possible solution is to link all stack-object tombstones together in a list, sorted by the address of the stack frame in which the object lies. When a pointer is created to a local object, the tombstone can simply be added to the beginning of the list. When a pointer is created to a parameter, the run-time system must scan down the list and insert in the middle, to keep it sorted. When a subroutine returns, the epilogue portion of the calling

---

#### EXAMPLE 7.94

Dangling reference detection with tombstones

sequence invalidates the tombstones at the head of the list, and removes them from the list.

Tombstones may be allocated from the heap itself or, more commonly, from a separate pool. The latter option avoids fragmentation problems, and makes allocation relatively fast, since the first tombstone on the free list is always the right size.

Tombstones can be expensive, both in time and in space. The time overhead includes (1) creation of tombstones when allocating heap objects or using a “pointer to” operator, (2) checking for validity on every access, and (3) double-indirection. Fortunately, checking for validity can be made essentially free on most machines by arranging for the address in an “invalid” tombstone to lie outside the program’s address space. Any attempt to use such an address will result in a hardware interrupt, which the operating system can reflect up into the language run-time system. We can also use our invalid address, in the pointer itself, to represent the constant `nil`. If the compiler arranges to set every pointer to `nil` at elaboration time, then the hardware will catch any use of an uninitialized pointer. (This technique works without tombstones, as well.)

The space overhead for tombstones can be significant. The simplest approach is never to reclaim them. Since a tombstone is usually significantly smaller than the object to which it refers, a program will waste less space by leaving a tombstone around forever than it would waste by never reclaiming the associated object. Even so, any long-running program that continually creates and reclaims objects will eventually run out of space for tombstones. A potential solution, which we will consider in Section 7.7.3, is to augment every tombstone with a *reference count*, and reclaim tombstones themselves when the reference count goes to zero.

Tombstones have a valuable side effect. Because of double-indirection, it is easy to change the location of an object in the heap. The run-time system need not locate every pointer that refers to the object; all that is required is to change the address in the tombstone. The principal reason to change heap locations is for *storage compaction*, in which all dynamically allocated blocks are “scooted together” at one end of the heap in order to eliminate external fragmentation. Tombstones are not widely used in language implementations, but the Macintosh operating system (versions 9 and below) uses them internally, for references to system objects such as file and window descriptors.

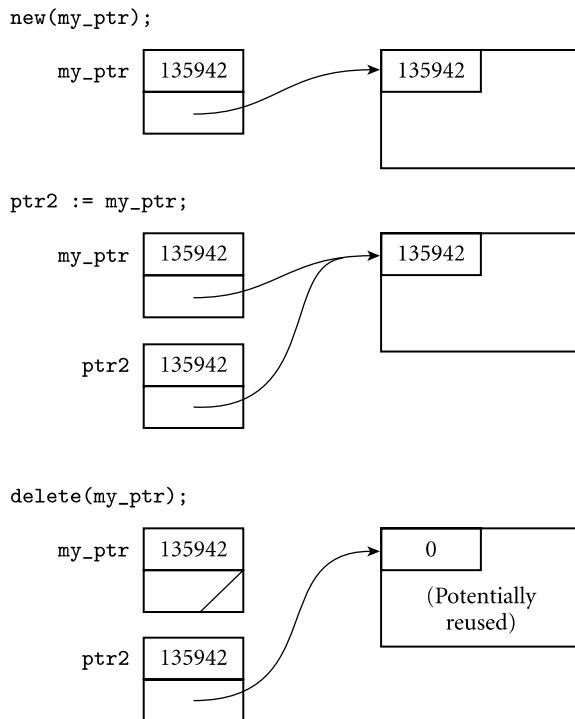
### **Locks and Keys**

*Locks and keys* [FL80] are an alternative to tombstones. Their disadvantages are that they work only for objects in the heap, and they provide only probabilistic protection from dangling pointers. Their advantage is that they avoid the need to keep tombstones around forever (or to figure out when to reclaim them). Again the idea is simple: Every pointer is a tuple consisting of an address and a key. Every object in the heap begins with a lock. A pointer to an object in the heap is valid only if the key in the pointer matches the lock in the object (Figure 7.16). When the run-time system allocates a new heap object, it generates a new key

---

#### **EXAMPLE 7.95**

Dangling reference detection with locks and keys



**Figure 7.16 Locks and Keys.** A valid pointer contains a key that matches the lock on an object in the heap. A dangling reference is unlikely to match.

value. These can be as simple as serial numbers, but should avoid “common” values such as zero and one. When an object is reclaimed, its lock is changed to some arbitrary value (e.g., zero) so that the keys in any remaining pointers will not match. If the block is subsequently reused for another purpose, we expect it to be very unlikely that the location that used to contain the lock will be restored to its former value by coincidence. ■

Like tombstones, locks and keys incur significant overhead. They add an extra word of storage to every pointer and to every block in the heap. They increase the cost of copying one pointer into another. Most significantly, they incur the cost of comparing locks and keys on every access (or every provably nonredundant access). It is unclear whether the lock and key check is cheaper or more expensive than the tombstone check. A tombstone check may result in two cache misses (one for the tombstone and one for the object); a lock and key check is unlikely to cause more than one. On the other hand, the lock and key check requires a significantly longer instruction sequence on most machines.

To minimize time and space overhead, most compilers do not by default generate code to check for dangling references. Most Pascal compilers allow the programmer to request dynamic checks, which are usually implemented with locks and keys. In most implementations of C, even optional checks are unavailable.

### 7.7.3 Garbage Collection

Explicit reclamation of heap objects is a serious burden on the programmer and a major source of bugs (memory leaks and dangling references). The code required to keep track of object lifetimes makes programs more difficult to design, implement, and maintain. An attractive alternative is to have the language implementation notice when objects are no longer useful and reclaim them automatically. Automatic reclamation (otherwise known as *garbage collection*) is more or less essential for functional languages: `delete` is a very imperative sort of operation, and the ability to construct and return arbitrary objects from functions means that many objects that would be allocated on the stack in an imperative language must be allocated from the heap in a functional language, to give them unlimited extent.

Over time, automatic garbage collection has become popular for imperative languages as well. It can be found in, among others, Clu, Cedar, Modula-3, Java, C#, and all the major scripting languages. Automatic collection is difficult to implement, but the difficulty pales in comparison to the convenience enjoyed by programmers once the implementation exists. Automatic collection also tends to be slower than manual reclamation, though it eliminates any need to check for dangling references.

#### DESIGN & IMPLEMENTATION

##### Garbage collection

Garbage collection presents a classic tradeoff between convenience and safety on the one hand and performance on the other. Manual storage reclamation, implemented correctly by the application program, is almost invariably faster than any automatic garbage collector. It is also more predictable: automatic collection is notorious for its tendency to introduce intermittent “hiccups” in the execution of real-time or interactive programs.

Ada takes the unusual position of refusing to take a stand: the language design makes automatic garbage collection possible, but implementations are not required to provide it (most don’t), and programmers can request manual reclamation with a built-in routine called `Unchecked_Deallocation`. The Ada 95 version of the language provides extensive facilities whereby programmers can implement their own storage managers (garbage collected or not), with different types of pointers corresponding to different storage “pools.”

In a similar vein, the Real Time Specification for Java allows the programmer to create so-called *scoped memory* areas that are accessible to only a subset of the currently running threads. When all threads with access to a given area terminate, the area is reclaimed in its entirety. Objects allocated in a scoped memory area are never examined by the garbage collector; performance anomalies due to garbage collection can therefore be avoided by providing scoped memory to every real-time thread.

### Reference Counts

When is an object no longer useful? One possible answer is: when no pointers to it exist.<sup>10</sup> The simplest garbage collection technique simply places a counter in each object that keeps track of the number of pointers that refer to the object. When the object is created, this *reference count* is set to one, to represent the pointer returned by the `new` operation. When one pointer is assigned into another, the run-time system decrements the reference count of the object formerly referred to by the assignment's left-hand side and increments the count of the object referred to by the right-hand side. On subroutine return, the calling sequence epilogue must decrement the reference count of any object referred to by a local pointer that is about to be destroyed. When a reference count reaches zero, its object can be reclaimed. Recursively, the run-time system must decrement counts for any objects referred to by pointers within the object being reclaimed, and reclaim those objects if their counts reach zero. To prevent the collector from following garbage addresses, each pointer must be set to `nil` at elaboration time.

In order for reference counts to work, the language implementation must be able to identify the location of every pointer. When a subroutine returns, it must be able to tell which words in the stack frame represent pointers; when an object in the heap is reclaimed, it must be able to tell which words within the object represent pointers. The standard technique to track this information relies on *type descriptors* generated by the compiler. There is one descriptor for every distinct type in the program, plus one for the stack frame of each subroutine and one for the set of global variables. Most descriptors are simply a table that lists the offsets within the type at which pointers can be found, together with the addresses of descriptors for the types of the objects referred to by those pointers. For a tagged variant record (discriminated union) type, the descriptor is a bit more complicated: it must contain a list of values (or ranges) for the tag, together with a table for the corresponding variant. For *untagged* variant records, there is no acceptable solution: reference counts work only if the language is strongly typed (but see the discussion of "Conservative Collection" on page 389).

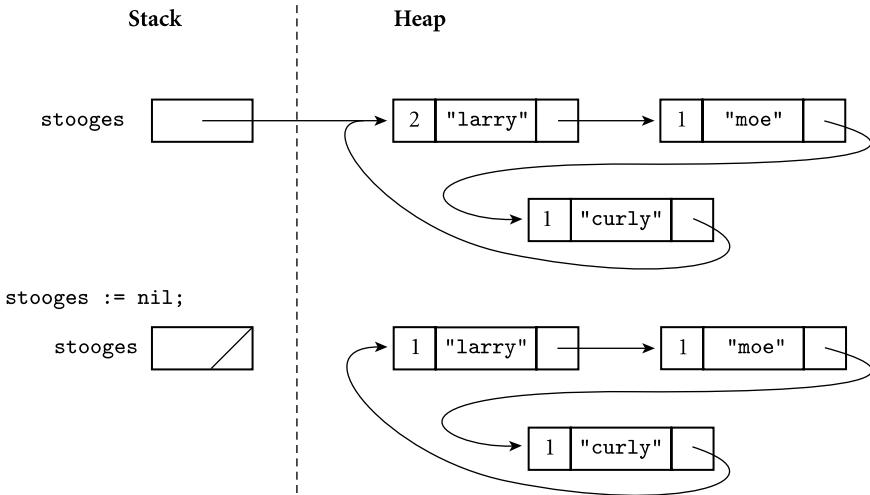
#### EXAMPLE 7.96

Reference counts and circular structures

The most important problem with reference counts stems from their definition of a "useful object." While it is definitely true that an object is useless if no references to it exist, it may also be useless when references *do* exist. As shown in Figure 7.17, reference counts may fail to collect circular structures. They work well only for structures that are guaranteed to be noncircular. Many language implementations use reference counts for variable-length strings; strings never contain references to anything else. Perl uses reference counts for all dynamically allocated data; the manual warns the programmer to break cycles manually when data aren't needed anymore. Some purely functional languages may also be able to use reference counts safely in all cases, if the lack of an assignment statement

---

**10** Throughout the following discussion we will use the pointer-based terminology of languages with a value model of variables. The techniques apply equally well, however, to languages with a reference model of variables.



**Figure 7.17 Reference counts and circular lists.** The list shown here cannot be found via any program variable, but because the list is circular, every cell contains a nonzero count.

prevents them from introducing circularity. Finally, reference counts can be used to reclaim tombstones. While it is certainly possible to create a circular structure with tombstones, the fact that the programmer is responsible for explicit deallocation of heap objects implies that reference counts will fail to reclaim tombstones only when the programmer has failed to reclaim the objects to which they refer. ■

#### Tracing Collection

A better definition of a “useful” object is one that can be reached by following a chain of valid pointers starting from something that has a name (i.e., something outside the heap). According to this definition, the blocks in the bottom half of Figure 7.17 are useless, even though their reference counts are nonzero. Tracing collectors work by recursively exploring the heap, starting from external pointers, to determine what is useful.

**Mark-and-Sweep** The classic mechanism to identify useless blocks, under this more accurate definition, is known as *mark-and-sweep*. It proceeds in three main steps, executed by the garbage collector when the amount of free space remaining in the heap falls below some minimum threshold.

1. The collector walks through the heap, tentatively marking every block as “useless.”
2. Beginning with all pointers outside the heap, the collector recursively explores all linked data structures in the program, marking each newly discovered block as “useful.” (When it encounters a block that is already marked as

“useful,” the collector knows it has reached the block over some previous path, and returns without recursing.)

3. The collector again walks through the heap, moving every block that is still marked “useless” to the free list.

Several potential problems with this algorithm are immediately apparent. First, both the initial and final walks through the heap require that the collector be able to tell where every “in-use” block begins and ends. In a language with variable-size heap blocks, every block must begin with an indication of its size, and of whether it is currently free. Second, the collector must be able in Step 2 to find the pointers contained within each block. The standard solution is to place a pointer to a type descriptor near the beginning of each block.

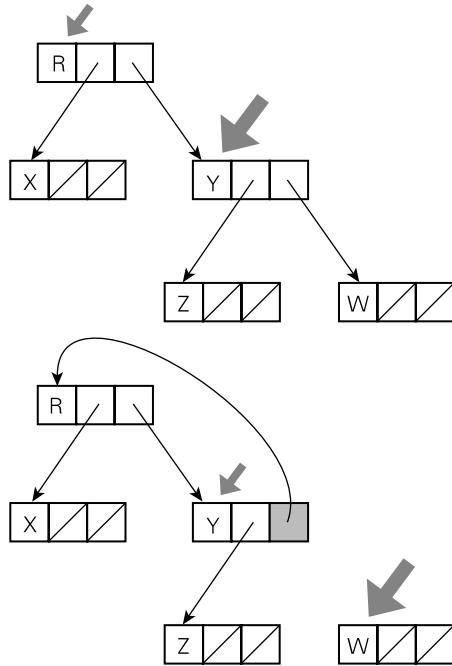
The space overhead for bookkeeping information in heap blocks is not as large as it might at first appear. If every type descriptor contains an indication of size, then a heap block that includes the address of its type descriptor need not include its size as a separate field (though the extra indirection required to find the size in the descriptor makes walking the heap more expensive). Moreover, since a type descriptor must be word-aligned on most machines, the two low-order bits of its address are guaranteed to be zero. If we are willing to mask these bits out before using the address, we can use them to store the “free” and “useful” flags.

**Pointer Reversal** The exploration step (Step 2) of mark-and-sweep collection is naturally recursive. The obvious implementation needs a stack whose maximum depth is proportional to the longest chain through the heap. In practice, the space for this stack may not be available: after all, we run garbage collection when we’re about to run out of space!<sup>11</sup> An alternative implementation of the exploration step uses a technique first suggested by Schorr and Waite [SW67] to embed the equivalent of the stack in already-existing fields in heap blocks. More specifically, as the collector explores the path to a given block, it *reverses* the pointers it follows, so that each points *back* to the previous block instead of forward to the next. This pointer-reversal technique is illustrated in Figure 7.18. As it explores, the collector keeps track of the current block and the block from whence it came (the two gray arrows in the figure).

When it returns from block W to block Y, the collector uses the reversed pointer in Y to restore its notion of previous block (R in our example). It then flips the reversed pointer back to W and updates its notion of current block to Y. If the block to which it has returned contains additional pointers, the collector proceeds forward again; otherwise it returns across the previous reversed pointer and tries again. At most one pointer in every block will be reversed at any given time. This pointer must be marked, probably by means of another bookkeeping

---

<sup>11</sup> In many language implementations, the stack and heap grow toward each other from opposite ends of memory; if the heap is full, the stack can’t grow. In a system with virtual memory the distance between the two may theoretically be enormous, but the space that backs them up on disk is still limited, and shared between them.



**Figure 7.18** **Heap exploration via pointer reversal.** The block currently under examination is indicated by the large gray arrow. The previous block is indicated by the small gray arrow. As the garbage collector moves from one block to the next, it changes the pointer it follows to refer back to the previous block. When it returns to a block it restores the pointer. Each reversed pointer must be marked, to distinguish it from other, forward pointers in the same block. We assume in this figure that the root node R is outside the heap, so none of its pointers are reversed.

field at the beginning of each block. (We could mark the pointer by setting one of its low-order bits, but the cost in time would probably be prohibitive: we'd have to search the block on every visit.)

**Stop-and-Copy** In a language with variable-size heap blocks, the garbage collector can reduce external fragmentation by performing storage compaction, as noted in the preceding discussion of tombstones. Compaction with tombstones is easier because there is only a single pointer to each object. Many garbage collectors employ a technique known as *stop-and-copy* that achieves compaction while simultaneously eliminating Steps 1 and 3 in the standard mark and sweep algorithm. Specifically, they divide the heap into two regions of equal size. All allocation happens in the first half. When this half is (nearly) full, the collector begins its exploration of reachable data structures. Each reachable block is copied into the second half of the heap, with no external fragmentation. The old version of the block, in the first half of the heap, is overwritten with a “useful” flag and a pointer to the new location. Any other pointer that refers to the same block (and

is found later in the exploration) is set to point to the new location. When the collector finishes its exploration, all useful objects have been moved (and compacted) into the second half of the heap, and nothing in the first half is needed anymore. The collector can therefore swap its notion of first and second halves, and the program can continue. Obviously, this algorithm suffers from the fact that only half of the heap can be used at any given time, but in a system with virtual memory it is only the virtual space that is underutilized; each “half” of the heap can occupy most of physical memory as needed. Moreover, by eliminating Steps 1 and 3 of standard mark and sweep, stop and copy incurs overhead proportional to the number of nongarbage blocks, rather than the total number of blocks.

**Generational Collection** To further reduce the cost of collection, some garbage collectors employ a “generational” technique, exploiting the observation that most dynamically allocated objects are short lived. The heap is divided into multiple regions (often two). When space runs low the collector first examines the youngest region (the “nursery”), which it assumes is likely to have the highest proportion of garbage. Only if it is unable to reclaim sufficient space in this region does the collector examine the next-older region. Any object that survives some small number of collections (often one) in its current region is promoted (moved) to the next older region, in a manner reminiscent of stop-and-copy. Promotion requires, of course, that pointers from old objects to new objects be

## DESIGN & IMPLEMENTATION

### Reference counts v. tracing

Reference counts require a counter field in every heap object. For small objects such as cons cells, this space overhead may be significant. The ongoing expense of updating reference counts when pointers are changed can also be significant in a program with large amounts of pointer manipulation. Other garbage collection techniques, however, have similar overheads. Tracing generally requires a reversed pointer indicator in every heap block, which reference counting does not, and generational collectors must generally incur overhead on every pointer assignment in order to keep track of pointers into the newest section of the heap.

The two principal tradeoffs between reference counting and tracing are the inability of the former to handle cycles and the tendency of the latter to “stop the world” periodically in order to reclaim space. On the whole, implementors tend to favor reference counting for applications in which circularity is not an issue, and tracing collectors in the general case. The “stop the world” problem can be addressed with *incremental* or *parallel* collectors, which execute concurrently with the rest of the program, but these tend to have higher total overhead. Efficient, effective garbage collection techniques remain an active area of research.

updated to reflect the new locations. While such old-to-new pointers tend to be rare, a generational collector must track them in an explicit data structure (updated at pointer assignment time) in order to avoid scanning the older portions of the heap in order to find them. A collector for a long-running system, which cannot afford to leak storage, must be able in the general case to examine the entire heap, but in most cases the overhead of collection will be proportional to the size of the youngest region only.

**Conservative Collection** Language implementors have traditionally assumed that automatic storage reclamation is possible only in languages that are strongly typed: both reference counts and tracing collection require that we be able to find the pointers within an object. If we are willing to admit the possibility that some garbage will go unreclaimed, it turns out that we can implement mark-and-sweep collection without being able to find pointers [BW88]. The key is to observe that the number of blocks in the heap is much smaller than the number of possible bit patterns in an address. The probability that a word in memory that is not a pointer into the heap will happen to contain a bit pattern that looks like such a pointer is relatively small. If we assume, conservatively, that everything that seems to point to a heap block is in fact a valid pointer, then we can proceed with mark-and-sweep collection. When space runs low, the collector (as usual) tentatively marks all blocks in the heap as useless. It then scans all word-aligned quantities in the stack and in global storage. If any of these “pointers” contains the address of a block in the heap, the collector marks that block as useful. Recursively, the collector then scans all word-aligned quantities in the block and marks as useful any other blocks whose addresses are found therein. Finally (as usual), the collector reclaims any blocks that are still marked useless. The algorithm is completely safe (in the sense that it never reclaims useful blocks) as long as the programmer never “hides” a pointer. In C, for example, the collector is unlikely to function correctly if the programmer casts a pointer to `int` and then `xors` it with a constant, with the expectation of restoring and using the pointer at a later time. In addition to sometimes leaving garbage unreclaimed, conservative collection suffers from the inability to perform compaction: the collector can never be sure which “pointers” should be changed.

## 7.8 Lists

A list is defined recursively as either the empty list or a pair consisting of an object (which may be either a list or an atom) and another (shorter) list. Lists are ideally suited to programming in functional and logic languages, which do most of their work via recursion and higher-order functions (to be described in Section 10.5). In Lisp, in fact, a program *is* a list, and can extend itself at run time by constructing a list and executing it (this capability will be examined further in Section 10.3.5; it depends heavily on the fact that Lisp delays almost all semantic checking until run time).

Lists can also be used in imperative programs. Clu provides a built-in type constructor for lists, and a list class is easy to write in most object-oriented languages. Several scripting languages, notably Perl and Python, provide extensive list support. In any language with records and pointers, the programmer can build lists by hand. Since many of the standard list operations tend to generate garbage, lists work best in a language with automatic garbage collection.

We have already discussed certain aspects of lists in ML (Section 7.2.4) and Lisp (Section 7.7.1). As we noted in those sections, lists in ML are homogeneous: every element of the list must have the same type. Lisp lists, by contrast, are heterogeneous: any object may be placed in a list, as long as it is never used in an inconsistent fashion.<sup>12</sup> The different approaches to type in ML and in Lisp lead to different implementations. An ML list is usually a chain of blocks, each of which contains an element and a pointer to the next block. A Lisp list is a chain of cons cells, each of which contains *two* pointers, one to the element and one to the next cons cell (see Figures 7.12 and 7.13, pages 371 and 372). For historical reasons, the two pointers in a cons cell are known as the *car* and the *cdr*; they represent the head of the list and the remaining elements, respectively. In both semantics (homogeneity versus heterogeneity) and implementation (chained blocks versus cons cells), Clu resembles ML, while Python and Prolog (to be discussed in Section 11.2) resemble Lisp. ■

#### EXAMPLE 7.98

##### Lists in ML and Lisp

#### EXAMPLE 7.99

##### List notation

Both ML and Lisp provide convenient notation for lists. An ML list is enclosed in square brackets, with elements separated by commas: [a, b, c, d]. A Lisp list is enclosed in parentheses, with elements separated by white space: (a b c d). In both cases, the notation represents a *proper* list: one whose innermost pair consists of the final element and the empty list. In Lisp, it is also possible to construct an *improper* list, whose final pair contains two elements. (Strictly speaking, such a list does not conform to the standard recursive definition.) Lisp systems provide a more general but cumbersome *dotted* list notation that captures both proper and improper lists. A dotted list is either an atom (possibly nil) or a pair consisting of two dotted lists separated by a period and enclosed in parentheses. The dotted list (a . (b . (c . (d . nil)))) is the same as (a b c d). The list (a . (b . (c . d))) is improper; its final cons cell contains a pointer to d in the second position, where a pointer to a list is normally required. ■

Both ML and Lisp provide a wealth of built-in polymorphic functions to manipulate arbitrary lists. Because programs are lists in Lisp, Lisp must distinguish between lists that are to be evaluated and lists that are to be left “as is” as structures. To prevent a literal list from being evaluated, the Lisp programmer may quote it: (quote (a b c d)), abbreviated '(a b c d). To evaluate an internal list (e.g., one returned by a function), the programmer may pass it to the built-in function eval. In ML, programs are not lists, so a literal list is always a structural aggregate.

---

**12** Recall that objects are self-descriptive in Lisp. The only type checking occurs when a function “deliberately” inspects an argument to see whether it is a list or an atom of some particular type.

**EXAMPLE 7.100**

Basic list operations in Lisp

The most fundamental operations on lists are those that construct them from their components or extract their components from them. In Lisp:

|                        |                         |
|------------------------|-------------------------|
| (cons 'a '(b))         | $\Rightarrow$ (a b)     |
| (car '(a b))           | $\Rightarrow$ a         |
| (car nil)              | $\Rightarrow$ ??        |
| (cdr '(a b c))         | $\Rightarrow$ (b c)     |
| (cdr '(a))             | $\Rightarrow$ nil       |
| (cdr nil)              | $\Rightarrow$ ??        |
| (append '(a b) '(c d)) | $\Rightarrow$ (a b c d) |

As in Chapter 6, we have used  $\Rightarrow$  to mean “evaluates to.” The car and cdr of the empty list (nil) are defined to be nil in Common Lisp; in Scheme they result in a dynamic semantic error.

**EXAMPLE 7.101**

Basic list operations in ML

In ML the equivalent operations are written as follows.

|                 |                                  |
|-----------------|----------------------------------|
| a :: [b]        | $\Rightarrow$ [a, b]             |
| hd [a, b]       | $\Rightarrow$ a                  |
| hd []           | $\Rightarrow$ run-time exception |
| tl [a, b, c]    | $\Rightarrow$ [b, c]             |
| tl [a]          | $\Rightarrow$ nil                |
| tl []           | $\Rightarrow$ run-time exception |
| [a, b] @ [c, d] | $\Rightarrow$ [a, b, c, d]       |

Run-time exceptions may be *caught* by the program if desired; further details will appear in Section 8.5.

Both ML and Lisp provide many additional list functions, including ones that test a list to see if it is empty; return the length of a list; return the *n*th element of a list, or a list consisting of all but the first *n* elements; reverse the order of the

**DESIGN & IMPLEMENTATION****Car and cdr**

The names of the functions car and cdr are historical accidents: they derive from the original (1959) implementation of Lisp on the IBM 704 at MIT. The machine architecture included 15-bit “address” and “decrement” fields in some of the (36-bit) loop-control instructions, together with additional instructions to load an index register from, or store it to, one of these fields within a 36-bit memory word. The designers of the Lisp interpreter decided to make cons cells mimic the internal format of instructions, so they could exploit these special instructions. In now archaic usage, memory words were also known as “registers.” What might appropriately have been called “first” and “rest” pointers thus came to be known as the CAR (contents of address of register) and CDR (contents of decrement of register). The 704, incidentally, was also the machine on which Fortran was first developed, and the first commercial machine to include hardware floating point and magnetic core memory.

elements of a list; search a list for elements matching some predicate; or apply a function to every element of a list, returning the results as a list.

Miranda, Haskell, and Python provide lists that resemble those of ML, but with an important additional mechanism, known as *list comprehensions*. A common form of list comprehension comprises an expression, an enumerator, and one or more filters. In Miranda and Haskell, the following denotes a list of the squares of all odd numbers less than 100.

**EXAMPLE 7.102**

## List comprehensions

```
[i*i | i <- [1..100], i `mod` 2 == 1]
```

Here the vertical bar means “such that”; the left arrow is roughly equivalent to “is a member of.” (Python syntax is slightly different.) We could of course write an equivalent expression with a collection of appropriate functions. The brevity of the list comprehension syntax, however, can sometimes lead to remarkably elegant programs (see, for example, Exercise 7.32). ■

## 7.9 Files and Input/Output

Input/output (I/O) facilities allow a program to communicate with the outside world. In discussing this communication, it is customary to distinguish between *interactive* I/O and I/O with files. Interactive I/O generally implies communication with human users or physical devices, which work in parallel with the running program, and whose input to the program may depend on earlier output from the program (e.g., prompts). Files generally correspond to storage outside the program’s address space, implemented by the operating system. Files may be further categorized into those that are *temporary* and those that are *persistent*. Temporary files exist for the duration of a single program run; their purpose is to store information that is too large to fit in the memory available to the program. Persistent files allow a program to read data that existed before the program began running, and to write data that will continue to exist after the program has ended.

I/O is one of the most difficult aspects of a language to design, and one that displays the least commonality from one language to the next. Some languages provide built-in `file` data types and special syntactic constructs for I/O. Oth-

### DESIGN & IMPLEMENTATION

#### I/O

Regardless of the level of language integration, the design of I/O facilities is complicated by the tension between “power” and portability: designers generally want to take advantage of (and provide access to) all the features supported by the underlying operating system. At the same time, they want to minimize the amount of work required to move a program from one system to another.

ers relegate I/O entirely to library packages, which export a (usually opaque) `file` type and a variety of input and output subroutines. The principal advantage of language integration is the ability to employ non-subroutine-call syntax and to perform operations (e.g., type checking on subroutine calls with varying numbers of parameters) that may not otherwise be available to library routines. A purely library-based approach to I/O, on the other hand, may keep a substantial amount of “clutter” out of the language definition.

#### IN MORE DEPTH

After a brief introduction to interactive and file-based I/O, we focus mainly on the common case of *text files*. The data in a text file are stored in character form but may be converted to and from internal types during `read` and `write` operations. As examples, we consider the text I/O facilities of Fortran, Ada, C, and C++.

## 7.10 Equality Testing and Assignment

For simple, primitive data types such as integers, floating-point numbers, or characters, equality testing and assignment are relatively straightforward operations, with obvious semantics and obvious implementations (bit-wise comparison or copy). For more complicated or abstract data types, however, both semantic and implementation subtleties arise.

Consider for example the problem of comparing two character strings. Should the expression `s = t` determine whether `s` and `t`

- are aliases for one another?
- occupy storage that is bit-wise identical over its full length?
- contain the same sequence of characters?
- would appear the same if printed?

The second of these tests is probably too low level to be of interest in most programs; it suggests the possibility that a comparison might fail because of garbage in currently unused portions of the space reserved for a string. The other three alternatives may all be of interest in certain circumstances, and may generate different results.

In many cases the definition of equality boils down to the distinction between l-values and r-values: in the presence of references, should expressions be considered equal only if they refer to the same object, or also if the objects to which they refer are in some sense equal? The first option (refer to the same object) is known as a *shallow* comparison. The second (refer to equal objects) is called a *deep* comparison. For complicated data structures (e.g., lists or graphs) a deep comparison may require recursive traversal.

In imperative programming languages assignment operations may also be deep or shallow. Under a reference model of variables, a shallow assignment `a := b` will make `a` refer to the object to which `b` refers. A deep assignment will create a copy of the object to which `b` refers, and make `a` refer to the copy. Under a value model of variables, a shallow assignment will copy the value of `b` into `a`, but if that value is a pointer (or a record containing pointers), then the objects to which the pointer(s) refer will not be copied.

**EXAMPLE 7.103**

Equality testing in Scheme

Most programming languages employ both shallow comparisons and shallow assignment. A few (notably Python and the various dialects of Lisp) provide more than one option for comparison. Scheme, for example, has three equality-testing functions:

```
(eq? a b) ; do a and b refer to the same object?
(eqv? a b) ; are a and b provably semantically equivalent?
(equal? a b) ; do a and b have the same recursive structure?
```

The intent behind the `eq?` predicate is to make the implementation as fast as possible while still producing useful results for many types of operands. The intent behind `eqv?` is to provide as intuitively appealing a result as possible for as wide a range of types as possible.

The `eq?` predicate behaves as one would expect for Booleans, symbols (names), and pairs (things built by `cons`), but can have implementation-defined behavior on numbers, characters, and strings.

|                                             |                                                                                  |
|---------------------------------------------|----------------------------------------------------------------------------------|
| <code>(eq? #t #t)</code>                    | $\implies \#t$ ( <code>true</code> )                                             |
| <code>(eq? 'foo 'foo)</code>                | $\implies \#t$                                                                   |
| <code>(eq? '(a b) '(a b))</code>            | $\implies \#f$ ( <code>false</code> ); created by separate <code>cons</code> -es |
| <code>(let ((p '(a b)))   (eq? p p))</code> | $\implies \#t$ ; created by the same <code>cons</code>                           |
| <code>(eq? 2 2)</code>                      | $\implies \text{unspecified}$                                                    |
| <code>(eq? "foo" "foo")</code>              | $\implies \text{unspecified}$                                                    |

In any particular implementation, numeric, character, and string tests will always work the same way; if `(eq? 2 2)` returns `true`, then `(eq? 37 37)` will return `true` also. Implementations are free to choose whichever behavior results in the fastest code.

The exact rules that govern the situations in which `eqv?` is guaranteed to return `true` or `false` are quite involved. Among other things, they specify that `eqv?` should behave as one might expect for numbers, characters, and nonempty strings, and that two objects will never test `true` for `eqv?` if there are any circumstances under which they would behave differently. (Conversely, however, `eqv?` is allowed to return `false` for certain objects—functions, for example—that would behave identically in all circumstances.) The `eqv?` predicate is “less discriminating” than `eq?`, in the sense that `eqv?` will never return `false` when `eq?` returns `true`.

For structures (lists), `eqv?` returns `false` if its arguments refer to different root `cons` cells. In many programs this is not the desired behavior. The `equal?`

predicate recursively traverses two lists to see if their internal structure is the same and their leaves are `eqv?`. The `equal?` predicate may lead to an infinite loop if the programmer has used the imperative features of Scheme to create a circular list. ■

Deep assignments are relatively rare. They are used primarily in distributed computing, and in particular for parameter passing in remote procedure call (RPC) systems. These will be discussed in Section 12.4.4.

For user-defined abstractions, no single language-specified mechanism for equality testing or assignment is likely to produce the desired results in all cases. Languages with sophisticated data abstraction mechanisms usually allow the programmer to define the comparison and assignment operators for each new data type—or to specify that equality testing and/or assignment is not allowed.

### CHECK YOUR UNDERSTANDING

---

45. What are *dangling references*? How are they created, and why are they a problem? Discuss the comparative advantages of *tombstones* and *locks and keys* as a means of solving the problem.
  46. What is *garbage*? How is it created, and why is it a problem? Discuss the comparative advantages of *reference counts* and *tracing collection* as a means of solving the problem.
  47. Summarize the differences among mark-and-sweep, stop-and-copy, and generational garbage collection.
  48. What is *pointer reversal*? What problem does it address?
  49. What is “conservative” garbage collection? How does it work?
  50. Do dangling references and garbage ever arise in the same programming language? Why or why not?
  51. Why was automatic garbage collection so slow to be adopted by imperative programming languages?
  52. What are the advantages and disadvantages of allowing pointers to refer to objects that do not lie in the heap?
  53. Why are lists so heavily used in functional programming languages?
  54. Why is equality testing more subtle than it first appears?
- 

## 7.11 Summary and Concluding Remarks

This section concludes the third of our five core chapters on language design (names [from Part I], control flow, types, subroutines, and classes). In the first

two sections we looked at the general issues of type systems and type checking. In the remaining sections we examined the most important composite types: records and variants, arrays and strings, sets, pointers and recursive types, lists, and files. We noted that types serve two principal purposes: they provide implicit context for many operations, freeing the programmer from the need to specify that context explicitly, and they allow the compiler to catch a wide variety of common programming errors. A *type system* consists of a set of built-in types; a mechanism to define new types; and rules for *type equivalence*, *type compatibility*, and *type inference*. Type equivalence determines when two names or values have the same type. Type compatibility determines when a value of one type may be used in a context that “expects” another type. Type inference determines the type of an expression based on the types of its components or (sometimes) the surrounding context. A language is said to be *strongly typed* if it never allows an operation to be applied to an object that does not support it; a language is said to be *statically typed* if it enforces strong typing at compile time.

In our general discussion of types we distinguished between the denotational, constructive, and abstraction-based points of view, which regard types, respectively, in terms of their values, their substructure, and the operations they support. We introduced terminology for the common built-in types and for enumerations, subranges, and the common type *constructors*. We discussed several different approaches to type equivalence, compatibility, and inference, including (on the PLP CD) a detailed examination of the inference rules of ML. We also examined type *conversion*, *coercion*, and *nonconverting casts*. In the area of type equivalence, we contrasted the *structural* and *name-based* approaches, noting that while name equivalence appears to have gained in popularity, structural equivalence retains its advocates.

In our survey of composite types, we spent the most time on records, arrays, and recursive types. Key issues for records include the syntax and semantics of variant records, whole-record operations, type safety, and the interaction of each of these with memory layout. Memory layout is also important for arrays, in which it interacts with binding time for shape; static, stack, and heap-based allocation strategies; efficient array traversal in numeric applications; the interoperability of pointers and arrays in C; and the available set of whole-array and *slice-based* operations.

For recursive data types, much depends on the choice between the *value* and *reference models* of variables/names. Recursive types are a natural fall-out of the reference model; with the value model they require the notion of a *pointer*: a variable whose value is a reference. The distinction between values and references is important from an implementation point of view: it would be wasteful to implement built-in types as references, so languages with a reference model generally implement built-in and user-defined types differently. Java reflects this distinction in the language semantics, calling for a value model of built-in types and a reference model for objects of user-defined type classes.

Recursive types are generally used to create linked data structures. In most cases these structures must be allocated from a heap. In some languages, the pro-

grammer is responsible for deallocating heap objects that are no longer needed. In other languages, the language run-time system identifies and reclaims such *garbage* automatically. Explicit deallocation is a burden on the programmer, and leads to the problems of *memory leaks* and *dangling references*. While language implementations almost never attempt to catch memory leaks (see Exploration 3.28 and Exercise 7.30, however, for some ideas on this subject) *tombstones* or *locks and keys* are sometimes used to catch dangling references. Automatic garbage collection can be expensive but has proven increasingly popular. Most garbage-collection techniques rely either on *reference counts* or on some form of recursive exploration (tracing) of currently accessible structures. Techniques in this latter category include *mark-and-sweep*, *stop-and-copy*, and *generational* collection.

Few areas of language design display as much variation as I/O. Our discussion (largely on the PLP CD) distinguished between *interactive I/O*, which tends to be very platform specific, and *file-based I/O*, which subdivides into *temporary files*, used for voluminous data within a single program run, and *persistent files*, used for off-line storage. Files also subdivide into those that represent their information in a binary form that mimics layout in memory and those that convert to and from character-based *text*. In comparison to binary files, text files generally incur both time and space overhead, but they have the important advantages of portability and human readability.

In our examination of types, we saw many examples of language innovations that have served to improve the clarity and maintainability of programs, often with little or no performance overhead. Examples include the original idea of user-defined types (Algol 68), enumeration and subrange types (Pascal), the integration of records and variants (Pascal), and the distinction between subtypes and derived types in Ada. In Chapter 9 we will examine what many consider the most important innovation of the past thirty years, namely object orientation.

In some cases, the distinctions between languages are less a matter of evolution than of fundamental differences in philosophy. We have already mentioned the choice between the value and reference models of variables/names. In a similar vein, most languages have adopted static typing, but Smalltalk, Lisp, and the many scripting languages work well with dynamic types. Most statically typed languages have adopted name equivalence, but ML and Modula-3 work well with structural equivalence. Most languages have moved away from type coercions, but C++ embraces them: together with operator overloading, they make it possible to define terse, type-safe I/O routines outside the language proper.

As in the previous chapter, we saw several cases in which a language's convenience, orthogonality, or type safety appears to have been compromised in order to simplify the compiler, or to make compiled programs smaller or faster. Examples include the lack of an equality test for records in most languages, the requirement in Pascal and Ada that the variant portion of a record lie at the end, the limitations in many languages on the maximum size of sets, the lack of type checking for I/O in C, and the general lack of dynamic semantic checks in many language implementations. We also saw several examples of language features in-

troduced at least in part for the sake of efficient implementation. These include packed types, multilength numeric types, with statements, decimal arithmetic, and C-style pointer arithmetic.

At the same time, one can identify a growing willingness on the part of language designers and users to tolerate complexity and cost in language implementation in order to improve semantics. Examples here include the type-safe variant records of Ada; the standard-length numeric types of Java and C#; the variable-length strings and string operators of Icon, Java, and C#; the late binding of array bounds in Ada; and the wealth of whole-array and slice-based array operations in Fortran 90. One might also include the polymorphic type inference of ML. Certainly one should include the trend toward automatic garbage collection. Once considered too expensive for production-quality imperative languages, garbage collection is now standard not only in such experimental languages as Clu and Cedar, but in Ada, Modula-3, Java, and C# as well. Many of these features, including variable-length strings, slices, and garbage collection, have been embraced by scripting languages.

## 7.12 Exercises

- 7.1 Most modern Algol-family languages use some form of name equivalence for types. Is structural equivalence a bad idea? Why or why not?
- 7.2 In the following code, which of the variables will a compiler consider to have compatible types under structural equivalence? Under strict name equivalence? Under loose name equivalence?

```
type T = array [1..10] of integer
S = T
A : T
B : T
C : S
D : array [1..10] of integer
```

- 7.3 Consider the following declarations.

```
1. type cell -- a forward declaration
2. type cell_ptr = pointer to cell
3. x : cell
4. type cell = record
5. val : integer
6. next : cell_ptr
7. y : cell
```

Should the declaration at line 4 be said to introduce an alias type? Under strict name equivalence, should x and y have the same type? Explain.

- 7.4 Suppose you are implementing an Ada compiler, and must support arithmetic on 32-bit fixed-point binary numbers with a programmer-specified number of fractional bits. Describe the code you would need to generate to add, subtract, multiply, or divide two fixed-point numbers. You should assume that the hardware provides arithmetic instructions only for integers and IEEE floating point. You may assume that the integer instructions preserve full precision; in particular, integer multiplication produces a 64-bit result. Your description should be general enough to deal with operands and results that have different numbers of fractional bits.
- 7.5 When Sun Microsystems ported Berkeley Unix from the Digital VAX to the Motorola 680x0 in the early 1980s, many C programs stopped working and had to be repaired. In effect, the 680x0 revealed certain classes of program bugs that one could “get away with” on the VAX. One of these classes of bugs occurred in programs that use more than one size of integer (e.g., short and long) and arose from the fact that the VAX is a little-endian machine, while the 680x0 is big-endian (Section 5.2). Another class of bugs occurred in programs that manipulate both null and empty strings. It arose from the fact that location zero in a process’s address space on the VAX always contained a zero, while the same location on the 680x0 is not in the address space, and will generate a protection error if used. For both of these classes of bugs, give examples of program fragments that would work on a VAX but not on a 680x0.
- 7.6 Ada provides two “remainder” operators, `rem` and `mod` for integer types, defined as follows [Ame83, Sec. 4.5.5]:

Integer division and remainder are defined by the relation  $A = (A/B)*B + (A \bmod B)$ , where  $(A \bmod B)$  has the sign of A and an absolute value less than the absolute value of B. Integer division satisfies the identity  $(-A)/B = - (A/B) = A/(-B)$ .

The result of the modulus operation is such that  $(A \bmod B)$  has the sign of B and an absolute value less than the absolute value of B; in addition, for some integer value N, this result must satisfy the relation  $A = B*N + (A \bmod B)$ .

Give values of A and B for which `A rem B` and `A mod B` differ. For what purposes would one operation be more useful than the other? Does it make sense to provide both, or is it overkill?

Consider also the `%` operator of C and the `mod` operator of Pascal. The designers of these languages could have picked semantics resembling those of either Ada’s `rem` or its `mod`. Which did they pick? Do you think they made the right choice?

- 7.7 Consider the problem of performing range checks on set expressions in Pascal. Given that a set may contain many elements, some of which may be known at compile time, describe the information that a compiler might maintain in order to track both the elements known to belong to the set

and the possible range of unknown elements. Then explain how to update this information for the following set operations: union, intersection, and difference. The goal is to determine (1) when subrange checks can be eliminated at run time and (2) when subrange errors can be reported at compile time. Bear in mind that the compiler *cannot* do a perfect job: some unnecessary run-time checks will inevitably be performed, and some operations that must always result in errors will not be caught at compile time. The goal is to do as good a job as possible at reasonable cost.

- 7.8 Suppose we are compiling for a machine with 1-byte characters, 2-byte shorts, 4-byte integers, and 8-byte reals, and with alignment rules that require the address of every primitive data element to be an even multiple of the element's size. Suppose further that the compiler is not permitted to reorder fields. How much space will be consumed by the following array?

```
A : array [0..9] of record
 s : short
 c : char
 t : short
 d : char
 r : real
 i : integer
```

- 7.9 Show how variant records in Pascal or unions in C can be used to interpret the bits of a value of one type as if they represented a value of some other type. Explain why the same technique does not work in Ada. If you have access to an Ada manual, describe how an `unchecked` pragma can be used to get around the Ada rules.
- 7.10 Are variant records a form of polymorphism? Why or why not?
- 7.11 Pascal does not permit the tag field of a variant record to be passed to a subroutine by reference (i.e., as a `var` parameter). Why not?
- 7.12 Explain how to implement dynamic semantic checks to catch references to uninitialized fields of a tagged variant record in Pascal. Changing the value of the tag field should cause all fields of the variant part of the record to become uninitialized. Suppose you want to avoid adding flag fields within the record itself (e.g., to avoid changing the offsets of fields in a systems program). How much harder is your task?
- 7.13 Explain how to implement dynamic semantic checks to catch references to uninitialized fields of an *untagged* variant record in Pascal. Any assignment to a field of a variant should cause all fields of other variants to become uninitialized. Any assignment that changes the record from one variant to another should also cause all other fields of the new variant to be uninitialized. Again, suppose you want to avoid adding flag fields within the untagged record itself. How much harder is your task?

**7.14** We noted in Section 7.3.4 that Pascal and Ada require the variant portions of a record to occur at the end, to save space when a particular record is constrained to have a comparatively small variant part. Could a compiler rearrange fields to achieve the same effect, without the restriction on the declaration order of fields? Why or why not?

**7.15** Give Ada code to map from lowercase to uppercase letters, using

- (a) an array
- (b) a function

Note the similarity of syntax: in both cases `upper('a')` is '`A`'.

**7.16** In Section 7.4 we discussed how to differentiate between the constant and variable portions of an array reference, in order to efficiently access the subparts of array and record objects. An alternative approach is to generate naive code and count on the compiler's code improver to find the constant portions, group them together, and calculate them at compile time. Discuss the advantages and disadvantages of each approach.

**7.17** Explain how to extend Figure 7.7 to accommodate subroutine arguments that are passed by value, but whose shape is not known until the subroutine is called at run time.

**7.18** Explain how to obtain the effect of Fortran 90's `allocate` statement for one-dimensional arrays using pointers in C. You will probably find that your solution does not generalize to multidimensional arrays. Why not? If you are familiar with C++, show how to use its `class` facilities to solve the problem.

**7.19** Consider the following C declaration, compiled on a 32-bit Pentium machine.

```
struct {
 int n;
 char c;
} A[10][10];
```

If the address of `A[0][0]` is 1000 (decimal), what is the address of `A[3][7]`?

**7.20** Consider the following Pascal variable declarations.

```
var A : array [1..10, 10..100] of real;
 i : integer;
 x : real;
```

Assume that a real number occupies eight bytes and that `A`, `i`, and `x` are global variables. In something resembling assembly language for a RISC machine, show the code that a reasonable compiler would generate for the following assignment: `x := A[3,i]`. Explain how you arrived at your answer.

- 7.21** Suppose  $A$  is a  $10 \times 10$  array of (4-byte) integers, indexed from  $[0][0]$  through  $[9][9]$ . Suppose further that the address of  $A$  is currently in register  $r1$ , the value of integer  $i$  is currently in register  $r2$ , and the value of integer  $j$  is currently in register  $r3$ .

Give pseudo-assembly language for a code sequence that will load the value of  $A[i][j]$  into register  $r1$  (a) assuming that  $A$  is implemented using (row-major) contiguous allocation; (b) assuming that  $A$  is implemented using row pointers. Each line of your pseudocode should correspond to a single instruction on a typical modern machine. You may use as many registers as you need. You need not preserve the values in  $r1$ ,  $r2$ , and  $r3$ . You may assume that  $i$  and  $j$  are in bounds, and that addresses are 4 bytes long.

Which code sequence is likely to be faster? Why?

- 7.22** In Examples 7.69 and 7.70, show the code that would be required to access  $A[i, j, k]$  if subscript bounds checking were required.
- 7.23** Pointers and recursive type definitions complicate the algorithm for determining structural equivalence of types. Consider, for example, the following definitions.

```
type A = record
 x : pointer to B
 y : real
type B = record
 x : pointer to A
 y : real
```

The simple definition of structural equivalence given in Section 7.2.1 (expand the subparts recursively until all you have is a string of built-in types and type constructors; then compare them) does not work: we get an infinite expansion (type  $A = \text{record } x : \text{pointer to record } x : \text{pointer to record } x : \text{pointer to record } \dots$ ). The obvious reinterpretation is to say two types  $A$  and  $B$  are equivalent if any sequence of field selections, array subscripts, pointer dereferences, and other operations that takes one down into the structure of  $A$ , and that ends at a built-in type, always ends at the same built-in type when used to dive into the structure of  $B$  (and encounters the same field names along the way). Under this reinterpretation,  $A$  and  $B$  above have the same type. Give an algorithm based on this reinterpretation that could be used in a compiler to determine structural equivalence. (*Hint:* The fastest approach is due to J. Král [Krá73]. It is based on the algorithm used to find the smallest deterministic finite automaton that accepts a given regular language. This algorithm was outlined in Example 2.13 [page 53]; details can be found in any automata theory textbook [e.g., [HMU01]].)

- 7.24** Explain the meaning of the following C declarations.

```
double *a[n];
double (*b)[n];
```

```
double (*c[n])();
double (*d())[n];
```

- 7.25** In Ada 83, as in Pascal, pointers (access variables) can point only to objects in the heap. Ada 95 allows a new kind of pointer, the `access all` type, to point to other objects as well, provided that those objects have been declared to be aliased:

```
type int_ptr is access all Integer;
foo : aliased Integer;
ip : int_ptr;
...
ip := foo'Access;
```

The `'Access` attribute is roughly equivalent to C’s “address of” (`&`) operator. How would you implement `access all` types and aliased objects? How would your implementation interact with automatic garbage collection (assuming it exists) for objects in the heap?

- 7.26** As noted in Section 7.7.2, Ada 95 forbids an `access all` pointer from referring to any object whose lifetime is briefer than that of the pointer’s type. Can this rule be enforced completely at compile time? Why or why not?
- 7.27** In the discussion of pointers in Section 7.7, we assumed implicitly that every pointer into the heap points to the *beginning* of a dynamically allocated block of storage. In some languages, including Algol 68 and C, pointers may also point to data *inside* a block in the heap. If you were trying to implement dynamic semantic checks for dangling references or, alternatively, automatic garbage collection, how would your task be complicated by the existence of such “internal pointers”?
- 7.28** (a) A tracing garbage collector in a typesafe language can find and reclaim all *unreachable* objects. It will not necessarily reclaim all *useless* objects—those that will never be used again. Explain.  
 (b) With future technology, might it be possible to design a garbage collector that will reclaim all useless objects? Again, explain.
- 7.29** (a) Occasionally one encounters the suggestion that a garbage-collected language should provide a `delete` operation as an optimization: by explicitly `delete-ing` objects that will never be used again, the programmer might save the garbage collector the trouble of finding and reclaiming those objects automatically, thereby improving performance. What do you think of this suggestion? Explain.  
 (b) Alternatively, one might allow the programmer to “tenure” an object, so that it will never be a candidate for reclamation. Is this a good idea?
- 7.30** In Example 7.96 we noted that reference counts can be used to reclaim tombstones, failing only when the programmer neglects to manually delete

the object to which a tombstone refers. Explain how to leverage this observation to catch memory leaks at run time. Does your solution work in all cases? Explain.

- 7.31 In Example 7.96 we also noted that functional languages can safely use reference counts if the lack of an assignment statement prevents them from introducing circularity. This isn't strictly true: constructs like the Lisp `letrec` can also be used to make cycles. How might you address this problem?
- 7.32 Here is a skeleton for the standard quicksort algorithm in Haskell:

```
quicksort [] = []
quicksort (a : l) = quicksort [...] ++ [a] ++ quicksort [...]
```

The `++` operator denotes list concatenation (similar to `@` in ML). The `:` operator is equivalent to ML's `::` or Lisp's `cons`. Show how to express the two elided expressions as list comprehensions.

© 7.33–7.37 In More Depth.

## 7.13 Explorations

- 7.38 Some language definitions specify a particular representation for data types in memory, while others specify only the semantic behavior of those types. For languages in the latter class, some implementations guarantee a particular representation, while others reserve the right to choose different representations in different circumstances. Which approach do you prefer? Why?
- 7.39 If you have access to a compiler that provides optional dynamic semantic checks for out-of-bounds array subscripts, use of an inappropriate record variant, and/or dangling or uninitialized pointers, experiment with the cost of these checks. How much do they add to the execution time of programs that make a significant number of checked accesses? Experiment with different levels of optimization (code improvement) to see what effect it has on the overhead of checks.
- 7.40 Investigate the *typestate* mechanism employed by Strom et al. in the Hermes programming language [SBG<sup>+</sup>91]. Discuss its relationship to the notion of definite assignment in Java and C# (Section 6.1.3).
- 7.41 Investigate the notion of *type conformance*, employed by Black et al. in the Emerald programming language [BHJ<sup>+</sup>87]. Discuss how conformance relates to the type inference of ML and to the class-based typing of object-oriented languages.
- 7.42 Write a library package that might be used by a language implementation to manage sets of elements drawn from a very large base type (e.g., `integer`). You should support membership tests, union, intersection, and difference.

Does your package allocate memory from the heap? If so, what would a compiler that assumed the use of your package need to do to make sure that space was reclaimed when no longer needed?

- 7.43 Learn about SETL [SDDS86], a programming language based on sets, designed by Jack Schwartz of New York University. List the mechanisms provided as built-in set operations. Compare this list with the set facilities of other programming languages. What data structure(s) might a SETL implementation use to represent sets in a program?
- 7.44 Implement your favorite garbage collection algorithm in Ada 95. Alternatively, implement a special pointer class in C++ for which storage is garbage-collected. You'll want to use templates (generics) so that your class can be instantiated for arbitrary pointed-to types.
- 7.45 Experiment with the cost of garbage collection in your favorite language implementation. What kind of collector does it use? Can you create artificial programs for which it performs particularly well or poorly? (*Hint:* Check to see if your machine and operating system allow user-level programs to access a low-cost, high-resolution clock register.)

⌚ 7.46–7.48 In More Depth.

## 7.14 Bibliographic Notes

References to general information on the various programming languages mentioned in this chapter can be found in Appendix A, and in the Bibliographic Notes for Chapters 1 and 6. Welsh, Sneeringer, and Hoare [WSH77] provide a critique of the original Pascal definition, with a particular emphasis on its type system. Tanenbaum's comparison of Pascal and Algol 68 also focuses largely on types [Tan78]. Cleaveland [Cle86] provides a book-length study of many of the issues in this chapter. Pierce [Pie02] provides a formal and detailed modern coverage of the subject. The ACM Special Interest Group on Programming Languages launched a biennial workshop on *Types in Language Design and Implementation* in 2003.

What we have referred to as the denotational model of types originates with Hoare [DDH72]. Denotational formulations of the overall semantics of programming languages are discussed in the Bibliographic Notes for Chapter 4. A related but distinct body of work uses algebraic techniques to formalize data abstraction; key references include Guttag [Gut77] and Goguen et al. [GTW78]. Milner's original paper [Mil78] is the seminal reference on type inference in ML. Mairson [Mai90] proves that the cost of unifying ML types is  $O(2^n)$ , where  $n$  is the length of the program. Fortunately, the cost is linear in the size of the program's type expressions, so the worst case arises only in programs whose semantics are too complex for a human being to understand anyway.

Hoare [Hoa75] discusses the definition of recursive types under a reference model of variables. Cardelli and Wegner survey issues related to polymorphism, overloading, and abstraction [CW85]. The new Character Model standard for the World Wide Web provides a remarkably readable introduction to the subtleties and complexities of international character sets [Wor05]. Conway's game of Life, which appeared in Figure 7.8, was first described by Martin Gardner in his "Mathematical Games" column in *Scientific American* [Gar70].

Tombstones are due to Lomet [Lom75, Lom85]. Locks and keys are due to Fischer and LeBlanc [FL80]. The latter also discuss how to check for various other dynamic semantic errors in Pascal, including those that arise with variant records. Constant-space (pointer-reversing) mark-and-sweep garbage collection is due to Schorr and Waite [SW67]. Stop-and-copy collection was developed by Fenichel and Yochelson [FY69], based on ideas due to Minsky. Deutsch and Bobrow [DB76] describe an *incremental* garbage collector that avoids the "stop-the-world" phenomenon. Wilson and Johnstone [WJ93] describe a more recent incremental collector. The conservative collector described at the end of Section 7.7.3 is due to Boehm and Weiser [BW88]. Cohen [Coh81] surveys garbage-collection techniques as of 1981; Wilson [Wil92b] and Jones and Lins [JL96] provide more recent views.

# 8

## Subroutines and Control Abstraction

In the introduction to Chapter 3, we defined *abstraction* as a process by which the programmer can associate a name with a potentially complicated program fragment, which can then be thought of in terms of its purpose or function, rather than in terms of its implementation. We sometimes distinguish between *control abstraction*, in which the principal purpose of the abstraction is to perform a well-defined operation, and *data abstraction*, in which the principal purpose of the abstraction is to represent information.<sup>1</sup> We will consider data abstraction in more detail in Chapter 9.

Subroutines are the principal mechanism for control abstraction in most programming languages. A subroutine performs its operation on behalf of a *caller*, who waits for the subroutine to finish before continuing execution. Most subroutines are parameterized: the caller passes arguments that influence the subroutine's behavior, or provide it with data on which to operate. Arguments are also called *actual parameters*. They are mapped to the subroutine's *formal parameters* at the time a call occurs. A subroutine that returns a value is usually called a *function*. A subroutine that does not return a value is usually called a *procedure*. Most languages require subroutines to be declared before they are used, though a few (including Fortran, C, and Lisp) do not. Declarations allow the compiler to verify that every call to a subroutine is consistent with the declaration—that is, that it passes the right number and types of arguments.

As noted in Section 3.2.2, the storage consumed by parameters and local variables can in most languages be allocated on a stack. We therefore begin this chapter, in Section 8.1, by reviewing the layout of the stack. We then turn in Section 8.2 to the *calling sequences* that serve to maintain this layout. In the process, we revisit the use of static chains to access nonlocal variables in nested subroutines and consider (on the PLP CD) an alternative mechanism, known as a *display*, that serves

---

**I** The distinction between control and data abstraction is somewhat fuzzy, because the latter usually encapsulates not only information, but also the operations that access and modify that information. Put another way, most data abstractions include control abstraction.

a similar purpose. We also consider subroutine inlining and the representation of closures. To illustrate some of the possible implementation alternatives, we present (again on the PLP CD) a pair of case studies: the SGI MIPSpro C compiler for the MIPS instruction set, and the GNU gpc Pascal compiler for the x86 instruction set, as well as the *register window* mechanism of the Sparc instruction set.

In Section 8.3 we look more closely at subroutine parameters. We consider parameter-passing *modes*, which determine the operations that a subroutine can apply to its formal parameters and the effects of those operations on the corresponding actual parameters. We also consider conformant arrays, named and default parameters, variable numbers of arguments, and function return mechanisms. In Section 8.4 we turn to *generic* subroutines and modules (classes), which support explicit parametric polymorphism, as defined in Section 3.6.3. Where conventional parameters allow a subroutine to operate on many different values, generic parameters allow it to operate on data of many different *types*.

In Section 8.5, we consider the handling of exceptional conditions. While exceptions can sometimes be confined to the current subroutine, in the general case they require a mechanism to “pop out of” a nested context without returning, so that recovery can occur in the calling context. Finally, in Section 8.6, we consider *coroutines*, which allow a program to maintain two or more execution contexts, and to switch back and forth among them. Coroutines can be used to implement iterators (Section 6.5.3), but they have other uses as well, particularly in simulation and in server programs. In Chapter 12 we will use them as the basis for concurrent (“quasiparallel”) threads.

## 8.1 Review of Stack Layout

### EXAMPLE 8.1

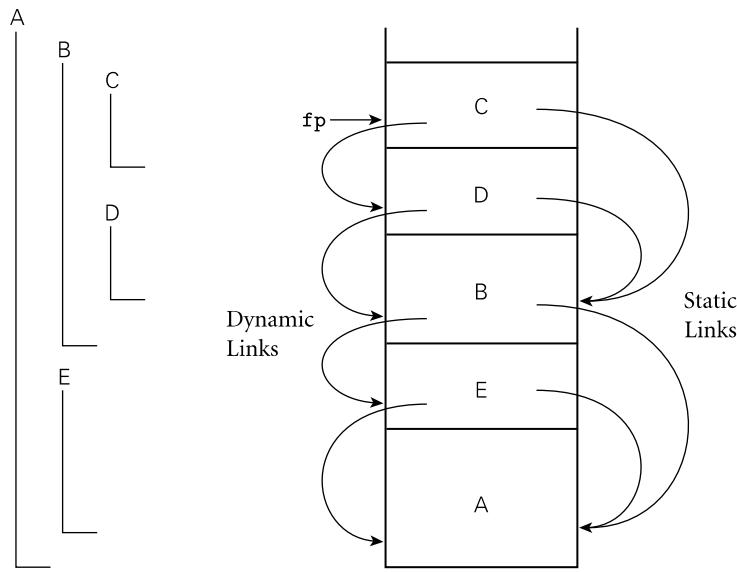
Layout of run-time stack  
(reprise)

In Section 3.2.2 we discussed the allocation of space on a subroutine call stack (Figure 3.2, page 110). Each routine, as it is called, is given a new *stack frame*, or *activation record*, at the top of the stack. This frame may contain arguments and/or return values, bookkeeping information (including the return address and saved registers), local variables, and/or temporaries. When a subroutine returns, its frame is popped from the stack. ■

At any given time, the *stack pointer* register contains the address of either the last used location at the top of the stack or the first unused location, depending on convention. The *frame pointer* register contains an address within the frame. Objects in the frame are accessed via displacement addressing with respect to the frame pointer. If the size of an object (e.g., a local array) is not known at compile time, then the object is placed in a variable-size area at the top of the frame; its address and dope vector are stored in the fixed-size portion of the frame, at a statically known offset from the frame pointer (Figure 7.7, page 354). If there are no variable-size objects, then every object within the frame has a statically known offset from the stack pointer, and the implementation may dispense with

### EXAMPLE 8.2

Offsets from frame pointer



**Figure 8.1** Example of subroutine nesting, taken from Figure 3.5. Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.

the frame pointer, freeing up a register for other use. If the size of an argument is not known at compile time, then the argument may be placed in a variable-size portion of the frame *below* the other arguments, with its address and dope vector at known offsets from the frame pointer. Alternatively, the caller may simply pass a temporary address and dope vector, counting on the called routine to copy the argument into the variable-size area at the top of the frame. ■

In a language with nested subroutines and static scoping (e.g., Pascal, Ada, ML, Common Lisp, or Scheme), objects that lie in surrounding subroutines and are thus neither local nor global can be found by maintaining a *static chain* (Figure 8.1). Each stack frame contains a reference to the frame of the lexically surrounding subroutine. This reference is called the *static link*. By analogy, the saved value of the frame pointer, which will be restored on subroutine return, is called the *dynamic link*. The static and dynamic links may or may not be the same, depending on whether the current routine was called by its lexically surrounding routine, or by some other routine nested in that surrounding routine. ■

Whether or not a subroutine is called directly by the lexically surrounding routine, we can be sure that the surrounding routine is active; there is no other way that the current routine could have been visible, allowing it to be called. Consider for example, the subroutine nesting shown in Figure 8.1. If subroutine D is called directly from B, then clearly B's frame will already be on the stack. How else could D be called? It is not visible in A or E, because it is nested inside of

#### EXAMPLE 8.3

Static and dynamic links

#### EXAMPLE 8.4

Visibility of nested routines

B. A moment's thought makes clear that it is only when control enters B (placing B's frame on the stack) that D comes into view. It can therefore be called by C, or by any other routine (not shown) that is nested inside of C or D, but only because these are also within B.

## 8.2 Calling Sequences

In Section 3.2.2 we also mentioned that maintenance of the subroutine call stack is the responsibility of the *calling sequence*—the code executed by the caller immediately before and after a subroutine call—and of the *prologue* (code executed at the beginning) and *epilogue* (code executed at the end) of the subroutine itself. Sometimes the term “calling sequence” is used to refer to the combined operations of the caller, the prologue, and the epilogue.

Tasks that must be accomplished on the way into a subroutine include passing parameters, saving the return address, changing the program counter, changing the stack pointer to allocate space, saving registers (including the frame pointer) that contain important values and that may be overwritten by the callee, changing the frame pointer to refer to the new frame, and executing initialization code for any objects in the new frame that require it. Tasks that must be accomplished on the way out include passing return parameters or function values, executing finalization code for any local objects that require it, deallocating the stack frame (restoring the stack pointer), restoring other saved registers (including the frame pointer), and restoring the program counter. Some of these tasks (e.g., passing parameters) must be performed by the caller, because they differ from call to call. Most of the tasks, however, can be performed either by the caller or the callee. In general, we will save space if the callee does as much work as possible: tasks performed in the callee appear only once in the target program, but tasks performed in the caller appear at every call site, and the typical subroutine is called in more than one place.

### Saving and Restoring Registers

Perhaps the trickiest division-of-labor issue pertains to saving registers. As we noted in Section 5.5.2, the ideal approach is to save precisely those registers that are both in use in the caller and needed for other purposes in the callee. Because of separate compilation, however, it is difficult (though not impossible) to determine this intersecting set. A simpler solution is for the caller to save all registers that are in use, or for the callee to save all registers that it will overwrite.

Calling sequence conventions for many processors, including the MIPS and x86 described in the case studies of Section 8.2.2, strike something of a compromise: registers not reserved for special purposes are divided into two sets of approximately equal size. One set is the caller's responsibility, the other is the callee's responsibility. A callee can assume that there is nothing of value in any of the registers in the caller-saves set; a caller can assume that no callee will destroy

the contents of any registers in the callee-saves set. In the interests of code size, the compiler uses the callee-saves registers for local variables and other long-lived values whenever possible. It uses the caller-saves set for transient values, which are less likely to be needed across calls. The result of these conventions is that the caller-saves registers are seldom saved by either party: the callee knows that they are the caller's responsibility, and the caller knows that they don't contain anything important.

### Maintaining the Static Chain

In languages with nested subroutines, at least part of the work required to maintain the static chain must be performed by the caller, rather than the callee, because this work depends on the lexical nesting depth of the caller. The standard approach is for the caller to compute the callee's static link and to pass it as an extra, hidden parameter. The following subcases arise.

1. The callee is nested (directly) inside the caller. In this case, the callee's static link should refer to the caller's frame. The caller therefore passes its own frame pointer as the callee's static link.
2. The callee is  $k \geq 0$  scopes "outward"—closer to the outer level of lexical nesting. In this case, all scopes that surround the callee also surround the caller (otherwise the callee would not be visible). The caller dereferences its own static link  $k$  times and passes the result as the callee's static link.

### A Typical Calling Sequence

#### EXAMPLE 8.5

A typical calling sequence

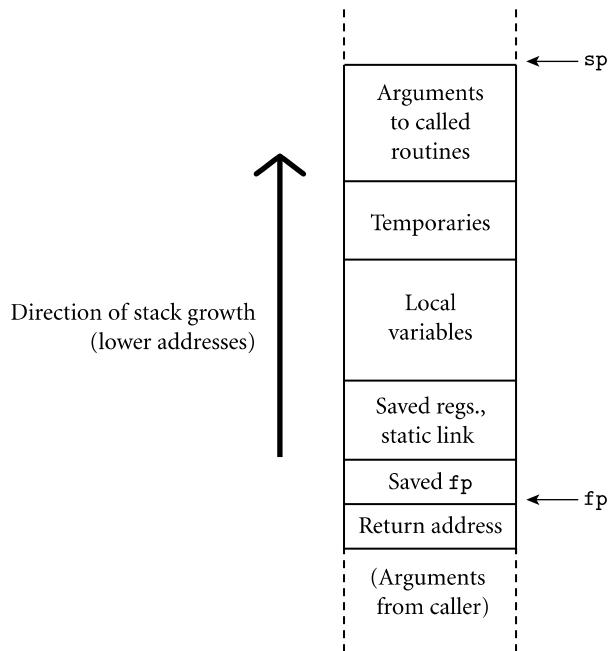
Figure 8.2 shows one plausible layout for a stack frame, consistent with Figure 3.2. The stack pointer (*sp*) points to the first unused location on the stack (or the last used location, depending on the compiler and machine). The frame pointer (*fp*) points to a location near the bottom of the frame. Space for all arguments is reserved in the stack, even if the compiler passes some of them in registers (the callee will need a place to save them if it calls a nested routine).

To maintain this stack layout, the calling sequence might operate as follows. The caller

1. saves any caller-saves registers whose values will be needed after the call.
2. computes the values of arguments and moves them into the stack or registers.
3. computes the static link (if this is a language with nested subroutines) and passes it as an extra, hidden argument.
4. uses a special subroutine call instruction (sometimes called "branch and link") to jump to the subroutine, simultaneously passing the return address on the stack or in a register.

In its prologue, the callee

1. allocates a frame by subtracting an appropriate constant from the *sp*.



**Figure 8.2 A typical stack frame.** Though we draw it growing upward on the page, the stack actually grows downward toward lower addresses on most machines. Arguments are accessed at positive offsets from the `fp`. Local variables and temporaries are accessed at negative offsets from the `fp`. Arguments to be passed to called routines are assembled at the top of the frame, using positive offsets from the `sp`.

2. saves the old frame pointer into the stack, and assigns it an appropriate new value.
3. saves any callee-saves registers that may be overwritten by the current routine (including the static link and return address, if they were passed in registers).

After the subroutine has completed, the epilogue

1. moves the return value (if any) into a register or a reserved location in the stack.
2. restores callee-saves registers if needed.
3. restores the `fp` and the `sp`.
4. jumps back to the return address.

Finally, the caller

1. moves the return value to wherever it is needed.
2. restores caller-saves registers if needed.

### ***Special Case Optimizations***

Many parts of the calling sequence, prologue, and epilogue can be omitted in common cases. If the hardware passes the return address in a register, then a *leaf routine* (a subroutine that makes no additional calls before returning)<sup>2</sup> can simply leave it there; it does not need to save it in the stack. Likewise it need not save the static link or any caller-saves registers.

A subroutine with no local variables and nothing to save or restore may not even need a stack frame on a RISC machine. The simplest subroutines (e.g., library routines to compute the standard mathematical functions) may not touch memory at all, except to fetch instructions: they may take their arguments in registers, compute entirely in (caller-saves) registers, call no other routines, and return their results in registers. As a result they may be extremely fast.

#### **8.2.1 Displays**

One disadvantage of static chains is that access to an object in a scope  $k$  levels out requires that the static chain be dereferenced  $k$  times. If a local object can be loaded into a register with a single (displacement mode) memory access, an object  $k$  levels out will require  $k + 1$  memory accesses. This number can be reduced to a constant by use of a *display*.

##### **IN MORE DEPTH**

A display is a small array that replaces the static chain. The  $j$ th element of the display contains a reference to the frame of the most recently active subroutine at lexical nesting level  $j$ . If the currently active routine is nested  $i > 3$  levels deep, then elements  $i - 1$ ,  $i - 2$ , and  $i - 3$  of the display contain the values that would have been the first three links of the static chain. An object  $k$  levels out can be found at a statically known offset from the address stored in element  $j = i - k$  of the display.

For most programs the cost of maintaining a display in the subroutine calling sequence tends to be slightly higher than that of maintaining a static chain. At the same time, the cost of dereferencing the static chain has been reduced by modern compilers, which tend to do a good job of caching the links in registers when appropriate. These observations, combined with the trend toward languages (those descended from C in particular) in which subroutines do not nest, have made displays less common today than they were in the 1970s.

---

**2** A leaf routine is so named because it is a leaf of the *subroutine call graph*, a data structure mentioned in Exercise 3.10.

### 8.2.2 Case Studies: C on the MIPS; Pascal on the x86

Calling sequences differ significantly from machine to machine and even compiler to compiler (though typically a hardware manufacturer publishes a suggested set of conventions for a given architecture, to promote interoperability among program components produced by different compilers). Some of the most significant differences can be found in a comparison of CISC and RISC conventions.

- Compilers for CISC machines tend to pass arguments on the stack; compilers for RISC machines tend to pass arguments in registers.
- Compilers for CISC machines usually dedicate a register to the frame pointer; compilers for RISC machines often do not.
- Compilers for CISC machines often rely on special purpose instructions to implement parts of the calling sequence; available instructions on a RISC machine are typically much simpler.

The use of the stack to pass arguments reflects the technology of the 1970s, when register sets were significantly smaller and memory access was significantly faster (in comparison to processor speed) than is the case today. Most CISC instruction sets include push and pop instructions that combine a store or load with automatic update of the stack pointer. The push instruction, in particular, was traditionally used to pass arguments to subroutines, effectively allocating stack space on demand. The resulting instability in the value of the *sp* made it difficult (though not impossible) to use that register as the base for access to local variables. A separate frame pointer made code generation easier and, perhaps more important, made it practical to locate local variables from within a simple symbolic debugger.

#### IN MORE DEPTH

On the PLP CD we look in some detail at the stack layout conventions and calling sequences of a representative pair of compilers: the SGI MIPSpro C compiler for the 64-bit MIPS architecture, and the GNU Pascal compiler (*gpc*) for the 32-bit x86. The former illustrates the heavy use of registers on modern RISC machines. The latter, while adjusted somewhat to reflect modern implementations of the x86, still retains vestiges of its CISC ancestry, with heavier use of the stack. It also illustrates the use of the static chain to accommodate nested subroutines, and the creation of closures when such routines are passed as parameters.

### 8.2.3 Register Windows

As an alternative to saving and restoring registers on subroutine calls and returns, the original Berkeley RISC machines [PD80, Pat85] introduced a hardware

mechanism known as *register windows*. The basic idea is to map the ISA's limited set of register names onto some subset (window) of a much larger collection of physical registers, and to change the mapping when making subroutine calls. Old and new mappings overlap a bit, allowing arguments to be passed (and function results returned) in the intersection.

#### IN MORE DEPTH

We consider register windows in more detail on the PLP CD. They have appeared in several commercial processors, most notably the Sun Sparc and the Intel IA-64 (Itanium).

### 8.2.4 In-Line Expansion

As an alternative to stack-based calling conventions, many language implementations allow certain subroutines to be expanded in-line at the point of call. In-line expansion avoids a variety of overheads, including space allocation, branch delays from the call and return, maintaining the static chain or display, and (often) saving and restoring registers. It also allows the compiler to perform code improvements such as global register allocation and common subexpression elimination across the boundaries between subroutines, something that most compilers can't do otherwise.

In many implementations the compiler chooses which subroutines to expand in-line and which to compile conventionally. In some languages, the programmer can suggest that particular routines be in-lined. In C++ and C99, the keyword `inline` can be prefixed to a function declaration:

```
inline int max(int a, int b) {return a > b ? a : b;}
```

In Ada, the programmer can request in-line expansion with a *significant comment*, or *pragma*:

#### DESIGN & IMPLEMENTATION

##### Hints and directives

As noted in the sidebar on page 292, the decision to make `inline` a semantically neutral hint in C (and the similar treatment of most pragmas in Ada) acknowledges that while programmer suggestions may sometimes improve the quality of generated code, advances in compiler technology may change the balance in the future. By contrast, the use of pointer arithmetic in place of array subscripts, as discussed in the sidebar on page 377, is more of a *directive* than a hint, and may complicate the generation of high-quality code from legacy programs.

#### EXAMPLE 8.6

Requesting an `inline` subroutine

```

function max(a, b : integer) return integer is
begin
 if a > b then return a; else return b; end if;
end max;
pragma inline(max);

```

Ada provides a large variety of pragmas. As a rule, they do not affect the semantics of the program; they simply offer suggestions (“hints”) to the compiler. In some cases (e.g., the `elaborate` pragma, which controls the order in which headers for library packages are examined), the compiler is required to follow the suggestion. In other cases (including `inline`), it is not. The `inline` keyword in C++ and C99 is likewise a suggestion; the compiler can ignore it. ■

In Section 6.6.2 we noted the similarity between in-line expansion and macros, but argued that the former is semantically preferable. In fact, in-line expansion is semantically neutral: it is purely an implementation technique, with no effect on the meaning of the program. In comparison to real subroutine calls, in-line expansion has the obvious disadvantage of increasing code size, since the entire body of the subroutine appears at every call site. In-line expansion is also not an option in the general case for recursive subroutines. For the occasional case in which a recursive call is possible but unlikely, it may be desirable to generate a true recursive subroutine, but to expand one instance of it in-line at each call site. Consider the following C routine for use in hash-table lookup.

```

range_t bucket_contents(bucket *b, domain_t x)
{
 if (b->key == x)
 return b->val;
 else if (b->next == 0)
 return ERROR;
 else
 return bucket_contents(b->next, x);
}

```

We can expand this code in-line if we make the nested invocation a true subroutine call. Since most hash chains are only one bucket long, the nested call will usually not occur. The in-line expansion will be faster than a true subroutine call, and smaller than in-line code that handles the general case. ■

## DESIGN & IMPLEMENTATION

### Inline and modularity

Probably the most important argument for in-line expansion is that it allows programmers to adopt a very modular programming style, with lots of tiny subroutines, without sacrificing performance. This modular programming style is essential for object-oriented languages, as we shall see in Chapter 9.

 **CHECK YOUR UNDERSTANDING**

1. What is a subroutine *calling sequence*? What does it do? What is meant by the subroutine *prologue* and *epilogue*?
2. How do calling sequences typically differ in CISC and RISC compilers?
3. Describe how to maintain the *static chain* during a subroutine call.
4. What is a *display*? How does it differ from a static chain?
5. What are the purposes of the *stack pointer* and *frame pointer* registers? Why does a subroutine often need both?
6. Why do RISC machines typically pass subroutine parameters in registers rather than on the stack?
7. Why do subroutine calling conventions often give the caller responsibility for saving half the registers and the callee responsibility for saving the other half?
8. If work can be done in either the caller or the callee, why do we typically prefer to do it in the callee?
9. Why do compilers typically allocate space for arguments in the stack, even when they pass them in registers?
10. List the optimizations that can be made to the subroutine calling sequence in important special cases (e.g., *leaf routines*).
11. How does an *in-line subroutine* differ from a *macro*?
12. Under what circumstances is it desirable to expand a subroutine in-line?

## 8.3 Parameter Passing

Most subroutines are parameterized: they take arguments that control certain aspects of their behavior, or specify the data on which they are to operate. Parameter names that appear in the declaration of a subroutine are known as *formal parameters*. Variables and expressions that are passed to a subroutine in a particular call are known as *actual parameters*. We have been referring to actual parameters as *arguments*. In the following two subsections, we discuss the most common parameter-passing *modes*, most of which are implemented by passing values, references, or closures. In Section 8.3.3 we will look at additional mechanisms, including conformant array parameters, missing and default parameters, named parameters, and variable-length argument lists. Finally in Section 8.3.4 we will consider mechanisms for returning values from functions.

**EXAMPLE 8.8**

Infix operators

As we noted in Section 6.1, most languages use a prefix notation for calls to user-defined subroutines, with the subroutine name followed by a parenthesized argument list. Lisp places the function name inside the parentheses, as in `(max a b)`. ML allows the programmer to specify that certain names represent infix operators, which appear between a pair of arguments:

```
infixr 8 tothe; (* exponentiation *)
fun x tothe 0 = 1.0
| x tothe n = x * (x tothe(n-1)); (* assume n >= 0 *)
```

The `infixr` declaration indicates that `tosome` will be a right-associative binary infix operator, at precedence level 8 (multiplication and division are at level 7, addition and subtraction at level 6). Fortran 90 also allows the programmer to define new infix operators, but it requires their names to be bracketed with periods (e.g., `A .cross . B`), and it gives them all the same precedence. Smalltalk uses infix (or “mixfix”) notation (without precedence) for all its operations. ■

**EXAMPLE 8.9**

Control abstraction in Lisp and Smalltalk

The uniformity of Lisp and Smalltalk syntax makes control abstraction particularly effective: user-defined subroutines (functions in Lisp, “messages” in Smalltalk) use the same style of syntax as built-in operations. As an example, consider `if...then...else`:

```
if a > b then max := a else max := b; (* Pascal *)
(if (> a b) (setf max a) (setf max b)) ; Lisp
(a > b) ifTrue: [max <- a] ifFalse: [max <- b]. "Smalltalk"
```

In Pascal or C it is clear that `if...then...else` is a built-in language construct: it does not look like a subroutine call. In Lisp and Smalltalk, on the other hand, the analogous conditional constructs are syntactically indistinguishable from user-defined operations. They are in fact defined in terms of simpler concepts, rather than being built-in, though they require a special mechanism to evaluate their arguments in normal, rather than applicative, order (Section 6.6.2). ■

### 8.3.1 Parameter Modes

In our discussion of subroutines so far, we have glossed over the semantic rules that govern parameter passing, and that determine the relationship between actual and formal parameters. Some languages, including C, Fortran, ML, and Lisp, define a single set of rules, which apply to all parameters. Other languages, including Pascal, Modula, and Ada, provide two or more sets of rules, corresponding to different parameter-passing *modes*. As in many aspects of language design, the semantic details are heavily influenced by implementation issues.

Suppose for the moment that `x` is a global variable in a language with a value model of variables, and that we wish to pass `x` as a parameter to subroutine `p`:

`p(x);`

**EXAMPLE 8.10**

Passing a subroutine argument

From an implementation point of view, we have two principal alternatives: we may provide  $p$  with a copy of  $x$ 's value, or we may provide it with  $x$ 's address. The two most common parameter-passing modes, called *call by value* and *call by reference*, are designed to reflect these implementations. ■

With value parameters, each actual parameter is assigned into the corresponding formal parameter when a subroutine is called; from then on, the two are independent. With reference parameters, each formal parameter introduces, within the body of the subroutine, a new name for the corresponding actual parameter. If the actual parameter is also visible within the subroutine under its original name (as will generally be the case if it is declared in a surrounding scope), then the two names are *aliases* for the same object, and changes made through one will be visible through the other. In most languages (Fortran is an exception, as we shall see) an actual parameter that is to be passed by reference must be an l-value; it cannot be the result of an arithmetic operation, or any other value without an address.

### **EXAMPLE 8.11**

Value and reference parameters

As a simple example, consider the following pseudocode.

```

x : integer -- global
procedure foo(y : integer)
 y := 3
 print x
 ...
 x := 2
 foo(x)
 print x

```

If  $y$  is passed to  $\text{foo}$  by value, then the assignment inside  $\text{foo}$  has no visible effect— $y$  is private to the subroutine—and the program prints 2 twice. If  $y$  is passed to  $\text{foo}$  by reference, then the assignment inside  $\text{foo}$  changes  $x$ — $y$  is just a local name for  $x$ —and the program prints 3 twice. ■

### **Variations on Value and Reference Parameters**

In Pascal, parameters are passed by value by default; they are passed by reference if preceded by the keyword `var` in their subroutine header's formal parameter list. Parameters in C are always passed by value, though the effect for arrays is un-

## DESIGN & IMPLEMENTATION

### Parameter modes

While it may seem odd to introduce parameter modes (a semantic issue) in terms of implementation, the distinction between value and reference parameters is fundamentally an implementation issue. Most languages with more than one mode (Ada is the principal exception) might fairly be characterized as an attempt to paste acceptable semantics onto the desired implementation, rather than to find an acceptable implementation of the desired semantics.

**EXAMPLE 8.12**

Emulating call-by-reference  
in C

usual: because of the interoperability of arrays and pointers in C (Section 7.7.1), what is passed by value is a pointer; changes to array elements accessed through this pointer are visible to the caller. To allow a called routine to modify a variable other than an array in the caller's scope, the C programmer must pass the address of the variable explicitly:

```
void swap(int *a, int *b) { int t = *a; *a = *b; *b = t; }
...
swap(&v1, &v2);
```

Fortran passes all parameters by reference, but it does not require that every actual parameter be an l-value. If a built-up expression appears in an argument list, the compiler creates a temporary variable to hold the value, and passes this variable by reference. A Fortran subroutine that needs to modify the values of its formal parameters without modifying its actual parameters must copy the values into local variables and modify those instead.

**Call by Sharing** In languages like Smalltalk, Lisp, ML, and Clu, which use a reference model of variables, an actual parameter is already a reference to an object. Instead of passing the value of the actual parameter or a reference to the actual parameter (neither of which makes sense), these languages provide a single parameter-passing mode in which the actual and formal parameters refer to the same object. Clu calls this mode *call by sharing*. For variables that are implemented as addresses, call by sharing is usually implemented by passing the address. For variables that refer to immutable objects (numbers, characters, etc.) and that are implemented as values, call by sharing is usually implemented by passing the value.

In Java, parameters of primitive types are passed by value; object parameters are passed by sharing. A similar approach is the default in C#, but because the language allows users to create both value (`struct`) and reference (`class`) types, both cases are considered call by value. When desired, parameters can be passed by reference instead, by labeling both formal and *actual* parameters with the `ref` or `out` keyword. Both of these modes are implemented by passing an address; they differ in that a `ref` argument must be *definitely assigned* prior to the call, as described in Section 6.1.3; an `out` argument need not. If a variable of `class` (reference) type is passed as a `ref` or `out` parameter, it may end up referring to a different value as a result of subroutine execution.

**The Ambiguity of Call by Reference** In a language that provides both value and reference parameters (e.g., Pascal or Modula), there are two principal reasons why the programmer might choose one over the other. First, if the called routine is supposed to change the value of an actual parameter, then the programmer must pass the parameter by reference. Conversely, to ensure that the called routine cannot modify the parameter, the programmer can pass the parameter by value. Second, the implementation of value parameters requires copying actuals to formals, a potentially time-consuming operation when arguments are large. Reference parameters can be implemented simply by passing an address. (Of

course, accessing a parameter that is passed by reference requires an extra level of indirection. If the parameter is used often enough, the cost of this indirection may outweigh the cost of copying the actual parameter.)

The potential inefficiency of large value parameters sometimes prompts programmers to pass an argument by reference when passing by value would be semantically more appropriate. Pascal programmers, for example, were commonly taught to use `var` (reference) parameters both for arguments that need to be modified and for arguments that are very large. Unfortunately, the latter justification often leads to buggy code, in which a subroutine modifies an actual parameter that the caller meant to leave unchanged.

**Read-Only Parameters** To combine the efficiency of reference parameters and the safety of value parameters, Modula-3 provides a `READONLY` parameter mode. Any formal parameter whose declaration is preceded by `READONLY` cannot be changed by the called routine: the compiler prevents the programmer from using that formal parameter on the left-hand side of any assignment statement, reading it from a file, or passing it by reference to any other subroutine. Small `READONLY` parameters are generally implemented by passing a value; larger `READONLY` parameters are implemented by passing an address. As in Fortran, a Modula-3 compiler will create a temporary variable to hold the value of any built-up expression passed as a large `READONLY` parameter.

The equivalent of `READONLY` parameters is also available in C, which allows any variable or parameter declaration to be preceded by the keyword `const`. `Const` variables are “elaboration-time constants,” as described in Section 3.2. `Const` parameters are particularly useful when passing addresses:

```
void append_to_log(const huge_record *r) { ...
...
append_to_log(&my_record);
```

Here the keyword `const` applies to the record to which `r` points;<sup>3</sup> the caller must pass the address of its record explicitly, but can be assured that the callee will not change the record’s contents. ■

One traditional problem with parameter modes—and with the `READONLY` mode in particular—is that they tend to confuse the key pragmatic issue (does the implementation pass a value or a reference?) with two semantic issues: is the callee allowed to change the formal parameter and, if so, will the changes be reflected in the actual parameter? C keeps the pragmatic issue separate, by forcing the programmer to pass references explicitly with pointers. Still, its `const` mode

---

<sup>3</sup> Following the usual rules for parsing C declarations (page 377) `r` is a pointer to a `huge_record` whose value is constant. If we wanted `r` to be a constant that points to a `huge_record`, we should need to say `huge_record * const r`.

serves double duty: is the intent of `const foo *p` to protect the actual parameter from change, or to document the fact that the subroutine thinks of the formal parameter as a constant rather than a variable, or both?

### Parameter Modes in Ada

Ada provides three parameter-passing modes, called `in`, `out`, and `in out`. `In` parameters pass information from the caller to the callee; they can be read by the callee but not written. `Out` parameters pass information from the callee to the caller. In Ada 83 they can be written by the callee but not read; in Ada 95 they can be both read and written, but they begin their life uninitialized. `In out` parameters pass information in both directions; they can be both read and written. Changes to `out` or `in out` parameters always change the actual parameter.

For parameters of scalar and access (pointer) types, Ada specifies that all three modes are to be implemented by copying values. For these parameters, then, `in` is call by value, `out` is what some authors call *call by result* (the value of the formal parameter is copied into the actual parameter when the subroutine returns), and `in out` is *call by value/result*, a mode first introduced in Algol-W. For parameters of most constructed types, however, Ada specifically permits an implementation to pass either values or addresses. In most languages, these two different mechanisms would lead to different semantics: changes made to an `in out` parameter that is passed as an address will affect the actual parameter immediately; changes made to an `in out` parameter that is passed as a value will not affect the actual parameter until the subroutine returns. Return for a moment to Example 8.11:

#### EXAMPLE 8.14

Reference and value/result parameters

```
x : integer -- global
procedure foo(y : integer)
 y := 3
 print x
 ...
 x := 2
 foo(x)
 print x
```

We already noted that if `y` is passed by reference the program will print 3 twice. If `y` is passed by value/result, it will print 2 and then 3. ■

One possible way to hide the distinction between reference and value/result would be to outlaw the creation of aliases, as Euclid does. Ada takes a simpler tack: a program that can tell the difference between value and address-based implementations of (nonscalar, nonpointer) `in out` parameters is said to be “*erroneous*”—incorrect, but in a way that the language implementation is not required to catch.

Ada’s semantics for parameter passing allow a single set of modes to be used not only for subroutine parameters, but also for communication among concurrently executing tasks (to be discussed in Chapter 12). When tasks are executing

on separate machines, with no memory in common, passing the address of an actual parameter is not a practical option. Most Ada compilers pass large arguments to subroutines as addresses; they pass them to the entry points of tasks by copying.

### **References in C++**

Programmers who switch to C after some experience with Pascal, Modula, or Ada (or with call by sharing in Java or Lisp) are often frustrated by C's lack of reference parameters. As noted above, one can always arrange to modify an object by passing its address, but then the formal parameter is a pointer, and must be explicitly dereferenced whenever it is used. C++ addresses this problem by introducing an explicit notion of a *reference*. Reference parameters are specified by preceding their name with an ampersand in the header of the function:

```
void swap(int &a, int &b) { int t = a; a = b; b = t; }
```

In the code of this *swap* routine, *a* and *b* are *ints*, not pointers to *ints*; no dereferencing is required. Moreover, the caller passes as arguments the variables whose values are to be swapped, rather than passing their addresses. ■

As in C, a C++ parameter can be declared to be *const* to ensure that it is not modified. For large types, *const* reference parameters in C++ provide the same combination of speed and safety found in the *READONLY* parameters of Modula-3: they can be passed by address, but cannot be changed by the called routine.

References in C++ see their principal use as parameters, but they can appear in other contexts as well. Any variable can be declared to be a reference:

```
int i;
int &j = i;
...
i = 2;
j = 3;
cout << i; // prints 3
```

Here *j* is a reference to (an alias for) *i*. The initializer in the declaration is required; it identifies the object for which *j* is an alias. Moreover it is not possible later to change the object to which *j* refers; it will always refer to *i*.

Any change to *i* or *j* can be seen by reading the other. Most C++ compilers implement references with addresses. In this example, *i* will be assigned a location that contains an integer, while *j* will be assigned a location that contains the address of *i*. Despite their different implementation, however, there is no semantic difference between *i* and *j*; the exact same operations can be applied to either, with precisely the same results. ■

While there is seldom any reason to create aliases on purpose in straight-line code, references in C++ are highly useful for at least one purpose other than parameters: namely function returns. Some objects—file buffers, for example—do not support a copy operation, and therefore cannot be passed or returned by

### **EXAMPLE 8.15**

Reference parameters in C++

### **EXAMPLE 8.16**

References as aliases in C++

value. One can always return a pointer, but just as with subroutine parameters, the subsequent dereferencing operations can be cumbersome.

**EXAMPLE 8.17**

Returning a reference from a function

```
cout << a << b << c;
```

is short for

```
((cout.operator<<(a)).operator<<(b)).operator<<(c);
```

Without references, << and >> would have to return a pointer to their stream:

```
((cout.operator<<(a))->operator<<(b))->operator<<(c);
```

or

```
((cout.operator<<(a)).operator<<(b)).operator<<(c);
```

This change would spoil the cascading syntax of the operator form:

```
((cout << a) << b) << c;
```

It should be noted that the ability to return references from functions is not new in C++: Algol 68 provides the same capability. The object-oriented features of C++, and its operator overloading, make reference returns particularly useful.

### Closures as Parameters

A closure (a reference to a subroutine, together with its referencing environment) may be passed as a parameter for any of several reasons. The most obvious of these arises when the parameter is declared to be a subroutine (sometimes called a formal subroutine). In Standard Pascal one might write the following.

```
procedure apply_to_A(function f(n : integer) : integer;
 var A : array [low..high : integer] of integer);
var i : integer;
begin
 for i := low to high do A[i] := f(A[i]);
end;
```

Early versions of Pascal did not include the full header of the subroutine parameter (e.g., *f*) in the header of the routine (e.g., *apply\_to\_A*) to which it was being passed. This omission made it difficult or impossible to check at compile time to make sure that the actual and formal parameters expected the same number and types of arguments. The situation in Fortran is similar: Fortran 77 allows a subroutine to be passed as a parameter but cannot check statically for consistent use. Fortran 90 allows (but does not require) the programmer to specify the parameter's interface.

**EXAMPLE 8.18**

Subroutines as parameters in Pascal

Several languages provide first-class subroutine *types*, supporting not only subroutine parameters, but also subroutine variables. In Modula-2 we could write the following.

**EXAMPLE 8.19**

Subroutine types in Modula-2

```

TYPE int_to_int = PROCEDURE(INTEGER) : INTEGER;
PROCEDURE apply_to_A(f : int_to_int; A : ARRAY OF INTEGER);
VAR i : CARDINAL; (* unsigned integer *)
BEGIN
 FOR i := 0 TO HIGH(A) DO A[i] := f(A[i]); END;
END apply_to_A;

```

**EXAMPLE 8.20**

Subroutine pointers in C and C++

C and C++ support *pointers* to subroutines, both as parameters and as variables:

```

void apply_to_A(int (*f)(int), int A[], int A_size)
{
 int i;
 for (i = 0; i < A_size; i++) A[i] = f(A[i]);
}

```

The syntax `f(n)` is used not only when `f` is the name of a function, but also when `f` is a pointer to a function; the pointer need not be dereferenced explicitly.

Ada 83 does not permit subroutines to be passed as parameters. Some of the same effect can be obtained through generic subroutines (to be discussed in Section 8.4), but not enough; Ada 95 provides first-class pointer-to-subroutine types.

Subroutines are routinely passed as parameters (and returned as results) in functional languages. A list-based version of `apply_to_A` would look something like this in Scheme (for the meanings of `car`, `cdr`, and `cons`, see Section 7.8):

```

(define apply-to-L (lambda (f l)
 (if (null? l) '()
 (cons (f (car l)) (apply-to-L f (cdr l)))))))

```

Because Scheme (like Lisp) is not statically typed, there is no need to specify the type of `f`. At run time, a Scheme implementation will announce a dynamic se-

**DESIGN & IMPLEMENTATION****Anonymous delegates in C# 2.0**

C#, which calls its first-class subroutines *delegates*, is a rarity among statically typed imperative languages: though it does not permit subroutines (methods) to nest in the general case, it does allow anonymous delegates (comparable to the `lambda` expressions of Lisp or Scheme) to appear inside other methods. If a delegate refers to objects declared in the surrounding method, then those objects have unlimited extent. When a program assigns an anonymous delegate into a variable, or returns it from a method, the run-time system creates a closure object, into which it copies (references to) any objects referenced by the delegate that may need to outlive the scope in which they are declared. This implementation incurs the cost of dynamic (heap-based) allocation only when it is needed, allowing local variables to remain in the stack in the common case. Python also provides for local variables with unlimited extent, but like Lisp and Scheme it performs type checking at run time.

mantic error in `(f (car l))` if `f` is not a function, and in `(null? l), (car l),` or `(cdr l)` if `l` is not a list.

**EXAMPLE 8.22**

First-class subroutines in ML

```
fun apply_to_L(f, l) =
 case l of
 nil => nil
 | h :: t => f(h) :: apply_to_L(f, t);
```

The type of `apply_to_L` is `('a -> 'b) * 'a list -> 'b list.`

As described in Section 3.5, referencing environments are required in closures only for nested subroutines. C and C++ get by with simple subroutine pointers because they have no nested subroutines. Similarly, Modula-2 can pass simple addresses because it allows only outermost routines to appear as arguments. Modula-3 is a bit more general: it allows inner subroutines to be passed as parameters (and uses closures to represent them), but though it also allows subroutines to be returned from functions or assigned into variables, it limits these cases to outermost routines, thereby avoiding the need for objects of unlimited extent (again, see Section 3.5).

### 8.3.2 Call by Name

Explicit subroutine parameters are not the only language feature that requires a closure to be passed as a parameter. In general, a language implementation must pass a closure whenever the eventual use of the parameter requires the restoration of a previous referencing environment. Interesting examples occur in the *call by name* parameters of Algol 60 and Simula, the label parameters of Algol 60 and Algol 68, and the *call by need* parameters of Miranda, Haskell, and R.

**IN MORE DEPTH**

When Algol 60 was defined, most programmers programmed in assembly language (Fortran was only a few years old, and Lisp was even newer). The assembly languages of the day made heavy use of macros, and it was natural for the Algol designers to propose a parameter-passing mechanism that mimicked the behavior of macros, namely normal-order argument evaluation (Section 6.6.2). It was also natural, given common practice in assembly language, to allow a `goto` to jump to a label that was passed as a parameter. Call-by-name parameters have some interesting and powerful applications, but they are more difficult to implement (and more expensive to use) than one might at first expect: they require the passing of closures. Label parameters are typically implemented by closures as well. Both call-by-name and label parameters tend to lead to inscrutable code; modern languages encourage programmers to use explicit formal subroutines (Section 8.3.1) and structured exceptions (Section 8.5) instead.

|                 | implementation mechanism | permissible operations | change to actual? | alias? |
|-----------------|--------------------------|------------------------|-------------------|--------|
| value           | value                    | read, write            | no                | no     |
| in, const       | value or reference       | read only              | no                | maybe  |
| out (Ada)       | value or reference       | write only             | yes               | maybe  |
| value/result    | value                    | read, write            | yes               | no     |
| var, ref        | reference                | read, write            | yes               | yes    |
| sharing         | value or reference       | read, write            | yes               | yes    |
| in out (Ada)    | value or reference       | read, write            | yes               | maybe  |
| name (Algol 60) | closure (thunk)          | read, write            | yes               | yes    |

**Figure 8.3 Parameter passing modes.** Column 1 indicates common names for modes. Column 2 indicates implementation via passing of values, references, or closures. Column 3 indicates whether the callee can read or write the formal parameter. Column 4 indicates whether changes to the formal parameter affect the actual parameter. Column 5 indicates whether changes to the formal or actual parameter, during the execution of the subroutine, may be visible through the other.

### 8.3.3 Special Purpose Parameters

Figure 8.3 contains a summary of the common parameter-passing modes. In this subsection we examine other aspects of parameter passing.

#### Conformant Arrays

As we saw in Section 7.4.2, the binding time for array dimensions and bounds varies greatly from language to language, ranging from compile time (Basic and Pascal) to elaboration time (Ada and Fortran 90) to arbitrary times during execution (APL, Perl, and Common Lisp). In several languages, the rules for parameters are looser than they are for variables. A formal array parameter whose shape is finalized at run time (in a language that usually determines shape at compile time) is called a *conformant*, or *open*, array parameter. Example 7.61 (page 355) illustrates the use of conformant arrays in Pascal, as does Example 8.18 (page 424). Modula-2 and C equivalents of the latter can be found in Examples 8.19 and 8.20, respectively. Because it passes arrays as pointers, C allows actual parameters of different shapes to be passed through the same parameter, but without any run-time checks to ensure that references by the called routine are within the bounds of the actual array.

#### Default (Optional) Parameters

In Section 3.3.6 we noted that the principal use of dynamic scope is to change the default behavior of a subroutine. We also noted that the same effect can be achieved with *default* parameters. A default parameter is one that need not necessarily be provided by the caller; if it is missing, then a preestablished default value will be used instead.

**EXAMPLE 8.23**

Default parameters in Ada

One common use of default parameters is in I/O library routines (described in Section 7.9.3). In Ada, for example, the `put` routine for integers has the following declaration in the `text_IO` library package.

```
type field is integer range 0..integer'last;
type number_base is integer range 2..16;
default_width : field := integer'width;
default_base : number_base := 10;
procedure put(item : in integer;
 width : in field := default_width;
 base : in number_base := default_base);
```

Here the declaration of `default_width` uses the built-in type *attribute* `width` to determine the maximum number of columns required to print an integer in decimal on the current machine (e.g., a 32-bit integer requires no more than 11 columns, including the optional minus sign).

Any formal parameter that is “assigned” a value in its subroutine heading is optional in Ada. In our `text_IO` example, the programmer can call `put` with one, two, or three arguments. No matter how many are provided in a particular call, the code for `put` can always assume it has all three parameters. The implementation is straightforward: in any call in which actual parameters are missing, the compiler pretends as if the defaults had been provided; it generates a calling sequence that loads those defaults into registers or pushes them onto the stack, as appropriate. On a 32-bit machine, `put(37)` will print the string “37” in an 11-column field (with nine leading blanks) in base 10 notation. `Put(37, 4)` will print “37” in a four-column field (two leading blanks), and `put(37, 4, 8)` will print “45” ( $37 = 45_8$ ) in a four-column field.

Because the `default_width` and `default_base` variables are part of the `text_IO` interface, the programmer can change them if desired. When using default values in calls with missing actuals, the compiler loads the defaults from the variables of the package. As noted in Section 7.9.3, there are overloaded instances of `put` for all the built-in types. In fact, there are two overloaded instances of `put` for every type, one of which has an additional first parameter that specifies the output file to which to write a value.<sup>4</sup> It should be emphasized that there is nothing special about I/O as far as default parameters are concerned: defaults can be used in any subroutine declaration. In addition to Ada, default parameters appear in C++, Common Lisp, Fortran 90, and Python. ■

**Named Parameters**

In all of our discussions so far we have been assuming that parameters are *positional*: the first actual parameter corresponds to the first formal parameter, the

---

**4** The real situation is actually a bit more complicated: the `put` routine for integers is nested inside `integer_IO`, a generic package that is in turn inside of `text_IO`. The programmer must *instantiate* a separate version of the `integer_IO` package for each variety (size) of integer type.

second actual to the second formal, and so on. In some languages, including Ada, Common Lisp, Fortran 90, Modula-3, and Python, this need not be the case. These languages allow parameters to be *named*. Named parameters (also called *keyword* parameters) are particularly useful in conjunction with default parameters. Positional notation allows us to write `put(37, 4)` to print “37” in a four-column field, but it does not allow us to print in octal in a field of default width: any call (with positional notation) that specifies a base must also specify a width, explicitly, because the width parameter precedes the base in `put`’s parameter list. Named parameters provide the Ada programmer with a way around this problem:

```
put(item => 37, base => 8);
```

Because the parameters are named, their order does not matter; we can also write

```
put(base => 8, item => 37);
```

We can even mix the two approaches, using positional notation for the first few parameters, and names for all the rest:

```
put(37, base => 8);
```

In addition to allowing parameters to be specified in arbitrary order, omitting any intermediate default parameters for which special values are not required, named parameter notation has the advantage of documenting the purpose of each parameter. For a subroutine with a very large number of parameters, it can be difficult to remember which is which. Named notation makes the meaning of arguments explicit in the call, as in the following hypothetical example.

```
format_page(columns => 2,
 window_height => 400, window_width => 200,
 header_font => Helvetica, body_font => Times,
 title_font => Times_Bold, header_point_size => 10,
 body_point_size => 11, title_point_size => 13,
 justification => true, hyphenation => false,
 page_num => 3, paragraph_indent => 18,
 background_color => white);
```

### Variable Numbers of Arguments

Lisp, Python, and C and its descendants are unusual in that they allow the user to define subroutines that take a variable number of arguments. Examples of such subroutines can be found in Section 7.9.3: the `printf` and `scanf` functions of C’s `stdio` I/O library. In C, `printf` can be declared as follows.

```
int printf(char *format, ...)
{ ... }
```

The ellipsis (`...`) in the function header is a part of the language syntax. It indicates that there are additional parameters following the format, but that their

types and numbers are unspecified. Since C and C++ are statically typed, additional parameters are not type safe. They are type safe in Common Lisp and Python, however, thanks to dynamic typing.

Within the body of a function with a variable-length argument list, the C or C++ programmer must use a collection of standard routines to access the extra arguments. Originally defined as macros, these routines have implementations that vary from machine to machine, depending on how arguments are passed to functions; today the necessary support is often built into the compiler. For `printf`, variable arguments would be used as follows in C.

**EXAMPLE 8.26**

Variable number of arguments in C

```
#include <stdarg.h> /* macros and type definitions */
int printf(char *format, ...)
{
 va_list args;
 va_start(args, format);
 ...
 char cp = va_arg(args, char);
 ...
 double dp = va_arg(args, double);
 ...
 va_end(args);
}
```

Here `args` is defined as an object of type `va_list`, a special (implementation-dependent) type used to enumerate the elided parameters. The `va_start` routine takes the last declared parameter (in this case, `format`) as its second argument. It initializes its first argument (in this case `args`) so that it can be used to enumerate the rest of the caller's actual parameters. At least one formal parameter must be declared; they can't all be elided.

Each call to `va_arg` returns the value of the next elided parameter. Two examples appear above. Each specifies the expected type of the parameter, and assigns the result into a variable of the appropriate type. If the expected type is different from the type of the actual parameter, chaos can result. In `printf`, the `%X` placeholders in the `format` string are used to determine the type: `printf` contains a large `switch` statement, with one arm for each possible `X`. The arm for `%c` contains a call to `va_arg(args, char)`; the arm for `%f` contains a call to `va_arg(args, double)`. All C floating-point types are extended to double precision before being passed to a subroutine, so there is no need inside `printf` to worry about the distinction between `floats` and `doubles`. `Scanf`, on the other hand, must distinguish between pointers to `floats` and pointers to `doubles`. The call to `va_end` allows the implementation to perform any necessary cleanup operations (e.g., deallocation of any heap space used for the `va_list`, or repair of any changes to the stack frame that might confuse the epilogue code). ■

Older versions of C use a slightly different set of macros, defined in `varargs.h`, for variable-length argument lists. The differences between the two

interfaces reflect the introduction of *function prototypes* (headers with complete information on parameter types) in C.<sup>5</sup>

Like C and C++, C# and recent versions of Java support variable numbers of parameters, but unlike their parent languages they do so in a typesafe manner, by requiring all trailing parameters to share a common type. In Java, for example, one can write

```
static void print_lines(String foo, String... lines) {
 System.out.println("First argument is \"" + foo + "\".");
 System.out.println("There are " +
 lines.length + " additional arguments:");
 for (String str: lines) {
 System.out.println(str);
 }
}
...
print_lines("Hello, world", "This is a message", "from your sponsor.");
```

Here again the ellipsis in the method header is part of the language syntax. Method `print_lines` has two arguments. The first, `foo`, is of type `String`; the second, `lines`, is of type `String...`. Within `print_lines`, `lines` functions as if it had type `String[]` (array of `String`). The caller, however, need not package the second and subsequent parameters into an explicit array; the compiler does this automatically, and the program prints

```
First argument is "Hello, world".
There are 2 additional arguments:
This is a message
from your sponsor.
```

#### EXAMPLE 8.28

Variable number of arguments in C#

The parameter declaration syntax is slightly different in C#:

```
static void print_lines(String foo, params String[] lines) {
 Console.WriteLine("First argument is \"" + foo + "\".");
 Console.WriteLine("There are " +
 lines.Length + " additional arguments:");
 for (int i = 0; i < lines.Length; i++) {
 Console.WriteLine(lines[i]);
 }
}
```

The calling syntax is the same.

---

**5** Prototypes were actually introduced in C++ and then adopted back into the parent language. C accepts the older syntax, as well as the newer, for the sake of backward compatibility. Only the newer is allowed in C++.

### 8.3.4 Function Returns

Many languages place restrictions on the types of objects that can be returned from a function. In Algol 60 and Fortran, a function must return a scalar value. In Pascal and early versions of Modula-2, it must return a scalar or a pointer. Most imperative languages are more flexible: Algol 68, Ada, C, and many (nonstandard) implementations of Pascal allow functions to return values of composite type. Modula-3 and Ada 95 allow a function to return a subroutine, implemented as a closure. C has no closures, but it allows a function to return a pointer to a subroutine. In functional languages such as Lisp and ML, returning a closure is commonplace.

The syntax by which a function indicates the value to be returned varies greatly. In languages like Lisp, ML, and Algol 68, which do not distinguish between expressions and statements, the value of a function is simply the value of its body, which is itself an expression.

In several early imperative languages, including Algol 60, Fortran, and Pascal, a function specifies its return value by executing an assignment statement whose left-hand side is the name of the function. This approach has an unfortunate interaction with the usual static scope rules (Section 3.3.1): the compiler must forbid any immediately nested declaration that would hide the name of the function, since the function would then be unable to return. This special case is avoided in more recent imperative languages by introducing an explicit `return` statement:

```
return expression
```

In addition to specifying a value, `return` causes the immediate termination of the subroutine. As noted in Section 6.2, this termination avoids the common Pascal idiom of placing a statement label on the last line of a subroutine, and then performing a `goto` to this label. A function that has figured out what to return but doesn't want to return yet can always assign the return value into a temporary variable, and then return it later:

```
rtn := expression
...
return rtn
```

Fortran separates early termination of a subroutine from the specification of return values: it specifies the return value by assigning to the function name, and has a `return` statement that takes no arguments.

Argument-bearing `return` statements and assignment to the function name share one additional shortcoming: they force the programmer to employ a temporary variable in incremental computations. Here is an example in Ada:

```
type int_array is array (integer range <>) of integer;
-- array of integers with unspecified integer bounds
function A_max(A : int_array) return integer is
rtn : integer;
```

### EXAMPLE 8.30

Incremental computation  
of a return value

```

begin
 rtn := integer'first;
 for i in A'first .. A'last loop
 if A(i) > rtn then rtn := A(i); end if;
 end loop;
 return rtn;
end A_max;

```

Here `rtn` must be declared as a variable so that the function can read it as well as write it. Because `rtn` is a local variable, most compilers will allocate it within the stack frame of `A_max`. The `return` statement must then perform an unnecessary copy to move that variable's value into the return location allocated by the caller.

**EXAMPLE 8.31**

Explicitly named return values in SR

```

procedure A_max(ref A[1:*]: int) returns rtn : int
 rtn := low(int)
 fa i := 1 to ub(A) ->
 if A[i] > rtn -> rtn := A[i] fi
 af
end

```

Here `rtn` can reside throughout its lifetime in the return location allocated by the caller. A similar facility can be found in Eiffel, in which every function contains an implicitly declared object named `Result`. This object can be both read and written, and is returned to the caller when the function returns.

 **CHECK YOUR UNDERSTANDING**

13. What is the difference between *formal* and *actual* parameters?
14. Describe four common parameter-passing modes. How does a programmer choose which one to use when?
15. Explain the rationale for `READONLY` parameters in Modula-3.
16. What parameter mode is typically used in languages with a reference model of variables?
17. Describe the parameter modes of Ada. How do they differ from the modes of most other Algol-family languages?
18. What does it mean for an Ada program to be *erroneous*?

---

**6** The `fa` in SR stands for “for all”; `ub` stands for “upper bound.” The `->` symbol is roughly equivalent to `do` and `then` in other languages. All structured statements in SR are terminated by spelling the opening keyword backwards. Semicolons between statements may be omitted if they occur at end-of-line.

19. Give an example in which it is useful to return a reference from a function in C++.
  20. List three reasons why a language implementation might implement a parameter as a closure.
  21. What is a *conformant (open) array*?
  22. What are *default parameters*? How are they implemented?
  23. What are *named (keyword) parameters*? Why are they useful?
  24. Explain the value of variable-length argument lists. What distinguishes such lists in Java and C# from their counterparts in C and C++?
  25. Describe three common mechanisms for specifying the return value of a function. What are their relative strengths and drawbacks?
- 

## 8.4 Generic Subroutines and Modules

Subroutines provide a natural way to perform an operation for a variety of different object (parameter) values. In large programs, the need also often arises to perform an operation for a variety of different object *types*. An operating system, for example, tends to make heavy use of queues, to hold processes, memory descriptors, file buffers, device control blocks, and a host of other objects. The characteristics of the queue data structure are independent of the characteristics of the items placed in the queue. Unfortunately, the standard mechanisms for declaring enqueue and dequeue subroutines in most languages require that the type of the items be declared, statically. In a language like Pascal or Fortran, this static declaration of item type means that the programmer must create separate copies of enqueue and dequeue for every type of item, even though the entire text of these copies (other than the type names in the procedure headers) is the same. In some languages (C is an obvious example) it is possible to define a queue of pointers to arbitrary objects, but use of such a queue requires type casts that abandon compile-time checking (Exercise 8.17).

Implicit parametric polymorphism, as suggested in Section 3.6.3, provides a way around the problem, allowing us to declare subroutines whose parameter types are incompletely specified but still type-safe. This approach has its drawbacks, however. As realized in Lisp (Section 10.3) or the various scripting languages, it delays type checking until run time. As realized in ML (Section 7.2.4), it makes the compiler substantially slower and more complicated, and it forces the adoption of a structural view of type equivalence (Section 7.2.1). An alternative, also mentioned in Section 3.6.3, is to provide an explicitly polymorphic *generic* facility that allows a collection of similar subroutines or modules—with different types in each—to be created from a single copy of

the source code. Languages that provide generics include Ada, C++ (which calls them *templates*), Clu, Eiffel, Modula-3, Java, and C#.

**EXAMPLE 8.32**

Generic queues in Ada and C++

**EXAMPLE 8.33**

Generic `min` function in Ada (reprise)

**EXAMPLE 8.34**

Generic parameters

Generic modules or classes are particularly valuable for creating *containers*: data abstractions that hold a collection of objects, but whose operations are generally oblivious to the type of those objects. Examples of containers include stack, queue, heap, set, and dictionary (mapping) abstractions, implemented as lists, arrays, trees, or hash tables. Ada and C++ examples of a generic queue appear in Figure 8.4.

Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right. A generic “minimum” function in Ada appears in Figure 3.17 (page 147). Another standard example is a sorting routine, which needs to be able to tell when objects are smaller or larger than each other, but does not need to know anything else about them.

Exactly what can be passed as a generic parameter varies from language to language. Java and C# pass only types. Ada and C++ are a bit more general. In particular, both allow values of ordinary (nongeneric) types, including subroutines and classes. We can see examples in Figure 8.4, where an integer parameter specifies the maximum length of the queue. In Ada, which supports dynamic arrays (Section 7.4.2), the value of `max_items` need not be known until run time; in C++ it must be a compile-time constant. Often, as in the case of a sorting routine, the generic code needs to be able to count on certain minimal properties of the type parameters. Appropriate *constraints* may be specified explicitly (as in Ada) or inferred by the compiler (as in C++). We will discuss constraints in more detail in Section 8.4.2.

### 8.4.1 Implementation Options

Generics can be implemented several ways. In most implementations of Ada and C++ they are a purely static mechanism: all the work required to create and use multiple instances of the generic code takes place at compile time. In the usual case, the compiler creates a *separate copy* of the code for every instance. (C++ goes farther, and arranges to type-check each of these instances independently.) If several queues are instantiated with the same set of arguments, then the compiler may share the code of the `enqueue` and `dequeue` routines among them. A clever compiler may arrange to share the code for a queue of integers with the code for a queue of single-precision floating-point numbers, if the two types have the same size, but this sort of optimization is not required, and the programmer should not be surprised if it doesn’t occur.

Java 5, by contrast, guarantees that *all* instances of a given generic will share the same code at run time. In effect, if T is a generic type parameter in Java, then objects of class T are treated as instances of the standard base class `Object`, except that the programmer does not have to insert explicit casts to use them as objects of class T, and the compiler guarantees, statically, that the elided casts will never fail. C# plots an intermediate course. Like C++, it will create specialized imple-

```

generic
 type item is private;
 -- can be assigned; other characteristics are hidden
 max_items : in integer := 100; -- 100 items max by default
package queue is
 procedure enqueue(it : in item);
 function dequeue return item;
private
 subtype index is integer range 1..max_items;
 items : array(index) of item;
 next_free, next_full : index := 1;
end queue;

package body queue is
 procedure enqueue(it : in item) is
begin
 items(next_free) := it;
 next_free := next_free mod max_items + 1;
end enqueue;
function dequeue return item is
 rtn : item := items(next_full);
begin
 next_full := next_full mod max_items + 1;
 return rtn;
end dequeue;
end queue;
...
package ready_list is new queue(process);
 -- assume type process has previously been declared
package int_queue is new queue(integer, 50);
 -- only 50 items long, instead of the default 100

```

**Figure 8.4** Generic array-based queues in Ada (left) and C++ (right). C++ calls its generics templates. Checks for overflow and underflow have been omitted for brevity of presentation.  
(continued)

mentations of a generic for different built-in or value types. Like Java, however, it requires that the generic code itself be demonstrably type safe, independent of the arguments provided in any particular instantiation. We will examine the tradeoffs among C++, Java, and C# generics in more detail in Section 8.4.4.

As we noted in Section 3.6.3, statically implemented generics have much in common with macros. The designers of Ada describe generics as “a restricted form of context-sensitive macro facility” [IBFW91, p. 236]. The designers of C++ describe templates as “a clever kind of macro that obeys the scope, naming, and type rules of C++” [Str91, p. 257]. The difference between macros and generics is much like the difference between macros and in-line subroutines (Sections 6.6.2 and 8.2.4): generics are integrated into the rest of the language, and are understood by the compiler, rather than being tacked on as an afterthought, to be

```

template<class item, int max_items = 100>
class queue {
 item items[max_items];
 int next_free;
 int next_full;
public:
 queue() {
 next_free = next_full = 0; // initialization
 }
 void enqueue(item it) {
 items[next_free] = it;
 next_free = (next_free + 1) % max_items;
 }
 item dequeue() {
 item rtn = items[next_full];
 next_full = (next_full + 1) % max_items;
 return rtn;
 }
};
...
queue<process> ready_list;
queue<int, 50> int_queue;

```

**Figure 8.4 (continued)**

expanded by a preprocessor. Generic parameters are type checked. Arguments to generic subroutines are evaluated exactly once. Names declared inside generic code obey the normal scoping rules. In Ada, which allows nested subroutines and modules, names passed as generic arguments are resolved in the referencing environment in which the instance of the generic was created, but all other names in the generic are resolved in the environment in which the generic itself was declared.

### 8.4.2 Generic Parameter Constraints

Because a generic is an abstraction, it is important that its interface (the header of its declaration) provide all the information that must be known by a user of the abstraction. Several languages, including Clu, Ada, Java, and C#, attempt to enforce this rule by *constraining* generic parameters. Specifically, they require that the operations permitted on a generic parameter type be explicitly declared. In the Ada portion of Figure 8.4, the generic clause said

```
type item is private;
```

A *private* type in Ada is one for which the only permissible operations are assignment, testing for equality and inequality, and accessing a few standard at-

---

#### EXAMPLE 8.35

Simple constraints in Ada

tributes (e.g., `size`). To prohibit testing for equality and inequality, the programmer can declare the parameter to be `limited private`. To allow additional operations, the programmer must provide additional information. In simple cases, it may be possible to specify a *type pattern* such as

```
type item is (<>);
```

Here the parentheses indicate that `item` is a discrete type, and will thus support such operations as comparison for ordering (`<`, `>`, etc.) and the attributes `first` and `last`. (As always in Ada, the “box” symbol, `<>`, is a placeholder for missing information: enumeration values, subrange bounds, etc.) ■

#### EXAMPLE 8.36

With constraints in Ada

In more complex cases, the Ada programmer can specify the operations of a generic type parameter by means of a trailing `with` clause. We saw a simple example in the “minimum” function of Figure 3.17 (page 147). The declaration of a generic sorting routine in Ada might be similar:

```
generic
 type T is private;
 type T_array is array (integer range <>) of T;
 with function "<"(a1, a2 : T) return boolean;
 procedure sort(A : in out T_array);
```

Without the `with` clause, procedure `sort` would be unable to compare elements of `A` for ordering, because type `T` is private. ■

Java and C# employ a particularly clean approach to constraints that exploits the ability of object-oriented types to *inherit* methods from a parent type or interface. We defer a full discussion of inheritance to Chapter 9. For now, we note that it allows the Java or C# programmer to require that a generic parameter support a particular set of methods. In Java, for example, we might declare and use our sorting routine as follows.

```
public static <T extends Comparable<T>> void sort(T A[]) {
 ...
 if (A[i].compareTo(A[j]) >= 0) ...
 ...
}
...
Integer[] myArray = new Integer[50];
sort(myArray);
```

Where C++ requires a `template<type_args>` prefix before a generic method, Java puts the type parameters immediately in front of the method’s return type. The `extends` clause constitutes a generic constraint: `Comparable` is an interface (a set of required methods) from the Java standard library that includes the method `compareTo`. This method returns  $-1$ ,  $0$ , or  $1$ , respectively, depending on whether the current object is less than, equal to, or greater than the object passed as a parameter. The compiler checks to make sure that the objects in any array passed to `sort` are of a type that implements `Comparable`, and are therefore guaranteed to provide `compareTo`. If `T` had needed additional interfaces (that is, if we

#### EXAMPLE 8.37

Generic sorting routine in Java

had wanted more constraints), they could have been specified with a comma-separated list: `<T extends I1, I2, I3>`.

#### EXAMPLE 8.38

Generic sorting routine in C#

```
static void sort<T>(T[] A) where T : IComparable {
 ...
 if (A[i].CompareTo(A[j]) >= 0) ...
 ...
}
...
int[] myArray = new int[50];
sort(myArray);
```

C# puts the type parameters after the name of the subroutine, and the constraints (the `where` clause) after the regular parameter list. The compiler is smart enough to recognize that `int` is a built-in type, and generates a customized implementation of `sort`, eliminating the need for Java's `Integer` wrapper class and producing faster code.

#### EXAMPLE 8.39

Generic sorting routine in C++

```
template<class T>
void sort(T A[], int A_size) { ... }
```

No mention is made of the need for a comparison operator. The body of a generic can (attempt to) perform arbitrary operations on objects of a generic parameter type, but if the generic is instantiated with a type that does not support that operation, the compiler will announce a static semantic error. Unfortunately, because the header of the generic does not necessarily specify which operations will be required, it can be difficult for the programmer to predict whether a particular instantiation will cause an error message. Worse, in some cases the type provided in a particular instantiation may support an operation required by the generic's code, but that operation may not do "the right thing." Suppose in our C++ sorting example that the code for `sort` makes use of the `<` operator. For `ints` and `doubles`, this operator will do what one would expect. For character strings, however, it will compare pointers, to see which referenced character has a lower address. If the programmer is expecting comparison for lexicographic ordering, the results may be surprising!

To avoid surprises, it is best to avoid implicit use of the operations of a generic parameter type. There are several ways to make things more explicit in C++ [Str91, pp. 271–277]: the comparison routine can be provided as a method of class `T`, an extra argument to the `sort` routine, or an extra generic parameter. To facilitate the first of these options, the programmer may choose to emulate Java or C#, encapsulating the required methods in an abstract base class from which the type `T` may inherit.

### 8.4.3 Implicit Instantiation

**EXAMPLE 8.40**

Generic class instance in C++

```
queue<int, 50> *my_queue = new queue<int, 50>(); // C++
```

**EXAMPLE 8.41**

Generic subroutine instance in Ada

```
procedure int_sort is new sort(integer, int_array, "<");
```

```
...
```

```
int_sort(my_array);
```

**EXAMPLE 8.42**

Implicit instantiation in C++

Other languages (C++, Java, and C# among them) do not require this. Instead they treat generic subroutines as a form of overloading. Given the C++ sorting routine of Example 8.39 and the following objects:

```
int ints[10];
double reals[50];
char *strings[30];
```

we can perform the following calls without instantiating anything explicitly.

```
sort(ints, 10);
sort(reals, 50);
sort(strings, 30);
```

In each case, the compiler will implicitly instantiate an appropriate version of the `sort` routine. Java and C# have similar conventions. To keep the language manageable, the rules for implicit instantiation in C++ are more restrictive than the rules for resolving overloaded subroutines in general. In particular, the compiler will not coerce a subroutine argument to match a type expression containing a generic parameter (Exercise 8.24).

### 8.4.4 Generics in C++, Java, and C#

Several of the key tradeoffs in the design of generics can be illustrated by comparing the features of C++, Java, and C#. C++ is by far the most ambitious of the three. Its templates are intended for almost any programming task that requires substantially similar but not identical copies of an abstraction. Java 5 and C# 2.0 provide generics purely for the sake of polymorphism. Java's design was heavily influenced by the desire for backward compatibility, not only with existing versions of the language, but with existing virtual machines and libraries. The C# designers, though building on an existing language, did not feel as constrained. They had been planning for generics from the outset, and were able to engineer substantial new support into the .NET virtual machine.

 IN MORE DEPTH

On the PLP CD we discuss C++, Java, and C# generics in more detail, and consider the impact of their differing designs on the quality of error messages, the speed and size of generated code, and the expressive power of the notation. We note in particular the very different mechanisms used to make generic classes and methods support as broad a class of generic arguments as possible.

 CHECK YOUR UNDERSTANDING

26. What is the principal purpose of generics? In what sense do generics serve a broader purpose in C++ and Ada than they do in Java and C#?
27. How does a generic subroutine differ from a macro?
28. Under what circumstances can a language implementation share code among separate instances of a generic?
29. Summarize the relative strengths and weaknesses of generic container classes and classes containing instances of a “generic reference type,” as defined in Section 7.2.2 (page 331).
30. What does it mean for a generic parameter to be *constrained*? Explain the difference between explicit and implicit constraints.
31. Why will C# accept `int` as a generic argument, but Java won’t?
32. Under what circumstances will C++ instantiate a generic function implicitly?

## 8.5 Exception Handling

Several times in the preceding chapters and sections we have referred to *exception handling* mechanisms. We have delayed detailed discussion of these mechanisms until now because exception handling generally requires the language implementation to “unwind” the subroutine call stack.

An exception can be defined as an unexpected—or at least unusual—condition that arises during program execution, and that cannot easily be handled in the local context. It may be detected automatically by the language implementation, or the program may *raise* it explicitly. The most common exceptions are various sorts of run-time errors. In an I/O library, for example, an input routine may encounter the end of its file before it can read a requested value, or it may find punctuation marks or letters on the input when it is expecting digits. To cope with such errors without an exception-handling mechanism, the programmer has basically three options, none of which is entirely satisfactory:

- I. “Invent” a value that can be used by the caller when a real value could not be returned.

2. Return an explicit “status” value to the caller, who must inspect it after every call. The status may be written into an extra, explicit parameter, stored in a global variable, or encoded as otherwise invalid bit patterns of a function’s regular return value.
3. Pass a closure (in languages that support them) for an error-handling routine that the normal routine can call when it runs into trouble.

The first of these options is fine in certain cases but does not work in the general case. Options 2 and 3 tend to clutter up the program, and impose overhead that we should like to avoid in the common case. The tests in option 2 are particularly offensive: they obscure the normal flow of events in the common case. Because they are so tedious and repetitive, they are also a common source of errors; one can easily forget a needed test. Exception-handling mechanisms address these issues by moving error-checking code “out of line,” allowing the normal case to be specified simply, and arranging for control to branch to a *handler* when appropriate.

#### **EXAMPLE 8.43**

ON conditions in PL/I

Exception handling was pioneered by PL/I, which includes an executable statement of the form

```
ON condition
 statement
```

The nested statement (often a GOTO or a BEGIN . . . END block) is a handler. It is not executed when the ON statement is encountered, but is “remembered” for future reference. It will be executed later if exception *condition* (e.g., OVERFLOW) arises. Because the ON statement is executable, the binding of handlers to exceptions depends on the flow of control at run time. ■

If a PL/I exception handler is invoked and then “returns” (i.e., does not perform a GOTO to somewhere else in the program), then one of two things will happen. For exceptions that the language designers considered to be fatal, the program itself will terminate. For “recoverable” exceptions, execution will resume at the statement following the one in which the exception occurred. Experience with PL/I indicates that both the dynamic binding of handlers to exceptions and the automatic resumption of code in which an exception occurred are confusing and error-prone.

More recent languages, including Clu, Ada, Modula-3, Python, C++, Java, C#, and ML, all provide exception-handling facilities in which handlers are lexically bound to blocks of code, and in which the execution of the handler *replaces* the yet-to-be-completed portion of the block. As a general rule, if an exception is not handled within the current subroutine, then the subroutine returns abruptly and the exception is raised at the point of call. If the exception is not handled in the calling routine, it continues to propagate back up the dynamic chain. If it is not handled in the program’s main routine, then a predefined outermost handler is invoked, and usually terminates the program.

In a sense, the dependence of exception handling on the order of subroutine calls might be considered a form of dynamic binding, but it is a much more re-

stricted form than is found in PL/I. Rather than say that a handler in a calling routine has been dynamically bound to an error in a called routine, we prefer to say that the handler is lexically bound to the expression or statement that *calls* the called routine. An exception that is not handled inside a called routine can then be modeled as an “exceptional return”; it causes the calling expression or statement to raise an exception, which is again handled lexically within its subroutine.

In practice, exception handlers tend to be used for three main purposes. First, ideally, a handler will perform some operation that allows the program to recover from the exception and continue execution. For example, in response to an “out of memory” exception in a storage management routine, a handler might request the operating system to allocate additional space to the application, after which it could complete the requested operation. Second, when an exception occurs in a given block of code but cannot be handled locally, it is often important to declare a local handler that cleans up any resources allocated in the local block, and then “reraises” the exception so that it will continue to propagate back to a handler that can (hopefully) recover. Third, if recovery is not possible, a handler can at least print a helpful error message before the program terminates.

As noted in Section 6.2, Common Lisp has an unusually rich set of features for nonlocal transfer of control. Not only does it support multilevel returns as a separate concept from exceptions, it also includes four versions of the exception handling mechanism. Two provide the usual “exceptional return” semantics; the others are designed to repair the problem and restart evaluation of some dynamically enclosing expression. Orthogonally, two perform their work in the referencing environment where the handler is declared; the others perform their work in the environment where the exception first arises. The latter option allows an abstraction to provide several alternative strategies for recovery from exceptions. The user of the abstraction can then specify, dynamically, which of these strategies should be used in a given context. We will consider Common Lisp further in Exercise 8.32 and Exploration 8.48. The “exceptional return” mechanism, with work performed in the environment of the handler, is known as `handler-case`; it provides semantics comparable to those of most other modern languages.

### 8.5.1 Defining Exceptions

In many languages, including Clu, Ada, Modula-3, Python, Java, C#, and ML, most dynamic semantic errors result in exceptions, which the program can then catch. The programmer can also define additional, application-specific exceptions. Examples of predefined exceptions include arithmetic overflow, division by zero, end-of-file on input, subscript and subrange errors, and null pointer dereference. The rationale for defining these as exceptions (rather than as fatal errors) is that they may arise in certain valid programs. Some other dynamic errors (e.g., return from a subroutine that has not yet designated a return value) are still fatal in most languages. In C++ and Common Lisp, most exceptions are

programmer-defined. (The `signal` library provided by many C and C++ implementations is independent of language-level exceptions; it allows a program to bind handlers dynamically to certain exceptions detected by the operating system.) In Ada, some of the predefined exceptions can be *suppressed* by means of a pragma.

**EXAMPLE 8.44**

What is an exception?

In Ada, `exception` is a built-in type; an exception is simply an object of this type:

```
declare empty_queue : exception;
```

In Modula-3, exceptions are another “kind” of object, akin to constants, types, variables, or subroutines:

```
EXCEPTION empty_queue;
```

In Python, C++, Java, and C#, an exception is an ordinary object, in the object-oriented sense of the word—a value of some class type:

```
class empty_queue { };
```

In ML, `exception` is a constructor, akin to `datatype` (as described in Section 7.2.4). ■

**EXAMPLE 8.45**

Parameterized exceptions

Most languages allow an exception to be “parameterized” so the code that raises the exception can pass information to the code that handles it. In C++/Java/C# and ML, the “parameters” of an exception are naturally expressed as the fields of the class or constructor:

```
class duplicate_in_set { // C++
 item dup; // element that was inserted twice
};

...
throw duplicate_in_set(d);

exception duplicate_in_set of item; (* ML *)
...
raise duplicate_in_set(d);
```

In Clu and Modula-3, the parameters are included in the exception declaration, much as they are in a subroutine header (the Modula-3 `empty_queue` in Example 8.44 has no parameters). Ada is unusual in that its exceptions are simply tags: they contain no information other than their name. ■

The `throw` statement (in C++/Java/C# and Common Lisp) or `raise` statement (in Ada, Modula-3, Python, and ML) allows the programmer to write code that will raise an exception at run time. A `throw` or `raise` statement is usually embedded in an `if` statement that checks to see if something has gone wrong. PL/I and Clu both use `signal` instead of `throw` or `raise`, and both provide semantics significantly different from those of other exception-handling languages. As noted earlier, PL/I handlers are dynamically bound; exceptions do not propagate back down the dynamic chain. In Clu, `signal` is always an “exceptional

return": it cannot be handled locally, but rather causes an immediate return from the current subroutine, forcing the caller to recover.

If a subroutine raises an exception but does not catch it internally, it may "return" in an unexpected way. This possibility is an important part of the routine's interface to the rest of the program. Consequently, several languages, including Clu, Modula-3, C++, and Java, include in each subroutine header a list of the exceptions that may propagate out of the routine. This list is mandatory in Modula-3: it is a run-time error if an exception arises that does not appear in the header and is not caught internally. The list is optional in C++: if it appears, the semantics are the same as in Modula-3; if it is omitted, all exceptions are permitted to propagate. Java adopts an intermediate approach: it segregates its exceptions into "checked" and "unchecked" categories. Checked exceptions must be declared in subroutine headers; unchecked exceptions need not. Unchecked exceptions are typically run-time errors that most programs will want to be fatal (subscript out of bounds, for example)—and that would therefore be a nuisance to declare in every function—but that a highly robust program may want to catch if they occur in library routines.

### 8.5.2 Exception Propagation

---

#### **EXAMPLE 8.46**

Exception handler in Ada

```
with text_IO; -- import I/O routines (and exceptions)
procedure read_rec ... is
begin
 ...
 begin
 ...
 -- potentially complicated sequence of operations
 -- involving many calls to text_IO.get
 ...
 exception
 when end_error => ...
 -- handler to catch any attempt to read past end-of-file
 -- in any of the I/O calls
 end;
 ...
end read_rec;
```

Here we have hypothesized a subroutine to read a record from a file. If the file has been corrupted, it may end in the middle of a record. Rather than check for end-of-file at every read (get) operation, we can place the entire series of reads inside a `begin...end` block that is protected by a single handler.

As written, the handler above will catch only the `end_error` exception, which is declared in package `text_IO`. In general, the `exception` part of a

`begin...end` block can have an arbitrary number of handlers, each for a different exception. The syntax of the handlers resembles that of an Ada `case` statement. As in a `case` statement, the final `when` clause can be written to catch all unnamed exceptions:

```
when others => ...
```

#### EXAMPLE 8.47

##### Exception handler in C++

Syntax in other languages is similar. In C++:

```
try {
 ...
 // protected block of code
 ...

} catch(end_of_file) {
 ...
}

} catch(io_error e) {
 // handler for any io_error other than end_of_file
 ...
}

} catch(...) {
 // handler for any exception not previously named
 // (in this case, the triple-dot ellipsis is a valid C++ token;
 // it does not indicate missing code)
}
```

The handlers attached to a block of code are always examined in order; control is transferred to the first one that *matches* the exception. In Ada, a handler matches if it names the propagating exception or if it is a “catch-all” `others` clause. In C++, a handler matches if it names a class from which the exception is derived or if it is a catch-all. In the current example, let us assume that `end_of_file` is a subclass of `io_error`. Then an `end_of_file` exception, if it arises, will be handled by the first of the three `catch` clauses. All other I/O errors will be caught by the second `catch` clause. All non-I/O errors will be caught by the third `catch` clause. Note that in the second `catch` clause we have declared a local name, `e`, for the exception object. Within the `catch` clause, we can refer to the members of `e`. This mechanism allows the code that raises (throws) the exception to pass information to the handler. The C++ standard library declares exceptions as a hierarchy of classes; programmers are encouraged to use and extend this hierarchy. Java and C#, whose handlers look just like those of C++, provide similar standard hierarchies.

If an exception propagates out of the scope in which it was declared, it can no longer be named by a handler, and thus can be caught only by a “catch-all” handler. Modula-3 avoids this problem by requiring all exceptions to be declared at the outermost level of lexical nesting. In most languages, an exception that is declared in a recursive subroutine will be caught by the innermost handler for that exception at run time. In a language with concurrency, one must also consider what will happen if an exception is not handled at the outermost level of a concurrent thread of control. In Modula-3, the entire program terminates

abnormally; in Ada and Java, the affected thread terminates quietly; in C# the behavior is implementation-defined.

### **Handlers on Expressions**

In an expression-oriented language such as ML or Common Lisp, an exception handler is attached to an expression, rather than to a statement. Since execution of the handler replaces the unfinished portion of the protected code when an exception occurs, a handler attached to an expression must provide a value for the expression. (In a statement-oriented language, the handler—like most statements—is executed for its side effects.) In ML, a handler looks like this:

```
val foo = (f(a) * b) handle Overflow => max_int;
```

Here `(f(a) * b)` is the protected expression, `handle` is a keyword, `Overflow` is a predefined exception (a value built from the `exc` constructor), and `max_int` is an expression (in this case a constant) whose value replaces the value of the expression in which the `Overflow` exception arose. Both the protected expression (`here (f(a) * b)`) and the handler (`here max_int`) could in general be arbitrarily complicated, with many nested function calls. Exceptions that arise within a nested call (and are not handled locally) propagate back down the dynamic chain, just as they do in Ada or C++. ■

### **Cleanup Operations**

In the process of searching for a matching handler, the exception-handling mechanism must “unwind” the run-time stack by reclaiming the stack frames of any subroutines from which the exception escapes. Reclaiming a frame requires not only that its space be popped from the stack, but also that any registers that were saved as part of the calling sequence be restored. (We discuss implementation issues in more detail in Section 8.5.4.)

In C++, an exception that leaves a scope, whether a subroutine or just a nested block, requires the language implementation to call *destructor* functions for any objects declared within that scope. Destructors (to be discussed in more detail in Section 9.3) are often used to deallocate heap space and other resources (e.g., open files). Similar functionality is provided in Common Lisp by an `unwind-protect` expression, and in Modula-3, Python, Java, and C# by means of `try...finally` constructs. Code in Modula-3 might look like this:

### **EXAMPLE 8.49**

Finally clause in Modula-3

## DESIGN & IMPLEMENTATION

### **Structured exceptions**

Exception handling mechanisms are among the most complex aspects of modern language design, from both a semantic and a pragmatic point of view. Programmers have used subroutines since before there were computers (they appear, among other places, in the 19th-century notes of Countess Ada Augusta Byron). Structured exceptions, by contrast, were not invented until the 1970s, and did not become commonplace until the 1980s.

```

TRY
 myStream := OpenRead(myFileName); (* protected block *)
 Parse(myStream);
FINALLY
 Close(myStream); (* cleanup code *)
END;

```

A FINALLY clause will be executed whenever control escapes from the corresponding protected block, whether the escape is due to normal completion, an exit from a loop, a return from the current subroutine, or the propagation of an exception. In fact, EXITS and RETURNS in Modula-3 are modeled as exceptions. We have assumed in our example that `myStream` is not bound to anything at the beginning of the code, and that it is harmless to `Close` a not-yet-opened stream. ■

**EXAMPLE 8.50**

Catch and finally in Java

A try block may have both a `finally` clause and exception handlers. In Java we might write

```

static void parse(FileReader s) throws IOException {...}
...
FileReader myStream = null;

try {
 myStream = new FileReader(new File("foo"));
 parse(myStream);
} catch(EOFException e) {
 System.out.println("Oops; input file too short.");
} finally {
 myStream.close();
}

```

Here the `finally` clause will be executed immediately before normal exit, immediately after executing the `catch` clause (if an `EOFException` arises in `parse`), or immediately before control escapes the `try` block due to some other exception. ■

If cleanup is appropriate only when a certain exception occurs, but not in the general case, then we will need to use a `catch` clause rather than a `finally` clause. If the exception itself cannot be handled locally we will then need to reraise it. The usual syntax for this purpose is a `throw` or `raise` statement without an argument, permitted only in a handler.

### 8.5.3 Example: Phrase-Level Recovery in a Recursive Descent Parser

In Section ⑩ 2.3.4 we presented a technique for phrase-level recovery from syntax errors in a recursive descent parser. The key idea was this: at the beginning of the subroutine whose job it is to parse a given nonterminal  $A$ , we check to see whether the upcoming input token is acceptable. If not, we announce an error and delete tokens until we find one in the FIRST or FOLLOW set of  $A$ . A good implementation of this idea requires an extra parameter for every parsing routine

(the context-specific FOLLOW set), a call to the error-checking routine in the beginning of every parsing routine, and a globally defined set of “starter” symbols that should not be deleted.

An attractive alternative approach is possible with exceptions. We can avoid the clutter of the extra parameters and the expense of the error-checking calls by declaring a single syntax\_error exception and then placing handlers for it at a small number of “clean points” in the parse. In many languages, for example, we could obtain simple but probably serviceable error recovery by placing one handler around the body of statement and another around declaration:

```

procedure statement
 try ...
 -- code to parse a statement
 except when syntax_error =>
 loop
 if next_token ∈ FIRST(statement)
 statement -- try again
 return
 elseif next_token ∈ FOLLOW(statement)
 return
 else get_next_token

```

The code for declaration is similar. For better quality repair, we might add handlers around the bodies of expression, aggregate, or other complex constructs. To guarantee that we can always recover from an error, we must ensure that all parts of the grammar lie inside at least one handler. At any point where a syntax error is detected (i.e., when a parsing routine is unable to predict, or when match sees an unexpected input token), we simply raise the syntax\_error exception. The exception will propagate back out of an arbitrary number of nested constructs (the number can be very large) until it encounters the innermost protected construct. At that point we will toss the remainder of the phrase and continue with the parse. ■

#### 8.5.4 Implementation of Exceptions

##### **EXAMPLE 8.52**

Stacked exception handlers

The most obvious implementation for exceptions maintains a linked-list stack of handlers. When control enters a protected block, the handler for that block is added to the head of the list. When an exception arises, either implicitly or as a result of a raise statement, the language run-time system pops the innermost handler off the list and calls it. The handler begins by checking to see if it matches the exception that occurred; if not, it simply reraises it:

```

if exception matches duplicate_in_set
 ...
else
 reraise exception

```

To implement propagation back down the dynamic chain, each subroutine has an implicit handler that performs the work of the subroutine epilogue code and then reraises the exception.

**EXAMPLE 8.53**

Multiple exceptions per handler

```
if exception matches end_of_file
...
elsif exception matches io_error
...
else
 ...
 -- "catch-all" handler
```

The problem with this implementation is that it incurs run-time overhead in the common case. Every protected block and every subroutine begins with code to push a handler onto the handler list, and ends with code to pop it back off the list. We can usually do better.

The only real purpose of the handler list is to determine which handler is active. Since blocks of source code tend to translate into contiguous blocks of machine-language instructions, we can capture the correspondence between handlers and protected blocks in the form of a table generated at compile time. Each entry in the table contains two fields: the starting address of a block of code and the address of the corresponding handler. The table is sorted on the first field. When an exception occurs, the language run-time system performs binary search in the table, using the program counter as key, to find the handler for the current block. If that handler reraises the exception, the process repeats: handlers themselves are blocks of code, and can be found in the table. The only subtlety arises in the case of the implicit handlers associated with propagation out of subroutines: such a handler must ensure that the reraise code uses the return address of the subroutine, rather than the current program counter, as the key for table lookup.

The cost of raising an exception is higher in this second implementation, by a factor logarithmic in the number of handlers in the program. But this cost is paid only when an exception actually occurs. On the assumption that exceptions are unusual events, the net impact on performance is clearly beneficial: the cost in the common case is zero. In its pure form the table-based approach requires that the compiler have access to the entire program, or that the linker provide a mechanism to glue subtables together. For a language like Java, in which code fragments are compiled independently, we can employ a hybrid approach in which the compiler creates a separate table for each subroutine, and each stack frame contains a pointer to the appropriate table.

### **Exception Handling without Exceptions**

It is worth noting that exceptions can sometimes be simulated in a language that does not provide them as a built-in. In Section 6.2 we noted that Pascal permits `gotos` to labels outside the current subroutine, that Algol 60 allows labels to be passed as parameters, and that PL/I allows them to be stored in variables. These

mechanisms permit the program to escape from a deeply nested context, but in a very unstructured way.

A much more attractive alternative appears in Scheme, which provides a general purpose function called `call-with-current-continuation`, sometimes abbreviated `call/cc`. This function takes a single argument  $f$ , which is itself a function. It calls  $f$ , passing as argument a continuation  $c$  (a closure) that captures the current program counter and referencing environment. At any point in the future,  $f$  can call  $c$  to reestablish the saved environment. If nested calls have been made, control pops out of them, as it does with exceptions. More generally, however,  $c$  can be saved in variables, returned explicitly by subroutines, or called repeatedly, even after control has returned from  $f$  (recall that closures in Scheme have unlimited extent; see Section 3.5). `call/cc` suffices to build a wide variety of control abstractions, including iterators and coroutines (Section 8.6) and the `exits` and `returns` of nonfunctional programs. It even subsumes the notion of returning from a subroutine, though it seldom replaces it in practice.

#### EXAMPLE 8.54

`Setjmp` and `longjmp` in C

Intermediate between the anarchy of nonlocal `gos` and the generality of `call/cc`, most versions of C (including the ISO standard) provide a pair of library routines entitled `setjmp` and `longjmp`. `Setjmp` takes as argument a buffer into which to capture a representation of the program's current state. This buffer can later be passed to `longjmp` to restore the captured state. `Setjmp` returns C's equivalent of a Boolean value: a 0 or a 1. The 0 indicates "normal" return; the 1 indicates "return" from a `longjmp`. The usual programming idiom looks like this:

```
if (!setjmp(buffer)) {
 /* protected code */
} else {
 /* handler */
}
```

When initially called, `setjmp` returns a 0, and control enters the protected code. If `longjmp(buffer)` is called anywhere within the protected code, or in subroutines called by that code, then `setjmp` will appear to return again,

#### DESIGN & IMPLEMENTATION

##### `Setjmp`

Because it saves many registers to memory, the usual implementation of `setjmp` is quite expensive—more so than entry to a protected block in the “obvious” implementation of exceptions described above. While implementors are free to use a more efficient, table-driven approach if desired, the usual implementation minimizes the complexity of the run-time system and eliminates the need for linker-supported integration of tables from separately compiled modules and libraries.

this time with a 1, causing control to enter the handler. Unlike the closure created by `call/cc`, the information captured by `setjmp` has limited extent; once the protected code completes, the behavior of `longjmp(buffer)` is undefined. ■

`Setjmp` and `longjmp` are usually implemented by saving the current machine registers in the `setjmp` buffer, and by restoring them in `longjmp`. There is no list of handlers; rather than “unwinding” the stack, the implementation simply tosses all the nested frames by restoring old values of the `sp` and `fp`. The problem with this approach is that the register contents at the beginning of the handler do not reflect the effects of the successfully completed portion of the protected code: they were saved before that code began to run. Any changes to variables that have been written through to memory will be visible in the handler, but changes that were cached in registers will be lost. To address this limitation, C allows the programmer to specify that certain variables are `volatile`. A `volatile` variable is one whose value in memory can change “spontaneously”—for example, as the result of activity by an I/O device or a concurrent thread of control. C implementations are required to store `volatile` variables to memory whenever they are written, and to load them from memory whenever they are read. If a handler needs to see changes to a variable that may be modified by the protected code, then the programmer must include the `volatile` keyword in the variable’s declaration.

### CHECK YOUR UNDERSTANDING

---

33. Describe the algorithm used to identify an appropriate handler when an exception is raised in a language like Ada or C++.
  34. Explain why it is useful to define exceptions as classes in C++, Java, and C#.
  35. Explain how to implement exceptions in a way that incurs no cost in the common case (when exceptions don’t arise).
  36. How do the exception handlers of a functional language like ML differ from those of an imperative language like C++?
  37. Describe the operations that must be performed by the implicit handler for a subroutine.
  38. Describe the `call-with-current-continuation` function of Scheme.
  39. Summarize the shortcomings of the `setjmp` and `longjmp` library routines of C.
  40. What is a `volatile` variable in C? Under what circumstances is it useful?
-

## 8.6 Coroutines

Given an understanding of the layout of the run-time stack, we can now consider the implementation of more general control abstractions—*coroutines* in particular. Like a continuation, a coroutine is represented by a closure (a code address and a referencing environment), into which we can jump by means of a nonlocal `goto`—in this case a special operation known as `transfer`. The principal difference between the two abstractions is that a continuation is a constant—it does not change once created—while a coroutine changes every time it runs. When we `goto` a continuation, our old program counter is lost, unless we explicitly create a new continuation to hold it. When we `transfer` from one coroutine to another, our old program counter is saved: the coroutine we are leaving is updated to reflect it. Thus if we perform a `goto` into the same continuation multiple times, each jump will start at precisely the same location, but if we perform a `transfer` into the same coroutine multiple times, each jump will take up where the previous one left off.

In effect, coroutines are execution contexts that exist concurrently but execute one at a time, and that transfer control to each other explicitly, by name. Coroutines can be used to implement iterators (Section 6.5.3) and threads (to be discussed in Chapter 12). They are also useful in their own right, particularly for certain kinds of servers and for discrete event simulation. Threads appear in a variety of languages, including Algol 68, Modula (1), Modula-3, Ada, SR, Occam, Java, and C#. They are also commonly provided (though with somewhat less attractive syntax and semantics) outside the language proper by means of library packages. Coroutines are less common as a user-level programming abstraction. Languages that provide them include Simula and Modula-2. We focus in the following subsections on the implementation of coroutines and (on the PLP CD) on their use in iterators (Section 8.6.3) and discrete event simulation (Section 8.6.4).

### EXAMPLE 8.55

#### Explicit interleaving of concurrent computations

As a simple example of an application in which coroutines might be useful, imagine that we are writing a “screen-saver” program, which paints a mostly black picture on the screen of an inactive workstation and keeps the picture moving, to avoid phosphor or liquid-crystal “burn-in.” Imagine also that our screen-server performs “sanity checks” on the file system in the background, looking for corrupted files. We could write our program as follows.

```
loop
 -- update picture on screen
 -- perform next sanity check
```

The problem with this approach is that successive sanity checks (and to a lesser extent successive screen updates) are likely to depend on each other. On most systems, the file-system checking code has a deeply nested control structure containing many loops. To break it into pieces that can be interleaved with the screen updates, the programmer must follow each check with code that saves the state

of the nested computation, and must precede the following check with code that restores that state.

**EXAMPLE 8.56**

## Interleaving coroutines

A much more attractive approach is to cast the operations as coroutines:<sup>7</sup>

```

us, cfs : coroutine

coroutine update_screen
 -- initialize
 detach
 loop
 ...
 transfer(cfs)
 ...

coroutine check_file_system
 -- initialize
 detach
 for all files
 ...
 transfer(us)
 ...
 transfer(us)
 ...
 transfer(us)
 ...
begin -- main
us := new update_screen
cfs := new check_file_system
transfer(us)

```

The syntax here is based loosely on that of Simula. When first created, a coroutine performs any necessary initialization operations, and then `detaches` itself from the main program. The `detach` operation creates a coroutine object to which

**DESIGN & IMPLEMENTATION**
**Threads and coroutines**

As we shall see in Section 12.2.4, it is easy to build a simple thread package given coroutines. Most programmers would agree, however, that threads are substantially easier to use, because they eliminate the need for explicit `transfer` operations. This contrast—a lot of extra functionality for a little extra implementation complexity—probably explains why coroutines as an explicit programming abstraction are relatively rare.

---

<sup>7</sup> Threads could also be used in this example, and might in fact serve our needs better. Coroutines suffice because there is a small number of execution contexts (namely two) and because it is easy to identify points at which one should transfer to the other.

control can later be transferred, and returns a reference to this coroutine to the caller. The `transfer` operation saves the current program counter in the current coroutine object and resumes the coroutine specified as a parameter. The main body of the program plays the role of an initial, default coroutine.

Calls to `transfer` from within the body of `check_file_system` can occur at arbitrary places, including nested loops and conditionals. A coroutine can also call subroutines, just as the main program can, and calls to `transfer` may appear inside these routines. The context needed to perform the “next” sanity check is captured by the program counter, together with the local variables of `check_file_system` and any called routines, at the time of the `transfer`.

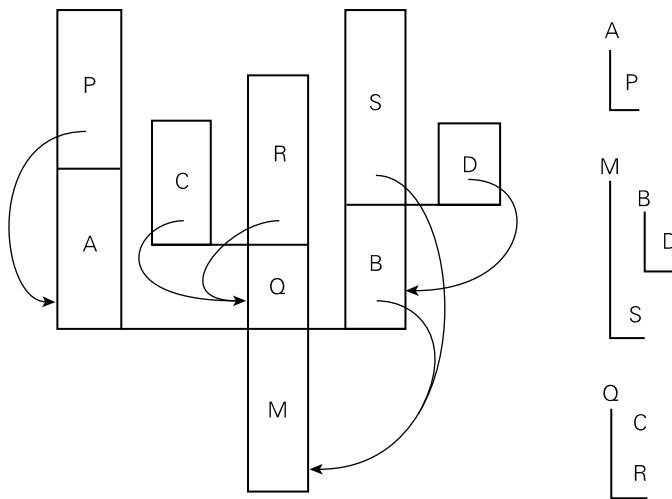
As in Example 8.55, the programmer must specify when to stop checking the file system and update the screen; coroutines make the job simpler by providing a `transfer` operation that eliminates the need to save and restore state explicitly. To decide where to place the calls to `transfer`, we must consider both performance and correctness. For performance, we must avoid doing too much work between calls, so that screen updates aren’t too infrequent. For correctness, we must avoid doing a `transfer` in the middle of any check that might be compromised by file access in `update_screen`. Parallel threads (to be described in Chapter 12) would eliminate the first of these problems by ensuring that the screen updater receives a share of the processor on a regular basis, but would complicate the second problem: we should need to synchronize the two routines explicitly if their references to files could interfere. ■

### 8.6.1 Stack Allocation

Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack: their subroutine calls and returns, taken as a whole, do not occur in last-in-first-out order. If each coroutine is declared at the outermost level of lexical nesting (as required in Modula-2), then their stacks are entirely disjoint: the only objects they share are global, and thus statically allocated. Most operating systems make it easy to allocate one stack and to increase its portion of the virtual address space as necessary during execution. It is usually not easy to allocate an arbitrary number of such stacks; space for coroutines is something of an implementation challenge.

The simplest solution is to give each coroutine a fixed amount of statically allocated stack space. This approach is adopted in Modula-2, which requires the programmer to specify the size and location of the stack when initializing a coroutine. It is a run-time error for the coroutine to need additional space. Some Modula-2 implementations catch the overflow and halt with an error message; others display abnormal behavior. If the coroutine uses less space than it is given, the excess is simply wasted.

If stack frames are allocated from the heap, as they are in most Lisp and Scheme implementations, then the problems of overflow and internal fragmentation are avoided. At the same time, the overhead of each subroutine call is sig-



**Figure 8.5** A *cactus stack*. Each branch to the side represents the creation of a coroutine (A, B, C, and D). The static nesting of blocks is shown at right. Static links are shown with arrows. Dynamic links are indicated simply by vertical arrangement: each routine has called the one above it.

nificantly increased. An intermediate option is to allocate the stack in large, fixed-size “chunks.” At each call, the subroutine calling sequence checks to see whether there is sufficient space in the current chunk to hold the frame of the called routine. If not, another chunk is allocated and the frame is put there instead. At each subroutine return, the epilogue code checks to see whether the current frame is the last one in its chunk. If so, the chunk is returned to a “free chunk” pool.

In any of these implementations, subroutine calls can use the ordinary central stack if the compiler is able to verify that they will not perform a transfer before returning [Sco91].

#### EXAMPLE 8.57

##### Cactus stacks

If coroutines can be created at arbitrary levels of lexical nesting (as they can in Simula), then two or more coroutines may be declared in the same non-global scope, and must thus share access to objects in that scope. To implement this sharing, the run-time system must employ a so-called *cactus stack* (named for its resemblance to the Saguaro cacti of the American Southwest; see Figure 8.5).

Each branch off the stack contains the frames of a separate coroutine. The dynamic chain of a given coroutine ends in the block in which the coroutine began execution. The *static* chain of the coroutine, however, extends down into the remainder of the cactus, through any lexically surrounding blocks. In addition to the coroutines of Simula, cactus stacks are needed for the threads of several parallel languages, including Ada. “Returning” from the main block of a coroutine will generally terminate the program as a whole. Because a coroutine only runs when specified as the target of a transfer, there is never any need to terminate it explicitly: if it is running it can transfer to something else, which never transfers back. When a given coroutine is no longer needed, the Modula-2 pro-

grammer can simply reuse its stack space. In Simula, the space will be reclaimed via garbage collection when it is no longer accessible.

### 8.6.2 Transfer

To transfer from one coroutine to another, the run-time system must change the program counter (PC), the stack, and the contents of the processor's registers. These changes are encapsulated in the `transfer` operation: one coroutine calls `transfer`; a different one returns. Because the change happens inside `transfer`, changing the PC from one coroutine to another simply amounts to remembering the right return address: the old coroutine calls `transfer` from one location in the program; the new coroutine returns to a potentially different location. If `transfer` saves its return address in the stack, then the PC will change automatically as a side effect of changing stacks.

#### **EXAMPLE 8.58**

##### Switching coroutines

```
transfer:
 push all registers other than sp (including ra)
 *current_coroutine := sp
 current_coroutine := r1 -- argument passed to transfer
 sp := *r1
 pop all registers other than sp (including ra)
 return
```

The data structure that represents a coroutine or thread is called a *context block*. In a simple coroutine package, the context block contains a single value: the coroutine's `sp` as of its most recent `transfer`. (A thread package generally places additional information in the context block, such as an indication of priority, or pointers to link the thread onto various scheduling queues. Some coroutine or thread packages choose to save registers in the context block, rather than at the top of the stack; either approach works fine.)

In Modula-2, the coroutine creation routine initializes the coroutine's stack to look like the frame of `transfer`, with a return address and register contents initialized to permit a "return" into the beginning of the coroutine's code. The

#### **DESIGN & IMPLEMENTATION**

##### Coroutine stacks

Many languages require coroutines or threads to be declared at the outermost level of lexical nesting, to avoid the complexity of noncontiguous stacks. Most thread libraries for sequential languages (the POSIX standard `pthread` library among them) likewise require or at least permit the use of contiguous stacks.

creation routine sets the `sp` value in the context block to point into this artificial frame, and returns a pointer to the context block. To begin execution of the coroutine, some existing routine must `transfer` to it.

In Simula (and in the code in Example 8.56), the coroutine creation routine begins to execute the new coroutine immediately, as if it were a subroutine. After the coroutine completes any application-specific initialization, it performs a `detach` operation. `Detach` sets up the coroutine stack to look like the frame of `transfer`, with a return address that points to the following statement. It then allows the creation routine to return to its own caller.

In all cases, `transfer` expects a pointer to a context block as argument; by dereferencing the pointer it can find the `sp` of the next coroutine to run. A global (static) variable, called `current_coroutine` in Example 8.58, contains a pointer to the context block of the currently running coroutine. This pointer allows `transfer` to find the location in which it should save the old `sp`.

### 8.6.3 Implementation of Iterators

Given an implementation of coroutines, iterators are almost trivial: one coroutine is used to represent the main program; a second is used to represent the iterator. Additional coroutines may be needed if iterators nest.

---

#### © IN MORE DEPTH

Additional detail appears on the PLP CD. We also consider a second, simpler implementation of iterators that keeps all state in a single stack, and a third that moves most of the work of coroutine management out of the run-time library and into the compiler.

---

### 8.6.4 Discrete Event Simulation

One of the most important applications of coroutines (and the one for which Simula was designed and named) is *discrete event simulation*. Simulation in general refers to any process in which we create an abstract model of some real-world system and then experiment with the model in order to infer properties of the real-world system. Simulation is desirable when experimentation with the real world would be complicated, dangerous, expensive, or otherwise impractical. A *discrete event* simulation is one in which the model is naturally expressed in terms of events (typically interactions among various interesting objects) that happen at specific times. Discrete event simulation is usually not appropriate for the simulation of continuous processes, such as the growth of crystals or the flow of water over a surface, unless these processes are captured at the level of individual particles.

---

 **IN MORE DEPTH**

On the PLP CD we consider a traffic simulation, in which events model interactions among automobiles, intersections, and traffic lights. We use a separate coroutine for each trip to be taken by car. At any given time we run the coroutine with the earliest expected arrival time at an upcoming intersection. We keep inactive routines in a priority queue ordered by those arrival times.

---

 **CHECK YOUR UNDERSTANDING**

41. What was the first high-level programming language to provide coroutines?
  42. What is the difference between a *Coroutine* and a *thread*?
  43. Why doesn't the *transfer* library routine need to change the program counter when switching between coroutines?
  44. Describe three alternative means of allocating coroutine stacks. What are their relative strengths and weaknesses?
  45. What is a *cactus stack*? What is its purpose?
  46. What is *discrete event simulation*? What is its connection with coroutines?
- 

## 8.7 Summary and Concluding Remarks

This chapter has focused on the subject of control abstraction, and on subroutines in particular. Subroutines allow the programmer to encapsulate code behind a narrow interface, which can then be used without regard to its implementation. Control abstraction is crucial to the design and maintenance of any large software system. It is particularly effective from an aesthetic point of view in languages like Lisp and Smalltalk, which use the same syntax for both built-in and user-defined control constructs.

We began our study of subroutines in Section 8.1 by reviewing the management of the subroutine call stack. We then considered the *calling sequences* used to maintain the stack, with extra sections on the PLP CD devoted to *displays*; case studies for the MIPSpro C compiler and the GNU x86 Pascal compiler (gpc); and the *register windows* of the Sparc. After a brief consideration of inline expansion, we turned in Section 8.3 to the subject of parameters. We first considered parameter-passing *modes*, all of which are implemented by passing values, references, or closures. We noted that the goals of semantic clarity and implementation speed sometimes conflict: it is usually most efficient to pass a large parameter by reference, but the aliasing that results can lead to program bugs. In Section 8.3.3 we considered special parameter-passing mechanisms, including conformant arrays, default (optional) parameters, named parameters,

and variable-length parameter lists. We noted that default and named parameters provide an attractive alternative to the use of dynamic scope. In Section 8.4 we considered the design and implementation of generic subroutines and modules. Generics allow a control abstraction to be parameterized (at compile time) in terms of the types of its parameters, rather than just their values.

In the final two major sections we considered exception-handling mechanisms, which allow a program to “unwind” in a well-structured way from a nested sequence of subroutine calls, and coroutines, which allow a program to maintain two or more execution contexts, and to switch back and forth among them. As examples, we considered the use of exceptions for phrase-level recovery from syntax errors in a recursive descent parser, and (on the PLP CD) the use of coroutines for discrete event simulation. In Chapter 12, we will consider the extension of coroutines to *threads*, which run (or appear to run) in parallel with one another.

In several cases we can discern an evolving consensus about the sorts of control abstractions that a language should provide. The limited parameter-passing modes of languages like Fortran and Algol 60 have been replaced by more extensive or flexible options. The standard positional notation for arguments has been augmented in languages like Ada and C++ with default and named parameters. Less-structured error-handling mechanisms, such as label parameters, nonlocal gotos, and dynamically bound handlers, have been replaced by structured exception handlers that are lexically scoped within subroutines, and can be implemented at zero cost in the common (no-exception) case. In many cases, implementing these newer features has required that compilers and run-time systems become more complex. Occasionally, as in the case of call-by-name parameters, label parameters, or nonlocal gotos, features that were semantically confusing were also difficult to implement, and abandoning them has made compilers simpler. In yet other cases language features that are useful but difficult to implement continue to appear in some languages but not in others. Examples in this category include first-class subroutines, coroutines, iterators, continuations, and local objects with unlimited extent.

## 8.8 Exercises

- 8.1 Describe as many ways as you can in which functions in Algol-family programming languages differ from functions in mathematics.
- 8.2 Using your favorite language and compiler, write a program that determines the order in which subroutine parameters are evaluated.
- 8.3 Consider the following (erroneous) program in C.

```

void foo() {
 int i;
 printf("%d ", i++);
}
main() {
 int j;
 for (j = 1; j <= 10; j++) foo();
}

```

Local variable *i* in subroutine *foo* is never initialized. On many systems, however, the program will display repeatable behavior, printing 0 1 2 3 4 5 6 7 8 9. Suggest an explanation. Also explain why the behavior on other systems might be different, or nondeterministic.

- 8.4 The standard calling sequence for the Digital VAX instruction set employs not only a stack pointer (*sp*) and frame pointer (*fp*), but a separate *arguments pointer* (*ap*) as well. Under what circumstances might this separate pointer be useful? In other words, when might it be handy not to have to place arguments at statically known offsets from the *fp*?
- 8.5 Suppose you wish to minimize the size of closures in a language implementation that uses a display to access nonlocal objects. Assuming a language like Pascal or Ada, in which subroutines have limited extent (Section 3.5), explain how an appropriate display for a formal subroutine can be calculated when that routine is finally called, starting with only (1) the value of the frame pointer, saved in the closure at the time that the closure was created, (2) the subroutine return addresses found in the stack at the time the formal subroutine is finally called, and (3) static tables created by the compiler. How costly is your scheme?
- 8.6 Write (in the language of your choice) a procedure or function that will have four different effects, depending on whether arguments are passed by value, by reference, by value/result, or by name.
- 8.7 Consider an expression like *a* + *b* that is passed to a subroutine in Fortran. Is there any semantically meaningful difference between passing this expression as a reference to an unnamed temporary (as Fortran does) or passing it by value (as one might, for example, in Pascal)?
- 8.8 Consider the following subroutine in Fortran 77.

```

subroutine shift(a, b, c)
integer a, b, c
a = b
b = c
end

```

Suppose we want to call *shift(x, y, 0)* but we don't want to change the value of *y*. Knowing that built-up expressions are passed as temporaries, we decide to call *shift(x, y+0, 0)*. Our code works fine at first, but then

(with some compilers) fails when we enable optimization. What is going on? What might we do instead?

- 8.9** In some implementations of Fortran IV, the following code would print a 3. Can you suggest an explanation? How do you suppose more recent Fortran implementations get around the problem?

```
c main program
 call foo(2)
 print*, 2
 stop
 end
 subroutine foo(x)
 x = x + 1
 return
 end
```

- 8.10** Suppose you are writing a program in which all parameters must be passed by name. Can you write a subroutine that will swap the values of its actual parameters? Explain. (*Hint:* Consider mutually dependent parameters like *i* and *A[i]*.)

- 8.11** Can you write a *swap* routine in Java, or in any other language with only call-by-sharing parameters? What exactly should *swap* *do* in such a language?

- 8.12** As noted in Section 8.3.1, out parameters in Ada 83 can be written by the callee but not read. In Ada 95 they can be both read and written, but they begin their life uninitialized. Why do you think the designers of Ada 95 made this change? Does it have any drawbacks?

- 8.13** Fields of *packed* records (Example 7.39) cannot be passed by reference in Pascal. Likewise, when passing a subrange variable by reference, Pascal requires that all possible values of the corresponding formal parameter be valid for the subrange:

```
type small = 1..100;
begin
 R = record x, y : small; end;
 S = packed record x, y : small; end;
var a : 1..10;
 b : 1..1000;
 c : R;
 d : S;
procedure foo(var n : small);
begin
 n := 100;
 writeln(a);
end;
...
a := 2;
foo(b); (* ok *)
```

```

foo(a); (* static semantic error *)
foo(c.x); (* ok *)
foo(d.x); (* static semantic error *)

```

Using what you have learned about parameter-passing modes, explain these language restrictions.

- 8.14** Consider the following declaration in C.

```
double (*foo(double (*)(double, double[]), double)) (double, ...);
```

Describe in English the type of `foo`.

- 8.15** Does a program run faster when the programmer leaves optional parameters out of a subroutine call? Why or why not?
- 8.16** Why do you suppose that variable-length argument lists are so seldom supported by high-level programming languages?
- 8.17** In Section 7.2.2 we introduced the notion of a *generic reference type* (called `void *` in C) that refers to an object of unknown type. Using such references, implement a “poor man’s generic queue” in C, as suggested at the beginning of Section 8.4. Where do you need type casts? Why? Give an example of a use of the queue that will fail catastrophically at run time, due to the lack of type checking.
- 8.18** Rewrite the code of Figure 8.4 in Ada, Java, or C#.
- 8.19** (a) Give a generic solution to Exercise 6.15.  
 (b) Translate this solution into Ada, Java, or C#.
- 8.20** In your favorite language with generics, write code for simple versions of the following abstractions.
- (a) A stack, implemented as a linked list
  - (b) A priority queue, implemented as a skip list or a partially ordered tree embedded in an array
  - (c) A dictionary (mapping), implemented as a hash table
- 8.21** Figure 8.4 (C++ version) passes integer `max_items` to the `queue` abstraction as a generic parameter. Write an alternative version of the code that makes `max_items` a parameter to the `queue` constructor instead. What is the advantage of the generic parameter version?
- 8.22** Flesh out the C++ sorting routine of Example 8.39. Demonstrate that this routine does “the wrong thing” when asked to sort an array of `char*` strings.
- 8.23** In the discussion of Example 8.39 we mentioned three ways to make the need for comparisons more explicit when defining a generic sort routine in C++: we can make the comparison routine a method of the generic parameter class `T`, an extra argument to the sort routine, or an extra generic

parameter. Implement each of these options and discuss their comparative strengths and weaknesses.

- 8.24** Consider the C++ program shown in Figure 8.6. Explain why the final call to `first_n` generates a compile-time error, but the call to `last_n` does not. (Note that `first_n` is generic, but `last_n` is not.) Show how to modify the final call to `first_n` so that the compiler will accept it.
- 8.25** Consider the following code skeleton in C++.

```
#include <list>
using std::list;

class foo { ... };
class bar : public foo { ... };

static void print_all(list<foo*> &L) { ...

 list<foo*> LF;
 list<bar*> LB;
 ...
 print_all(LF); // works fine
 print_all(LB); // static semantic error
}
```

Explain why the compiler won't allow the second call. Give an example of bad things that could happen if it did.

- 8.26** In Section 8.3.1 we noted that Ada does not permit subroutines to be passed as parameters, but that some of the same effect can be achieved with generics. Suppose we want to apply a function to every member of an array. We might write the following in Ada.

```
generic
 type item is private;
 type item_array is array (integer range <>) of item;
 with function F(it : in item) return item;
 procedure apply_to_array(A : in out item_array);

procedure apply_to_array(A : in out item_array) is
begin
 for i in A'first..A'last loop
 A(i) := F(A(i));
 end loop;
end apply_to_array;
```

Given an array of integers, `scores`, and a function on integers, `foo`, we can write the following.

```
procedure apply_to_ints is new apply_to_array(integer, int_array, foo);
...
apply_to_ints(scores);
```

```

#include <iostream>
#include <list>
using std::cout;
using std::list;

template<class T> void first_n(list<T> p, int n) {
 for (typename list<T>::iterator li = p.begin(); li != p.end(); li++) {
 if (n-- <= 0) break;
 cout << *li << " ";
 }
 cout << "\n";
}

void last_n(list<int> p, int n) {
 for (list<int>::reverse_iterator li = p.rbegin(); li != p.rend(); li++) {
 if (n-- <= 0) break;
 cout << *li << " ";
 }
 cout << "\n";
}

class int_list_box {
 list<int> content;
public:
 int_list_box(list<int> l) { content = l; }
 operator list<int>() { return content; }
 // user-supplied operator for coercion/conversion
};

int main() {
 int i = 5;
 list<int> l;

 for (int i = 0; i < 10; i++) l.push_back(i);
 int_list_box b(l);

 first_n(l, i); // works
 last_n(b, i); // works (coerces b)
 first_n(b, i); // static semantic error
}

```

**Figure 8.6** Coercion and generics in C++. The compiler refuses to accept the final call to `first_n`.

How general is this mechanism? What are its limitations? Is it a reasonable substitute for formal (i.e., second-class, as opposed to third-class) subroutines?

- 8.27** Modify the code of Figure 8.4 (C++ version) or your solution to Exercise 8.21 to throw an exception if an attempt is made to enqueue an item in a full queue or to dequeue an item from an empty queue.
- 8.28** Algol 60 allows labels to be passed as parameters, and allows `gos` to these labels. One can imagine using these “nonlocal `gos`” to escape from a nested function in response to unexpected conditions, but modern language designers regard label parameters as a bad idea. In what way(s) is the exception handling of Ada, Modula-3, or C++/Java/C# better?
- 8.29** Building on Exercise 6.31, show how to implement exceptions using `call-with-current-continuation` in Scheme. Model your syntax after the `handler-case` of Common Lisp.
- 8.30** Given what you have learned about the implementation of structured exceptions, describe how you might implement the nonlocal `gos` of Pascal or the label parameters of Algol 60 (Section 6.2). Do you need to place any restrictions on how these features can be used?
- 8.31** Use coroutines to build support for iterators in Modula-2. Your code should allow the programmer to create new iterators easily. Try to hide as much of the implementation as possible inside a module. In particular, hide the use of `transfer` inside implementations of routines named `yield` (to be called by an iterator coroutine) and `next` (to be called in the body of a loop). Discuss any weaknesses you encounter in the abstraction facilities of the language.
- 8.32** In Common Lisp multilevel returns use `catch` and `throw`; exception handling in the style of most other modern languages uses `handler-case` and `error`. Show that the distinction between these is mainly a matter of style, rather than expressive power. In other words, show that each facility can be used to emulate the other.
- ⌚ **8.34–8.43** In More Depth.

## 8.9 Explorations

- 8.44** Obtain a copy of the GNU Ada translator `gnat`. Explore its subroutine calling conventions. How do they compare to those of `gpc`? Pay particular attention to language features present in Ada but not in Pascal, including declarations in nested blocks (Section 3.3.2), dynamic-size arrays (Section 7.4.2), value-result parameters (Section 8.3.1), optional and named parameters (Section 8.3.3), generic subroutines (Section 8.4), exceptions (Section 8.5), and concurrency (Section 12.2.3).
- 8.45** If you were designing a new imperative language, what set of parameter modes would you pick? Why?

- 8.46** Learn about references and the reference assignment operator in PHP. Discuss the similarities and differences between these and the references of C++. In particular, note that assignments in PHP can change the object to which a reference variable refers. Why does PHP allow this but C++ does not?
- 8.47** Find manuals for several languages with exceptions and look up the set of predefined exceptions—those that may be raised automatically by the language implementation. Discuss the differences among the sets defined by different languages. If you were designing an exception-handling facility, what exceptions, if any, would you make predefined? Why?
- 8.48** Learn the details of nonlocal control transfer in Common Lisp. Write a tutorial that explains `tagbody` and `go`; `block` and `return-from`; `catch` and `throw`; and `restart-case`, `restart-bind`, `handler-case`, `handler-bind`, `find-restart`, `invoke-restart`, `ignore-errors`, `signal`, and `error`. What do you think of all this machinery? Is it overkill? Be sure to give an example that illustrates the use of `handler-bind`.
- 8.49** If you have manuals for Common Lisp, Modula-3, and Java, compare the semantics they provide for `unwind-protect` and `try...finally`. Specifically, what happens if an exception arises within a cleanup clause?
- © **8.50–8.52** In More Depth.

## 8.10 Bibliographic Notes

Recursive subroutines became known primarily through McCarthy’s work on Lisp [McC60].<sup>8</sup> Stack-based space management for recursive subroutines developed with compilers for Algol 60 (see for example Randell and Russell [RR64]). (Because of issues of extent, subroutine space in Lisp requires more general, heap-based allocation.) Dijkstra [Dij60] presents an early discussion of the use of displays to access nonlocal data. Hanson [Han81] argues that nested subroutines are unnecessary.

Calling sequences and stack conventions for `gpc` are partially documented in the `texinfo` files distributed with `gcc`, on which `gpc` is based (see <http://www.gnu.org/software>). Documentation for the MIPSpro C compiler can be found at [techpubs.sgi.com](http://techpubs.sgi.com). Several of the details described on the PLP CD were “reverse engineered” by examining the output of the two compilers.

The Ada language rationale [IBFW91, Chap. 8] contains an excellent discussion of parameter-passing modes. Harbison [Har92, Secs. 6.2–6.3] describes

---

**8** John McCarthy (1927–), Professor Emeritus at Stanford University, is one of the founders of the field of Artificial Intelligence. He introduced Lisp in 1958, and also made key contributions to the early development of time-sharing and the use of mathematical logic to reason about computer programs. He received the ACM Turing Award in 1971.

the Modula-3 modes and compares them to those of other languages. Liskov and Guttag [LG86, p. 25] liken call-by-sharing in Clu to parameter passing in Lisp. Call-by-name parameters have their roots in the lambda calculus of Alonzo Church [Chu41], which we consider in more detail in Section 10.6.1. Thunks were first described by Ingberman [Ing61]. Fleck [Fle76] discusses the problems involved in trying to write a `swap` routine with call-by-name parameters (Exercise 8.10).

Garcia et al. provide a detailed comparison of generic facilities in ML, C++, Haskell, Eiffel, Java, and C# [GJL<sup>+</sup>03]. The C# generic facility is described by Kennedy and Syme [KS01]. Java generics are based on the work of Bracha et al. [BOSW98].

MacLaren [Mac77] describes exception handling in PL/I. The lexically scoped alternative of Ada, and of several more recent languages, draws heavily on the work of Goodenough [Goo75]. Ada's semantics are described formally by Luckam and Polak [LP80]. Liskov and Snyder [LS79] discuss exception handling in Clu. Friedman, Wand, and Haynes [FWH01, Chaps. 8–9] provide an excellent explanation of continuation-passing style in Scheme.

An early description of coroutines appears in the work of Conway [Con63], who uses them to represent the phases of compilation. Birtwistle et al. [BDMN73] provide a tutorial introduction to the use of coroutines for simulation in Simula 67. Cactus stacks date from at least the mid-1960s; they were supported directly in hardware by the Burroughs B6500 and B7500 computers [HD68]. Murer et al. [MOSS96] discuss the implementation of iterators in the Sather programming language (a descendant of Eiffel).



# Data Abstraction and Object Orientation

In Chapter 3 we presented several stages in the development of data abstraction, with an emphasis on the scoping mechanisms that control the visibility of names. We began with global variables, whose lifetime spans program execution. We then added local variables, whose lifetime is limited to the execution of a single subroutine; nested scopes, which allow subroutines themselves to be local; and static variables, whose lifetime spans execution but whose names are visible only within a single scope. These were followed by modules, which allow a collection of subroutines to share a set of static variables; module *types*, which allow the programmer to instantiate multiple instances of a given abstraction, and *classes*, which allow the programmer to define families of related abstractions.

Ordinary modules encourage a “manager” style of programming, in which a module exports an abstract type. Module types and classes allow the module itself to *be* the abstract type. The distinction becomes apparent in two ways: First, the explicit *create* and *destroy* routines typically exported from a manager module are replaced by creation and destruction of an instance of the module type. Second, invocation of a routine in a particular module instance replaces invocation of a general routine that expects a variable of the exported type as argument. Classes build on the module-as-type approach by adding mechanisms for *inheritance*, which allows new abstractions to be defined as refinements or extensions to existing ones, and *dynamic method binding*, which allows a new version of an abstraction to display newly refined behavior, even when used in a context that expects an earlier version. An instance of a class is known as an *object*; languages and programming techniques based on classes are said to be *object-oriented*.<sup>1</sup>

The stepwise evolution of data abstraction mechanisms presented in Chapter 3 is a useful way to organize ideas, but it does not completely reflect the historical

---

<sup>1</sup> In previous chapters we used the term “object” informally to refer to almost anything that can have a name. In this chapter we use it only to refer to an instance of a class.

development of language features. In particular, it would be inaccurate to suggest that object-oriented programming developed as an outgrowth of modules. Rather, all three of the fundamental concepts of object-oriented programming—encapsulation, inheritance, and dynamic method binding—have their roots in the Simula programming language, developed in the mid-1960s by Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Centre.<sup>2</sup> In comparison to modern object-oriented languages, Simula was weak in the data hiding part of encapsulation, and it was in this area that Clu, Modula, Euclid, and related languages made important contributions in the 1970s. At the same time, the ideas of inheritance and dynamic method binding were adopted and refined in Smalltalk over the course of the 1970s. Smalltalk employs a distinctive “message-based” programming model, with dynamic typing and unusual terminology and syntax. The dynamic typing tends to make Smalltalk implementations relatively slow, and delays the reporting of errors. The language is also tightly integrated into a graphical programming environment, making it difficult to port to other systems. For these reasons, Smalltalk is less widely used than one might expect, given the influence it has had on subsequent developments. More recent object-oriented languages, including Eiffel, C++, Modula-3, Ada 95, Python, Ruby, Java, and C# represent to a large extent a reintegration of the inheritance and dynamic method binding of Smalltalk with “mainstream” imperative syntax and semantics. In an alternative vein, Objective-C [App04] combines Smalltalk-style messaging and dynamic typing, in a relatively pure and unadulterated form, with traditional C syntax for intra-object operations. Object orientation has also become important in functional languages; the leading notation is CLOS, the Common Lisp Object System [Kee89; Ste90, Chap. 28].

In Section 9.1 we provide an overview of object-oriented programming and of its three fundamental concepts. We consider encapsulation and data hiding in more detail in Section 9.2. We then consider object initialization and finalization in Section 9.3, and dynamic method binding in Section 9.4. In Section 9.5 (mostly on the PLP CD) we consider the subject of *multiple inheritance*, in which a class is defined in terms of more than one existing class. As we shall see, multiple inheritance introduces some particularly thorny semantic and implementation challenges. Finally, in Section 9.6, we revisit the definition of object orientation, considering the extent to which a language can or should model everything as an object. Most of our discussion will focus on Smalltalk, Eiffel, C++, and Java, though we shall have occasion to mention Simula, Modula-3, Python, Ruby, Ada 95, C#, Objective-C, Oberon, and CLOS as well.

---

**2** Kristen Nygaard (1926–2002) was widely admired as a mathematician, computer language pioneer, and social activist. His career included positions with the Norwegian Defense Research Establishment, the Norwegian Operational Research Society, the Norwegian Computing Center, the Universities of Aarhus and Oslo, and a variety of labor, political, and social organizations. Ole-Johan Dahl (1931–2002) also held positions at the Norwegian Defense Research Establishment and the Norwegian Computing Center, and was the founding member of the Informatics department at Oslo. Together, Nygaard and Dahl shared the 2001 ACM Turing Award.

## 9.1 Object-Oriented Programming

With the development of ever-more complicated computer applications, data abstraction has become essential to software engineering. The abstraction provided by modules and module types has at least three important benefits.

1. It reduces *conceptual load* by minimizing the amount of detail that the programmer must think about at one time.
2. It provides *fault containment* by preventing the programmer from using a program component in inappropriate ways, and by limiting the portion of a program's text in which a given component can be used, thereby limiting the portion that must be considered when searching for the cause of a bug.
3. It provides a significant degree of *independence* among program components, making it easier to assign their construction to separate individuals, to modify their internal implementations without changing external code that uses them, or to install them in a library where they can be used by other programs.

Unfortunately, experience with modules and module types indicates that the reuse implied by the third of these points is difficult to achieve in practice. One often finds that a previously constructed module has almost, but not quite, the properties required by some new application. Perhaps one has a preexisting queue abstraction but would like to be able to insert and delete from either end, rather than being limited to first-in-first-out (FIFO) order. Perhaps one has a preexisting dialog box abstraction for a graphical user interface but without any mechanism to highlight a default response. Perhaps one has a package for symbolic math, but it assumes that all values are real numbers, rather than complex. In all these cases much of the advantage of abstraction will be lost if the programmer must copy the preexisting code, figure out how it works inside, and modify it by hand, rather than using it “as-is.” If it becomes necessary to change the abstraction at some point in the future (to fix a bug or implement an enhancement), the programmer will need to remember to fix all copies—a tedious and error-prone activity.

Object-oriented programming can be seen as an attempt to enhance opportunities for code reuse by making it easy to define new abstractions as *extensions* or *refinements* of existing abstractions. As a starting point for examples, consider a list of records. Figure 9.1 contains C++ code for the elements of such a list. The example employs a “module-as-type” style of abstraction: each element of a list is an object of class `list_node`. The class contains both *data members* (`prev`, `next`, `head_node`, and `val`) and *subroutine members* (`predecessor`, `successor`, `insert_before`, and `remove`). Subroutine members are called *methods* in many object-oriented languages; data members are also called *fields*. The keyword `this` in C++ refers to the object of which the currently executing method is a member. In Smalltalk and Objective-C, the equivalent keyword is `self`; in Eiffel it is `current`.

---

**EXAMPLE 9.1**

`List_node` class in C++

```

class list_err { // exception
public:
 char *description;
 list_err(char *s) {description = s;}
};

class list_node {
 list_node* prev;
 list_node* next;
 list_node* head_node;
public:
 int val; // the actual data in a node
 list_node() { // constructor
 prev = next = head_node = this; // point to self
 val = 0; // default value
 }
 list_node* predecessor() {
 if (prev == this || prev == head_node) return 0;
 return prev;
 }
 list_node* successor() {
 if (next == this || next == head_node) return 0;
 return next;
 }
 bool singleton() {
 return (prev == this);
 }
 void insert_before(list_node* new_node) {
 if (!new_node->singleton())
 throw new list_err("attempt to insert node already on list");
 prev->next = new_node;
 new_node->prev = prev;
 new_node->next = this;
 prev = new_node;
 new_node->head_node = head_node;
 }
 void remove() {
 if (singleton())
 throw new list_err(
 "attempt to remove node not currently on list");
 prev->next = next;
 next->prev = prev;
 prev = next = head_node = this; // point to self
 }
 ~list_node() { // destructor
 if (!singleton())
 throw new list_err("attempt to delete node still on list");
 }
};

```

**Figure 9.1** A simple class for list nodes in C++. In this example we envision a list of integers.

**EXAMPLE 9.2**

List class that uses  
list\_node

Given the existence of the list\_node class, we could define a list as follows.

```
class list {
 list_node header;
public:
 // no explicit constructor required;
 // implicit construction of 'header' suffices
 int empty() {
 return header.singleton();
 }
 list_node* head() {
 return header.successor();
 }
 void append(list_node *new_node) {
 header.insert_before(new_node);
 }
 ~list() { // destructor
 if (!header.singleton())
 throw new list_err("attempt to delete non-empty list");
 }
};
```

To create an empty list, one could then write

```
list* my_list_ptr = new list;
```

Records to be inserted into a list are created in much the same way:

```
list_node* elem_ptr = new list_node;
```

In C++, one can also simply declare an object of a given class:

```
list my_list;
list_node elem;
```

**EXAMPLE 9.3**

Declaration of in-line  
(expanded) objects

Our list class includes such an object (`header`) as a field. When created with `new`, an object is allocated in the heap; when created via elaboration of a declaration it is allocated statically or on the stack, depending on lifetime. In either case, creation causes the invocation of a programmer-specified initialization routine, known as a *constructor*. In C++ and its descendants, Java and C#, the name of the constructor is the same as that of the class itself. C++ also allows the programmer to specify a *destructor* method that will be invoked automatically when an object is destroyed, either by explicit programmer action or by return from the subroutine in which it was declared. The destructor's name is also the same as that of the class, but with a leading tilde (~).

#### **Public and Private Members**

The `public` label within the list of members of `list_node` separates members required by the implementation of the abstraction from members available to users of the abstraction. In the terminology of Section 3.3.4, members that appear after the `public` label are exported from the class; members that appear before

the label are not. The language also provides a `private` label, so the publicly visible portions of a class can be listed first if desired (or even intermixed). In many other languages, including C#, every data or subroutine member (field or method) is private unless individually labeled with the `public` keyword. Note that C++ classes are open scopes, as defined in Section 3.3.4; nothing needs to be explicitly imported.

Like packages in Ada or external (separately compiled) modules in Modula-2, C++ classes allow certain information to be left out of the declaration, and provided in a separate file not visible to users of the abstraction. In our running example, we could declare the public methods of `list_node` without providing their bodies:

```
class list_node {
 list_node* prev;
 list_node* next;
 list_node* head_node;
public:
 int val;
 list_node();
 list_node* predecessor();
 list_node* successor();
 bool singleton();
 void insert_before(list_node* new_node);
 void remove();
 ~list_node();
};
```

This somewhat abbreviated class declaration might then be put in a .h “header” file, with method bodies relegated to a .cc “implementation” file. (Conventions for separate compilation in C were discussed in Section 3.7. The file name suffixes used here are those expected by the GNU g++ compiler.) Within a .cc file, the header of a method definition must identify the class to which it belongs by using a `:: scope resolution operator`:

```
void list_node::insert_before(list_node* new_node) {
 if (!new_node->singleton())
 throw new list_err("attempt to insert node already on list");
 prev->next = new_node;
 new_node->prev = prev;
 new_node->next = this;
 prev = new_node;
 new_node->head_node = head_node;
}
```

### Tiny Subroutines

Object-oriented programs tend to make many more subroutine calls than do ordinary imperative programs, and the subroutines tend to be shorter. Lots of things that would be accomplished by direct access to record fields in a von Neumann language tend to be hidden inside object methods in an object-oriented

language. Many programmers in fact consider it bad style to declare public fields, because they give users of an abstraction direct access to the internal representation. Arguably, we should make the `val` field of `list_node` private, with `get_val` and `set_val` methods to read and write it.

C# provides a *property* mechanism specifically designed to facilitate the declaration of methods to “get” and “set” values. Using this mechanism, our `val` field could be written as follows.

```
class list_node {
 ...
 int val;
 public int Val {
 get { // presence of get and optional set
 return val; // methods means that Val is a property
 }
 set {
 val = value; // value is a keyword: argument to set
 }
 }
 ...
}
```

Users of the `list_node` class can now access the (private) `val` field through the (public) `Val` property as if it were a field:

```
list_node n;
...
int a = n.Val; // implicit call to get method
n.Val = 3; // implicit call to set method
```

## DESIGN & IMPLEMENTATION

### What goes in a class declaration?

Two rules govern the choice of what to put in the declaration of a class, rather than in separate definitions. First, the declaration must contain all the information that a programmer needs in order to use the abstraction correctly. Second, the declaration must contain all the information that the compiler needs in order to generate code. The second rule is generally broader: it tends to force information that is not required by the first rule into (the private part of) the interface, particularly in languages that use a value model of variables, instead of a reference model. If the compiler must generate code to allocate space (e.g., in stack frames) to hold a value of an object type, then it must know the size of the object; this is the rationale for including private fields in the class declaration. In addition, if the compiler is to expand any method calls in-line then it must have their code available. In-line expansion of the smallest, most common methods in an object-oriented program tends to be crucial for good performance.

A similar *indexer* mechanism can make objects of arbitrary classes look like arrays, with conventional subscript syntax in both l-value and r-value contexts. An example appears in the sidebar on page 351. In C++, operator overloading and references (Section 8.3.1, page 412) can be used to provide the equivalent of indexers, but not of properties. ■

### Derived Classes

**EXAMPLE 9.7**

Queue class derived from list

Suppose now that we already have a list abstraction and would like a queue abstraction. We could define the queue from scratch, but much of the code would look the same as in Figure 9.1. In an object-oriented language we have a better alternative: we can *derive* the queue from the list, allowing it to *inherit* preexisting fields and methods:

```
class queue : public list { // derive from list
public:
 // no specialized constructor or destructor required
 void enqueue(list_node* new_node) {
 append(new_node);
 }
 list_node* dequeue() {
 if (empty())
 throw new list_err("attempt to dequeue from empty queue");
 list_node* p = head();
 p->remove();
 return p;
 }
};
```

Here `queue` is said to be a *derived class* (also called a *child class* or *subclass*); `list` is said to be a *base class* (also called a *parent class* or *superclass*). The derived class automatically has all the fields and methods of the base class.<sup>3</sup> All the programmer needs to declare explicitly are members that a `queue` has but a `list` lacks: in this case, the `enqueue` and `dequeue` methods. We shall see examples shortly in which derived classes have new fields as well. ■

By deriving new classes from old ones, the programmer can create arbitrarily deep *class hierarchies*, with additional functionality at every level of the tree. The standard library for Smalltalk has as many as seven levels of derivation (Figure 9.2): class `FileStream` is derived from `ExternalStream`, which is in turn derived, in order, from `ReadWriteStream`, `WriteStream`, `PositionalStream`, `Stream`, and `Object`. (Unlike C++, Smalltalk has a single root superclass, `Object`, from which all other classes are derived. Java, C#, and Objective-C have a similar class, as does Eiffel; the latter refers to it as ANY.) ■

**EXAMPLE 9.8**

The Smalltalk class hierarchy

---

<sup>3</sup> Actually, users of a derived class in C++ can see the members of the base class only if the base class name is preceded with the keyword `public` in the first line of the derived class's declaration. We will discuss the visibility rules of C++ in more detail in Section 9.2.

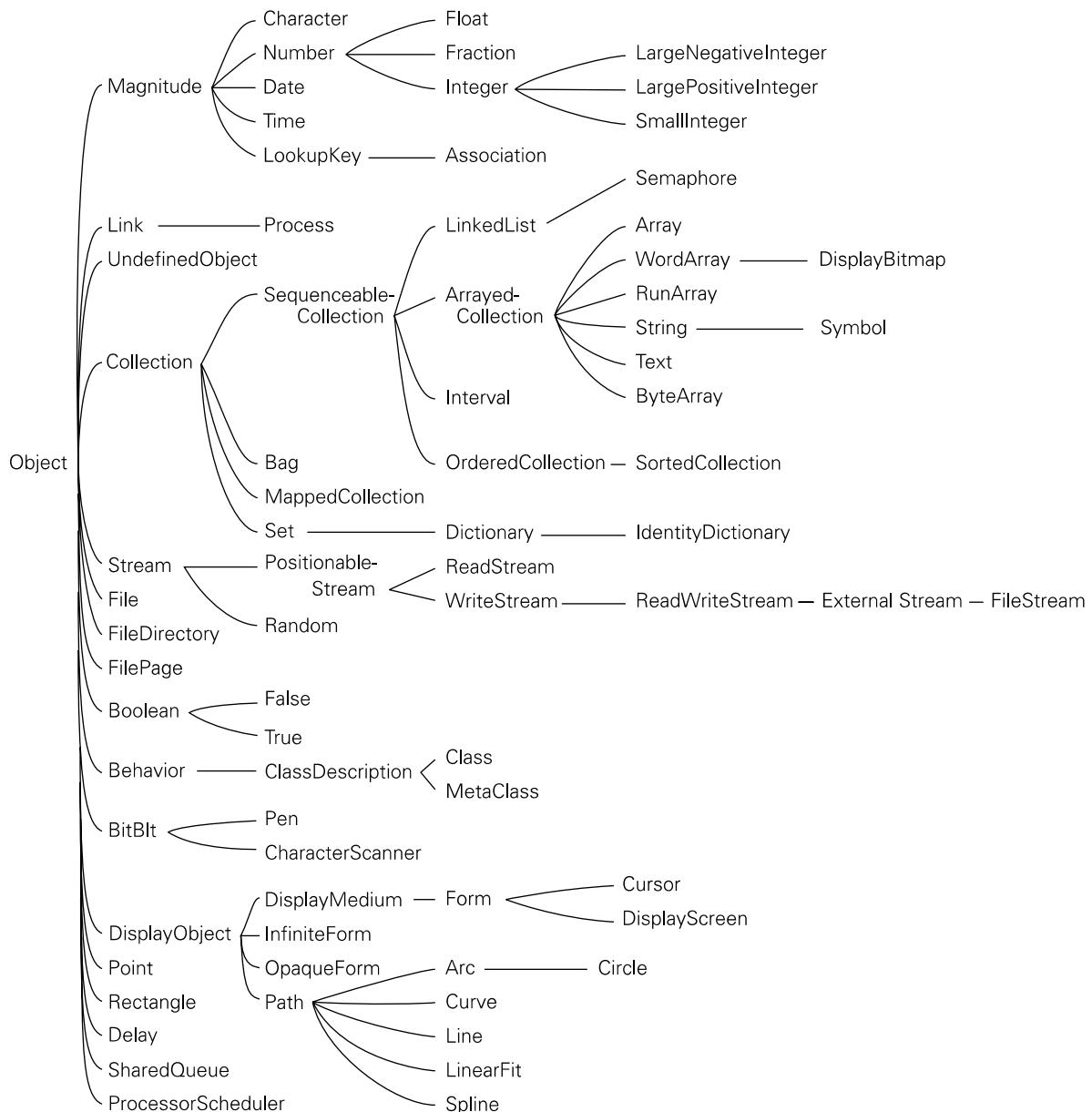


Figure 9.2 The standard class hierarchy of Smalltalk-80.

### General Purpose Base Classes

**EXAMPLE 9.9**

Base class for general purpose lists

The astute reader may have noticed that our original list abstraction made the unfortunate assumption that the data in every item was to be an integer. This assumption really isn't necessary. Given an inheritance mechanism, we can create a general purpose element base class that contains only the fields and methods needed to implement list operations:

```
class gp_list_node {
 gp_list_node* prev;
 gp_list_node* next;
 gp_list_node* head_node;
public:
 gp_list_node(); // assume method bodies given separately
 gp_list_node* predecessor();
 gp_list_node* successor();
 bool singleton();
 void insert_before(gp_list_node* new_node);
 void remove();
 ~gp_list_node();
};
```

Now we can use this general purpose class to derive lists and queues with specific types of fields:

```
class int_list_node : public gp_list_node {
public:
 int val; // the actual data in a node
 int_list_node() {
 val = 0;
 }
 int_list_node(int v) {
 val = v;
 }
};
```

Templates (generics) are commonly used to facilitate the construction of such type-specific classes; we will discuss this option further in Section 9.4.4.

### Overloaded Constructors

**EXAMPLE 9.10**

Overloaded  
int\_list\_node  
constructor

We have overloaded the constructor in `int_list_node`, providing two alternative implementations. One takes an argument; the other does not. Now the programmer can create `int_list_nodes` with or without specifying an initial value:

```
int_list_node element1; // val = 0
int_list_node *e_ptr = new int_list_node(13); // val = 13
```

In C++, the compiler ensures that constructors for base classes are executed before those of derived classes. In our example, the constructor for `gp_list_node` will be executed first, followed by the constructor for `int_list_node`. We will discuss constructors further in Section 9.3.

### Modifying Base Class Methods

#### EXAMPLE 9.11

Redefining a method in a derived class

In addition to defining new fields and methods, a derived class can hide or redefine members of base class(es). We will discuss data hiding in Section 9.2. To *redefine* a method of a base class, a derived class simply declares a new version. Suppose, for example, that we are creating an `int_list_node` class, but we want somewhat different semantics for the `remove` method. If written as in Figure 9.1, `gp_list_node::remove` will throw a `list_err` exception if the node to be removed is not currently on a list. If we want `int_list_node::remove` simply to return without doing anything in this situation, we can declare it that way explicitly:

```
class int_list_node : public gp_list_node {
public:
 ...
 void remove() {
 if (!singleton()) {
 prev->next = next;
 next->prev = prev;
 prev = next = head_node = this;
 }
 }
};
```

The disadvantage of this redefinition is that it pulls implementation details of `gp_list_node` into an `int_list_node` method, a potential violation of abstraction. (As a matter of fact, a C++ compiler will not accept the code above: as we shall see in Section 9.2, we would need to change the `gp_list_node` base class to make its `next` and `prev` fields visible to derived classes.) ■

A better approach is to leave the implementation details to the base class and simply catch the exception if it arises:

```
void int_list_node::remove() {
 try {
 gp_list_node::remove();
 } catch(list_err) {
 ; // do nothing
 }
}
```

This version of the code may be slightly slower than the previous one, depending on how `try` blocks are implemented, but it does a better job of maintaining abstraction. Note that the scope resolution operator (`::`) allows us to access the `remove` method of the base class explicitly, even though we have redefined it for `int_list_node`. ■

Other object-oriented languages provide other means of accessing the members of a base class. In Smalltalk, Objective-C, Java, and C#, one uses the keyword `base` or `super`:

#### EXAMPLE 9.12

Redefinition that builds on the base class method

#### EXAMPLE 9.13

Accessing base class members

```

gp_list_node::remove(); // C++
super.remove(); // Java
base.remove(); // C#
super remove. // Smalltalk
[super remove] // Objective-C

```

**EXAMPLE 9.14**

## Renaming methods in Eiffel

In Eiffel, one must explicitly *rename* methods inherited from a base class, in order to make them accessible:

```

class int_list_node
inherit
 gp_list_node
 rename
 remove as old_remove
 ... -- other renames
end

```

Within methods of `int_list_node`, the `remove` method of `gp_list_node` can be invoked as `old_remove`. C++ and Eiffel cannot use the keyword `super`, because it would be ambiguous in the presence of multiple inheritance.

**Containers/Collections**

In object-oriented programming, an abstraction that holds a collection of objects of some given class is often called a *container*. There are several different ways to build containers. In this section we have explored an approach in which objects are derived from a container element base class. The principal problem with this approach is that an object cannot be placed in a container unless its class is derived from the element class of the container. In order to put an *arbitrary* object into, say, a list, we can adopt an alternative approach, in which list nodes are separate objects containing *pointers* (or references) to the listed objects, rather than the data of the objects themselves. Examples of this approach can be found in the binary trees of Section 6.5.3. A third alternative is to make the list node a member (a subobject) of the listed object. In general, the design of consistent, intuitive, and useful class hierarchies is a complex and difficult art. Containers are only the tip of the iceberg.

 **CHECK YOUR UNDERSTANDING**

1. What are generally considered to be the three defining characteristics of object-oriented programming?
2. In what programming language of the 1960s does object orientation find its roots? Who invented that language? Summarize the evolution of the three defining characteristics since that time.
3. Name three important benefits of abstraction.
4. What are the more common names for *subroutine member* and *data member*?
5. What is a *property* in C#?

6. What is the purpose of the “private” part of an object interface? Why is it required?
  7. What is the purpose of the `::` operator in C++?
  8. What is a *container* class?
  9. Explain why in-line subroutines are particularly important in object-oriented languages.
  10. What are *constructors* and *destructors*?
  11. Give two other terms, each, for *base class* and *derived class*.
- 

## 9.2 Encapsulation and Inheritance

Encapsulation mechanisms enable the programmer to group data and the subroutines that operate on them together in one place, and to hide irrelevant details from the users of an abstraction. In the discussion above we have cast object-oriented programming as an extension of the “module-as-type” mechanisms of Simula and Euclid. It is also possible to cast object-oriented programming in a “module-as-manager” framework. In the first subsection below we consider the data-hiding mechanisms of modules in non-object-oriented languages. In the second subsection we consider the new data-hiding issues that arise when we add inheritance to modules to make classes. In the third subsection we briefly consider an alternative approach, in which inheritance is added to records, and (static) modules continue to provide data hiding.

### 9.2.1 Modules

Scope rules for data hiding were one of the principal innovations of Clu, Modula, Euclid, and other module-based languages of the 1970s. In Clu and Euclid, the declaration and definition (header and body) of a module always appear together. The header clearly states which of the module’s names are to be exported. If a Euclid module  $M$  exports a type  $T$ , by default the remainder of the program can do nothing with objects of type  $T$  other than pass them to subroutines exported from  $M$ .  $T$  is said to be an *opaque* type. If desired, the Euclid programmer can explicitly grant code outside the module the ability to perform bit-wise assignment and/or equality tests on values of type  $T$ , or to access  $T$ ’s fields (if a record), subscript it (if an array), or refer to its values by name (if an enumeration). In the following, for example, code outside module Database would be able to assign tuple variables to each other and to access their name fields, but not to check them for equality or to access their other fields.

---

**EXAMPLE 9.15**

Data hiding in Euclid

```

var Database : module
 exports (tuple with (:=, name))
 ...
 type tuple = record
 var name : packed array 1..80 of char
 ...
 end tuple
 ...

```

In Clu, a module (called a *cluster*) implements a single abstract type. Assignment and equality testing are permitted for that type, but because Clu uses a reference model for variables, these operations copy or compare references to objects, not the objects themselves.

**EXAMPLE 9.16**

## Opaque types in Modula-2

In Modula-2, programmers have the option of separating the header and body of a module. In Chapter 3 (Figures 3.7 and 3.8) we looked only at so-called “internal” modules, in which the two parts appear together. In an “external” module (meant for separate compilation), the header appears in one source file and the body in another. Unfortunately, there is no way to divide the header into public and private parts; everything in it is public (i.e., exported). The only concession to data hiding is that a type may be made opaque by listing only its name in the header:

```
TYPE T;
```

In this case variables of type T can only be assigned, compared for equality, and passed to the module’s subroutines. There is no way to disable assignment and comparison.

**EXAMPLE 9.17**

## Data hiding in Ada

Ada, which also allows the headers and bodies of modules (called *packages*) to be separated, eliminates the problems of Modula-2 by allowing the header of a package to be divided into public and private parts. A type can be exported opaquely by putting its definition in the private part of the header and simply naming it in the public part:

**DESIGN & IMPLEMENTATION****Opaque exports in Modula-2**

Because opaque types are not defined in a Modula-2 header module, there is no obvious way for the compiler to determine the size of an object (in the informal sense of the word) of an opaque type when compiling code that uses the module. Modula-2 therefore requires that all opaque types be pointers. Assuming that all pointers have the same size (which they do on most machines), objects of opaque type can then be allocated statically or on the stack without knowledge of internal structure. Some Modula-2 implementations permit certain additional opaque types, but only if they are implemented with the same number of bits as a pointer.

```

package foo is -- header
...
type T is private;
...
private -- definitions below here are inaccessible to users
...
type T is ... -- full definition
...
end foo;

```

Variables of *private* types can be assigned, compared for equality, or passed to subroutines of the package. Optionally, the Ada programmer can disable assignment and comparison with a more restrictive declaration:

```
type T is limited private; -- replaces third line of example
```

When the header and body of a module appear in separate files, a change to a module body never requires us to recompile any of the module's users. A change to the *private* part of a module header may require us to recompile the module's users, but never requires us to change their code. A change to the *public* part of a header is a change to the module's interface: it will often require us to change the code of users.

Because they affect only the visibility of names, static, manager-style modules introduce no special code generation issues. Storage for variables and other data inside a module is managed in precisely the same way as storage for data immediately outside the module. If the module appears in a global scope, then its data can be allocated statically. If the module appears within a subroutine, then its data can be allocated on the stack, at known offsets, when the subroutine is called, and reclaimed when it returns.

Module types, as in Euclid, are somewhat more complicated: they allow a module to have an arbitrary number of *instances*. The obvious implementation then resembles that of a record. If all of the data in the module have a statically known size, then each individual datum can be assigned a static offset within the module's storage. If the size of some of the data is not known until run time, then the module's storage can be divided into fixed-size and variable-size portions, with a dope vector (descriptor) at the beginning of the fixed-size portion. Instances of the module can be allocated statically, on the stack, or in the heap, as appropriate.

#### **The “this” Parameter**

One additional complication arises for subroutines inside a module. How do they know which variables to use? We could, of course, replicate the code for each subroutine in each instance of the module, just as we replicate the data. This replication would be highly wasteful, however, as the copies would vary only in the details of address computations. A better technique is to create a single instance of each module subroutine, and to pass that instance, at run time, the address of the storage of the appropriate module instance. This address takes the form of an

**EXAMPLE 9.18**

The hidden `this` parameter

extra, hidden first parameter for every module subroutine. A Euclid call of the form

```
my_stack.push(x)
```

is translated as if it were really

```
push(my_stack, x)
```

where `my_stack` is passed by reference. ■

### Making Do Without Module Headers

As noted in Section 3.7, Java packages and C++/C# namespaces can be spread across multiple compilation units (files). In C++ and C#, a single file can also contain pieces of more than one namespace. More significantly, Java and C# dispense with the notion of separate headers and bodies. While the programmer must still define the interface (and specify it via `public` declarations), there is no need to manually identify code that needs to be in the header for implementation reasons: instead the compiler is responsible for extracting this information automatically from the full text of the module. For software engineering purposes it may still be desirable to create preliminary versions of a module, against which other modules can be compiled, but this is optional. To assist in project management and documentation, many Java and C# implementations provide a tool that will extract from the complete text of a module the minimum information required by its users.

### 9.2.2 Classes

With the introduction of inheritance, object-oriented languages must supplement the scope rules of module-based languages to cover additional issues. For example, should private members of a base class be visible to methods of a derived class? Should public members of a base class always be public members of a derived class (i.e., be visible to users of the derived class)? How much control should a base class exercise over the visibility of its members in derived classes?

**EXAMPLE 9.19**

Private base class in C++

We glossed over most of these questions in our examples in Section 9.1. For example, we might want to hide the `append` method of a `queue`, since it is superseded by `enqueue`. To effect this hiding in C++, the definition of class `queue` can specify that its base class is to be `private`:

```
class queue : private list {
public:
 using list::empty;
 using list::head;
 // but NOT using list::append
 void enqueue(gp_list_node* new_node);
 gp_list_node* dequeue();
};
```

Here the appearance of `private` in the first line of the declaration indicates that public members of `list` will be visible to users of `queue` only if specifically made so by later parts of the declaration. We have made the `empty` and `head` methods visible by means of using declarations in `queue`'s public part. ■

In addition to the `public` and `private` labels, C++ allows members of a class to be designated `protected`. A protected member is visible only to methods of its own class or of classes derived from that class. In our examples, a protected member `M` of `list` would be accessible not only to methods of `list` itself, but also to methods of `queue`. Unlike public members, however, `M` would not be visible to arbitrary users of `list` or `queue` objects.

**EXAMPLE 9.20**

Protected base class in C++

The `protected` keyword can also be used when specifying a base class:

```
class derived : protected base { ... }
```

Here public members of the base class act like protected members of the derived class. ■

The basic philosophy behind the visibility rules of C++ can be summarized as follows.

- Any class can limit the visibility of its members. Public members are visible anywhere the class declaration is in scope. Private members are visible only inside the class's methods. Protected members are visible inside methods of the class or its descendants. (As an exception to the normal rules, a class can specify that certain other `friend` classes or subroutines should have access to its private members.)
- A derived class can restrict the visibility of members of a base class but can never increase it. Private members of a base class are never visible in a derived class. Protected and public members of a public base class are protected or public, respectively, in a derived class. Protected and public members of a protected base class are protected members of a derived class. Protected and public members of a private base class are private members of a derived class.
- A derived class that limits the visibility of members of a base class by declaring that base class `protected` or `private` can restore the visibility of individual members of the base class by inserting a `using` declaration in the `protected` or `public` portion of the derived class declaration.

Other object-oriented languages take different approaches to visibility. Eiffel is more flexible than C++ in the patterns of visibility it can support, but it does not adhere to the first of the C++ principles above. Derived classes in Eiffel can both restrict and increase the visibility of members of base classes. Every method (called a `feature` in Eiffel) can specify its own *export status*. If the status is `{NONE}`, then the member is effectively private (called *secret* in Eiffel). If the status is `{ANY}`, then the member is effectively public (called *generally available* in Eiffel). In the general case the status can be an arbitrary list of class names, in which case the feature is said to be *selectively available* to those classes and their descendants only. Any feature inherited from a base class can be given a new status in a derived class.

Java and C# follow C++ in the declaration of `public`, `protected`, and `private` members, but do not provide the `protected` and `private` designations for base classes; a derived class can neither increase *nor* restrict the visibility of members of a base class. (Of course, a derived class can always redefine a data or subroutine member with a method that generates a run-time error if used.) The `protected` keyword has a slightly different meaning in Java than it does in C++: a `protected` member of a Java class is visible not only within derived classes, but also within the entire package (namespace) in which the class is declared. A class member with no explicit access modifier in Java is visible throughout the package in which the class is declared, but *not* in any derived classes that reside in other packages. C# defines `protected` as C++ does, but provides an additional `internal` keyword that makes a member visible throughout the *assembly* in which the class appears. (An assembly is a collection of linked-together compilation units, comparable to a `.jar` file in Java.) Members of a C# class are `private` by default.

In Smalltalk and Objective-C, the issue of member visibility never arises: the language allows code at run time to attempt to make a call to any method name in any object. If the object has a method of the given name (with the right number of parameters), then the invocation proceeds; otherwise a run-time error results. There is no way in these languages to make a method available to some parts of a program but not to others. In a related vein, Python class members are always `public`.

### 9.2.3 Type Extensions

Smalltalk, Objective-C, Eiffel, C++, Java, and C# were all designed from the outset as object-oriented languages, either starting from scratch or from an existing language without a strong encapsulation mechanism. They all support a module-as-type approach to abstraction, in which a single mechanism (the class) provides both encapsulation and inheritance. Several other languages, including Modula-3, Ada 95, Oberon, CLOS, and Fortran 2003, can be characterized as object-oriented extensions to languages in which modules already provide encapsulation. (Neither Modula-3 nor Oberon is strictly an extension to Modula-2, but both draw heavily on the syntax and semantics of their common predecessor.) Rather than alter the existing module mechanism, these languages provide inheritance and dynamic method binding through a mechanism for *extending* records. In Ada 95, for example, our list and queue abstractions could be defined as in Figure 9.3.

---

**EXAMPLE 9.21**

List and queue abstractions  
in Ada 95

To control access to the structure of types, we hide them inside Ada packages. The procedures `initialize`, `finalize`, `enqueue`, and `dequeue` of `gp_list.queue` can convert their parameter `self` to a `list_ptr`, because `queue` is an extension of `list`. Package `gp_list.queue` is said to be a *child* of package `gp_list` because its name is prefixed with that of its parent. A child package in Ada 95 is similar to a derived class in Eiffel or C++, except that it is still a man-

```

package gp_list is
 list_err : exception;
 type gp_list_node is tagged private;
 -- 'tagged' means extendible; 'private' means opaque
 type gp_list_node_ptr is access all gp_list_node;
 -- 'all' means that this can point at 'aliased' nonheap data
 procedure initialize(self : access gp_list_node);
 procedure finalize(self : access gp_list_node);
 function predecessor(self : access gp_list_node) return gp_list_node_ptr;
 function successor(self : access gp_list_node) return gp_list_node_ptr;
 function singleton(self : access gp_list_node) return boolean;
 procedure insert_before(self : access gp_list_node;
 new_node : gp_list_node_ptr);
 procedure remove(self : access gp_list_node);

 type list is tagged private;
 type list_ptr is access all list;
 procedure initialize(self : access list);
 procedure finalize(self : access list);
 function empty(self : access list) return boolean;
 function head(self : access list) return gp_list_node_ptr;
 procedure append(self : access list; new_node : gp_list_node_ptr);
private
 type gp_list_node is tagged record
 prev, next, head_node : gp_list_node_ptr;
 end record;
 type list is tagged record
 header : aliased gp_list_node;
 -- 'aliased' means that an 'all' pointer can refer to this
 end record;
end gp_list;
...
package body gp_list is
 -- definitions of subroutines
 ...
end gp_list;
...
package gp_list.queue is -- 'child' of gp_list
 type queue is new list with private
 -- 'new' means it's a subtype; 'with' means it's an extension
 procedure initialize(self : access queue);
 procedure finalize(self : access queue);
 procedure enqueue(self : access queue;
 new_node : gp_list_node_ptr);
 function dequeue(self : access queue) return gp_list_node_ptr;
private
 type queue is new list with null record; -- no new fields
end gp_list.queue;

```

**Figure 9.3** List and queue abstractions in Ada 95. The tagged types `list` and `queue` provide inheritance; the packages provide encapsulation. An `int_list_node` could be derived from `gp_list_node` in a similar manner. Declaring `self` to have type `access XX` (instead of `XX_ptr`) causes the compiler to recognize the subroutine as a method of the tagged type. (continued)

```

package body gp_list.queue is
 procedure initialize(self : access queue) is
 begin
 initialize(list_ptr(self));
 end initialize;

 procedure finalize(self : access queue) is
 begin
 finalize(list_ptr(self));
 end finalize;

 procedure enqueue(self : access queue; new_node : gp_list_node_ptr) is
 begin
 append(list_ptr(self), new_node);
 end enqueue;

 function dequeue(self : access queue) return gp_list_node_ptr is
 rtn : gp_list_node_ptr;
 begin
 if empty(list_ptr(self)) then
 raise list_err;
 end if;
 rtn := head(list_ptr(self));
 remove(rtn);
 return rtn;
 end dequeue;
end gp_list.queue;

```

**Figure 9.3 (continued)**

ager, not a type. Like Eiffel, but unlike C++, Ada 95 allows the body of a child package to see the private parts of the parent package.

All of the list and queue subroutines in Figure 9.3 take an explicit first parameter; Ada 95, Oberon, and CLOS do not use “*object.method()*” notation. Modula-3 and Python do use this notation, but only as syntactic sugar: a call to *A.B(C, D)* is interpreted as a call to *B(A, C, D)*, where *B* is declared as a three-parameter subroutine. Arbitrary Ada code can pass an object of type *queue* to any routine that expects a *list*; as in Java, there is no way for a derived type to hide the public members of a base type. ■

### CHECK YOUR UNDERSTANDING

---

12. What is meant by an *opaque* export from a module?
13. What are *private* and *limited private* types in Ada?
14. Explain the significance of the *this* parameter in object-oriented languages.
15. How do Java and C# make do without explicit class headers?

16. Explain the distinction between `private`, `protected`, and `public` class members in C++.
  17. Explain the distinction between `private`, `protected`, and `public` base classes in C++.
  18. Describe the notion of *selective availability* in Eiffel.
  19. How do the rules for member name visibility in Smalltalk and Objective-C differ from the rules of most other object-oriented languages?
  20. Describe the key design difference between the object-oriented features of Smalltalk, Eiffel, and C++ on the one hand, and Oberon, Modula-3, and Ada 95 on the other.
- 

## 9.3 Initialization and Finalization

In Section 3.2 we defined the lifetime of an object to be the interval during which it occupies space and can thus hold data. Most object-oriented languages provide some sort of special mechanism to *initialize* an object automatically at the beginning of its lifetime. When written in the form of a subroutine, this mechanism is known as a *constructor*. Though the name might be thought to imply otherwise, a constructor does not allocate space; it initializes space that has already been allocated. A few languages provide a similar *destructor* mechanism to *finalize* an object automatically at the end of its lifetime. Several important issues arise.

*choosing a constructor:* An object-oriented language may permit a class to have zero, one, or many distinct constructors. In the latter case, different constructors may have different names, or it may be necessary to distinguish among them by number and types of arguments.

*references and values:* If variables are references, then every object must be created explicitly, and it is easy to ensure that an appropriate constructor is called. If variables are values, then object creation can happen implicitly as a result of elaboration. In this latter case, the language must either permit objects to begin their lifetime uninitialized, or it must provide a way to choose an appropriate constructor for every elaborated object.

*execution order:* When an object of a derived class is created in C++, the compiler guarantees that the constructors for any base classes will be executed, outermost first, before the constructor for the derived class. Moreover, if a class has members that are themselves objects of some class, then the constructors for the members will be called before the constructor for the object in which they are contained. These rules are a source of considerable syntactic and semantic complexity: when combined with multiple constructors, elaborated objects, and multiple inheritance they can sometimes induce a complicated

sequence of nested constructor invocations, with overload resolution, before control even enters a given scope. Other languages have simpler rules.

*garbage collection:* Most object-oriented languages provide some sort of constructor mechanism. Destructors are comparatively rare. Their principal purpose is to facilitate manual storage reclamation in languages like C++. If the language implementation collects garbage automatically, then the need for destructors is greatly reduced.

In the remainder of this section we consider these issues in more detail.

### 9.3.1 Choosing a Constructor

Smalltalk, Eiffel, C++, Java, and C# all allow the programmer to specify more than one constructor for a given class. In C++, Java, and C#, the constructors behave like overloaded subroutines: they must be distinguished by their numbers and types of arguments. In Smalltalk and Eiffel, different constructors can have different names; code that creates an object must name a constructor explicitly. In Eiffel one might say

```
class COMPLEX
creation
 new_cartesian, new_polar
feature {ANY}
 x, y : REAL

 new_cartesian(x_val, y_val : REAL) is
 do
 x := x_val; y := y_val
 end

 new_polar(rho, theta : REAL) is
 do
 x := rho * cos(theta)
 y := rho * sin(theta)
 end

 -- other public methods

feature {NONE}

 -- private methods

end -- class COMPLEX
...
a, b : COMPLEX
...
!!b.new_cartesian(0, 1)
!!a.new_polar(pi/2, 1)
```

---

**EXAMPLE 9.22**

Naming constructors in Eiffel

The `!!` operator is Eiffel's equivalent of `new`. Because class `COMPLEX` specified constructor ("creator") methods, the compiler will insist that every use of `!!` specify a constructor name and arguments. There is no straightforward analog of this code in C++; the fact that both constructors take two `real` arguments means that they could not be distinguished by overloading. ■

Smalltalk resembles Eiffel in the use of multiple named constructors, but it distinguishes more sharply between operations that pertain to an individual object and operations that pertain to a class of objects. Smalltalk also adopts an anthropomorphic programming model in which every operation is seen as being executed by some specific object in response to a request (a "message") from some other object. Since it makes little sense for an object  $O$  to create itself,  $O$  must be created by some other object (call it  $C$ ) that represents  $O$ 's class. Of course, because  $C$  is an object, it must itself belong to some class. The result of this reasoning is a system in which each class definition really introduces a *pair* of classes and a pair of objects to represent them. Objective-C and CLOS have similar dual hierarchies.

#### EXAMPLE 9.23

##### Metaclasses in Smalltalk

Consider, for example, the standard class named `Date`. Corresponding to `Date` is a single object (call it  $D$ ) that performs operations on behalf of the class. In particular, it is  $D$  that creates new objects of class `Date`. Because only objects execute operations (classes don't), we don't really need a name for  $D$ ; we can simply use the name of the class it represents:

```
todaysDate <- Date today
```

This code causes  $D$  to execute the `today` constructor of class `Date`, and assigns a reference to the newly created object into a variable named `todaysDate`.

So what is the class of  $D$ ? It clearly isn't `Date`, because  $D$  represents class `Date`. Smalltalk says that  $D$  is an object (in fact the only object) of the *metaclass* `Date class`. For technical reasons, it is also necessary for `Date class` to be represented by an object. To avoid an infinite regression, all objects that represent metaclasses are instances of a single class named `Metaclass`. ■

Modula-3 and Oberon provide no constructors at all: the programmer must initialize everything explicitly. Ada 95 supports constructors and destructors (called `Initialize` and `Finalize` routines) only for objects of types derived from the standard library type `Controlled`.

### 9.3.2 References and Values

Several object-oriented languages, including Simula, Smalltalk, Python, Ruby, and Java, use a programming model in which variables refer to objects. Other languages, including C++, Modula-3, Ada 95, and Oberon, allow a variable to have a value that *is* an object. Eiffel uses a reference model by default, but allows the programmer to specify that certain classes should be expanded, in which case variables of those classes will use a value model. In a similar vein, C# uses `struct` to define types whose variables are values, and `class` to define types whose variables are references.

With a reference model for variables every object is created explicitly, and it is easy to ensure that an appropriate constructor is called. With a value model for variables object creation can happen implicitly as a result of elaboration. In Modula-3, Ada 95, and Oberon, which don't really have constructors, elaborated objects begin life uninitialized and it is possible to accidentally attempt to use a variable before it has a value. In C++, the compiler ensures that an appropriate constructor is called for every elaborated object, but the rules it uses to identify constructors and their arguments can sometimes be confusing.

**EXAMPLE 9.24**

Declarations and  
constructors in C++

If a C++ variable of class type `foo` is declared with no initial value, then the compiler will call `foo`'s zero-argument constructor (if no such constructor exists, but other constructors do, then the declaration is a static semantic error—a call to a nonexistent subroutine):

```
foo b; // calls foo::foo()
```

If the programmer wants to call a different constructor, the declaration must specify constructor arguments to drive overload resolution:

```
foo b(10, 'x'); // calls foo::foo(int, char)
```

**EXAMPLE 9.25**

Copy constructors

The most common argument list consists of a single object, of the same or different class:

```
foo a;
bar b;
...
foo c(a); // calls foo::foo(foo&)
foo d(b); // calls foo::foo(bar&)
```

Usually the programmer's intent is to declare a new object whose initial value is "the same" as that of the existing object. In this case it is more natural to write

**DESIGN & IMPLEMENTATION****The value/reference tradeoff**

The reference model of variables is arguably more elegant than the value model, particularly for object-oriented languages, but it generally requires that objects be allocated from the heap, and imposes (in the absence of compiler optimizations) an extra level of indirection on every access. The value model tends to be more efficient but makes it difficult to control initialization. In languages like Java, an optimization known as *escape analysis* can sometimes allow the compiler to determine that references to a given object will always be contained within (will never escape) a given method. In this case the object can be allocated in the method's stack frame, avoiding the overhead of heap allocation and, more significantly, eventual garbage collection.

```

foo a; // calls foo::foo()
bar b; // calls bar::bar()
...
foo c = a; // calls foo::foo(foo&)
foo d = b; // calls foo::foo(bar&)

```

In recognition of this intent, a single-argument constructor in C++ is called a *copy constructor*. It is important to realize here that the equals sign (=) in these declarations indicates initialization, not assignment. The effect is *not* the same as that of the similar code fragment

```

foo a, c, d; // calls foo::foo() three times
bar b; // calls bar::bar()
...

```

## DESIGN & IMPLEMENTATION

### Initialization and assignment

The distinction between initialization and assignment in C++ can sometimes have a surprising effect on performance. Consider, for example, the seemingly innocuous declaration

```
foo a = b + c;
```

If `foo` is a nontrivial class, the compiler will need to create a hidden, temporary object to be the target of the `+` operation, roughly equivalent to the following.

```

foo t;
t = b.operator+(c);
foo a = t;

```

The generated code will then include calls to both the zero-argument constructor and the destructor for `t`, as well as a copy constructor to move `t` into `a`. The less elegant

```
foo a = b; a += c;
```

will call the copy constructor for `a`, followed by `operator+=`, avoiding the need for `t`. Programmers who create explicit temporary objects to break up complex expressions may see similar unexpected costs.

A similar issue arises in subroutine calls. A parameter that is passed by value typically induces an implicit call to a copy constructor. A parameter that is passed by reference does not. Of course the reference parameter imposes the cost of indirection on accesses within the subroutine. It also creates an alias, which may inhibit certain code improvements, as noted in Section 3.6.1. Which parameter mode will result in the fastest code will depend on details of the individual program. Unfortunately, C++ semantics are sufficiently complex that it is difficult for the typical programmer to evaluate this tradeoff in practice.

```
c = a; // calls foo::operator=(foo&)
d = b; // calls foo::operator=(bar&)
```

Here `c` and `d` are initialized with the zero-argument constructor, and the later use of the equals sign indicates *assignment*, not initialization. The distinction is a common source of confusion in C++ programs. It arises from the combination of a value model of variables and an insistence that every elaborated object be initialized by a constructor. In CLOS, which requires objects to be passed to methods as explicit first parameters, object creation and initialization relies on overloaded versions of subroutines named `make-instance` and `initialize-instance`. Because CLOS employs a reference model uniformly, the issue of initializing elaborated objects does not arise. ■

#### EXAMPLE 9.26

Eiffel constructors and expanded objects

In Eiffel, every variable is initialized to a default value. For built-in types (integer, floating-point, character, etc.), which are considered to be expanded, the default values are all zero. For references to objects, the default value is `void` (null). For variables of expanded class types, the defaults are applied recursively to members. As noted on page 490 new objects are created by invoking Eiffel's `!!` creation operator:

```
!!var.creator(args)
```

where `var` is a variable of some class type  $T$  and `creator` is a constructor for  $T$ . In the common case, `var` will be a reference, and the creation operator will allocate space for an object of class  $T$  and then call the object's constructor. This same syntax is permitted, however, when  $T$  is an expanded class type, in which case `var` will actually be an object, rather than a reference. In this case, the `!!` operator simply passes to the constructor (a reference to) the already-allocated object. ■

## DESIGN & IMPLEMENTATION

### Initialization of “expanded” objects

C++ inherits from C a design philosophy that emphasizes execution speed, minimal run-time support, and suitability for “systems” programming, in which the programmer needs to be able to write code whose mapping to assembly language is straightforward and self-evident. The use of a value model for variables in C++ is thus more than an attempt to be backward compatible with C; it reflects the desire to allocate variables statically or on the stack whenever possible, to avoid the overhead of dynamic allocation, deallocation, and frequent indirection. In later sections we shall see several other ramifications of the C++ philosophy, including manual storage reclamation (Section 9.3.4) and static method binding (Section 9.4.1).

### 9.3.3 Execution Order

As we have seen, C++ insists that every object be initialized before it can be used. Moreover, if the object's class (call it *B*) is derived from some other class (call it *A*), C++ insists on calling an *A* constructor before calling a *B* constructor, so the derived class is guaranteed never to see its inherited fields in an inconsistent state. When the programmer creates an object of class *B* (either via declaration or with a call to `new`), the creation operation specifies arguments for a *B* constructor. These arguments allow the C++ compiler to resolve overloading when multiple constructors exist. But where does the compiler obtain arguments for the *A* constructor? Adding them to the creation syntax (as Simula does) would be a clear violation of abstraction. The answer adopted in C++ is to allow the header of the constructor of a derived class to specify base class constructor arguments:

```
foo::foo(foo_params) : bar(bar_args) {
 ...
```

Here `foo` is derived from `bar`. The list `foo_params` consists of formal parameters for this particular `foo` constructor. Between the parameter list and the opening brace of the subroutine definition is a “call” to a constructor for the base class `bar`. The arguments to the `bar` constructor can be arbitrarily complicated expressions involving the `foo` parameters. The compiler will arrange to execute the `bar` constructor before beginning execution of the `foo` constructor. ■

Similar syntax allows the C++ programmer to specify constructor arguments or initial values for members of the class. In Figure 9.1, for example, we could have used this syntax to initialize `prev`, `next`, `head_node`, and `val` in the constructor for `list_node`:

```
list_node() : prev(this), next(this), head_node(this), val(0) {
 // empty body -- nothing else to do
}
```

Given that all of these members have simple (pointer or integer) types, there will be no significant difference in the generated code. But suppose we have members that are themselves objects of some nontrivial class:

```
class foo : bar {
 mem1_t member1; // mem1_t and
 mem2_t member2; // mem2_t are classes
 ...
}

foo::foo(foo_params) : bar(bar_args), member1(mem1_args),
 member2(mem2_args) {
 ...
}
```

Here the use of embedded calls in the header of the `foo` constructor causes the compiler to call the copy constructors for the member objects, rather than calling the default (zero-argument) constructors, followed by `operator=`. Both semantics and performance may be different as a result. ■

#### EXAMPLE 9.27

Specification of base class constructor arguments

#### EXAMPLE 9.28

Specification of member constructor arguments

**EXAMPLE 9.29**

Invocation of base class constructor in Java

Like C++, Java insists that a constructor for a base class be called before the constructor for a derived class. The syntax is a bit simpler, however: the initial line of the code for the derived class constructor may consist of a “call” to the base class constructor:

```
super(args);
```

(C# has a similar mechanism.) As noted in Section 9.1, `super` is a Java keyword that refers to the base class of the class in whose code it appears. If the call to `super` is missing, the Java compiler automatically inserts a call to the base class’s zero-argument constructor (in which case such a constructor must exist). ■

Because Java uses a reference model uniformly for all objects, any class members that are themselves objects will actually be *references*, rather than “expanded” objects (to use the Eiffel term). Java simply initializes such members to `null`. If the programmer wants something different, he or she must call `new` explicitly within the constructor of the surrounding class. Smalltalk and (in the common case) C# and Eiffel adopt a similar approach. In C#, members whose types are `struct`s are initialized by setting all of their fields to zero or `null`. In Eiffel, if a class contains members of an `expanded` class type, that type is required to have a single constructor, with no arguments; the Eiffel compiler arranges to call this constructor when the surrounding object is created.

Smalltalk, Eiffel, and CLOS are all more lax than C++ regarding the initialization of base classes. The compiler or interpreter arranges to call the constructor (creator, initializer) for each newly created object automatically, but it does *not* arrange to call constructors for base classes automatically; all it does is initialize base class data members to default (0 or `null`) values. If the derived class wants different behavior, its constructor(s) must call a constructor for the base class explicitly. Objective-C has no special notion of constructor: programmers must write and explicitly invoke their own initialization methods.

### 9.3.4 Garbage Collection

When a C++ object is destroyed, the destructor for the derived class is called first, followed by those of the base class(es), in reverse order of derivation. By far the most common use of destructors in C++ is manual storage reclamation. Suppose, for example, that we were to create a list or queue of character-string names:

```
class name_list_node : public gp_list_node {
 char *name; // pointer to the data in a node
public:
 name_list_node() {
 name = 0; // empty string
 }
 name_list_node(char *n) {
 name = new char[strlen(n)];
 strcpy(name, n); // copy argument into member
 }
}
```

**EXAMPLE 9.30**

Reclaiming space with destructors

```

~name_list_node() {
 if (name != 0) {
 delete[] name; // reclaim space
 }
}
;

```

The destructor in this class serves to reclaim space that was allocated in the heap by the constructor.

In languages with automatic garbage collection, there is much less need for destructors. In fact, the entire idea of destruction is suspect in a garbage-collected language, because the programmer has little or no control over when an object is going to be destroyed. Java and C# allow the programmer to declare a `finalize` method that will be called immediately before the garbage collector reclaims the space for an object, but the feature is not widely used.

### CHECK YOUR UNDERSTANDING

21. Does a constructor allocate space for an object? Explain.
22. What is a *metaclass* in Smalltalk?
23. Why is object initialization simpler in a language with a reference model of variables (as opposed to a value model)?
24. How does a C++ (or Java or C#) compiler tell which constructor to use for a given object? How does the answer differ for Eiffel and Smalltalk?
25. What is *escape analysis*?
26. Summarize the rules in C++ that determine the order in which constructors are called for a class, its base class(es), and the classes of its fields. How are these rules simplified in other languages?
27. Explain the difference between initialization and assignment in C++.
28. Why does C++ need destructors more than Eiffel does?

## 9.4 Dynamic Method Binding

One of the principal consequences of inheritance/type extension is that a derived class *D* has all the members—data and subroutines—of its base class *C*. As long as *D* does not hide any of the publicly visible members of *C* (see Exercise 9.13), it makes sense to allow an object of class *D* to be used in any context that expects an object of class *C*: anything we might want to do to an object of class *C* we can also do to an object of class *D*. In Ada terminology, a derived class that does

not hide any publicly visible members of its base class is a *subtype* of that base class.

**EXAMPLE 9.31**

Derived class objects in a base class context

```
class person { ... }
class student : public person { ... }
class professor : public person { ... }
```

Because both `student` and `professor` objects have all the properties of a `person` object, we should be able to use them in a `person` context:

```
student s;
professor p;
...
person *x = &s;
person *y = &p;
```

Moreover a subroutine like

```
void person::print_mailing_label() { ... }
```

would be polymorphic—capable of accepting arguments of multiple types:

```
s.print_mailing_label(); // i.e., print_mailing_label(s)
p.print_mailing_label(); // i.e., print_mailing_label(p)
```

As with other forms of polymorphism, we depend on the fact that `print_mailing_label` uses only those features of its formal parameter that all actual parameters will have in common. ■

**EXAMPLE 9.32**

Static and dynamic method binding

But now suppose that we have redefined `print_mailing_label` in each of the two derived classes. We might, for example, want to encode certain information (student's year in school, professor's home department) in the corner of the label. Now we have multiple versions of our subroutine—`student::print_mailing_label` and `professor::print_mailing_label`, rather than the single, polymorphic `person::print_mailing_label`. Which version we will get depends on the object:

```
s.print_mailing_label(); // student::print_mailing_label(s)
p.print_mailing_label(); // professor::print_mailing_label(p)
```

But what about

```
x->print_mailing_label(); // ??
y->print_mailing_label(); // ??
```

Does the choice of the method to be called depend on the types of the *variables* `x` and `y`, or on the classes of the *objects* `s` and `p` to which those variables refer? ■

The first option (use the type of the reference) is known as *static method binding*. The second option (use the class of the object) is known as *dynamic method binding*. Dynamic method binding is central to object-oriented programming. Imagine, for example, that our administrative computing program has created a list of persons who have overdue library books. The list may contain both students and professors. If we traverse the list and print a mailing label for each person, dynamic method binding will ensure that the correct printing routine is called for each individual. In this situation the definitions in the derived classes are said to *override* the definition in the base class.

### **Semantics and Performance**

#### **EXAMPLE 9.33**

The need for dynamic binding

```
class text_file {
 char *name;
 long position; // file pointer
public:
 void seek(long whence);
 ...
}
```

Now suppose we have a derived class `read_ahead_text_file`:

```
class read_ahead_text_file : public text_file {
 char *upcoming_characters;
public:
 void seek(long whence); // redefinition
 ...
}
```

The code for `read_ahead_text_file::seek` will undoubtedly need to change the value of the cached `upcoming_characters`. If the method is not dynamically dispatched, however, we cannot guarantee that this will happen: if we pass a `read_ahead_text_file` reference to a subroutine that expects a `text_file` reference as argument, and if that subroutine then calls `seek`, we'll get the version of `seek` in the base class. ■

Unfortunately, as we shall see in Section 9.4.3, dynamic method binding imposes run-time overhead. While this overhead is generally modest, it is nonetheless a concern for small subroutines in performance-critical applications. Smalltalk, Objective-C, Modula-3, Python, and Ruby use dynamic method binding for all methods. Java and Eiffel use dynamic method binding by default, but allow individual methods and (in Java) classes to be labeled `final` (Java) or `frozen` (Eiffel), in which case they cannot be overridden by derived classes,

and can therefore employ an optimized implementation. Simula, C++, C#, and Ada 95 use static method binding by default but allow the programmer to specify dynamic binding when desired. In these latter languages it is common terminology to distinguish between *overriding* a method that uses dynamic binding and (merely) *redefining* a method that uses static binding. For the sake of clarity, C# requires explicit use of the keywords `override` and `new` whenever a method in a derived class overrides or redefines (respectively) a method of the same name in a base class.

#### 9.4.1 Virtual and Nonvirtual Methods

##### EXAMPLE 9.34

Virtual methods in C++ and C#

```
class person {
public:
 virtual void print_mailing_label();
 ...
}
```

##### EXAMPLE 9.35

Virtual methods in Simula

```
CLASS Person;
 VIRTUAL: PROCEDURE PrintMailingLabel;
BEGIN
 ...
 PROCEDURE PrintMailingLabel...
 COMMENT body of subroutine
 ...
END Person;
```

##### EXAMPLE 9.36

Class-wide types in Ada 95

Ada 95 adopts a different approach. Rather than associate dynamic dispatch with particular methods, the Ada 95 programmer associates it with certain *references*. In our mailing label example, a formal parameter or an access variable (pointer) can be declared to be of the *class-wide* type `person'Class`, in which case all calls to all methods of that parameter or variable will be dispatched based on the class of the object to which it refers:

---

**4** C++ also uses the `virtual` keyword in certain circumstances to prefix the name of a base class in the header of the declaration of a derived class. This usage supports the very different purpose of *shared multiple inheritance*, which we will consider in Section 9.5.3.

```

type person is tagged record ...
type student is new person with ...
type professor is new person with ...

procedure print_mailing_label(r : person) is ...
procedure print_mailing_label(s : student) is ...
procedure print_mailing_label(p : professor) is ...

procedure print_appropriate_label(r : person'Class) is
begin
 print_mailing_label(r);
 -- calls appropriate overloaded version, depending
 -- on type of r at run time
end print_appropriate_label;

```

#### 9.4.2 Abstract Classes

##### **EXAMPLE 9.37**

Abstract methods in Java and C#

In most object-oriented languages it is possible to omit the body of a virtual method in a base class. In Java and C#, one does so by labeling both the class and the missing method as abstract:

```

abstract class person {
 ...
 public abstract void print_mailing_label();
 ...
}

```

##### **EXAMPLE 9.38**

Abstract methods in C++

The notation in C++ is somewhat less intuitive: one follows the subroutine declaration with an “assignment” to zero:

```

class person {
 ...
public:
 virtual void print_mailing_label() = 0;
 ...
}

```

C++ refers to abstract methods as *pure virtual* methods. In Simula all virtual methods are abstract.

Regardless of declaration syntax, a *class* is said to be abstract if it has at least one abstract method. It is not possible to declare an object of an abstract class, because it would be missing at least one member. The only purpose of an abstract class is to serve as a base for other, *concrete* classes. A concrete class (or one of its intermediate ancestors) must provide a real definition for every abstract method it inherits. The existence of an abstract method in a base class provides a “hook” for dynamic method binding; it allows the programmer to write code that calls methods of (references to) objects of the base class, under the assumption that appropriate concrete methods will be invoked at run time. Classes that have no members other than abstract methods—no fields or method bodies—are called

*interfaces* in Java and C#. They support a restricted, “mix-in” form of multiple inheritance, which we will consider in Section 9.5.4.<sup>5</sup>

### 9.4.3 Member Lookup

#### EXAMPLE 9.39

Vtables

With static method binding (as in Simula, C++, C#, or Ada 95), the compiler can always tell which version of a method to call, based on the type of the variable being used. With dynamic method binding, however, the object referred to by a reference or pointer variable must contain sufficient information to allow the code generated by the compiler to find the right version of the method at run time. The most common implementation represents each object with a record whose first field contains the address of a *virtual method table* (*vtable*) for the object’s class (see Figure 9.4). The vtable is an array whose *i*th entry indicates the address of the code for the object’s *i*th virtual method. All objects of a given class share the same vtable.

#### EXAMPLE 9.40

Implementation of a virtual method call

Suppose that the `this` (`self`) pointer for methods is passed in register `r1`, that `m` is the third method of class `foo`, and that `f` is a pointer to an object of class `foo`. Then the code to call `f->m()` looks something like this:

```
r1 := f
r2 := *r1 -- vtable address
r2 := *(r2 + (3-1) × 4) -- assuming 4 = sizeof (address)
call *r2
```

#### EXAMPLE 9.41

Implementation of single inheritance

On a typical RISC machine this calling sequence is two instructions (both of which access memory) longer than a call to a statically identified method. The extra overhead can be avoided whenever the compiler can deduce the type of the relevant object at compile time. The deduction is trivial for calls to methods of object-valued variables (as opposed to references and pointers).

If `bar` is derived from `foo`, we place its additional fields at the end of the “record” that represents it. We create a vtable for `bar` by copying the vtable for `foo`, replacing the entries of any virtual methods overridden by `bar`, and appending entries for any virtual methods declared in `bar` (see Figure 9.5). If we have an object of class `bar` we can safely assign its address into a variable of type `foo*`:

```
class foo { ...
class bar : public foo { ...
...
```

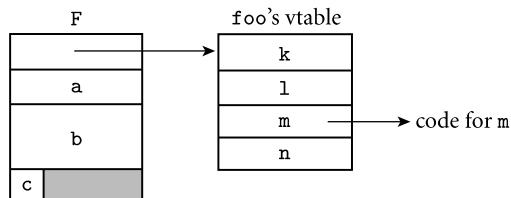
---

**5** An abstract virtual method in Eiffel is called a *deferred feature*. (Recall that all features are virtual.) An abstract class is called a *deferred class*. A concrete class is called an *effective class*. An interface in the Java or C# sense of the word is called a *fully deferred class*.

```

class foo {
 int a;
 double b;
 char c;
public:
 virtual void k(...);
 virtual int l(...);
 virtual void m();
 virtual double n(...);
 ...
} F;

```

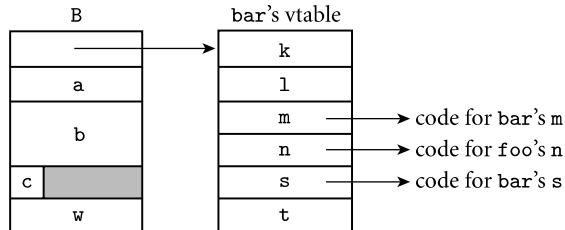


**Figure 9.4 Implementation of virtual methods.** The representation of object **F** begins with the address of the vtable for class **foo**. (All objects of this class will point to the same vtable.) The vtable itself consists of an array of addresses, one for the code of each virtual method of the class. The remainder of **F** consists of the representations of its fields.

```

class bar : public foo {
 int w;
public:
 void m(); //override
 virtual double s(...);
 virtual char *t(...);
 ...
} B;

```



**Figure 9.5 Implementation of single inheritance.** As in Figure 9.4, the representation of object **B** begins with the address of its class's vtable. The first four entries in the table represent the same members as they do for **foo**, except that one—**m**—has been overridden and now contains the address of the code for a different subroutine. Additional fields of **bar** follow the ones inherited from **foo** in the representation of **B**; additional virtual methods follow the ones inherited from **foo** in the vtable of class **bar**.

```

foo F;
bar B;
foo* q;
bar* s;
...
q = &B; // ok; references through q will use prefixes
 // of B's data space and vtable
s = &F; // static semantic error; F lacks the additional
 // data and vtable entries of a bar

```

In C++ (as in all the object-oriented languages we have considered, save Smalltalk, Objective-C, and CLOS), the compiler can verify the type correctness of this code statically. It does not know what the class of the object referred to by **q** will be at run time, but it knows that it will either be **foo** or something derived (directly or indirectly) from **foo**, and this ensures that it will have all the members that may be accessed by **foo**-specific code. ■

#### EXAMPLE 9.42

Casts in C++

C++ allows “backward” assignments by means of a `dynamic_cast` operator:

```
s = dynamic_cast<bar*>(q); // performs a run-time check
```

For backward compatibility C++ also supports traditional C-style casts of object pointers and references.

```
s = (bar*) q; // permitted, but risky
```

With a C-style cast it is up to the programmer to ensure that the actual object involved is of an appropriate type: no dynamic semantic check is performed. ■

#### EXAMPLE 9.43

Reverse assignment in Eiffel and C#

Java and C# employ the traditional cast notation but perform the dynamic check. Eiffel has a *reverse assignment* operator, ?=, which (like the C++ `dynamic_cast`) assigns an object reference into a variable if and only if the type at run time is acceptable:

#### DESIGN & IMPLEMENTATION

##### Reverse assignment

Implementations of Eiffel, Java, C#, and C++ typically support dynamic checks on reverse assignment by including in each vtable the address of a run-time *type descriptor*. In C++, `dynamic_cast` is permitted only on pointers and references of polymorphic types (classes with virtual methods), since objects of nonpolymorphic types do not have vtables. A separate `static_cast` operation can be used on nonpolymorphic types, but it performs no run-time check and is thus inherently unsafe when applied to a pointer of a derived class type.

#### DESIGN & IMPLEMENTATION

##### The fragile base class problem

Under certain circumstances, it can be desirable to perform method lookup at run time even when the language permits compile-time lookup. In Java, for example, programs are usually distributed in a portable “byte code” format that is either interpreted or, in some implementations, compiled immediately before execution. The standard “virtual machine” interpreter for byte code looks methods up at run time. By doing so it avoids what is known as the *fragile base class* problem. Java implementations depend on the presence of a large standard library. This library is expected to evolve over time. Though the designers of the library will presumably be careful to maximize backward compatibility—seldom if ever deleting any members of a class—it is likely that users of old versions of the library will on occasion attempt to run code that was written with a new version of the library in mind. In such a situation it would be disastrous to rely on static assumptions about the representation of library classes: code that tries to use a newly added library feature could end up accessing memory beyond the end of the available representation. Run-time method lookup, by contrast, will produce a helpful “member not found in your version of the class” dynamic error message.

```

class foo ...
class bar inherit foo ...
...
f : foo
b : bar
...
f := b -- always ok
b ?= f -- reverse assignment: b gets f if f refers to a bar object
 -- at run time; otherwise b gets void

```

C# provides an `as` operator that performs a similar function. ■

In Smalltalk, variables are *untyped* references. A reference to any object may be assigned into any variable. Only when code actually attempts to invoke an operation (send a “message”) at run time does the language implementation check to see whether the operation is supported by the object. The implementation is straightforward: fields of an object are never public; methods provide the only means of object interaction. The representation of an object begins with the address of a type descriptor. The type descriptor contains a dictionary that maps method names to code fragments. At run time, the Smalltalk interpreter performs a lookup operation in the dictionary to see if the method is supported. If not, it generates a “message not understood” error—the equivalent of a type clash error in Lisp. CLOS and Objective-C provide similar semantics and invite similar implementations. The Smalltalk/CLOS/Objective-C approach is arguably more flexible than that of more statically typed languages, but it incurs significant run-time cost, and delays the reporting of errors.

In addition to imposing the overhead of indirection, virtual methods often preclude the in-line expansion of subroutines at compile time. The lack of in-line subroutines can be a serious performance problem when subroutines are small and frequently called. Like C, C++ attempts to avoid run-time overhead whenever possible: hence its use of static method binding as the default, and its heavy reliance on object-valued variables, for which even virtual methods can be dispatched at compile time.

#### 9.4.4 Polymorphism

We have already noted that dynamic method binding introduces polymorphism (specifically, *subtype* polymorphism) into any code that expects a reference to an object of some base class `foo`. As long as objects of the derived class support the operations of the base class, the code will work equally well with references to objects of any class derived from `foo`. By declaring a reference parameter to be of class `foo`, for example, the programmer asserts that the subroutine uses only the “`foo` features” of the parameter, and will work on any object that provides those features.

One might be tempted to think that the combination of inheritance and dynamic method binding would eliminate the need for generics, but this is not the

**EXAMPLE 9.44**

Inheritance and method signatures

case. We can see an example in the `gp_list_node` class and its descendants of Section 9.1. By placing the structural aspects of an abstraction (in this case a list) in a base class, we make it easy to create type-specific lists: `int_list_node`, `float_list_node`, `student_list_node`, and so on. Unfortunately, base class methods like `predecessor` and `successor` return references of the base class type, which do not then support type-specific operations. To allow us to access the values stored in objects returned by the list-manipulation routines, we must perform an explicit type cast:

```
int_list_node_ptr q, r;
...
r = q->successor(); // error: type clash
gp_list_node_ptr p = q->successor();
cout << p.val; // error: gp_list_nodes have no val
r = (int_list_node_ptr) q->successor();
cout << r.val; // ok
```

The cast on the sixth line here is both awkward and unsafe. We can't use a `dynamic_cast` operation because `gp_list_node` has no virtual members, and hence (in C++) no vtable. We can confine the awkwardness to the definition of `int_list_node` by redefining methods:

```
int_list_node* int_list_node::predecessor() { // redefine
 return (int_list_node*) gp_list_node::predecessor();
}
int_list_node* int_list_node::successor() { // redefine
 return (int_list_node*) gp_list_node::successor();
}
```

**EXAMPLE 9.45**

Generics and inheritance

Unfortunately, redefining all of the appropriate arguments and return types of base class methods in every derived class is still a frustratingly tedious exercise, and the code is still unsafe: the compiler cannot verify type correctness. Generics get around both problems. In C++, we can write

```
template<class V>
class list_node {
 list_node<V>* prev;
 list_node<V>* next;
 list_node<V>* head_node;
public:
 V val;
 list_node<V>* predecessor() { ... }
 list_node<V>* successor() { ... }
 void insert_before(list_node<V>* new_node) { ...
 ...
 };
}
```

```

template<class V>
class list {
 list_node<V> header;
public:
 list_node<V>* head() { ... }
 void append(list_node<V> *new_node) { ... }
 ...
};

typedef list_node<int> int_list_node;
typedef list<int> int_list;
...
int_list numbers;
int_list_node* first_int;
...
first_int = numbers->head();
```

In a nutshell, generics exist for the purpose of abstracting over unrelated types, something that inheritance does not support. (NB: the type inference system of ML and related languages *does* suffice to abstract over unrelated types; ML does not require generics. On the other hand, while ML provides Euclid-like module types, it does not provide inheritance, and thus cannot be considered an object-oriented language.)

Eiffel, Java, and C# all provide generics as well. Java's version is somewhat simpler than the others: because object variables are always references, they always have the same size, and a single copy of the code can generally be shared by every instance of a generic. As a convenient shorthand, Eiffel allows the programmer to declare parameters and return values of methods to be of the same type as some “anchor” field of the class. Then if a derived class redefines the anchor, the parameters and return values are automatically redefined as well, without the need to specify them explicitly:

#### EXAMPLE 9.46

Like in Eiffel

#### DESIGN & IMPLEMENTATION

##### Generics and dynamic method dispatch

As noted in Section 3.6.3, generics (explicit parametric polymorphism) are usually implemented by creating multiple copies of the polymorphic code, one specialized for each needed concrete type. (Java is an exception: it uses a single copy. Other languages may share specializations when possible.) Subtype polymorphism is almost always implemented by creating a single copy of the code, and relying on vtables for dynamic method dispatch. So in object-oriented languages the two main forms of polymorphism—parametric and subtype—not only serve different purposes, they typically have very different implementations.

```

class gp_list_node ...
...
class gp_list
feature {NONE} -- private
 header : gp_list_node -- to be redefined by derived classes
feature {ALL} -- public
 head : like header is ... -- methods
 append(new_node : like header) is ...
...
end
...
class student_list_node inherit gp_list_node ...
...
class student_list
 inherit gp_list
 redefine header end
feature {NONE}
 header : student_list_node
 -- don't need to redefine head and append
end

```

The like mechanism does not eliminate the need for generics, but it makes it easier to define them, or to do without them in simple situations. ■

#### 9.4.5 Closures

---

##### EXAMPLE 9.47

Objects as closures

Because the dispatch of virtual methods is delayed until run time, dynamic method binding provides a mechanism similar to first-class subroutines (Sections 3.5 and 8.3.1). Code that looks like this in C++:

```

typedef void (*F_INT)(int);
// F_INT is a type: pointer to function from int to void
void p(int a) {
 ...
}
void q(F_INT f) {
 ...
 f(3);
 ...
}
q(&p);

```

can more or less be replaced by code that looks like this:

```

class foo {
public:
 virtual void f(int a) = 0;
};

```

```
void q(foo& obj) {
 ...
 obj.f(3);
 ...
}
class bar : public foo {
public:
 virtual void f(int a) {
 ...
 }
} my_obj;
q(my_obj);
```

This latter, object-oriented version of the code has an important advantage over the pointer-to-subroutine version in C++: by adding fields to class `bar` (and object `my_obj`), we can provide method `f` with data on which to operate, data that `q` knows nothing about. In effect, the fields of an object with a virtual method behave like the referencing environment of a closure in a language with nested scopes (recall that C and C++ do not require closures, because they do not have nested scopes). In Ada 95, which has both nested scopes and classes (tagged types), the entries in vtables must themselves be closures, not just subroutine addresses.

In some cases, virtual methods and first-class subroutines can complement each other. Suppose, for example, that we are writing a discrete event simulation, as described in Section 8.6.4. We might like a general mechanism that allows us to schedule a call to an arbitrary subroutine, with an arbitrary set of parameters, to occur at some future point in time. If the subroutines we want to have called vary in their numbers and types of parameters, we won't be able to pass them to a general purpose `schedule_at` routine. We can solve the problem with virtual methods, as shown in Figure 9.6. As we shall see in Section 12.2.3, this same technique is used in Modula-3 to encapsulate start-up arguments for newly created threads of control.

### ✓ CHECK YOUR UNDERSTANDING

29. Explain the difference between dynamic and static method binding (i.e., between *virtual* and *nonvirtual* methods).
30. Summarize the fundamental argument for dynamic method binding. Why do C++ and C# use static method binding by default?
31. Explain the distinction between *redefining* and *overriding* a method.
32. What is a *class-wide* type in Ada 95?
33. Explain the connection between dynamic method binding and polymorphism.

```

class fn_call {
public:
 virtual void trigger() = 0;
};

void schedule_at(fn_call& fc, time t) {
 ...
}

void foo(int a, double b, char c) {
 ...
}

class call_foo : public fn_call {
 int arg1;
 double arg2;
 char arg3;
 void (*ptr)(int, double, char);
public:
 call_foo(int a, double b, char c) : // constructor
 arg1(a), arg2(b), arg3(c) {
 // member initialization is all that is required
 }
 void trigger() {
 foo(arg1, arg2, arg3);
 }
};

call_foo cf(3, 3.14, 'x'); // declaration/constructor call
schedule_at(cf, now() + delay); // at some point in the future, the discrete event system
// will call cf.trigger(), which will cause a call to
// foo(3, 3.14, 'x')

```

**Figure 9.6 Subroutine pointers and virtual methods.** Class `call_foo` encapsulates a subroutine pointer and values to be passed to the subroutine. It exports a parameter-less subroutine that can be used to trigger the encapsulated call.

34. What is an *abstract* method (also called a *pure virtual* method in C++ and a *deferred feature* in Eiffel)?
35. What is *reverse assignment*? Why does it require a run-time check?
36. What is a *vtable*? How is it used?
37. What is the *fragile base class* problem?
38. Describe how virtual functions can be used to achieve the effect of subroutine closures.
39. What is an *abstract (deferred) class*?

40. Explain why generics may be useful in an object-oriented language, despite the extensive polymorphism already provided by inheritance.
  41. Explain the use of `like` in Eiffel.
- 

## 9.5 Multiple Inheritance

### EXAMPLE 9.49

Deriving from two base classes

At times it can be useful for a derived class to inherit features from more than one base class. Suppose, for example, that we want our administrative computing system to keep all students of the same year (freshmen, sophomores, juniors, seniors, unmatriculated) on some list. It may then be desirable to derive class `student` from both `person` and `gp_list_node`. In C++ we can say

```
class student : public person, public gp_list_node { ... }
```

Now an object of class `student` will have all the fields and methods of both a `person` and a `gp_list_node`. The declaration in Eiffel is analogous:

```
class student
inherit
 person
 gp_list_node
feature
 ...

```

Multiple inheritance also appears in CLOS and Python. Simula, Smalltalk, Objective-C, Modula-3, Ada 95, and Oberon have only single inheritance. Java, C#, and Ruby provide a limited, “mix-in” form of multiple inheritance, in which only one parent class is permitted to have fields.

### IN MORE DEPTH

Multiple inheritance introduces a wealth of semantic and pragmatic issues.

- Suppose two parent classes provide a method with the same name. Which one do we use in the child? Can we access both?
- Suppose two parent classes are both derived from some common “grandparent” class. Does the “grandchild” have one copy or two of the grandparent’s fields?
- Our implementation of single inheritance relies on the fact that the representation of an object of the parent class is a prefix of the representation of an object of a derived class. With multiple inheritance, how can *each* parent be a prefix of the child?

Multiple inheritance with a common “grandparent” is known as *repeated* inheritance. Repeated inheritance with separate copies of the grandparent is known

as *replicated* inheritance; repeated inheritance with a single copy of the grand-parent is known as *shared* inheritance. Shared inheritance is the default in Eiffel. Replicated inheritance is the default in C++. Both languages allow the programmer to obtain the other option when desired.

Much of the complexity disappears if we insist, as in Java or C#, that all but one of the parent classes consist of methods only. Both languages call such a class an **interface**.

---

## 9.6 Object-Oriented Programming Revisited

At the beginning of this chapter, we characterized object-oriented programming in terms of three fundamental concepts: encapsulation, inheritance, and dynamic method binding. Encapsulation allows the implementation details of an abstraction to be hidden behind a simple interface. Inheritance allows a new abstraction to be defined as an extension or refinement of some existing abstraction, obtaining some or all of its characteristics automatically. Dynamic method binding allows the new abstraction to display its new behavior even when used in a context that expects the old abstraction.

Different programming languages support these fundamental concepts to different degrees. In particular, languages differ in the extent to which they *require* the programmer to write in an object-oriented style. Some authors argue that a *truly* object-oriented language should make it difficult or impossible to write programs that are not object-oriented. From this purist point of view, an object-oriented language should present a *uniform object model* of computing, in which every data type is a class, every variable is a reference to an object, and every subroutine is an object method. Moreover, objects should be thought of in *anthropomorphic terms*: as active entities responsible for all computation.

Smalltalk and Ruby come close to this ideal. In fact, as described in the subsection below (mostly on the PLP CD), even such control flow mechanisms as selection and iteration are modeled as method invocations in Smalltalk. On the other hand, Modula-3 and Ada 95 are probably best characterized as von Neumann languages that *permit* the programmer to write in an object-oriented style if desired.

So what about C++? It certainly has a wealth of features, including several (multiple inheritance, elaborate access control, strict initialization order, destructors, generics) that are useful in object-oriented programs and that are not found in Smalltalk. At the same time, it has a wealth of problematic wrinkles. Its simple types are not classes. It has subroutines outside of classes. It uses static method binding and replicated multiple inheritance by default, rather than the more costly virtual alternatives. Its unchecked C-style type casts provide a major loophole for type checking and access control. Its lack of garbage collection is a major obstacle to the creation of correct, self-contained abstractions. Probably

most serious of all, C++ retains all of the low level mechanisms of C, allowing the programmer to escape or subvert the object-oriented model of programming entirely. It has been suggested that the best C++ programmers are those who did *not* learn C first: they are not as tempted to write “C-style” programs in the newer language. On balance, it is probably safe to say that C++ is an object-oriented language in the same sense that Common Lisp is a functional language. With the possible exception of garbage collection, C++ provides all of the necessary tools, but it requires substantial discipline on the part of the programmer to use those tools “correctly.”

### 9.6.1 The Object Model of Smalltalk

Smalltalk is to a large extent the canonical object-oriented language. The original version of Smalltalk was designed by Alan Kay as part of his doctoral work at the University of Utah in the late 1960s. It was then adopted by the Software Concepts Group at the Xerox Palo Alto Research Center (PARC), and went through five major revisions in the 1970s, culminating in the Smalltalk-80 language.<sup>6</sup>

#### IN MORE DEPTH

We have mentioned several features of Smalltalk in previous sections. A somewhat longer treatment can be found on the PLP CD, where we focus in particular on Smalltalk’s anthropomorphic programming model. A full introduction to the language is beyond the scope of this book.

## 9.7 Summary and Concluding Remarks

This has been the last of our five core chapters on language design: names (Chapter 3), control flow (Chapter 6), types (Chapter 7), subroutines (Chapter 8), and objects (Chapter 9).

We began in Section 9.1 by identifying three fundamental concepts of object-oriented programming: *encapsulation*, *inheritance*, and *dynamic method binding*. We also introduced the terminology of classes, objects, and methods. We had already seen encapsulation in the modules of Chapter 3. Encapsulation allows the details of a complicated data abstraction to be hidden behind a comparatively simple interface. Inheritance extends the utility of encapsulation by making it

---

**6** Alan Kay (1940–) joined PARC in 1972. In addition to developing Smalltalk and its graphical user interface, he conceived and promoted the idea of the laptop computer, well before it was feasible to build one. He became a Fellow at Apple Computer in 1984, and has subsequently held positions at Walt Disney and Hewlett-Packard. He received the ACM Turing Award in 2003.

easy for programmers to define new abstractions as refinements or extensions of existing abstractions. Inheritance provides a natural basis for polymorphic subroutines: if a subroutine expects an instance of a given class as argument, then an object of any class derived from the expected one can be used instead (assuming that it retains the entire existing interface). Dynamic method binding extends this form of polymorphism by arranging for a call to one of the parameter's methods to use the implementation associated with the class of the actual object at run time, rather than the implementation associated with the declared class of the parameter. We noted that some languages, including Modula-3, Ada 95, and Fortran 2003, support object orientation through a *type extension* mechanism, in which encapsulation is associated with modules, but inheritance and dynamic method binding are associated with a special form of record.

In later sections we covered object initialization and finalization, dynamic method binding, and (on the PLP CD) multiple inheritance in some detail. In many cases we discovered tradeoffs between functionality on the one hand and simplicity and execution speed on the other. Treating variables as references, rather than values, often leads to simpler semantics, but requires extra indirection. Garbage collection, as previously noted in Section 7.7.3, dramatically eases the creation and maintenance of software but imposes run-time costs. Dynamic method binding requires (in the general case) that methods be dispatched using vtables or some other lookup mechanism. Simple implementations of multiple inheritance impose overheads even when unused.

In several cases we saw time/space tradeoffs as well. In-line subroutines, as previously noted in Section 8.2.5, can dramatically improve the performance of code with many small subroutines, not only by eliminating the overhead of the subroutine calls themselves, but by allowing register allocation, common subexpression analysis, and other “global” code improvements to be applied across calls. At the same time, in-line expansion generally increases the size of object code. Exercises ⑩ 9.25 and ⑩ 9.27 explore similar tradeoffs in the implementation of multiple inheritance.

Despite its lack of multiple inheritance, Smalltalk is widely regarded as the purest and most flexible of the object-oriented languages. Its lack of compile-time type checking, however, together with its “message-based” model of computation and its need for dynamic method lookup, render its implementations rather slow. C++, with its object-valued variables, default static binding, minimal dynamic checks, and high-quality compilers, is largely responsible for the growing popularity of object-oriented programming. Improvements in reliability, maintainability, and code reuse may or may not justify the high-performance overhead of Smalltalk, or even the lower overhead of Eiffel. They almost certainly justify the relatively modest overhead of C++. With the ever-increasing size of software systems, the explosive growth of distributed computing on the Internet, and the development of highly portable object-oriented languages (Java) and binary object standards (.NET [WHA03], CORBA [Sie96]/JavaBeans [Sun97]), object-oriented programming will clearly play a central role in 21st-century computing.

## 9.8 Exercises

- 9.1 Some language designers argue that object orientation eliminates the need for nested subroutines. Do you agree? Why or why not?
- 9.2 Design a class hierarchy to represent syntax trees for the CFG of Figure 4.5 (page 175). Provide a method in each class to return the value of a node. Provide constructors that play the role of the `make_leaf`, `make_un_op`, and `make_bin_op` subroutines.
- 9.3 Repeat the previous exercise, but using a variant record (union) type to represent syntax tree nodes. Repeat again using type extensions. Compare the three solutions in terms of clarity, abstraction, type safety, and extensibility.
- 9.4 Rewrite the list and queue classes of Section 9.1 in such a way that objects not derived from a container base class can still be inserted in a list or queue. You will probably want to include a pointer to data, rather than the data itself, in each node of a list/queue.
- 9.5 In the spirit of Example 9.7, write a *double-ended queue* (deque) abstraction (pronounced “deck”), derived from a doubly linked list base class. Borrowing terminology from Icon, name your methods `put` (add at tail), `get` (remove at head), `push` (add at head), and `pull` (remove at tail).
- 9.6 Use templates (generics) to abstract your solutions to the previous two questions over the type of data in the container.
- 9.7 Repeat Exercise 9.5 in Python or Ruby. Write a simple program to demonstrate that generics are not needed to abstract over types. What happens if you mix objects of different types in the same deque?
- 9.8 Write a package body for the list abstraction of Figure 9.3.
- 9.9 Rewrite the list and queue abstractions in Eiffel, Java, and/or C#.
- 9.10 Using C++, Java, or C#, implement a `Complex` class in the spirit of Example 9.22.
- 9.11 Repeat the previous two exercises for Python and/or Ruby.
- 9.12 Compare Java `final` methods with C++ nonvirtual methods. How are they the same? How are they different?
- 9.13 In several object-oriented languages, including C++ and Eiffel, a derived class can hide members of the base class. In C++, for example, we can declare a base class to be `public`, `protected`, or `private`:

```
class B : public A { ...
 // public members of A are public members of B
 // protected members of A are protected members of B
 ...
}
```

```

class C : protected A { ...
 // public and protected members of A are protected members of C
...
class D : private A { ...
 // public and protected members of A are private members of D

```

In all cases, private members of A are inaccessible to methods of B, C, or D.

Consider the impact of protected and private base classes on dynamic method binding. Under what circumstances can a reference to an object of class B, C, or D be assigned into a variable of type A\*?

- 9.14** What happens to the implementation of a class if we redefine a data member? For example, suppose we have

```

class foo {
public:
 int a;
 char *b;
};

class bar : public foo {
public:
 float c;
 int b;
};

```

Does the representation of a bar object contain one b field or two? If two, are both accessible, or only one? Under what circumstances?

- 9.15** Discuss the relative merits of classes and type extensions. Which do you prefer? Why?
- 9.16** Building on the outline of Example 9.25, write a program that illustrates the difference between copy constructors and operator= in C++. Your code should include examples of each situation in which one of these may be called (don't forget parameter passing and function returns). Instrument the copy constructors and assignment operators in each of your classes so that they will print their names when called. Run your program to verify that its behavior matches your expectations.
- 9.17** What do you think of the decision, in C++, C#, and Ada 95, to use static method binding, rather than dynamic, by default? Is the gain in implementation speed worth the loss in abstraction and reusability? Assuming that we sometimes want static binding, do you prefer the method-by-method approach of C++ and C#, or the variable-by-variable approach of Ada 95? Why?

- 9.18 If `foo` is an abstract class in a C++ program, why is it acceptable to declare variables of type `foo*`, but not of type `foo`?

© 9.19–9.29 In More Depth.

## 9.9 Explorations

- 9.30 Return for a moment to Exercise 3.6. Build a (more complete) C++ version of the singly linked list library of Figure 3.18. Discuss the issue of storage management. Under what circumstances should one delete the elements of a list when deleting the list itself? What should the destructor for `list_node` do? Should it delete its `data` member? Should it recursively delete node `next`?
- 9.31 Learn about the *indexer* mechanism in C#, and use it to create a hash table class that can be indexed like an array. (In effect, create a simple version of the `System.Collections.Hashtable` container class.) Alternatively, use an overloaded version of operator `[]` to build a similar class in C++.
- 9.32 Several languages, including C++, Java, and C#, allow class declarations to nest. Java additionally allows classes to be nested inside functions. Learn the visibility rules associated with nested classes. Which members of an outer class are visible to methods of an inner class? Which members of an inner class are visible to methods of the outer class? Explain the distinction between `static` and non-`static` nested classes in Java. Which of these resembles the nested classes of C++ and C#? Finally, learn about anonymous delegates in C# 2.0 (introduced in the sidebar on page 425). What functionality do they provide that is not available in Java?
- 9.33 In Section 5.5.1 we noted that performance on pipelined processors depends critically on the ability of the hardware to successfully predict the outcome of branches, so that processing of subsequent instructions can begin before processing of the branch has completed. In object-oriented programs, however, knowing the outcome of a branch is not enough: because branches are so often dispatched through vtables, one must also predict the *destination*. Learn how branch prediction works in one or more modern processors. How well do these processors handle object-oriented programs?
- 9.34 Learn about *type hierarchy analysis* and *type propagation*, which can often be used in languages like C++ to infer the concrete type of objects at compile time, allowing the compiler to generate direct calls to methods, rather than indirecting through vtables. How effective are these techniques? What fraction of method calls are they able to optimize in typical benchmarks? What are their limitations? (You might start with the papers of Bacon and Sweeney [BS96] and Diwan et al. [DMM96].)

© 9.35–9.37 In More Depth.

## 9.10 Bibliographic Notes

Appendix A contains bibliographic citations for the various languages discussed in this chapter, including Simula, Smalltalk, C++, Eiffel, Java, C#, Modula-3, Python, Ruby, Ada 95, Oberon, and CLOS. Other object-oriented versions of Lisp include Loops [BS83a] and Flavors [Moo86].

Ellis and Stroustrup [ES90] provide extensive discussion of both semantic and pragmatic issues for C++. Chapters 16 through 19 of Stroustrup's text on C++ [Str97] contain a good introduction to the design and implementation of container classes. Deutsch and Schiffman [DS84] describe techniques to implement Smalltalk efficiently. Borning and Ingalls [BI82] discuss multiple inheritance in an extension to Smalltalk-80. Dolby describes how an optimizing compiler can identify circumstances in which a nested object can be expanded (in the Eiffel sense) while retaining reference semantics [Dol97]. Bacon and Sweeney [BS96] and Diwan et al. [DMM96] discuss techniques to infer the concrete type of objects at compile time, thereby avoiding the overhead of vtable indirection. Driesen presents an alternative to vtables that requires whole-program analysis but provides extremely efficient method dispatch, even in languages with dynamic typing and multiple inheritance [Dri93].

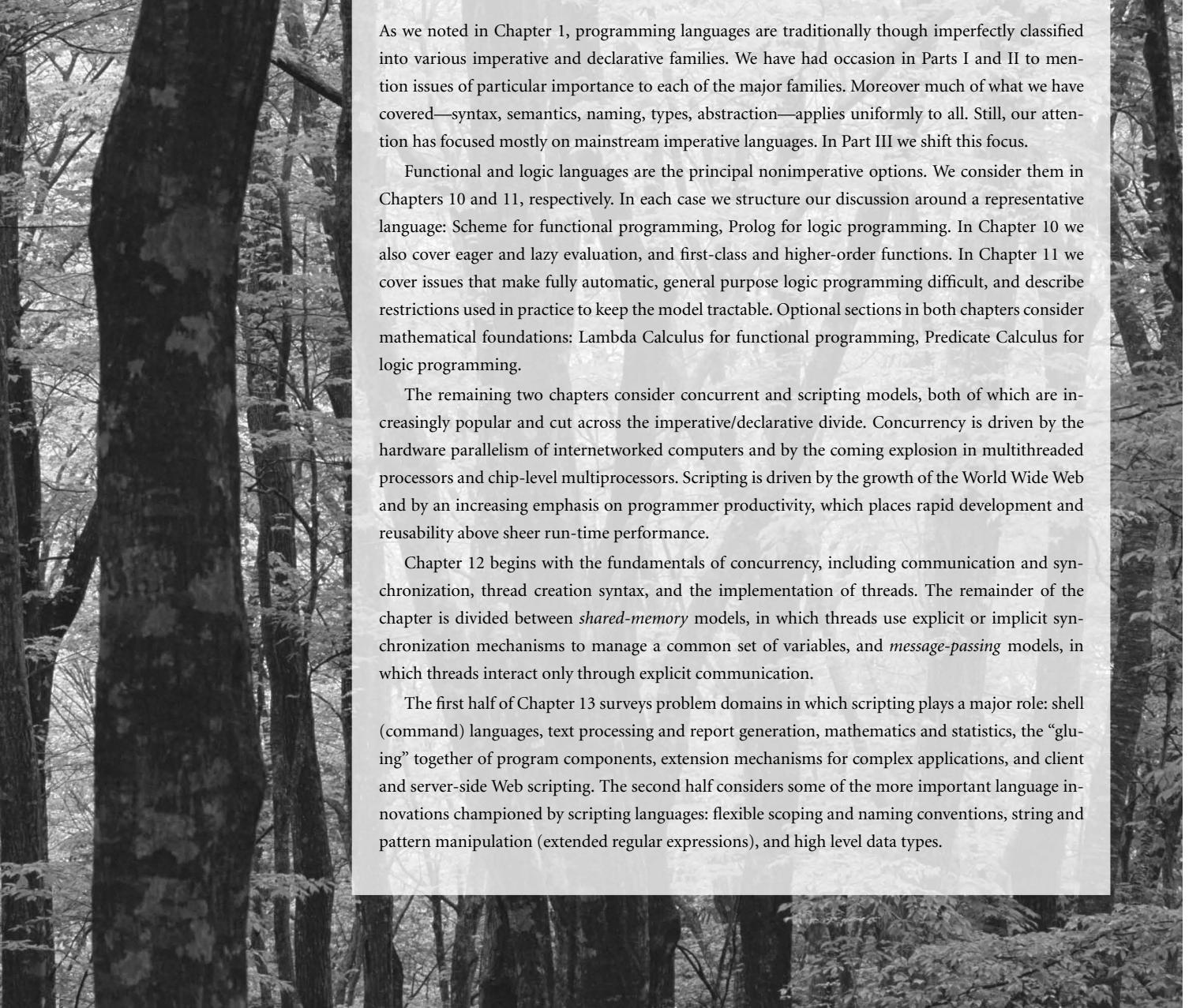
*Component* systems provide a standard for the specification of object interfaces, allowing code produced by arbitrary compilers for arbitrary languages to be joined together into a working program, often spanning a distributed collection of machines. CORBA [Sie96] is a component standard promulgated by the Object Management Group, a consortium of over 700 companies. .NET [WHA03] is a competing standard from Microsoft Corporation, based on their earlier ActiveX, DCOM, and OLE [Bro96] products. JavaBeans [Sun97] is a CORBA-compliant binary standard for components written in Java.

Many of the seminal papers in object-oriented programming have appeared in the proceedings of the ACM OOPSLA conferences (Object-Oriented Programming Systems, Languages, and Applications), held annually since 1986, and published as special issues of ACM SIGPLAN Notices. Wegner [Weg90] enumerates the defining characteristics of object orientation. Meyer [Mey92, Sec. 21.10] explains the rationale for dynamic method binding.





# Alternative Programming Models



As we noted in Chapter 1, programming languages are traditionally though imperfectly classified into various imperative and declarative families. We have had occasion in Parts I and II to mention issues of particular importance to each of the major families. Moreover much of what we have covered—syntax, semantics, naming, types, abstraction—applies uniformly to all. Still, our attention has focused mostly on mainstream imperative languages. In Part III we shift this focus.

Functional and logic languages are the principal nonimperative options. We consider them in Chapters 10 and 11, respectively. In each case we structure our discussion around a representative language: Scheme for functional programming, Prolog for logic programming. In Chapter 10 we also cover eager and lazy evaluation, and first-class and higher-order functions. In Chapter 11 we cover issues that make fully automatic, general purpose logic programming difficult, and describe restrictions used in practice to keep the model tractable. Optional sections in both chapters consider mathematical foundations: Lambda Calculus for functional programming, Predicate Calculus for logic programming.

The remaining two chapters consider concurrent and scripting models, both of which are increasingly popular and cut across the imperative/declarative divide. Concurrency is driven by the hardware parallelism of internetworked computers and by the coming explosion in multithreaded processors and chip-level multiprocessors. Scripting is driven by the growth of the World Wide Web and by an increasing emphasis on programmer productivity, which places rapid development and reusability above sheer run-time performance.

Chapter 12 begins with the fundamentals of concurrency, including communication and synchronization, thread creation syntax, and the implementation of threads. The remainder of the chapter is divided between *shared-memory* models, in which threads use explicit or implicit synchronization mechanisms to manage a common set of variables, and *message-passing* models, in which threads interact only through explicit communication.

The first half of Chapter 13 surveys problem domains in which scripting plays a major role: shell (command) languages, text processing and report generation, mathematics and statistics, the “gluing” together of program components, extension mechanisms for complex applications, and client and server-side Web scripting. The second half considers some of the more important language innovations championed by scripting languages: flexible scoping and naming conventions, string and pattern manipulation (extended regular expressions), and high level data types.



# 10

## Functional Languages

**Previous chapters of this text have focused** largely on imperative programming languages. In the current chapter and the next we emphasize functional and logic languages instead. While imperative languages are far more widely used, “industrial strength” implementations exist for both functional and logic languages, and both models have commercially important applications. Lisp has traditionally been popular for the manipulation of symbolic data, particularly in the field of artificial intelligence. In recent years functional languages—statically typed ones in particular—have become increasingly popular for scientific and business applications as well. Logic languages are widely used for formal specifications and theorem proving and, less widely, for many other applications.

Of course, functional and logic languages have a great deal in common with their imperative cousins. Naming and scoping issues arise under every model. So do types, expressions, and the control-flow concepts of selection and recursion. All languages must be scanned, parsed, and analyzed semantically. In addition, functional languages make heavy use of subroutines—more so even than most von Neumann languages—and the notions of concurrency and nondeterminacy are as common in functional and logic languages as they are in the imperative case.

As noted in Chapter 1, the boundaries between language categories tend to be rather fuzzy. One can write in a largely functional style in many imperative languages, and many functional languages include imperative features (assignment and iteration). The most common logic language—Prolog—provides certain imperative features as well. Finally, it is easy to build a logic programming system in most functional programming languages.

Because of the overlap between imperative and functional concepts, we have had occasion several times in previous chapters to consider issues of particular importance to functional programming languages. Most such languages depend heavily on polymorphism (the implicit parametric kind—Sections 3.6.3

and (§ 7.2.4). Most make heavy use of lists (Section 7.8). Several are dynamically scoped (Sections 3.3.6 and § 3.4.2). All employ recursion (Section 6.6) for repetitive execution, with the result that program behavior and performance depend heavily on the evaluation rules for parameters (Section 6.6.2). All have a tendency to generate significant amounts of temporary data, which their implementations reclaim through garbage collection (Section 7.7.3).

Our chapter begins with a brief introduction to the historical origins of the imperative, functional, and logic programming models. We then enumerate fundamental concepts in functional programming and consider how these are realized in the Scheme dialect of Lisp. More briefly, we also consider Common Lisp, ML, Miranda, Haskell, Sisal, and pH. We pay particular attention to issues of evaluation order and higher-order functions. For those with an interest in the theoretical foundations of functional programming, we provide (on the PLP CD) an introduction to functions, sets, and the lambda calculus. The formalism helps to clarify the notion of a “pure” functional language, and illuminates the differences between the pure notation and its realization in more practical programming languages.

## 10.1 Historical Origins

To understand the differences among programming models, it can be helpful to consider their theoretical roots, all of which predate the development of electronic computers. The imperative and functional models grew out of work undertaken by mathematicians Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, and others in the 1930s. Working largely independently, these individuals developed several very different formalizations of the notion of an algorithm, or *effective procedure*, based on automata, symbolic manipulation, recursive function definitions, and combinatorics. Over time, these various formalizations were shown to be equally powerful: anything that could be computed in one could be computed in the others. This result led Church to conjecture that *any* intuitively appealing model of computing would be equally powerful as well; this conjecture is known as *Church’s thesis*.

Turing’s model of computing was the *Turing machine*, an automaton reminiscent of a finite or pushdown automaton, but with the ability to access arbitrary cells of an unbounded storage “tape.”<sup>1</sup> The Turing machine computes in an imperative way, by changing the values in cells of its tape, just as a high-

---

<sup>1</sup> Alan Turing (1912–1954), for whom the Turing Award is named, was a British mathematician, philosopher, and computer visionary. As intellectual leader of Britain’s cryptanalytic group during World War II, he was instrumental in cracking the German “Enigma” code and turning the tide of the war. He also laid the theoretical foundations of modern computer science, conceived the general purpose electronic computer, and pioneered the field of Artificial Intelligence. Persecuted as a homosexual after the war, stripped of his security clearance, and sentenced to “treatment” with drugs, he committed suicide.

level imperative program computes by changing the values of variables. Church's model of computing is called the *lambda calculus*. It is based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter  $\lambda$ —hence the notation's name).<sup>2</sup> Lambda calculus was the inspiration for functional programming: one uses it to compute by substituting parameters into expressions, just as one computes in a high-level functional program by passing arguments to functions. The computing models of Kleene and Post are more abstract, and do not lend themselves directly to implementation as a programming language.

The goal of early work in computability was not to understand computers (aside from purely mechanical devices, computers did not exist) but rather to formalize the notion of an effective procedure. Over time, this work allowed mathematicians to formalize the distinction between a *constructive* proof (one that shows how to obtain a mathematical object with some desired property) and a *nonconstructive* proof (one that merely shows that such an object must exist, perhaps by contradiction, or counting arguments, or reduction to some other theorem whose proof is nonconstructive). In effect, a program can be seen as a constructive proof of the proposition that, given any appropriate inputs, there exist outputs that are related to the inputs in a particular, desired way. Euclid's algorithm, for example, can be thought of as a constructive proof of the proposition that every pair of nonnegative integers has a greatest common divisor.

Logic programming is also intimately tied to the notion of constructive proofs, but at a more abstract level. Rather than write a general constructive proof that works for all appropriate inputs, the logic programmer writes a set of *axioms* that allow the *computer* to discover a constructive proof for each particular set of inputs. Where the imperative programmer says

To compute the gcd of  $a$  and  $b$ , check to see if  $a$  and  $b$  are equal. If so, print one of them and stop. Otherwise, replace the larger one by their difference and repeat.

and the functional programmer says

The gcd of  $a$  and  $b$  is defined to be  $a$  when  $a$  and  $b$  are equal, and to be the gcd of  $c$  and  $d$  when  $a$  and  $b$  are unequal, where  $c$  is the smaller of  $a$  and  $b$ , and  $d$  is their difference. To compute the gcd of a given pair of numbers, expand and simplify this definition until it terminates.

the logic programmer says

The proposition  $\text{gcd}(a, b, g)$  is true if (1)  $a$ ,  $b$ , and  $g$  are all equal, or (2) there exist numbers  $c$  and  $d$  such that  $c$  is the minimum of  $a$  and  $b$  (i.e.,  $\min(a, b, c)$ )

---

**2** Alonzo Church (1903–1995) was a member of the mathematics faculty at Princeton University from 1929 to 1967, and at UCLA from 1967 to 1990. While at Princeton he supervised the doctoral theses of, among many others, Alan Turing, Stephen Kleene, Michael Rabin, and Dana Scott. His codiscovery, with Turing, of uncomputable problems was a major breakthrough in understanding the limits of mathematics.

is true),  $d$  is their difference (i.e.,  $\text{minus}(a, b, d)$  is true), and  $\text{gcd}(c, d, g)$  is true. To compute the  $\text{gcd}$  of a given pair of numbers, search for a number  $g$  (and various numbers  $c$  and  $d$ ) for which these two rules allow one to prove that  $\text{gcd}(a, b, g)$  is true. ■

We will consider logic programming in more detail in Chapter 11.

## 10.2 Functional Programming Concepts

In a strict sense of the term, *functional programming* defines the outputs of a program as a mathematical function of the inputs, with no notion of internal state, and thus no side effects. Among the common functional programming languages, Miranda, Haskell, Sisal, pH, and Backus's FP proposal [Bac78] are purely functional; Lisp/Scheme and ML include imperative features. To make functional programming practical, functional languages provide a number of features, the following among them, that are often missing in imperative languages.

- First-class function values and higher-order functions
- Extensive polymorphism
- List types and operators
- Recursion
- Structured function returns
- Constructors (aggregates) for structured objects
- Garbage collection

In Section 3.5.2 we defined a first-class value to be one that can be passed as a parameter, returned from a subroutine, or (in a language with side effects) assigned into a variable. Under a strict interpretation of the term, first-class status also requires the ability to create (compute) new values at run time. In the case of subroutines, this notion of first-class status requires that we be able to create a subroutine whose behavior is determined dynamically. Subroutines are second-class values in most imperative languages, but first-class values (in the strict sense of the term) in all functional programming languages. A *higher-order function* takes a function as an argument or returns a function as a result.

Polymorphism is important in functional languages because it allows a function to be used on as general a class of arguments as possible. As we have seen in Sections 7.1 and 7.2.4, Lisp and its dialects are dynamically typed, and thus inherently polymorphic, while ML, Miranda, Haskell, and their relatives obtain polymorphism through the mechanism of type inference. Lists are important in functional languages because they have a natural recursive definition, and are easily manipulated by operating on their first element and (recursively) the re-

mainder of the list. Recursion is important because in the absence of side effects it provides the only means of doing anything repeatedly.

Several of the items in our list of functional language features (recursion, structured function returns, constructors, garbage collection) can be found in some but not all imperative languages. Fortran 77 has no recursion, nor does it allow structured types (i.e., arrays) to be returned from functions. Pascal and early versions of Modula-2 allow only simple and pointer types to be returned from functions. As we saw in Section 7.1.5, several imperative languages, including Ada, C, and Fortran 90, provide aggregate constructs that allow a structured value to be specified in-line. In most imperative languages, however, such constructs are lacking or incomplete. (It is extremely unusual, for example, to find one that can specify an [unnamed] functional value.) A pure functional language must provide completely general aggregates: it cannot build a structured object via assignment to subcomponents. Finally, though garbage collection is increasingly common in imperative languages, it is by no means universal, nor does it apply to objects allocated in the stack. Because of the desire to provide unlimited extent for first-class functions, functional languages tend to employ a (garbage-collected) heap for *all* dynamically allocated data (or at least for all data for which the compiler is unable to prove that stack allocation is safe).

Because Lisp was the original functional language and is still the most widely used, several characteristics of Lisp are commonly, though inaccurately, described as though they pertained to functional programming in general. We will examine these characteristics (in the context of Scheme) in Section 10.3. They include the following.

*homogeneity of programs and data:* A program in Lisp is itself a list, and can be manipulated with the same mechanisms used to manipulate data.

*self-definition:* The operational semantics of Lisp can be defined elegantly in terms of an interpreter written in Lisp.

*interaction with the user* through a “read-eval-print” loop.

Many programmers—probably most—who have written significant amounts of software in both imperative and functional styles find the latter more aesthetically appealing. Moreover experience with a variety of large commercial projects [Wad98a] suggests that the absence of side effects makes functional programs significantly easier to write, debug, and maintain than are their imperative counterparts. When passed a given set of arguments, a pure function can always be counted on to return the same results. Issues of undocumented side effects, misordered updates, and dangling or (in most cases) uninitialized references simply don’t occur. At the same time, most implementations of functional languages still fall short in terms of portability, richness of library packages, interfaces to other languages, and debugging and profiling tools. We will return to the tradeoffs between functional and imperative programming in Section 10.7.

## 10.3

### A Review/Overview of Scheme

**EXAMPLE 10.2**
**The read-eval-print loop**

Most Scheme implementations employ an interpreter that runs a “read-eval-print” loop. The interpreter repeatedly reads an expression from standard input (generally typed by the user), evaluates that expression, and prints the resulting value. If the user types

(+ 3 4)

the interpreter will print

7

If the user types

7

the interpreter will also print

7

(The number 7 is already fully evaluated.) To save the programmer the need to type an entire program verbatim at the keyboard, most Scheme implementations provide a `load` function that reads (and evaluates) input from a file:

`(load "my_Scheme_program")`

As we noted in Section 6.1, Scheme (like all Lisp dialects) uses *Cambridge Polish* notation for expressions. Parentheses indicate a function application (or in some cases the use of a macro). The first expression inside the left parenthesis indicates the function; the remaining expressions are its arguments. Suppose the user types

`((+ 3 4))`

When it sees the inner set of parentheses, the interpreter will call the function `+`, passing 3 and 4 as arguments. Because of the outer set of parentheses, it will then attempt to call 7 as a zero-argument function—a run-time error:

`eval: 7 is not a procedure`

Unlike the situation in almost all other programming languages, extra parentheses change the semantics of Lisp/Scheme programs:

`(+ 3 4)      ==> 7  
 ((+ 3 4))    ==> error`

Here the `==>` means “evaluates to.” This symbol is not a part of the syntax of Scheme itself.

**EXAMPLE 10.4**
**Quoting**

One can prevent the Scheme interpreter from evaluating a parenthesized expression by *quoting* it:

```
(quote (+ 3 4)) ==> (+ 3 4)
```

Here the result is a three-element list. More commonly, quoting is specified with a special shorthand notation consisting of a leading single quote mark:

```
'(+ 3 4) ==> (+ 3 4)
```

Though every expression has a type in Scheme, that type is generally not determined until run time. Most predefined functions check dynamically to make sure that their arguments are of appropriate types. The expression

```
(if (> a 0) (+ 2 3) (+ 2 "foo"))
```

will evaluate to 5 if *a* is positive but will produce a run-time type clash error if *a* is negative or zero. More significantly, as noted in Section 3.6.3, functions that make sense for arguments of multiple types are implicitly polymorphic:

```
(define min (lambda (a b) (if (< a b) a b)))
```

The expression `(min 123 456)` will evaluate to 123; `(min 3.14159 2.71828)` will evaluate to 2.71828.

User-defined functions can implement their own type checks using predefined *type predicate* functions:

```
(boolean? x) ; is x a Boolean?
(char? x) ; is x a character?
(string? x) ; is x a string?
(symbol? x) ; is x a symbol?
(number? x) ; is x a number?
(pair? x) ; is x a (not necessarily proper) pair?
(list? x) ; is x a (proper) list?
```

(This is not an exhaustive list.)

A *symbol* in Scheme is comparable to what other languages call an identifier. The lexical rules for identifiers vary among Scheme implementations but are in general much looser than they are in other languages. In particular, identifiers are permitted to contain a wide variety of punctuation marks:

```
(symbol? 'x$_%:&=*!) ==> #t
```

The symbol `#t` represents the Boolean value true. False is represented by `#f`. Note the use here of quote ('); the symbol begins with `x`.

To create a function in Scheme one evaluates a *lambda expression*:<sup>3</sup>

#### EXAMPLE 10.7

Liberal syntax for symbols

#### EXAMPLE 10.8

Lambda expressions

**3** A word of caution for readers familiar with Common Lisp: A *lambda* expression in Scheme *evaluates to* a function. A *lambda* expression in Common Lisp *is* a function (or, more accurately, is automatically coerced to be a function, without evaluation). The distinction becomes important whenever *lambda* expressions are passed as parameters or returned from functions: they must be quoted in Common Lisp (with `function` or `#'`) to prevent evaluation. Common Lisp also distinguishes between a symbol's *value* and its meaning as a function; Scheme does not: if a symbol represents a function, then the function is the symbol's value.

`(lambda (x) (* x x))`  $\implies$  function

The first “argument” to `lambda` is a list of formal parameters for the function (in this case the single parameter `x`). The remaining “arguments” (again just one in this case) constitute the body of the function. As we shall see in Section 10.4, Scheme differentiates between functions and so-called *special forms* (`lambda` among them), which resemble functions but have special evaluation rules. Strictly speaking, only functions have arguments, but we will also use the term informally to refer to the subexpressions that look like arguments in a special form. ■

A `lambda` expression does not give its function a name; this can be done using `let` or `define` (to be introduced in the next subsection). In this sense, a `lambda` expression is like the aggregates that we used in Section 7.1.5 to specify array or record values.

#### EXAMPLE 10.9

Function evaluation

When a function is called, the language implementation restores the referencing environment that was in effect when the `lambda` expression was evaluated. It then augments this environment with bindings for the formal parameters and evaluates the expressions of the function body in order. The value of the last such expression (most often there is only one) becomes the value returned by the function:

`((lambda (x) (* x x)) 3)`  $\implies$  9 ■

#### EXAMPLE 10.10

If expressions

Simple conditional expressions can be written using `if`:

`(if (< 2 3) 4 5)`  $\implies$  4  
`(if #f 2 3)`  $\implies$  3

In general, Scheme expressions are evaluated in applicative order, as described in Section 6.6.2. Special forms such as `lambda` and `if` are exceptions to this rule. The implementation of `if` checks to see whether the first argument evaluates to `#t`. If so, it returns the value of the second argument, without evaluating the third argument. Otherwise it returns the value of the third argument, without evaluating the second. We will return to the issue of evaluation order in Section 10.4. ■

### 10.3.1 Bindings

#### EXAMPLE 10.11

Nested scopes with `let`

Names can be bound to values by introducing a nested scope.

```
(let ((a 3)
 (b 4)
 (square (lambda (x) (* x x)))
 (plus +))
 (sqrt (plus (square a) (square b)))) \implies 5
```

The special form `let` takes two arguments. The first of these is a list of pairs. In each pair, the first element is a name and the second is the value that the name is to represent within the second argument to `let`. The value of the construct as a whole is then the value of this second argument.

The scope of the bindings produced by `let` is `let`'s second argument only:

```
(let ((a 3))
 (let ((a 4)
 (b a))
 (+ a b))) ==> 7
```

Here `b` takes the value of the *outer* `a`. The way in which names become visible “all at once” at the end of the declaration list precludes the definition of recursive functions. For these one employs `letrec`:

```
(letrec ((fact
 (lambda (n)
 (if (= n 1) 1
 (* n (fact (- n 1)))))))
 (fact 5)) ==> 120
```

There is also a `let*` construct in which names become visible “one at a time” so that later ones can make use of earlier ones, but not vice versa. ■

As noted in Section 3.3, Scheme is statically scoped. (Common Lisp is also statically scoped. Most other Lisp dialects are dynamically scoped.) While `let` and `letrec` allow the user to create nested scopes, they do not affect the meaning of global names (names known at the outermost level of the Scheme interpreter). For these Scheme provides a special form called `define` that has the side effect of creating a global binding for a name:

```
(define hypot
 (lambda (a b)
 (sqrt (+ (* a a) (* b b)))))
(hypot 3 4)) ==> 5
```

#### EXAMPLE 10.12

Global bindings with `define`

#### EXAMPLE 10.13

Basic list operations

### 10.3.2 Lists and Numbers

Like all Lisp dialects, Scheme provides a wealth of functions to manipulate lists. We saw many of these in Section 7.8; we do not repeat them all here. The three most important are `car`, which returns the head of a list, `cdr` (“coulter”), which returns the rest of the list (everything after the head), and `cons`, which joins a head to the rest of a list:

```
(car '(2 3 4)) ==> 2
(cdr '(2 3 4)) ==> (3 4)
(cons 2 '(3 4)) ==> (2 3 4)
```

Also useful is the `null?` predicate, which determines whether its argument is the empty list. Recall that the notation `'(2 3 4)` indicates a *proper* list, in which the final element is the empty list:

```
(cdr '(2)) ==> ()
(cons 2 3) ==> (2 . 3) ; an improper list
```

For fast access to arbitrary elements of a sequence, Scheme provides a `vector` type that is indexed by integers, like an array, and may have elements of heterogeneous types, like a record. Interested readers are referred to the Scheme manual [ADH<sup>+</sup>98] for further information.

Scheme also provides a wealth of numeric and logical (Boolean) functions and special forms. The language manual describes a hierarchy of five numeric types: `integer`, `rational`, `real`, `complex`, and `number`. The last two levels are optional: implementations may choose not to provide any numbers that are not real. Most but not all implementations employ arbitrary-precision representations of both integers and rationals, with the latter stored internally as (numerator, denominator) pairs.

### 10.3.3 Equality Testing and Searching

As described in Section 7.10, Scheme provides three different equality-testing functions. The `eq?` function tests whether its arguments refer to the same object; `eqv?` tests whether its arguments are provably semantically equivalent; `equal?` tests whether its arguments have the same recursive structure, with `eqv?` leaves.

To search for elements in lists, Scheme provides two sets of functions, each of which has variants corresponding to the three different forms of equality. The functions `memq`, `memv`, and `member` take an element and a list as argument, and return the longest suffix of the list (if any) beginning with the element:

```
(memq 'z '(x y z w)) ==> (z w)
(memq '(z) '(x y (z) w)) ==> #f
(member '(z) '(x y (z) w)) ==> ((z) w)
```

The `memq`, `memv`, and `member` functions perform their comparisons using `eq?`, `eqv?`, and `equal?`, respectively. They return `#f` if the desired element is not found. It turns out that Scheme's conditional expressions (e.g., `if`) treat anything other than `#f` as true.<sup>4</sup> One therefore often sees expressions of the form

```
(if (memq desired-element list-that-might-contain-it) ...)
```

#### EXAMPLE 10.14

List search functions

The functions `assq`, `assv`, and `assoc` search for values in *association lists* (otherwise known as *A-lists*). A-lists were introduced in Section ⑩ 3.4.2 in the context of name lookup for languages with dynamic scoping (a picture can be found in Figure ⑩ 3.21, page 28-CD). An A-list is a dictionary implemented as a list of pairs. The first element of each pair is a key of some sort; the second element is information corresponding to that key. `Assq`, `assv`, and `assoc` take a key and an A-list as argument, and return the first pair in the list, if there is one, whose first element is `eq?`, `eqv?`, or `equal?`, respectively, to the key. If there is no matching pair, `#f` is returned.

---

<sup>4</sup> One of the more confusing differences between Scheme and Common Lisp is that Common Lisp uses the empty list () for false, while most implementations of Scheme (including all that conform to the version 5 standard) treat it as true.

### 10.3.4 Control Flow and Assignment

#### EXAMPLE 10.16

Multiway conditional  
expressions

```
(cond
 ((< 3 2) 1)
 ((< 4 3) 2)
 (else 3)) ==> 3
```

The arguments to `cond` are pairs. They are considered in order from first to last. The value of the overall expression is the value of the second element of the first pair in which the first element evaluates to `#t`. If none of the first elements evaluates to `#t`, then the overall value is `#f`. The symbol `else` is permitted only as the first element of the last pair of the construct, where it serves as syntactic sugar for `#t`.

Recursion, of course, is the principal means of doing things repeatedly in Scheme. Many issues related to recursion were discussed in Section 6.6; we do not repeat that discussion here.

For programmers who wish to make use of side effects, Scheme provides assignment, sequencing, and iteration constructs. Assignment employs the special form `set!` and the functions `set-car!` and `set-cdr!`:

```
(let ((x 2)
 (l '(a b)))
 (set! x 3)
 (set-car! l '(c d))
 (set-cdr! l '(e))
 ... x ==> 3
 ... l ==> ((c d) e))
```

The return values of the various varieties of `set!` are implementation-dependent.

#### EXAMPLE 10.17

Assignment

Sequencing uses the special form `begin`:

```
(begin
 (display "hi ")
 (display "mom"))
```

#### EXAMPLE 10.19

Iteration

Iteration uses the special form `do` and the function `for-each`:

```
(define iter-fib (lambda (n)
 ; print the first n+1 Fibonacci numbers
 (do ((i 0 (+ i 1))) ; initially 0, inc'ed in each iteration
 (a 0 b) ; initially 0, set to b in each iteration
 (b 1 (+ a b))) ; initially 1, set to sum of a and b
 ((= i n) b) ; termination test and final value
 (display b) ; body of loop
 (display " "))) ; body of loop
```

```
(for-each (lambda (a b) (display (* a b)) (newline))
 '(2 4 6)
 '(3 5 7))
```

The first argument to do is a list of triples, each of which specifies a new variable, an initial value for that variable, and an expression to be evaluated and assigned (via `set!`) to that variable at the end of each iteration. The second argument to do is a pair that specifies the termination condition and the expression to be returned. At the end of each iteration all new values of loop variables (e.g., `a` and `b`) are computed using the current values. Only after all new values are computed are the assignments actually performed.

The function `for-each` takes as argument a function and a sequence of lists. There must be as many lists as the function takes arguments, and the lists must all be of the same length. `For-each` calls its function argument repeatedly, passing successive sets of arguments from the lists. In the example shown here, the unnamed function produced by the `lambda` expression will be called on the arguments 2 and 3, 4 and 5, and 6 and 7. The interpreter will print

```
6
20
42
()
```

The last line is the return value of `for-each`, assumed here to be the empty list. The language definition allows this value to be implementation-dependent; the construct is executed for its side effects. ■

Two other control-flow constructs have been mentioned in previous chapters. `Delay` and `force` (Section 6.6.2) permit the lazy evaluation of expressions. `Call-with-current-continuation` (`call/cc`; Section 6.2.2) allows the current program counter and referencing environment to be saved in the form of a closure, and passed to a specified subroutine. We will discuss `delay` and `force` further in Section 10.4.

## DESIGN & IMPLEMENTATION

### Iteration in functional programs

It is important to distinguish between iteration as a notation for repeated execution and iteration as a means of orchestrating side effects. As we noted in Section 6.6.1, one can define iteration as syntactic sugar for tail recursion, and in fact Sisal and pH do precisely that (with special syntax to facilitate the passing of values from one iteration to the next). Such a notation may still be entirely side-effect free—entirely functional. Assignment and I/O are the truly imperative features of Scheme. We think of iteration as imperative because most Scheme programs that use it have assignments or I/O in their loops.

### 10.3.5 Programs as Lists

As should be clear by now, a program in Scheme takes the form of a list. In technical terms, we say that Lisp and Scheme are *homoiconic*: self-representing. A parenthesized string of symbols (in which parentheses are balanced) is called an *S-expression* regardless of whether we think of it as a program or as a list. In fact, a program *is* a list, and can be constructed, de-constructed, and otherwise manipulated with all the usual list functions.

Just as `quote` can be used to inhibit the evaluation of a list that appears as an argument in a function call, Scheme provides an `eval` function that can be used to evaluate a list that has been created as a data structure:

```
(define compose
 (lambda (f g)
 (lambda (x) (f (g x)))))
((compose car cdr) '(1 2 3)) ==> 2

(define compose2
 (lambda (f g)
 (eval (list 'lambda '(x) (list f (list g 'x)))
 (scheme-report-environment 5))))
((compose2 car cdr) '(1 2 3)) ==> 2
```

In the first of these declarations, `compose` takes as arguments a pair of functions `f` and `g`. It returns as result a function that takes as parameter a value `x`, applies `g` to it, then applies `f`, and finally returns the result. In the second declaration, `compose2` performs the same function, but in a different way. The function `list` returns a list consisting of its (evaluated) arguments. In the body of `compose2`, this list is the *unevaluated* expression `(lambda (x) (f (g x)))`. When passed to `eval`, this list evaluates to the desired function. The second argument of `eval` specifies the referencing environment in which the expression is to be evaluated. In our example we have specified the environment defined by the Scheme version 5 report [ADH<sup>+</sup>98].

#### Eval **and** Apply

The original description of Lisp [MAE<sup>+</sup>65] included a *self-definition* of the language: code for a Lisp interpreter, written in Lisp. Though Scheme differs in a number of ways from this early Lisp (most notably in its use of lexical scoping), such a *metacircular* interpreter can still be written easily [AS96, Chapter 4]. The code is based on the functions `eval` and `apply`. The first of these we have just seen. The second, `apply`, takes two arguments: a function and a list. It achieves the effect of calling the function, with the elements of the list as arguments.

The functions `eval` and `apply` can be defined as mutually recursive. When passed a number or a string, `eval` simply returns that number or string. When passed a symbol, it looks that symbol up in the specified environment and returns

the value to which it is bound. When passed a list it checks to see whether the first element of the list is one of a small number of symbols that name so-called *primitive* special forms, built into the language implementation. For each of these special forms (`lambda`, `if`, `define`, `set!`, `quote`, etc.) `eval` provides a direct implementation. For other lists, `eval` calls itself recursively on each element and then calls `apply`, passing as arguments the value of the first element (which must be a function) and a list of the values of the remaining elements. Finally, `eval` returns what `apply` returned.

When passed a function  $f$  and a list of arguments  $l$ , `apply` inspects the internal representation of  $f$  to see whether it is primitive. If so it invokes the built-in implementation. Otherwise it retrieves (from the representation of  $f$ ) the referencing environment in which  $f$ 's lambda expression was originally evaluated. To this environment it adds the names of  $f$ 's parameters, with values taken from  $l$ . Call this resulting environment  $e$ . Next `apply` retrieves the list of expressions that make up the body of  $f$ . It passes these expressions, together with  $e$ , one at a time to `eval`. Finally, `apply` returns what the `eval` of the last expression in the body of  $f$  returned.

#### EXAMPLE 10.21

Eval-apply trace of a simple expression

As an example, consider the function `cadr`, defined as `(lambda (x) (car (cdr x)))`. Suppose that this function is represented internally as a three-element list  $C$  consisting of a surrounding referencing environment (an A-list, in this case the global one), a list of parameters (in this case the one-element list  $(x)$ ), and a list of body expressions (in this case the one-element list  $((car (cdr (x))))$ ). Suppose also that  $p$  has been defined to be the list  $(a\ b)$ . To evaluate the expression  $(cadr\ p)$ , a Scheme interpreter written in Scheme would execute `(eval '(cadr p) (scheme-report-environment 5))`. When called, `eval` would begin its work by evaluating the `car` of its first argument—namely `cadr`—via a recursive call. This call would return the function  $c$  to which `cadr` is bound, represented internally as the three-element list  $C$ . Next `eval` would call itself recursively on  $p$ , returning the list  $(a\ b)$ . Finally, `eval` would execute `(apply\ c\ '(a\ b))` and return the result. Internally, `apply` would notice that  $c$  is represented by the list  $(E\ (x)\ (car\ (cdr\ (x))))$ , where  $E$  represents the global environment A-list. It would then execute `(eval\ '(car\ (cdr\ (x)))\ (cons\ (cons\ 'x\ '(a\ b))\ E))` and return the result. We do not trace the remainder of the recursion here. It terminates with primitive special forms in `eval` and primitive functions in `apply`. The latter include `car`, `cdr`, and `cons`. ■

#### Formalizing Self-Definition

The idea of self-definition—a Scheme interpreter written in Scheme—may seem a bit confusing unless one keeps in mind the distinction between the Scheme code that constitutes the interpreter and the Scheme code that the interpreter is interpreting. In particular, the interpreter is not running itself, though it could run a *copy* of itself. What we really mean by “self-definition” is that for all expressions  $E$ , we get the same result by evaluating  $E$  under the interpreter  $I$  that we get by evaluating  $E$  directly.

**EXAMPLE 10.22**

Denotational semantics of Scheme

Suppose now that we wish to formalize the semantics of Scheme as some as-yet-unknown mathematical function  $\mathcal{M}$  that takes a Scheme expression as an argument and returns the expression's value. (This value may be a number, a list, a function, or a member of any of a small number of other domains.) How might we go about this task? For certain simple strings of symbols we can define a value directly: strings of digits, for example, map onto the natural numbers. For more complex expressions, we note that

$$\forall E[\mathcal{M}(E) = (\mathcal{M}(I))(E)]$$

Put another way,

$$\mathcal{M}(I) = \mathcal{M}$$

Suppose now that we let  $H(\mathcal{F}) = \mathcal{F}(I)$  where  $\mathcal{F}$  can be any function that takes a Scheme expression as its argument. Clearly

$$H(\mathcal{M}) = \mathcal{M}$$

Our desired function  $\mathcal{M}$  is said to be a *fixed point* of  $H$ . Because  $H$  is well defined (it simply applies its argument to  $I$ ), we can use it to obtain a rigorous definition of  $\mathcal{M}$ . The tools to do so come from the field of denotational semantics, a subject beyond the scope of this book.<sup>5</sup>

### 10.3.6 Extended Example: DFA Simulation

**EXAMPLE 10.23**

Simulating a DFA in Scheme

To conclude our introduction to Scheme, we present a complete program to simulate the execution of a DFA (deterministic finite automaton). The code appears in Figure 10.1. We invoke the program by calling the function `simulate`, passing it a DFA description and an input string. The DFA description is a list of three items: the start state, the transition function, and a list of final states. The transition function is represented by a list of pairs. The first element of each pair is another pair, whose first element is a state and whose second element is an input symbol. If the current state and next input symbol match the first element of a pair, then the finite automaton enters the state given by the second element of the pair.

As it runs, the automaton accumulates as a list a trace of the states through which it has traveled, ending with the symbol `accept` or `reject`. For example, if we type

---

**5** Actually,  $H$  has an infinite number of fixed points. What we want (and what denotational semantics will give us) is the *least* fixed point: the one that defines a value for as few strings of symbols as possible, while still producing the “correct” value for numbers and other simple strings. Another example of least fixed points appears in Section 15.4.2.

```

(define simulate
 (lambda (dfa input)
 (cons (car dfa) ; start state
 (if (null? input)
 (if (infinal? dfa) '(accept) '(reject))
 (simulate (move dfa (car input)) (cdr input))))))

(define final?
 (lambda (dfa)
 (memq (car dfa) (caddr dfa)))))

(define move
 (lambda (dfa symbol)
 (let ((curstate (car dfa)) (trans (cadr dfa)) (finals (caddr dfa)))
 (list
 (if (eq? curstate 'error)
 'error
 (let ((pair (assoc (list curstate symbol) trans)))
 (if pair (cadr pair) 'error)))
 trans
 finals))))

```

**Figure 10.1** Scheme program to simulate the actions of a DFA. The functions `cadr` and `caddr` are defined as `(lambda (x) (car (cdr x)))` and `(lambda (x) (car (cdr (cdr x))))`, respectively. Scheme provides a large collection of such abbreviations.

```

(simulate
 '(q0 ; start state
 (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0) ; transition fn
 ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
 (q0)) ; final states
 '(0 1 1 0 1)) ; input string

```

then the Scheme interpreter will print

```
(q0 q2 q3 q2 q0 q1 reject)
```

Careful examination of the DFA in this example will reveal that it accepts precisely those strings of zeros and ones in which each digit appears an even number of times. If we change the input string to 010010 the interpreter will print

```
(q0 q2 q3 q1 q3 q2 q0 accept)
```

### CHECK YOUR UNDERSTANDING

1. What mathematical formalism underlies functional programming?
2. List several distinguishing characteristics of functional programming languages.

3. Briefly describe the behavior of the Lisp/Scheme *read-eval-print* loop.
  4. What is a *first-class* value?
  5. Explain the difference between `let`, `let*`, and `letrec` in Scheme.
  6. Explain the difference between `eq?`, `eqv?`, and `equal?`.
  7. Describe three ways in which Scheme programs can depart from a purely functional programming model.
  8. What is an *association list*?
  9. What does it mean for a language to be *homoiconic*?
  10. What is an *S-expression*?
  11. Outline the behavior of `eval` and `apply`.
- 

## 10.4 Evaluation Order Revisited

In Section 6.6.2 we observed that the subcomponents of many expressions can be evaluated in more than one order. In particular, one can choose to evaluate function arguments before passing them to a function, or to pass them unevaluated. The former option is called *applicative-order* evaluation; the latter is called *normal-order* evaluation. Like most imperative languages, Scheme uses applicative order in most cases. Normal order, which arises in the macros and call-by-name parameters of imperative languages, is available in special cases.

Suppose, for example, that we have defined the following function.

```
(define double (lambda (x) (+ x x)))
```

Evaluating the expression `(double (* 3 4))` in applicative order (as Scheme does), we have

```
(double (* 3 4))
 => (double 12)
 => (+ 12 12)
 => 24
```

Under normal-order evaluation we would have

```
(double (* 3 4))
 => (+ (* 3 4) (* 3 4))
 => (+ 12 (* 3 4))
 => (+ 12 12)
 => 24
```

Here we end up doing extra work: normal order causes us to evaluate `(* 3 4)` twice. ■

**EXAMPLE 10.25**

Normal-order avoidance of unnecessary work

In other cases, applicative-order evaluation can end up doing extra work. Suppose we have defined the following.

```
(define switch (lambda (x a b c)
 (cond ((< x 0) a)
 ((= x 0) b)
 ((> x 0) c))))
```

Evaluating the expression `(switch -1 (+ 1 2) (+ 2 3) (+ 3 4))` in applicative order we have

```
(switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⇒ (switch -1 3 (+ 2 3) (+ 3 4))
⇒ (switch -1 3 5 (+ 3 4))
⇒ (switch -1 3 5 7)
⇒ (cond ((< -1 0) 3)
 ((= -1 0) 5)
 ((> -1 0) 7))
⇒ (cond (#t 3)
 ((= -1 0) 5)
 ((> -1 0) 7))
⇒ 3
```

Under normal-order evaluation we would have

```
(switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⇒ (cond ((< -1 0) (+ 1 2))
 ((= -1 0) (+ 2 3))
 ((> -1 0) (+ 3 4)))
⇒ (cond (#t (+ 1 2))
 ((= -1 0) (+ 2 3))
 ((> -1 0) (+ 3 4)))
⇒ (+ 1 2)
⇒ 3
```

Here normal-order evaluation avoids evaluating `(+ 2 3)` or `(+ 3 4)`. ■

Under both evaluation orders we must provide exceptions to the rules in certain cases. Specifically, special forms such as `cond` must take *unevaluated* arguments, even under otherwise applicative-order evaluation, and arithmetic and logical functions such as `+` and `<` must actually yield values, even under otherwise normal-order evaluation, rather than passing their arguments on to something else.

In our overview of Scheme we have differentiated on several occasions between special forms and functions. Arguments to functions are always passed by sharing (Section 8.3.1), and are evaluated before they are passed (i.e., in applicative order). Arguments to special forms are passed unevaluated—in other words, by name. Each special form is free to choose internally when (and if) to evaluate its parameters. `Cond`, for example, takes a sequence of unevaluated pairs as arguments. It evaluates their `cars` internally, one at a time, stopping when it finds one that evaluates to `#t`.

Together, special forms and functions are known as *expression types* in Scheme. Some expression types are *primitive*, in the sense that they must be built into the language implementation. Others are *derived*; they can be defined in terms of primitive expression types. In an eval/apply based interpreter, primitive special forms are built into eval; primitive functions are recognized by apply. We have seen how the special form lambda can be used to create derived functions, which can be bound to names with let. Scheme provides an analogous special form, syntax-rules, that can be used to create derived special forms. These can then be bound to names with let-syntax. Derived special forms are known as *macros* in Scheme, but they are only loosely related to the macros of other languages. In the terminology of this book, Scheme macros are functions whose arguments are passed by name instead of by sharing. They are immune to the problems discussed in Section 6.6.2, and may be implemented in any way that is consistent with their semantics. The macros of C and C++, by contrast, are a low level mechanism for textual expansion.

### 10.4.1 Strictness and Lazy Evaluation

Evaluation order can have an effect not only on execution speed but on program correctness as well. A program that encounters a dynamic semantic error or an infinite regression in an “unneeded” subexpression under applicative-order evaluation may terminate successfully under normal-order evaluation. A (side-effect-free) function is said to be *strict* if it requires all of its arguments to be defined, so that its result will not depend on evaluation order. A function is said to be *nonstrict* if it does not impose this requirement. A *language* is said to be strict if it requires all functions to be strict. A language is said to be nonstrict if it permits the definition of nonstrict functions. Expressions in a strict language can safely be evaluated in applicative order. Expressions in a nonstrict language cannot. ML and (with the exception of macros) Scheme are strict. Miranda and Haskell are nonstrict.

*Lazy evaluation* (as described here—see footnote on page 294) gives us the advantage of normal-order evaluation (not evaluating unneeded subexpressions) while running within a constant factor of the speed of applicative-order evaluation for expressions in which everything is needed. The trick is to tag every argu-

#### DESIGN & IMPLEMENTATION

##### Lazy evaluation

One of the beauties of a purely functional language is that it makes lazy evaluation a completely transparent performance optimization: the programmer can think in terms of nonstrict functions and normal-order evaluation, counting on the implementation to avoid the cost of repeated evaluation. For languages with imperative features, however, this characterization does not hold: lazy evaluation is *not* transparent in the presence of side effects.

ment internally with a “memo” that indicates its value, if known. Any attempt to evaluate the argument sets the value in the memo as a side effect, or returns the value (without recalculating it) if it is already set. Lazy evaluation is particularly useful for “infinite” data structures, as described in Section 6.6.2. It can also be useful in programs that need to examine only a prefix of a potentially long list (see Exercise 10.11). Lazy evaluation is used for all arguments in Miranda and Haskell. It is available in Scheme through explicit use of `delay` and `force`. (Recall that the first of these is a special form that creates a (memo, closure) pair; the second is a function that returns the value in the memo, using the closure to calculate it first if necessary.) Where normal-order evaluation can be thought of as function evaluation using call-by-name parameters, lazy evaluation is sometimes said to employ “call by need.” In addition to Miranda and Haskell, call by need can be found in the R scripting language, widely used by statisticians.

The principal problem with lazy evaluation is its behavior in the presence of side effects. If an argument contains a reference to a variable that may be modified by an assignment, then the value of the argument will depend on whether it is evaluated before or after the assignment. Likewise, if the argument contains an assignment, values elsewhere in the program may depend on when evaluation occurs. These problems do not arise in Miranda or Haskell because they are purely functional: there are no side effects. Scheme leaves the problem up to the programmer, but requires that every use of a `delay-ed` expression be enclosed in `force`, making it relatively easy to identify the places where side effects are an issue. ML provides no built-in mechanism for lazy evaluation. The same effect can be achieved with assignment and explicit functions (Exercise 10.13), but the code is rather awkward.

#### 10.4.2 I/O: Streams and Monads

A major source of side effects can be found in traditional I/O, including the built-in functions `read` and `display` of Scheme: `read` will generally return a different value every time it is called, and multiple calls to `display`, though they never return a value, must occur in the proper order if the program is to be considered correct.

One way to avoid these side effects is to model input and output as *streams*: unbounded-length lists whose elements are generated lazily. We saw an example of a stream in Section 6.6.2, where we used Scheme’s `delay` and `force` to implement a “list” of the natural numbers. Similar code in ML appears in Exercise 10.13.<sup>6</sup>

---

**EXAMPLE 10.26**

Stream-based program execution

If we model input and output as streams, then a program takes the form

```
(define output (my-prog input))
```

---

**6** Note that `delay` and `force` automatically *memoize* their stream so that values are never computed more than once. Exercise 10.13 asks the reader to write a memoizing version of a non-memoizing stream.

When it needs an input value, function `my_prog` forces evaluation of the `car` of `input`, and passes the `cdr` on to the rest of the program. To drive execution, the language implementation repeatedly forces evaluation of the `car` of `output`, prints it, and repeats.

```
(define driver (lambda (s)
 (if (null? s) '() ; nothing left
 (display (car s))
 (driver (cdr s))))))
(driver output)
```

**EXAMPLE 10.27**

Interactive I/O with streams

To make things concrete, suppose we want to write a purely functional program that prompts the user for a sequence of numbers (one at a time!) and prints their squares. If Scheme employed lazy evaluation of input and output streams (it doesn't), then we could write

```
(define squares (lambda (s)
 (cons "please enter a number\n"
 (let ((n (car s)))
 (if (eof-object? n) '()
 (cons (* n n) (cons #\newline (squares (cdr s))))))))
(define output (squares input)))
```

Prompts, inputs, and outputs (i.e., `squares`) would be interleaved naturally in time. In effect, lazy evaluation would *force* things to happen in the proper order: The `car` of `output` is the first prompt. The `cadr` of `output` is the first square, a value that requires evaluation of the `car` of `input`. The `caddr` of `output` is the second prompt. The `cadddr` of `output` is the second square, a value that requires evaluation of the `cadr` of `input`.

Streams formed the basis of the I/O system in early versions of Haskell. Unfortunately, while they successfully encapsulate the imperative nature of interaction at a terminal, they don't work very well for graphics or random access to files. More recent versions of Haskell employ a more general concept known as *monads*. In the context of the Haskell language, a monad is an abstract data type that supports a notion of sequencing. The values of the I/O monad are *actions* that the programmer can force to occur in a specified order.

**EXAMPLE 10.28**

The Haskell I/O monad

Member functions of the Haskell I/O monad take actions as arguments or return actions as results. The `getChar` function, for example, returns an action which, when invoked, will read a character of input; `getChar` is said to be of type `I#IO Char`. The `putChar` function returns an action which, when invoked, will write a character of output; `putChar` is said to be of type `Char -> I#IO ()`. In general, the notation `I#IO t` denotes the type of an action which, when invoked, will return a result of type `t`.

The Haskell I/O monad distinguishes between the *definition* of an action and its *invocation*. Actions can be defined as components of arbitrarily complex data structures in purely functional code. But defining an action does *not* cause it to occur. For that we need the built-in operator `do`. (`Do` is actually syntactic sugar

**EXAMPLE 10.29**

Invocation of actions with  
do

for a pair of more fundamental operators, `>>` and `>>=`; we ignore those operators here.) A trivial Haskell program might look like this:

```
main = do putStrLn "hi, mom\n"
```

When evaluated, `do` causes the invocation of the action returned by `putStrLn`. In general, `do` accepts a sequence of actions, separated by semicolons or newlines, which it invokes in order:

```
do putStrLn "hi, "
 putStrLn "mom\n"
```

**EXAMPLE 10.30**

Functional composition of  
actions

Because actions can be manipulated like ordinary values, we can compose them with arbitrary functions. The `putStrLn` function can be defined in terms of `putChar`:

```
putStrLn :: String -> IO []
-- fn. from string to null-typed action sequence
putStrLn s = sequence (map putChar s)
```

Strings in Haskell are simply lists of characters. The `map` function is assumed to take a function  $f$  and a list  $l$  as argument, and to return a list that contains the results of applying  $f$  to the elements of  $l$ :

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (h:t) = f h : map f t -- `:' is like cons in Scheme
```

Since `putChar` returns an action but does not invoke it, we must pass the result of `map` to a function that will invoke the actions in a list:

```
sequence :: [IO ()] -> IO ()
sequence [] = return ()
sequence (a:more) = do a; sequence more
```

`Sequence` accepts a list of (null-typed) actions as argument and returns a single action consisting of the sequential composition of the actions in the list. If `main` were to evaluate `sequence L`, the actions in `L` would occur.

## DESIGN & IMPLEMENTATION

### Monads

Monads are, in some sense, the conceptual cost of adopting a purely functional model of computation. They acknowledge that the physical world is imperative, and that a language that needs to interact with the physical world in non-trivial ways must include imperative features. The beauty of monads is that they confine those features to a relatively small fraction of the typical program, where their side effects will not interfere with the bulk of the computation.

**EXAMPLE 10.31**

Streams and the I/O monad

In our examples `main` is allowed to use `do` because `main` is of type `IO ()`. Uses of `do` cannot occur in purely functional code. The typical Haskell program contains a small amount of high level imperative code to sequence its I/O operations, while most of the program—both the computation of values and the determination of the order in which any nontrivial set of actions should occur—is purely functional. For a program whose I/O can be expressed in terms of streams, this top-level structure may consist of a single line:

```
main = interact my_program
```

The library function `interact` is of type `(String -> String) -> IO ()`. It takes as argument a function from strings to strings (in this case `my_program`). It calls this function, passing the contents of standard input as argument, and writes the result to standard output. Internally, `interact` uses the function `getContents`, which returns the program's input as a lazily-evaluated string: a stream. In a more sophisticated program, `main` may orchestrate much more complex I/O actions, including graphics and random access to files.

## 10.5 Higher-Order Functions

**EXAMPLE 10.32**

Map function in Scheme

A function is said to be a *higher-order function* (also called a *functional form*) if it takes a function as an argument or returns a function as a result. We have seen several examples already of higher-order functions: `call/cc` (sec-continuations), `for-each` (Example 10.19), `compose` (Example 10.20), and `apply` (page 535). We also saw a Haskell version of the higher-order function `map` in Section 10.4.2. The Scheme version of `map` is slightly more general. Like `for-each`, it takes as argument a function and a *sequence* of lists. There must be as many lists as the function takes arguments, and the lists must all be of the same length. `Map` calls its function argument on corresponding sets of elements from the lists:

```
(map * '(2 4 6) '(3 5 7)) ==> (6 20 42)
```

Where `for-each` is executed for its side effects, and has an implementation-dependent return value, `map` is purely functional: it returns a list composed of the values returned by its function argument.

Programmers in Scheme (or in ML, Haskell, or other functional languages) can easily define other higher-order functions. Suppose, for example, that we want to be able to “fold” the elements of a list together, using an associative binary operator:

```
(define fold (lambda (f l i)
 (if (null? l) i ; i is commonly the identity element for f
 (f (car l) (fold f (cdr l) i))))
```

Now `(fold + '(1 2 3 4 5) 0)` gives us the sum of the first five natural numbers, and `(fold * '(1 2 3 4 5) 1)` gives us their product.

**EXAMPLE 10.34**

Combining higher-order functions

One of the most common uses of higher-order functions is to build new functions from existing ones:

```
(define total (lambda (l) (fold + 1 0)))
(total '(1 2 3 4 5)) ==> 15

(define total-all (lambda (l)
 (map total l)))
(total-all '((1 2 3 4 5)
 (2 4 6 8 10)
 (3 6 9 12 15))) ==> (15 30 45)

(define make-double (lambda (f) (lambda (x) (f x x))))
(define twice (make-double +))
(define square (make-double *))
```

***Currying*****EXAMPLE 10.35**

Partial application with currying

A common operation, named for logician Haskell Curry, is to replace a multi-argument function with a function that takes a single argument and returns a function that expects the remaining arguments:

```
(define curried-plus (lambda (a) (lambda (b) (+ a b))))
((curried-plus 3) 4) ==> 7
(define plus-3 (curried-plus 3))
(plus-3 4) ==> 7
```

Among other things, currying gives us the ability to pass a “partially applied” function to a higher-order function:

```
(map (curried-plus 3) '(1 2 3)) ==> (4 5 6)
```

**EXAMPLE 10.36**

General purpose curry function

It turns out that we can write a general purpose function that “curries” its (binary) function argument:

```
(define curry (lambda (f) (lambda (a) (lambda (b) (f a b)))))
(((curry +) 3) 4) ==> 7
(define curried-plus (curry +))
```

**EXAMPLE 10.37**

Tuples as ML function arguments

ML, Miranda, and Haskell make it especially easy to define curried functions. Consider the following function in ML.

```
fun plus (a, b) : int = a + b;
==> val plus = fn : int * int -> int
```

Recall that the last line is printed by the ML interpreter, and indicates the inferred type of `plus`. The type declaration is required to disambiguate the overloaded `+` operator. Though one may think of `plus` as a function of two arguments, the ML definition says that all functions take a *single* argument. What we have declared is a function that takes a two-element *tuple* as argument. To call `plus`, we juxtapose its name and the tuple that is its argument:

```
plus (3, 4);
==> val it = 7 : int
```

The parentheses here are not part of the function call syntax; they delimit the tuple (3, 4). ■

#### EXAMPLE 10.38

Optional parentheses on singleton arguments

We can declare a single-argument function without parenthesizing its formal argument:

```
fun twice n : int = n + n;
==> val twice = fn : int -> int
twice 2;
==> val it = 4 : int
```

We can add parentheses in either the declaration or the call if we want, but because there is no comma inside, no tuple is implied:

```
fun double (n) : int = n + n;
twice (2);
==> val it = 4 : int
twice 2;
==> val it = 4 : int
double (2);
==> val it = 4 : int
double 2;
==> val it = 4 : int
```

Ordinary parentheses can be placed around any expression in ML. ■

#### EXAMPLE 10.39

Simple curried function in ML

Now consider the definition of a curried function:

```
fun curried_plus a = fn b : int => a + b;
==> val curried_plus = fn : int -> int -> int
```

## DESIGN & IMPLEMENTATION

### Higher-order functions

If higher-order functions are so powerful and useful, why aren't they more common in imperative programming languages? There would appear to be at least two important answers. First, much of the power of first-class functions depends on the ability to create new functions on the fly, and for that we need a function *constructor*: something like Scheme's `lambda` or ML's `fn`. Though they appear in certain recent languages, notably Python and C#, function constructors are a significant departure from the syntax and semantics of traditional imperative languages. Second, the ability to specify functions as return values, or to store them in variables (if the language has side effects), requires either that we eliminate function nesting (something that would again erode the ability of programs to create functions with desired behaviors on the fly) or that we give local variables unlimited extent, thereby increasing the cost of storage management.

Note the type of `curried_plus: int -> int -> int` groups implicitly as `int -> (int -> int)`. Where `plus` is a function mapping a pair (tuple) of integers to an integer, `curried_plus` is a function mapping an integer to a function that maps an integer to an integer:

```
curried_plus 3;
==> val it = fn : int -> int

plus 3;
==> Error: operator domain (int * int) and operand (int) don't agree
```

**EXAMPLE 10.40**

Shorthand notation for currying

To make it easier to declare functions like `curried_plus`, ML allows a sequence of operands in the formal parameter position of a function declaration:

```
fun curried_plus a b : int = a + b;
==> val curried_plus = fn : int -> int -> int
```

This form is simply shorthand for the declaration in the previous example; it does not declare a function of two arguments. `Curried_plus` has a single formal parameter, `a`. Its return value is a function with formal parameter `b` that in turn returns `a + b`.

**EXAMPLE 10.41**

Folding (reduction) in ML

Using tuple notation, our `fold` function might be declared as follows in ML.

```
fun fold (f, l, i) =
 case l of
 nil => i
 | h :: t => f (h, fold (f, t, i));
==> val fold = fn : ('a * 'b -> 'b) * 'a list * 'b -> 'b
```

**EXAMPLE 10.42**

Curried fold in ML

The curried version would be declared as follows.

```
fun curried_fold f l i =
 case l of
 nil => i
 | h :: t => f (h, curried_fold f t i);
==> val fold = fn : ('a * 'b -> 'b) -> 'a list -> 'b -> 'b

curried_fold plus;
==> val it = fn : int list -> int -> int
curried_fold plus [1, 2, 3, 4, 5];
==> val it = fn : int -> int
curried_fold plus [1, 2, 3, 4, 5] 0;
==> val it = 15 : int
```

Note again the difference in the inferred types of the functions.

It is of course possible to define `curried_fold` by nesting occurrences of the explicit `fn` notation within the function's body. The shorthand notation, however, is substantially more intuitive and convenient. Note also that ML's syntax for function calls—juxtaposition of function and argument—makes the use of a curried function more intuitive and convenient than it is in Scheme:

**EXAMPLE 10.43**

Currying in ML v. Scheme

```
curried_fold plus [1, 2, 3, 4, 5] 0; (* ML *)
((curried_fold +) '(1 2 3 4 5)) 0) ; Scheme
```

## 10.6 Theoretical Foundations

**EXAMPLE 10.44**

Declarative  
(nonconstructive) function  
definition

Mathematically, a function is a single-valued mapping: it associates every element in one set (the *domain*) with (at most) one element in another set (the *range*). In conventional notation, we indicate the domain and range of, say, the square root function by writing

$$\text{sqrt} : \mathcal{R} \longrightarrow \mathcal{R}$$

We can also define functions using conventional set notation:

$$\text{sqrt} \equiv \{(x, y) \in \mathcal{R} \times \mathcal{R} \mid y = x^2\}$$

Unfortunately, this notation is *nonconstructive*: it doesn't tell us how to *compute* square roots. Church designed the lambda calculus to address this limitation. ■

**IN MORE DEPTH**

Lambda calculus is a *constructive* notation for function definitions. Any computable function can be written as a lambda expression. Computation amounts to macro substitution of arguments into the function definition, followed by reduction to simplest form via simple and mechanical rewrite rules. The order in which these rules are applied captures the distinction between applicative and normal-order evaluation, as described in Section 6.6.2. Conventions on the use of certain simple functions (e.g., the identity function) allow selection, structures, and even arithmetic to be captured as lambda expressions. Recursion is captured through the notion of *fixed points*.

## 10.7 Functional Programming in Perspective

Side-effect-free programming is a very appealing idea. As discussed in Sections 6.1.2 and 6.3, side effects can make programs both hard to read and hard to compile. By contrast, the lack of side effects makes expressions referentially transparent—*independent of evaluation order*. Programmers and compilers of a purely functional language can employ *equational reasoning*, in which the equivalence of two expressions at any point in time implies their equivalence at all times.

Unfortunately, there are common programming idioms in which the canonical side effect—assignment—plays a central role. Critics of functional programming often point to these idioms as evidence of the need for imperative language

features. I/O is one example. We have seen (in Section 10.4) that sequential access to files can be modeled in a functional manner using streams. For graphics and random file access we have also seen that the monads of Haskell can cleanly isolate the invocation of actions from the bulk of the language, and allow the full power of equational reasoning to be applied to both the computation of values and the determination of the order in which I/O actions should occur.

Other commonly cited examples of “naturally imperative” idioms include the following.

*initialization of complex structures:* The heavy reliance on lists in Lisp, ML, and Haskell reflects the ease with which functions can build new lists out of the components of old lists. Other data structures—multidimensional arrays in particular—are much less easy to put together incrementally, particularly if the natural order in which to initialize the elements is not strictly row-major or column-major.

*summarization:* Many programs include code that scans a large data structure or a large amount of input data, counting the occurrences of various items or patterns. The natural way to keep track of the counts is with a dictionary data structure in which one repeatedly updates the count associated with the most recently noticed key.

*in-place mutation:* In programs with very large data sets, one must economize as much as possible on memory usage, to maximize the amount of data that will fit in memory or the cache. Sorting programs, for example, need to sort in place, rather than copying elements to a new array or list. Matrix-based scientific programs, likewise, need to update values in place.

## DESIGN & IMPLEMENTATION

### Side effects and compilation

As noted in Section 10.2, side-effect freedom has a strong conceptual appeal: it frees the programmer from concern over undocumented access to nonlocal variables, misordered updates, aliases, and dangling pointers. Side-effect freedom also has the potential, at least in theory, to allow the compiler to generate faster code: like aliases, side effects often preclude the caching of values in registers (Section 3.6.1) or the use of constant and copy propagation (Sections 15.3 and 15.4).

So what are the technical obstacles to generating fast code for functional programs? The trivial update problem is certainly a challenge, as is the cost of heap management for values with unlimited extent. Type checking imposes significant run-time costs in languages descended from Lisp but not in those descended from ML. Memoization is expensive in Miranda and Haskell, though so-called *strictness analysis* may allow the compiler to eliminate it in cases where applicative order evaluation is provably equivalent. These challenges are all the subject of continuing research.

These last three idioms are examples of what has been called the *trivial update problem*. If the use of a functional language forces the underlying implementation to create a new copy of the entire data structure every time one of its elements must change, then the result will be very inefficient. In imperative programs, the problem is avoided by allowing an existing structure to be modified in place.

One can argue that while the trivial update problem causes trouble in Lisp and its relatives, it does not reflect an inherent weakness of functional programming per se. What is required for a solution is a combination of convenient notation—to access arbitrary elements of a complex structure—and an implementation that is able to determine when the old version of the structure will never be used again, so it can be updated in place instead of being copied.

Sisal and pH combine array types and iterative syntax with purely functional semantics. The iterative constructs are defined as syntactic sugar for tail recursive functions. When nested, these constructs can easily be used to initialize a multidimensional array. The semantics of the language say that each iteration of the loop returns a new copy of the entire array. The compiler can easily verify, however, that the old copy is never used after the return, and can therefore arrange to perform all updates in place. Similar optimizations could be performed in the absence of the imperative syntax, but they require somewhat more complex analysis. Cann reports [Can92] that the Livermore Sisal compiler is able to eliminate 99–100% of all copy operations in standard numeric benchmarks.

Significant strides in both the theory and practice of functional programming have been made in recent years. Wadler [Wad98b] argues persuasively that the principal remaining obstacles to the widespread adoption of functional languages are social and commercial, not technical: most programmers have been trained in an imperative style; software libraries and development environments for functional programming are not yet as mature as those of their imperative cousins. It seems likely that the coming decade will see a significant increase in the use of functional languages, pure functional languages in particular.



### CHECK YOUR UNDERSTANDING

---

12. What is the difference between *normal-order* and *applicative-order* evaluation? What is *lazy* evaluation?
13. What is the difference between a function and a *special form* in Scheme?
14. What does it mean for a function to be *strict*?
15. What is *memoization*?
16. How can one accommodate I/O in a purely functional programming model?
17. What is a *higher-order* function (also known as a *functional form*)? Give three examples.
18. What is *currying*? What purpose does it serve in practical programs?
19. What is the *trivial update problem* in functional programming?

20. Summarize the arguments for and against side-effect-free programming.
  21. Why do functional languages make such heavy use of lists?
- 

## 10.8 Summary and Concluding Remarks

In this chapter we have focused on the functional model of computing. Where an imperative program computes principally through iteration and side effects (i.e., the modification of variables), a functional program computes principally through substitution of parameters into functions. We began by enumerating a list of key issues in functional programming, including first-class and higher-order functions, polymorphism, control flow and evaluation order, and support for list-based data. We then turned to a concrete example—the Scheme dialect of Lisp—to see how these issues may be addressed in a programming language. We also considered, more briefly, ML and its descendants, Miranda and Haskell.

For imperative programming languages, the underlying formal model is often taken to be a Turing machine. For functional languages, the model is the lambda calculus. Both models evolved in the mathematical community as a means of formalizing the notion of an effective procedure, as used in constructive proofs. Aside from hardware-imposed limits on arithmetic precision, disk and memory space, and so on, the full power of lambda calculus is available in functional languages. While a full treatment of the lambda calculus could easily consume another book, we provided an overview on the PLP CD. We considered rewrite rules, evaluation order, and the Church-Rosser theorem. We noted that conventions on the use of very simple notation provide the computational power of integer arithmetic, selection, recursion, and structured data types.

For practical reasons, many functional languages extend the lambda calculus with additional features, including assignment, I/O, and iteration. Lisp dialects, moreover, are *homoiconic*: programs look like ordinary data structures, and can be created, modified, and executed on the fly.

Lists feature prominently in most functional programs, largely because they can easily be built incrementally, without the need to allocate and then modify state as separate operations. Many functional languages provide other structured data types as well. In Sisal, an emphasis on iterative syntax, tail recursive semantics, and high-performance compilers allows multidimensional array-based functional programs to achieve performance comparable to that of imperative programs.

## 10.9 Exercises

- 10.1 Is the `define` primitive of Scheme an imperative language feature? Why or why not?

- 10.2** It is possible to write programs in a purely functional subset of an imperative language such as C, but certain limitations of the language quickly become apparent. What features would need to be added to your favorite imperative language to make it genuinely useful as a functional language? (*Hint:* What does Scheme have that C lacks?)
- 10.3** Some authors characterize functional programming as one form of declarative programming. Others characterize functional programming as a separate computational model, co-equal with imperative and declarative programming. Which characterization do you prefer? Why?
- 10.4** Explain the connection between short-circuit Boolean expressions and normal-order evaluation. Why is `cond` a special form in Scheme, rather than a function?
- 10.5** Write a program in your favorite imperative language that has the same input and output as the Scheme program of Figure 10.1. Can you make any general observations about the usefulness of Scheme for symbolic computation, based on your experience?
- 10.6** Suppose we wish to remove adjacent duplicate elements from a list (e.g., after sorting). The following Scheme function accomplishes this goal.

```
(define unique
 (lambda (L)
 (cond
 ((null? L) L)
 ((null? (cdr L)) L)
 ((eqv? (car L) (car (cdr L))) (unique (cdr L)))
 (else (cons (car L) (unique (cdr L)))))))
```

Write a similar function that uses the imperative features of Scheme to modify `L` “in place,” rather than building a new list. Compare your function to the code above in terms of brevity, conceptual clarity, and speed.

- 10.7** Write tail-recursive versions of the following.

(a) ;; compute integer log, base 2  
`;; (number of bits in binary representation)`  
`;; works only for positive integers`  
`(define log2`  
 `(lambda (n)`  
 `(if (= n 1) 0 (+ 1 (log2 (quotient (+ n 1) 2))))))`

(b) ;; find minimum element in a list  
`(define min`  
 `(lambda (l)`  
 `(cond`  
 `((null? l) '())`  
 `((null? (cdr l)) (car l))`

```
((null? (cdr l)) (car l))
(#t (let ((a (car l))
 (b (min (cdr l))))
 (if (< b a) b a)))))
```

- 10.8** Write purely functional Scheme functions to
- return all *rotations* of a given list. For example, `(rotate '(a b c d e))` should return `((a b c d e) (b c d e a) (c d e a b) (d e a b c) (e a b c d))` (in some order).
  - return a list containing all elements of a given list that satisfy a given predicate. For example, `(filter (lambda (x) (< x 5)) '(3 9 5 8 2 4 7))` should return `(3 2 4)`.
- 10.9** Write a purely functional Scheme function that returns a list of all permutations of a given list. For example, given `(a b c)` it should return `((a b c) (b a c) (b c a) (a c b) (c a b) (c b a))` (in some order).
- 10.10** Modify the Scheme program of Figure 10.1 to simulate an NFA (nondeterministic finite automaton), rather than a DFA. (The distinction between these automata is described in Section 2.2.1.) Since you cannot “guess” correctly in the face of a multi-valued transition function, you will need either to use explicitly coded backtracking to search for an accepting series of moves (if there is one), or keep track of *all* possible states that the machine could be in at a given point in time.
- 10.11** Consider the problem of determining whether two trees have the same *fringe*: the same set of leaves in the same order, regardless of internal structure. An obvious way to solve this problem is to write a function `flatten` that takes a tree as argument and returns an ordered list of its leaves. Then we can say

```
(define same-fringe
 (lambda (T1 T2)
 (equal (flatten T1) (flatten T2))))
```

Write a straightforward version of `flatten` in Scheme. How efficient is `same-fringe` when the trees differ in their first few leaves? How would your answer differ in a language like Haskell, which uses lazy evaluation for all arguments? How hard is it to get Haskell’s behavior in Scheme, using `delay` and `force`?

- 10.12** We have noted that lists in ML are homogeneous, while lists in Lisp/Scheme may contain elements of varying types. Discuss the advantages and disadvantages of homogeneity.
- 10.13** We can use encapsulation within functions to delay evaluation in ML:

```
datatype 'a delayed_list =
 pair of 'a * 'a delayed_list
 | promise of unit -> 'a * 'a delayed_list;
```

```

fun head (pair (h, r)) = h
| head (promise (f)) = let val (a, b) = f () in a end;
fun rest (pair (h, r)) = r
| rest (promise (f)) = let val (a, b) = f () in b end;

```

Now given

```

fun next_int (n) = (n, promise (fn () => next_int (n + 1)));
val naturals = promise (fn () => next_int (1));

```

we have

|                               |                 |
|-------------------------------|-----------------|
| head (naturals)               | $\Rightarrow 1$ |
| head (rest (naturals))        | $\Rightarrow 2$ |
| head (rest (rest (naturals))) | $\Rightarrow 3$ |
| ...                           |                 |

The delayed list `naturals` is effectively of unlimited length. It will be computed out only as far as actually needed. If a value is needed more than once, however, it will be recomputed every time. Show how to use pointers and assignment (Section 7.7.1, page 375) to memoize the values of a `delayed_list` so that elements are computed only once.

- 10.14** In Example 10.27 we showed how to implement interactive I/O in terms of the lazy evaluation of streams. Unfortunately, our code would not work as written, because Scheme uses applicative-order evaluation. We can make it work, however, with calls to `delay` and `force`.

Suppose we define `input` to be a function that returns an “istream”—a promise that when forced will yield a pair, the `cdr` of which is an istream:

```
(define input (lambda () (delay (cons (read) (input))))))
```

Now we can define the driver to expect an “ostream”—an empty list or a pair, the `cdr` of which is an ostream.

```
(define driver
 (lambda (s)
 (if (null? s) '()
 (display (car s))
 (driver (force (cdr s))))))
```

Note the use of `force`.

Show how to write the function `squares` so that it takes an istream as argument and returns an ostream. You should then be able to type `(driver (squares (input)))` and see appropriate behavior.

- 10.15** Write new versions of `cons`, `car`, and `cdr` that operate on streams. Using them, rewrite the code of the previous exercise to eliminate the calls to `delay` and `force`. Note that the stream version of `cons` will need to

avoid evaluating its second argument; you will need to learn how to define macros (derived special forms) in Scheme.

- 10.16** Write the standard quicksort algorithm in Scheme, without using any imperative language features. Be careful to avoid the trivial update problem; your code should run in expected time  $n \log n$ .

Rewrite your code using arrays (you will probably need to consult a Scheme manual for further information). Compare the running time and space requirements of your two sorts.

- 10.17** Write `insert` and `find` routines that manipulate binary search trees in Scheme (consult an algorithms text if you need more information). Explain why the trivial update problem does *not* impact the asymptotic performance of `insert`.

- 10.18** Write an LL(1) parser generator in purely functional Scheme. If you consult Figure 2.23, remember that you will need to use tail recursion in place of iteration. Assume that the input CFG consists of a list of lists, one per nonterminal in the grammar. The first element of each sublist should be the nonterminal; the remaining elements should be the right-hand sides of the productions for which that nonterminal is the left-hand side. You may assume that the sublist for the start symbol will be the first one in the list. If we use quoted strings to represent grammar symbols, the calculator grammar of Figure 2.15 would look like this:

```
'(("program" ("stmt_list" "$$"))
 ("stmt_list" ("stmt" "stmt_list") ())
 ("stmt" ("id" ":=" "expr") ("read" "id") ("write" "expr"))
 ("expr" ("term" "term_tail"))
 ("term" ("factor" "factor_tail"))
 ("term_tail" ("add_op" "term" "term_tail") ())
 ("factor_tail" ("mult_op" "factor" "FT") ())
 ("add_op" ("+") ("-"))
 ("mult_op" ("*") ("/"))
 ("factor" ("id") ("number") ("(" "expr" ")")))
```

Your output should be a parse table that has this same format, except that every right-hand side is replaced by a *pair* (a 2-element list) whose first element is the predict set for the corresponding production, and whose second element is the right-hand side. For the calculator grammar, the table looks like this:

```
((("program" (($$ "id" "read" "write") ("stmt_list" $$)))
 ("stmt_list" (("id" "read" "write") ("stmt" "stmt_list") (($$) ()))
 ("stmt"
 ((("id") ("id" ":=" "expr"))
 ((("read") ("read" "id")))
 ((("write") ("write" "expr"))))
 ("expr" ((("id" "number") ("term" "term_tail")))))
```

```

("term" ((((" " id" "number") ("factor" "factor_tail"))))
("term_tail"
 (("+" "-") ("add_op" "term" "term_tail"))
 (($$" ") "id" "read" "write") ())
("factor_tail"
 (("*" "/") ("mult_op" "factor" "factor_tail"))
 (($$" ") "+" "-" "id" "read" "write") ())
("add_op" (("+") ("+")) (("-") ("-")))
("mult_op" (("*") ("*")) (("/") ("/")))
("factor"
 (("id") ("id")))
 (("number") ("number")))
 ((("") ("(" "expr" ")")))))

```

(*Hint:* You may want to define a `right_context` function that takes a nonterminal  $B$  as argument and returns a list of all pairs  $(A, \beta)$ , where  $A$  is a nonterminal and  $\beta$  is a list of symbols, such that for some potentially different list of symbols  $\alpha$ ,  $A \longrightarrow \alpha B \beta$ . This function is useful for computing FOLLOW sets. You may also want to build a tail-recursive function that recomputes FIRST and FOLLOW sets until they converge. You will find it easier if you do not include  $\epsilon$  in either set, but rather keep a separate estimate, for each nonterminal, of whether it may generate  $\epsilon$ .)

- 10.19** Write an ML version of the code in Figure 10.1. Alternatively (or in addition), solve Exercises 10.10, 10.11, or 10.16 in ML.

© **10.20–10.23** In More Depth.

## 10.10 Explorations

- 10.24** Read the original self-definition of Lisp [MAE<sup>+</sup>65]. Compare it to a similar definition of Scheme [AS96, Chapter 4]. What is different? What has stayed the same? What is built into `apply` and `eval` in each definition? What do you think of the whole idea? Does a metacircular interpreter really define anything, or is it “circular reasoning”?
- 10.25** Read the Turing Award lecture of John Backus [Bac78], in which he argues for functional programming. How does his FP notation compare to the Lisp and ML language families?
- 10.26** Learn more about monads in Haskell. What exactly *is* a monad? What are monads used for other than I/O? What is their relationship to continuations?
- 10.27** We have seen that Lisp and ML include such imperative features as assignment and iteration. How important are these? What do languages like Haskell give up (conversely, what do they gain) by insisting on a purely

functional programming style? In a similar vein, what do you think of attempts in several recent imperative languages (notably Python and C#—see the sidebar on page 547) to facilitate functional programming with function constructors and unlimited extent?

- 10.28** Investigate the compilation of functional programs. What special issues arise? What techniques are used to address them? Starting places for your search might include the compiler texts of Appel [App97], Wilhelm and Maurer [WM95], and Grune et al. [GBJL01].

© **10.29–10.31** In More Depth.

## 10.11

### Bibliographic Notes

Lisp, the original functional programming language, dates from the work of McCarthy and his associates in the late 1950s. Bibliographic references for Lisp, Scheme, ML, Miranda, Haskell, Sisal, and pH can be found in Appendix A. Historically important dialects of Lisp include Lisp 1.5 [MAE<sup>+</sup>65], MacLisp [Moo78] (no relation to the Apple Macintosh), and Interlisp [TM81].

The book by Abelson and Sussman [AS96], used for introductory programming classes at MIT and elsewhere, is a classic guide to fundamental programming concepts, and to functional programming in particular. Additional historical references can be found in the paper by Hudak [Hud89], which surveys the field from the point of view of Haskell.

The lambda calculus was introduced by Church in 1941 [Chu41]. A classic reference is the text of Curry and Feys [CF58]. Barendregt's book [Bar84] is a standard modern reference. Michaelson [Mic89] provides an accessible introduction to the formalism, together with a clear explanation of its relationship to Lisp and ML. Stansifer [Sta95, Sec. 7.6] provides a good informal discussion and correctness proof for the fixed-point combinator **Y** (see Exercise © 10.21).

John Backus, one of the original developers of Fortran, argued forcefully for a move to functional programming in his 1977 Turing Award lecture [Bac78]. His functional programming notation is known as FP. Peyton Jones [Pey87, Pey92], Wilhelm and Maurer [WM95, Chap. 3], Appel [App97, Chap. 15], and Grune et al. [GBJL01, Chap. 7] discuss the implementation of functional languages.

Wadler describes the use of monads [Wad97]. In other articles he relates experience with several “real-world” applications of functional programming [Wad98a] and discusses the remaining barriers to more widespread use of functional languages [Wad98b]. Hughes provides an articulate statement of the benefits of the functional style [Hug89]. Online discussions of functional programming in general, and of lazy functional programming in particular, can be found in the *comp.lang.functional* newsgroup. A frequently asked questions list for this group, last updated in 2002, can be found at [www.cs.nott.ac.uk/Department/Staff/gmh/faq.html](http://www.cs.nott.ac.uk/Department/Staff/gmh/faq.html). There are also newsgroups devoted to ML (*comp.lang.ml*) and Scheme (*comp.lang.scheme*).



# Logic Languages

**Having considered functional languages in some detail**, we now turn to the other principal declarative paradigm: logic languages. The overlap between imperative and functional concepts in programming language design has led us to discuss the latter at numerous points throughout the text. We have had less occasion to remark on features of logic programming languages. Logic, of course, is used heavily in the design of digital circuits, and most programming languages provide a logical (Boolean) type and operators. Logic is also heavily used in the formal study of language semantics, specifically in *axiomatic semantics*.<sup>1</sup> It was only in the 1970s, however, with the work of Alain Colmerauer and Philippe Roussel of the University of Aix–Marseille in France and Robert Kowalski and associates at the University of Edinburgh in Scotland, that researchers began to employ the process of logical deduction as a general purpose model of computing.

We introduce the basic concepts of logic programming in Section 11.1. We then survey the most widely used logic language, Prolog, in Section 11.2. We consider, in turn, the concepts of resolution and unification, support for lists and arithmetic, and the search-based execution model. After presenting an extended example based on the game of tic-tac-toe, we turn to the more advanced topics of imperative control flow and database manipulation.

Much as functional programming is based on the formalism of lambda calculus, Prolog and other logic languages are based on *first-order predicate calculus*. A brief introduction to this formalism appears in Section ⑩ 11.3 on the PLP CD. Where functional languages capture the full capabilities of the lambda calculus, however (within the limits, at least, of memory and other resources), logic

---

**I** Axiomatic semantics models each statement or expression in the language as a *predicate transformer*—an inference rule that takes a set of conditions known to be true initially and derives a new set of conditions guaranteed to be true after the construct has been evaluated. The study of formal semantics is beyond the scope of this book.

languages do not capture the full power of predicate calculus. We consider the relevant limitations as part of a general evaluation of logic programming in Section 11.4.

## 11.1 Logic Programming Concepts

Logic programming systems allow the programmer to state a collection of *axioms* from which theorems can be proven. The user of a logic program states a theorem, or *goal*, and the language implementation attempts to find a collection of axioms and inference steps (including choices of values for variables) that together imply the goal. Of the several existing logic languages, Prolog is by far the most widely used.

### EXAMPLE 11.1

Horn clauses

In almost all logic languages, axioms are written in a standard form known as a *Horn clause*. A Horn clause consists of a *head*,<sup>2</sup> or *consequent* term  $H$ , and a *body* consisting of terms  $B_i$ :

$$H \leftarrow B_1, B_2, \dots, B_n$$

The semantics of this statement are that when the  $B_i$  are all true, we can deduce that  $H$  is true as well. When reading aloud, we say “ $H$ , if  $B_1, B_2, \dots$ , and  $B_n$ .” Horn clauses can be used to capture most, but not all, logical statements. (We return to the issue of completeness in Section 11.3.) ■

### EXAMPLE 11.2

Resolution

In order to derive new statements, a logic programming system combines existing statements, canceling like terms, through a process known as *resolution*. If we know that  $A$  and  $B$  imply  $C$ , for example, and that  $C$  implies  $D$ , we can deduce that  $A$  and  $B$  imply  $D$ :

$$\begin{array}{c} C \leftarrow A, B \\ D \leftarrow C \\ \hline D \leftarrow A, B \end{array}$$

In general, terms like  $A$ ,  $B$ ,  $C$ , and  $D$  may consist not only of constants (“Rochester is rainy”) but also of *predicates* applied to *atoms* or to *variables*:  $\text{rainy}(\text{Rochester})$ ,  $\text{rainy}(\text{Seattle})$ ,  $\text{rainy}(X)$ . ■

### EXAMPLE 11.3

Unification

During resolution, free variables may acquire values through *unification* with expressions in matching terms, much as variables acquire types in ML (Section 7.2.4):

---

**2** Note that the word *head* is used for two different things in Prolog: the head of a Horn clause and the head of a list. The distinction between these is usually clear from context.

```
flowery(X) ← rainy(X)
```

rainy(Rochester)

```
flowery(Rochester)
```

In the following section we consider Prolog in more detail. We return to formal logic, and to its relationship to Prolog, in Section 11.3. ■

## 11.2 Prolog

Much as a Scheme interpreter evaluates functions in the context of a referencing environment in which other functions and constants have been defined, a Prolog interpreter runs in the context of a *database of clauses* (Horn clauses) that are assumed to be true. Each clause is composed of *terms*, which may be constants, variables, or *structures*. A constant is either an atom or a number. A structure can be thought of as either a logical predicate or a data structure.

Atoms in Prolog are similar to symbols in Lisp. Lexically, an atom looks like an identifier beginning with a lowercase letter, a sequence of “punctuation” characters, or a quoted character string:

```
foo my_Const + 'Hi, Mom'
```

Numbers resemble the integers and floating-point constants of other programming languages. A variable looks like an identifier beginning with an uppercase letter:

```
Foo My_var X
```

Variables can be *instantiated* to (i.e., can take on) arbitrary values at run time as a result of unification. The scope of every variable is limited to the clause in which it appears. There are no declarations. As in Lisp, type checking occurs only when a program attempts to use a value in a particular way at run time. ■

Structures consist of an atom called the *functor* and a list of arguments:

```
rainy(rochester)
teaches(scott, cs254)
bin_tree(foo, bin_tree(bar, glarch))
```

Prolog requires the opening parenthesis to come immediately after the functor, with no intervening space. Arguments can be arbitrary terms: constants, variables, or (nested) structures. Internally, a Prolog implementation can represent a structure using Lisp-like cons cells. Conceptually, the programmer may prefer to think of certain structures (e.g., `rainy`) as logical predicates. We use the term *predicate* to refer to the combination of a functor and an “arity” (number of arguments). The predicate `rainy` has arity 1. The predicate `teaches` has arity 2. ■

### EXAMPLE 11.4

Atoms, variables, scope, and type

### EXAMPLE 11.5

Structures and predicates

**EXAMPLE 11.6**

Facts and rules

The clauses in a Prolog database can be classified as *facts* or *rules*, each of which ends with a period. A fact is a Horn clause without a right-hand side. It looks like a single term (the implication symbol is implicit):

```
rainy(rochester).
```

A rule has a right-hand side:

```
snowy(X) :- rainy(X), cold(X).
```

The token `:-` is the implication symbol; the comma indicates “and.” ( $X$  is snowy if  $X$  is rainy and  $X$  is cold.)

It is also possible to write a clause with an empty left-hand side. Such a clause is called a *query*, or a *goal*. Queries do not appear in Prolog programs. Rather, one builds a database of facts and rules and then initiates execution by giving the Prolog interpreter (or the compiled Prolog program) a query to be answered (i.e., a goal to be proven).

**EXAMPLE 11.7**

Queries

In most implementations of Prolog, queries are entered with a special `?-` version of the implication symbol. If we were to type the following:

```
rainy(seattle).
rainy(rochester).
?- rainy(C).
```

the Prolog interpreter would respond with

```
C = seattle
```

Of course,  $C = rochester$  would also be a valid answer, but Prolog will find `seattle` first, because it comes first in the database. (Dependence on ordering is one of the ways in which Prolog departs from pure logic; we discuss this issue further in Section 11.2.4.) If we want to find all possible solutions, we can ask the interpreter to continue by typing a semicolon:

```
C = seattle;
C = rochester
```

If we type another semicolon, the interpreter will indicate that no further solutions are possible:

```
C = seattle;
C = rochester;
no
```

Similarly, given

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```

the query

```
?- snowy(C).
```

will yield only one solution.

### 11.2.1 Resolution and Unification

#### EXAMPLE 11.8

Resolution in Prolog

The *resolution principle*, due to Robinson [Rob65], says that if  $C_1$  and  $C_2$  are Horn clauses and the head of  $C_1$  matches one of the terms in the body of  $C_2$ , then we can replace the term in  $C_2$  with the body of  $C_1$ . Consider the following example.

```
takes(jane_doe, his201).
takes(jane_doe, cs254).
takes(ajit_chandra, art302).
takes(ajit_chandra, cs254).
classmates(X, Y) :- takes(X, Z), takes(Y, Z).
```

Here if we let  $X$  be `jane_doe` and  $Z$  be `cs254`, we can replace the first term on the right-hand side of the last clause with the (empty) body of the second clause, yielding the new rule

```
classmates(jane_doe, Y) :- takes(Y, cs254).
```

In other words,  $Y$  is a classmate of `jane_doe` if  $Y$  takes `cs254`.

The pattern-matching process used to associate  $X$  with `jane_doe` and  $Z$  with `cs254` is known as *unification*. Variables that are given values as a result of unification are said to be *instantiated*.

The unification rules for Prolog are as follows.

- A constant unifies only with itself.
- Two structures unify if and only if they have the same functor and the same number of arguments, and the corresponding arguments unify recursively.
- A variable unifies with anything. If the other thing has a value, then the variable is instantiated. If the other thing is an uninstantiated variable, then the two variables are associated in such a way that if either is given a value later, that value will be shared by both.

#### EXAMPLE 11.9

Unification in Prolog and ML

Unification of structures in Prolog is very much akin to ML's unification of the types of formal and actual parameters. A formal parameter of type `int * 'b list`, for example, will unify with an actual parameter of type `'a * real list` in ML by instantiating `'a` to `int` and `'b` to `real`.

Equality in Prolog is defined in terms of “unifiability.” The goal `= (A, B)` succeeds if and only if  $A$  and  $B$  can be unified. For the sake of convenience, the goal

**EXAMPLE 11.10**

Equality and unification

may be written as  $A = B$ ; the infix notation is simply syntactic sugar. In keeping with the rules above, we have

```
?- a = a.
yes % constant unifies with itself
?- a = b.
no % but not with another constant
?- foo(a, b) = foo(a, b).
yes % structures are recursively identical
?- X = a.
X = a; % variable unifies with constant
no % only once
?- foo(a, b) = foo(X, b).
X = a; % arguments must unify
no % only one possibility
```

**EXAMPLE 11.11**

Unification without instantiation

It is possible for two variables to be unified without instantiating them. If we type

```
?- A = B.
```

the interpreter will respond

```
A = _123
B = _123
```

where  $_123$  is an underscore followed by some arbitrary (implementation-dependent) integer that represents the (shared) location of  $A$  and  $B$ . In a similar vein, suppose we are given the following rules.

```
takes_lab(S) :- takes(S, C), has_lab(C).
has_lab(D) :- meets_in(D, R), is_lab(R).
```

( $S$  takes a lab class if  $S$  takes  $C$  and  $C$  is a lab class. Moreover  $D$  is a lab class if  $D$  meets in room  $R$  and  $R$  is a lab.) An attempt to resolve these rules will unify the head of the second with the second term in the body of the first, causing  $C$  and  $D$  to be unified, even though neither is instantiated.

**11.2.2 Lists****EXAMPLE 11.12**

List notation in Prolog

Like equality checking, list manipulation is a sufficiently common operation in Prolog to warrant its own notation. The construct  $[a, b, c]$  is syntactic sugar for the structure  $.(a, .(b, .(c, [])))$ , where  $[]$  is the empty list and  $.$  is a built-in cons-like predicate. This notation should be familiar to users of ML. Prolog adds an extra convenience, however: an optional vertical bar that delimits the “tail” of the list. Using this notation,  $[a, b, c]$  could be expressed as  $[a \mid [b, c]]$ ,  $[a, b \mid [c]]$ , or  $[a, b, c \mid []]$ . The vertical-bar notation is particularly handy when the tail of the list is a variable:

```

member(X, [X|T]).
member(X, [H|T]) :- member(X, T).

sorted([]). % empty list is sorted
sorted([X]). % singleton is sorted
sorted([A, B | T]) :- A <= B, sorted([B | T]).
 % compound list is sorted if first two elements are in order and
 % remainder of list (after first element) is sorted

```

Here `=<` is a built-in predicate that operates on numbers. Note that `[a, b | c]` is the *improper* list `.(a, .(b, c))`. The sequence of tokens `[a | b, c]` is syntactically invalid.

One of the interesting things about Prolog resolution is that it does not in general distinguish between “input” and “output” arguments (there are certain exceptions, such as the `is` predicate described in the following subsection). Thus given

```

append([], A, A).
append([H | T], A, [H | L]) :- append(T, A, L).

```

we can type

```

?- append([a, b, c], [d, e], L).
L = [a, b, c, d, e]
?- append(X, [d, e], [a, b, c, d, e]).
X = [a, b, c]
?- append([a, b, c], Y, [a, b, c, d, e]).
Y = [d, e]

```

This example highlights the difference between functions and predicates. The former have a clear notion of inputs (arguments) and outputs (results); the latter do not. In an imperative or functional language we apply functions to arguments to generate results. In a logic language we search for values for which a predicate is true.

### 11.2.3 Arithmetic

#### EXAMPLE 11.14

Arithmetic and the `is` predicate

The usual arithmetic operators are available in Prolog, but they play the role of predicates, not of functions. Thus `+(2, 3)`, which may also be written `2 + 3`, is a two-argument structure, not a function call. In particular, it will not unify with 5:

```

?- (2 + 3) = 5.
no

```

To handle arithmetic, Prolog provides a built-in predicate, `is`, that unifies its first argument with the arithmetic value of its second argument:

```

?- is(X, 1+2).
X = 3
?- X is 1+2.
X = 3 % infix is also ok
?- 1+2 is 4-1.
no % first argument (1+2) is already instantiated
?- X is Y.
<error> % second argument (Y) must already be instantiated
?- Y is 1+2, X is Y.
X = 3
Y = 3 % Y is instantiated by the time it is needed

```

#### 11.2.4 Search/Execution Order

So how does Prolog go about answering a query (satisfying a goal)? What it needs is a sequence of resolution steps that will build the goal out of clauses in the database, or a proof that no such sequence exists. In the realm of formal logic, one can imagine two principal search strategies.

- Start with existing clauses and work forward, attempting to derive the goal. This strategy is known as *forward chaining*.
- Start with the goal and work backward, attempting to “unresolve” it into a set of preexisting clauses. This strategy is known as *backward chaining*.

If the number of existing rules is very large, but the number of facts is small, it is possible for forward chaining to discover a solution more quickly than backward chaining. In most circumstances, however, backward chaining turns out to be more efficient. Prolog is defined to use backward chaining.

Because resolution is associative and commutative (Exercise 11.5), a backward-chaining theorem prover can limit its search to sequences of resolutions in which terms on the right-hand side of a clause are unified with the heads of other clauses one by one in some particular order (e.g., left to right). The resulting search can be described in terms of a tree of subgoals, as shown in Figure 11.1. The Prolog interpreter (or program) explores this tree depth first, from left to right. It starts at the beginning of the database, searching for a rule  $R$  whose head can be unified with the top-level goal. It then considers the terms in the body of  $R$  as subgoals, and attempts to satisfy them, recursively, left to right. If at any point a subgoal fails (cannot be satisfied), the interpreter returns to the previous subgoal and attempts to satisfy it in a different way (i.e., to unify it with the head of a different clause).

The process of returning to previous goals is known as *backtracking*. It strongly resembles the control flow of generators in Icon (Section 6.5.4). Whenever a unification operation is “undone” in order to pursue a different path through the search tree, variables that were given values or associated with one another as a result of that unification are returned to their uninstantiated or unassociated

---

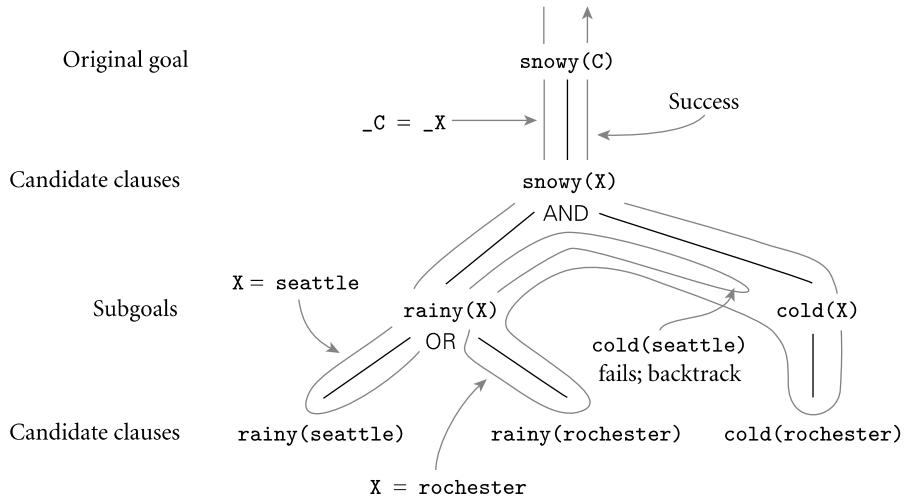
#### EXAMPLE 11.15

##### Search tree exploration

```

rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).

```



**Figure 11.1 Backtracking search in Prolog.** The tree of potential resolutions consists of alternating AND and OR levels. An AND level consists of subgoals from the right-hand side of a rule, all of which must be satisfied. An OR level consists of alternative database clauses whose head will unify with the subgoal above; one of these must be satisfied. The notation  $_C = _X$  is meant to indicate that while both  $C$  and  $X$  are uninstantiated, they have been associated with one another in such a way that if either receives a value in the future it will be shared by both.

### EXAMPLE 11.16

#### Backtracking and instantiation

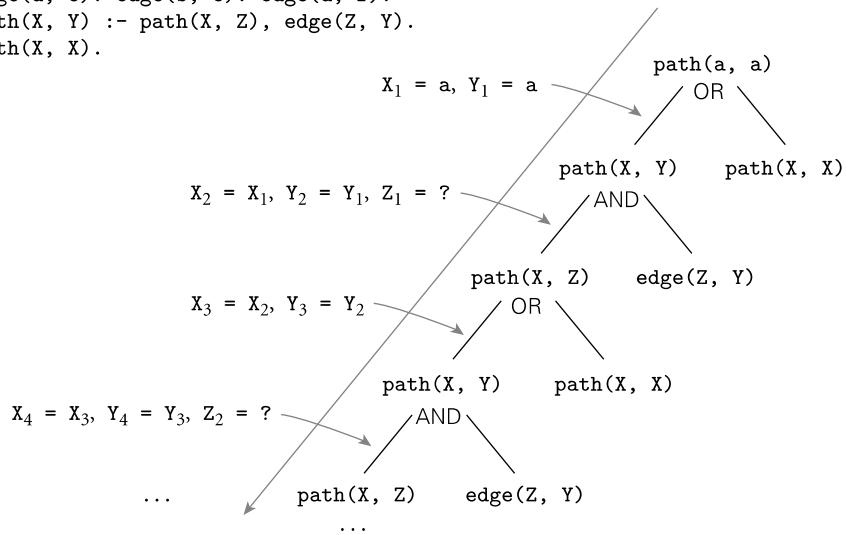
state. In Figure 11.1, for example, the binding of  $X$  to `seattle` is broken when we backtrack to the `rainy(X)` subgoal. The effect is similar to the breaking of bindings between actual and formal parameters in an imperative programming language, except that Prolog couches the bindings in terms of unification rather than subroutine calls. ■

Space management for backtracking search in Prolog usually follows the single-stack implementation of iterators described in Section 8.6.3. The interpreter pushes a frame onto its stack every time it begins to pursue a new subgoal  $G$ . If  $G$  fails, the frame is popped from the stack and the interpreter begins to backtrack. If  $G$  succeeds, control returns to the “caller” (the parent in the search tree), but  $G$ ’s frame remains on the stack. Later subgoals will be given space *above* this dormant frame. If subsequent backtracking causes the interpreter to search for alternative ways of satisfying  $G$ , control will be able to resume where it last left off. Note that  $G$  will not fail unless all of its subgoals (and all of its siblings to the right in the search tree) have also failed, implying that there is nothing above  $G$ ’s frame in the stack. At the top level of the interpreter, a semicolon typed by the user is treated the same as failure of the most recently satisfied subgoal.

```

edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).

```



**Figure 11.2** Infinite regression in Prolog. With this database even a simple query like `?- path(a, a)` will never terminate: the interpreter will never find the trivial branch.

The fact that clauses are ordered, and that the interpreter considers them from first to last, means that the results of a Prolog program are deterministic and predictable. In fact, the combination of ordering and depth-first search means that the Prolog programmer must often consider the order to ensure that recursive programs will terminate. Suppose, for example, that we have a database describing a directed acyclic graph:

```

edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, X).
path(X, Y) :- edge(Z, Y), path(X, Z).

```

The last two clauses tell us how to determine whether there is a path from node  $X$  to node  $Y$ . If we were to reverse the order of the terms on the right-hand side of the final clause, then the Prolog interpreter would search for a node  $Z$  that is reachable from  $X$  before checking to see whether there is an edge from  $Z$  to  $Y$ . The program would still work, but it would not be as efficient. ■

#### EXAMPLE 11.17

Order of rule evaluation

#### EXAMPLE 11.18

Infinite regression

```

path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).

```

From a logical point of view, our database still defines the same relationships. A Prolog interpreter, however, will no longer be able to find answers. Even a simple query like `?- path(a, a)` will never terminate. To see why, consider

Figure 11.2. The interpreter first unifies `path(a, a)` with the left-hand side of `path(X, Y) :- path(X, Z), edge(Z, Y).` It then considers the goals on the right-hand side, the first of which (`path(X, Z)`) unifies with the left-hand side of the very same rule, leading to an infinite regression. In effect, the Prolog interpreter gets lost in an infinite branch of the search tree, and never discovers finite branches to the right. We could avoid this problem by exploring the tree in breadth-first order, but that strategy was rejected by Prolog's designers because of its expense: it can require substantially more space, and does not lend itself to a stack-based implementation.

### 11.2.5 Extended Example: Tic-Tac-Toe

#### EXAMPLE 11.19

##### Tic-tac-toe in Prolog

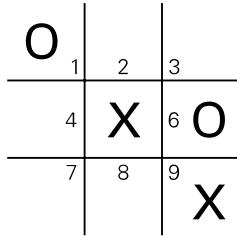
In the previous subsection we saw how the order of clauses in the Prolog database, and the order of terms within a right-hand side, can affect both the efficiency of a Prolog program and its ability to terminate. Ordering also allows the Prolog programmer to indicate that certain resolutions are *preferred*, and should be considered before other, “fallback” options. Consider, for example, the problem of making a move in tic-tac-toe. (Tic-tac-toe is a game played on a  $3 \times 3$  grid of squares. Two players, X and 0, take turns placing markers in empty squares. A player wins if he or she places three markers in a row, horizontally, vertically, or diagonally.)

Let us number the squares from 1 to 9 in row-major order. Further, let us use the Prolog fact `x(n)` to indicate that player X has placed a marker in square  $n$ , and `o(m)` to indicate that player 0 has placed a marker in square  $m$ . For simplicity, let us assume that the computer is player X, and that it is X’s turn to move. We should like to be able to issue a query `?- move(A)` that will cause the Prolog interpreter to choose a good square A for the computer to occupy next.

Clearly we need to be able to tell whether three given squares lie in a row. One way to express this is

```
ordered_line(1, 2, 3). ordered_line(4, 5, 6).
ordered_line(7, 8, 9). ordered_line(1, 4, 7).
ordered_line(2, 5, 8). ordered_line(3, 6, 9).
ordered_line(1, 5, 9). ordered_line(3, 5, 7).
line(A, B, C) :- ordered_line(A, B, C).
line(A, B, C) :- ordered_line(A, C, B).
line(A, B, C) :- ordered_line(B, A, C).
line(A, B, C) :- ordered_line(B, C, A).
line(A, B, C) :- ordered_line(C, A, B).
line(A, B, C) :- ordered_line(C, B, A).
```

It is easy to prove that there is no winning strategy for tic-tac-toe: either player can force a draw. Let us assume, however, that our program is playing against a less-than-perfect opponent. Our task then is never to lose, and to maximize our chances of winning if our opponent makes a mistake. The following rules work well.



**Figure 11.3** A “split” in tac-tac-toe. If X takes the bottom center square (square 8), no future move by O will be able to stop X from winning the game—O cannot block both the 2–5–8 line and the 7–8–9 line.

```

move(A) :- good(A), empty(A).

full(A) :- x(A).
full(A) :- o(A).
empty(A) :- not(full(A)).

% strategy:
good(A) :- win(A). good(A) :- block_win(A).
good(A) :- split(A). good(A) :- strong_build(A).
good(A) :- weak_build(A).

```

The initial rule indicates that we can satisfy the goal `move(A)` by choosing a good, empty square. The `not` is a built-in predicate that succeeds if its argument (a goal) cannot be proven; we discuss it further in the following subsection. Square `n` is empty if we cannot prove it is full; that is, if neither `x(n)` nor `o(n)` is in the database.

The key to strategy lies in the ordering of the last five rules. Our first choice is to win:

```
win(A) :- x(B), x(C), line(A, B, C).
```

Our second choice is to prevent our opponent from winning:

```
block_win(A) :- o(B), o(C), line(A, B, C).
```

Our third choice is to create a “split”—a situation in which our opponent cannot prevent us from winning on the next move (see Figure 11.3):

```

split(A) :- x(B), x(C), different(B, C),
 line(A, B, D), line(A, C, E), empty(D), empty(E).
same(A, A).
different(A, B) :- not(same(A, B)).

```

Here we have again relied on the built-in predicate `not`.

Our fourth choice is to build toward three in a row (i.e., to get two in a row) in such a way that the obvious blocking move won’t allow our opponent to build toward three in a row:

```
strong_build(A) :- x(B), line(A, B, C), empty(C), not(risky(C)).
risky(C) :- o(D), line(C, D, E), empty(E).
```

Barring that, our fifth choice is to build toward three in a row in such a way that the obvious blocking move won't give our opponent a split:

```
weak_build(A) :- x(B), line(A, B, C), empty(C), not(double_risky(C)).
double_risky(C) :- o(D), o(E), different(D, E), line(C, D, F),
line(C, E, G), empty(F), empty(G).
```

If none of these goals can be satisfied, our final, default choice is to pick an unoccupied square, giving priority to the center, the corners, and the sides in that order:

```
good(5).
good(1). good(3). good(7). good(9).
good(2). good(4). good(6). good(8). ■
```

### CHECK YOUR UNDERSTANDING

1. What mathematical formalism underlies logic programming?
2. What is a *Horn clause*?
3. Briefly describe the process of *resolution* in logic programming.
4. What is a *unification*? Why is it important in logic programming?
5. What are *clauses*, *terms*, and *structures* in Prolog? What are *facts*, *rules*, and *queries*?
6. Explain how Prolog differs from imperative languages in its handling of arithmetic.
7. Describe the difference between *forward chaining* and *backward chaining*. Which is used in Prolog by default?
8. Describe the Prolog search strategy. Discuss *backtracking* and the *instantiation* of variables.

## 11.2.6 Imperative Control Flow

We have seen that the ordering of clauses and of terms in Prolog is significant, with ramifications for efficiency, termination, and choice among alternatives. In addition to simple ordering, Prolog provides the programmer with several explicit control-flow features. The most important of these features is known as the *cut*.

The cut is a zero-argument predicate written as an exclamation point: `!`. As a subgoal it always succeeds, but with a crucial side effect: it commits the interpreter to whatever choices have been made since unifying the parent goal with the

**EXAMPLE 11.20**

The cut

left-hand side of the current rule, including the choice of that unification itself. For example, recall our definition of list membership:

```
member(X, [X|T]).
member(X, [H|T]) :- member(X, T).
```

If a given atom *a* appears in list *L* *n* times, then the goal `?- member(a, L)` can succeed *n* times. These “extra” successes may not always be appropriate. They can lead to wasted computation, particularly for long lists, when `member` is followed by a goal that may fail:

```
prime_candidate(X) :- member(X, candidates), prime(X).
```

Suppose that `prime(X)` is expensive to compute. To determine whether *a* is a prime candidate, we first check to see whether it is a member of the `candidates` list, and then check to see whether it is prime. If `prime(a)` fails, Prolog will backtrack and attempt to satisfy `member(a, candidates)` again. If *a* is in the `candidates` list more than once, then the subgoal will succeed again, leading to reconsideration of the `prime(a)` subgoal, even though that subgoal is doomed to fail. We can save substantial time by cutting off all further searches for *a* after the first is found:

```
member(X, [X|T]) :- !.
member(X, [H|T]) :- member(X, T).
```

The cut on the right-hand side of the first rule says that if *X* is the head of *L*, we should not attempt to unify `member(X, L)` with the left-hand side of the second rule; the cut commits us to the first rule. ■

**EXAMPLE 11.21**

Not and its implementation

An alternative way to ensure that `member(X, L)` succeeds no more than once is to embed a use of `not` in the second clause:

```
member(X, [X|T]).
member(X, [H|T]) :- not(X = H), member(X, T).
```

This code will display the same high-level behavior but is slightly less efficient: now the interpreter will actually consider the second rule, abandoning it only after (re)unifying *X* with *H* and reversing the sense of the test.

It turns out that `not` is actually implemented by a combination of the cut and two other built-in predicates, `call` and `fail`:

```
not(P) :- call(P), !, fail.
not(P).
```

The `call` predicate takes a term as argument and attempts to satisfy it as a goal (terms are first-class values in Prolog). The `fail` predicate always fails. ■

In principle, it is possible to replace all uses of the cut with uses of `not`—to confine the cut to the implementation of `not`. Doing so often makes a program easier to read. As we have seen, however, it often makes it less efficient. In some cases, explicit use of the cut may actually make a program *easier* to read. Consider our tic-tac-toe example. If we type semicolons at the program, it will continue to

**EXAMPLE 11.22**

Pruning unwanted answers with the cut

generate a series of increasingly poor moves from the same board position, even though we only want the first move. We can cut off consideration of the others by using the cut:

```
move(A) :- good(A), empty(A), !.
```

To achieve the same effect with `not` we would have to do more major surgery (Exercise 11.8). ■

In general, the cut can be used whenever we want the effect of `if...then...else`:

```
statement :- condition, !, then_part.
statement :- else_part.
```

The `fail` predicate can be used in conjunction with a “generator” to implement a loop. We have already seen how to effect a generator by driving a set of rules “backward.” Recall our definition of `append`:

```
append([], A, A).
append([H | T], A, [H | L]) :- append(T, A, L).
```

To enumerate the ways in which a list can be partitioned into pairs, we can follow a use of `append` with `fail`:

```
print_partitions(L) :- append(A, B, L),
 write(A), write(' '), write(B), nl,
 fail.
```

The `nl` predicate prints a newline character. The query `print_partitions([a, b, c])` produces the following output.

```
[] [a, b, c]
[a] [b, c]
[a, b] [c]
[a, b, c] []
no
```

In some cases, we may have a generator that produces an unbounded sequence of values. The following, for example, generates all of the natural numbers.

```
natural(1).
natural(N) :- natural(M), N is M+1.
```

We can use this generator in conjunction with a “cut-fail” combination to iterate over the first  $n$  numbers:

```
my_loop(N) :- natural(I), I <= N,
 write(I), nl, % loop body (nl prints a newline)
 I = N, !, fail.
```

As long as  $I$  is less than  $N$ , the equality predicate will fail, and backtracking will pursue another alternative for `natural`. If  $I = N$  succeeds, however, then the cut

#### **EXAMPLE 11.23**

Using the cut for selection

#### **EXAMPLE 11.24**

Looping with `fail`

#### **EXAMPLE 11.25**

Looping with an unbounded generator

will be executed, committing us to the current (final) choice of  $I$ , and terminating the loop. ■

This programming idiom—an unbounded generator with a test-cut-fail terminator—is known as *generate-and-test*. Like the iterative constructs of Scheme (pages 533–534), it is generally used in conjunction with side effects. One such side effect, clearly, is I/O. Another—modification of the database—is considered in the following subsection.

Prolog provides a variety of I/O features. In addition to `write` and `n1`, which print to the current output file, the `read` predicate can be used to read terms from the current input file. Individual characters are read and written with `get` and `put`. Input and output can be redirected to different files using `see` and `tell`. Finally, the built-in predicates `consult` and `reconsult` can be used to read database clauses from a file, so they don't have to be typed into the interpreter by hand.

#### **EXAMPLE 11.26**

Character input with `get`

The predicate `get` attempts to unify its argument with the next *printable* character of input, skipping over ASCII characters with codes below 32. In effect, it behaves as if it were implemented in terms of the simpler predicates `get0` and `repeat`:

```
get(X) :- repeat, get0(X), X >= 32, !.
```

The `get0` predicate attempts to unify its argument with the single next character of input, regardless of value and, like `get`, cannot be resatisfied during backtracking. The `repeat` predicate, by contrast, can succeed an arbitrary number of times; it behaves as if it were implemented with the following pair of rules.

```
repeat.
repeat :- repeat.
```

Within the above definition of `get`, backtracking will return to `repeat` as often as needed to produce a printable character (one with ASCII code at least 32). In general, `repeat` allows us to turn any predicate with side effects into a generator. ■

### 11.2.7 Database Manipulation

#### **EXAMPLE 11.27**

Prolog programs as data

Clauses in Prolog are simply collections of terms, connected by the built-in predicates `:−` and `,`, both of which can be written in either infix or prefix form:

|                                                                                                          |                                                                                                                                                                                     |
|----------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>rainy(rochester). rainy(seattle). cold(rochester). snowy(X) :- rainy(X),             cold(X).</pre> | $\left. \right\} \equiv ', '(rainy(rochester),             ', '(rainy(seattle),             ', '(cold(rochester),             ':-(snowy(X), ', '(rainy(X),             cold(X)))))$ |
|----------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Here the single quotes around the prefix commas serve to distinguish them from the commas that separate the arguments of a predicate. ■

**EXAMPLE 11.28**

Modifying the Prolog database

The structural nature of clauses and database contents implies that Prolog, like Scheme, is *homoiconic*: it can represent itself. It can also modify itself. A running Prolog program can add clauses to its database with the built-in predicate `assert`, or remove them with `retract`:

```
?- rainy(X).
X = seattle;
X = rochester;
no
?- assert(rainy(syracuse)).
yes
?- rainy(X).
X = seattle;
X = rochester;
X = syracuse;
no
?- retract(rainy(rochester)).
yes
?- rainy(X).
X = seattle;
X = syracuse;
no
```

**EXAMPLE 11.29**

Tic-tac-toe (full game)

Figure 11.4 contains a complete Prolog program for tic-tac-toe. It uses `assert`, `retract`, the cut, `fail`, `repeat`, and `write` to play an entire game. Moves are added to the database with `assert`. They are cleared with `retract` at the beginning of each game. This way the user can play multiple games without restarting the interpreter.

**DESIGN & IMPLEMENTATION****Homoiconic languages**

As we have noted, both Lisp/Scheme and Prolog are *homoiconic*. A few other languages—notably Snobol, Forth, and Tcl—share this property. What is its significance? For most programs the answer is: not much. As long as we write the sorts of programs that we’d write in other languages, the fact that programs and data look the same is really just a curiosity. It becomes something more if we are interested in *metacomputing*—the creation of programs that create or manipulate other programs, or that extend themselves. Metacomputing requires, at the least, that we have true first-class functions in the strict sense of the term—that is, that we be able to generate new functions whose behavior is determined dynamically. A homoiconic language can simplify metacomputing by eliminating the need to translate between internal (data structure) and external (syntactic) representations of programs or program extensions.

```

ordered_line(1, 2, 3). ordered_line(4, 5, 6). ordered_line(7, 8, 9).
ordered_line(1, 4, 7). ordered_line(2, 5, 8). ordered_line(3, 6, 9).
ordered_line(1, 5, 9). ordered_line(3, 5, 7).
line(A, B, C) :- ordered_line(A, B, C). line(A, B, C) :- ordered_line(A, C, B).
line(A, B, C) :- ordered_line(B, A, C). line(A, B, C) :- ordered_line(B, C, A).
line(A, B, C) :- ordered_line(C, A, B). line(A, B, C) :- ordered_line(C, B, A).

full(A) :- x(A). full(A) :- o(A). empty(A) :- not(full(A)).
% NB: empty must be called with an already-instantiated A.
same(A, A). different(A, B) :- not(same(A, B)).

move(A) :- good(A), empty(A), !.

% strategy:
good(A) :- win(A). good(A) :- block_win(A). good(A) :- split(A).
good(A) :- strong_build(A). good(A) :- weak_build(A).
good(5). good(1). good(3). good(7). good(9). good(2). good(4). good(6). good(8).

win(A) :- x(B), x(C), line(A, B, C).
block_win(A) :- o(B), o(C), line(A, B, C).
split(A) :- x(B), x(C), different(B, C), line(A, B, D), line(A, C, E), empty(D), empty(E).
strong_build(A) :- x(B), line(A, B, C), empty(C), not(risky(C)).
weak_build(A) :- x(B), line(A, B, C), empty(C), not(double_risky(C)).
risky(A) :- o(D), line(C, D, E), empty(E).
double_risky(C) :- o(D), o(E), different(D, E), line(C, D, F), line(C, E, G), empty(F), empty(G).

all_full :- full(1), full(2), full(3), full(4), full(5), full(6), full(7), full(8), full(9).
done :- ordered_line(A, B, C), x(A), x(B), x(C), write('I won.'), nl.
done :- all_full, write('Draw.'), nl.

getmove :- repeat, write('Please enter a move: '), read(X), empty(X), assert(o(X)).
makemove :- move(X), !, assert(x(X)).
makemove :- all_full.

printsquare(N) :- o(N), write(' o ').
printsquare(N) :- x(N), write(' x ').
printsquare(N) :- empty(N), write(' ').
printboard :- printsquare(1), printsquare(2), printsquare(3), nl,
 printsquare(4), printsquare(5), printsquare(6), nl,
 printsquare(7), printsquare(8), printsquare(9), nl.
clear :- x(A), retract(x(A)), fail.
clear :- o(A), retract(o(A)), fail.

% main goal:
play :- not(clear), repeat, getmove, respond.
respond :- ordered_line(A, B, C), o(A), o(B), o(C),
 printboard, write('You won.'), nl. % Shouldn't ever happen!
respond :- makemove, printboard, done.

```

Figure 11.4 Tic-tac-toe program in Prolog.

**EXAMPLE 11.30**

The functor predicate

Individual terms in Prolog can be created, or their contents extracted, using the built-in predicates `functor`, `arg`, and `=...`. The goal `functor(T, F, N)` succeeds if and only if `T` is a term with functor `F` and number of arguments `N`:

```
?- functor(foo(a, b, c), foo, 3).
yes
?- functor(foo(a, b, c), F, N).
F = foo
N = 3
?- functor(T, foo, 3).
T = foo(_10, _37, _24)
```

**EXAMPLE 11.31**

Creating terms at run time

The goal `arg(N, T, A)` succeeds if and only if its first two arguments (`N` and `T`) are instantiated, `N` is a natural number, `T` is a term, and `A` is the `N`th argument of `T`:

```
?- arg(3, foo(a, b, c), A).
A = c
```

Using `functor` and `arg` together, we can create an arbitrary term:

```
?- functor(T, foo, 3), arg(1, T, a), arg(2, T, b), arg(3, T, c).
T = foo(a, b, c)
```

Alternatively, we can use the (infix) `=...` predicate, which “equates” a term with a list:

```
?- T =.. [foo, a, b, c].
T = foo(a, b, c)

?- foo(a, b, c) =.. [F, A1, A2, A3].
F = foo
A1 = a
A2 = b
A3 = c
```

Note that

```
?- foo(a, b, c) = F(A1, A2, A3).
```

and

```
?- F(A1, A2, A3) = foo(a, b, c).
```

do not work: the term preceding a left parenthesis must be an atom, not a variable.

**EXAMPLE 11.32**

Pursuing a dynamic goal

Using `=...` and `call`, the programmer can arrange to pursue (attempt to satisfy) a goal created at run time:

```
param_loop(L, H, F) :- natural(I), I >= L, I <= H,
 G =.. [F, I], call(G),
 I = H, !, fail.
```

The goal `param_loop(5, 10, write)` will produce the following output.

```
5678910
no
```

If we want the numbers on separate lines we can write

```
writeln(X) :- write(X), nl.
...
?- param_loop(5, 10, writeln).
```

Taken together, the predicates described above allow a Prolog program to create and decompose clauses, and to add and subtract them from the database. So far, however, the only mechanism we have for *perusing* the database (i.e., to determine its contents) is the built-in search mechanism. To allow programs to “reason” in more general ways, Prolog provides a `clause` predicate that attempts to match its two arguments against the head and body of some existing clause in the database:

```
?- clause(snowy(X), B).
X = _19
B = rainy(_19), cold(_19);
no
```

Here we have discovered (by entering a query and requesting further matches with a semicolon) that there is a single rule in the database whose head is a single-argument term with functor `snowy`. The body of that rule is the conjunction

## DESIGN & IMPLEMENTATION

### Reflection

A language mechanism is said to be *reflective* if it allows a program to reason about its own structure. A language is said to be *fully reflective* if it allows a program to reason about all aspects of its current structure and state. Fully reflective languages are still just research prototypes, but limited forms of reflection appear in several languages. The `clause` predicate in Prolog is a noteworthy example. Given the functor and arity of a starting goal, it allows a program to explore the substructure under that goal in the database. Using `clause`, the programmer can in fact write a *metacircular interpreter* (i.e., an implementation of `call`, see Exercise 11.12) or an evaluator that uses a nonstandard search order (e.g., breadth-first or forward-chaining, see Exercise 11.13). Other languages with significant reflection facilities include Java, C#, Perl, PHP, Tcl, Python, and Ruby, all of which allow a program to inspect and reason about its complete type structure. A few languages (e.g., Python) allow a program to inspect its own source code as text, but this is not as powerful as the homoiconic inspection of Prolog or Scheme, which allows a program to *reason about* its own code structure directly.

`B = rainy(_19), cold(_19)`, where `_19` is the (uninstantiated) argument of the head of the rule. Prolog requires that the first argument to `clause` be sufficiently instantiated that its functor can be determined.

A clause with no body (a fact) matches the body `true`:

```
?- clause(rainy(rochester), true).
yes
```

Note that `clause` is quite different from `call`: it does not attempt to satisfy a goal, simply to match it against an existing clause:

```
?- clause(snowy(rochester)).
no
```

Various other built-in predicates can also be used to “deconstruct” the contents of a clause. The `var` predicate takes a single argument; it succeeds as a goal if and only if its argument is an uninstantiated variable. The `atom` and `integer` predicates succeed as goals if and only if their arguments are atoms and integers, respectively. The `name` predicate takes two arguments. It succeeds as a goal if and only if its first argument is an atom and its second is a list composed of the ASCII codes for the characters of that atom.

## 11.3 Theoretical Foundations

### EXAMPLE 11.34

Predicates as mathematical objects

In mathematical logic, a *predicate* is a function that maps constants (atoms) or variables to the values true and false. If `rainy` is a predicate, for example, we might have `rainy(Seattle) = true` and `rainy(Tijuana) = false`. *Predicate calculus* provides a notation and inference rules for constructing and reasoning about *propositions* (*statements*) composed of predicate applications, *operators* (and, or, not, etc.), and the *quantifiers*  $\forall$  and  $\exists$ . Logic programming formalizes the search for variable values that will make a given proposition true.

### IN MORE DEPTH

In conventional logical notation there are many ways to state a given proposition. Logic programming is built on *clausal form*, which provides a unique expression for every proposition. Many though not all clausal forms can be cast as a collection of Horn clauses, and thus translated into Prolog. On the PLP CD we trace the steps required to translate an arbitrary proposition into clausal form. We also characterize the cases in which this form can and cannot be translated into Prolog.

## 11.4 Logic Programming in Perspective

In the abstract, logic programming is a very compelling idea: it suggests a model of computing in which we simply list the logical properties of an unknown value,

and then the computer figures out how to find it (or tells us it doesn't exist). Unfortunately, the current state of the art falls quite a bit short of the vision, for both theoretical and practical reasons.

### 11.4.1 Parts of Logic Not Covered

As noted in Section 11.3, Horn clauses do not capture all of first-order predicate calculus. In particular, they cannot be used to express statements whose clausal form includes a disjunction with more than one nonnegated term. We can sometimes get around this problem in Prolog by using the `not` predicate, but the semantics are not the same (see Section 11.4.3).

### 11.4.2 Execution Order

While logic is inherently declarative, most logic languages explore the tree of possible resolutions in deterministic order. Prolog provides a variety of predicates, including the `cut`, `fail`, and `repeat`, to control that execution order (Section 11.2.6). It also provides predicates, including `assert`, `retract`, and `call`, to manipulate its database explicitly during execution.

#### DESIGN & IMPLEMENTATION

##### Implementing logic

Predicate calculus is a significantly higher-level notation than lambda calculus. It is much more abstract—much less algorithmic. It is natural, therefore, that a language like Prolog not provide the full power of predicate calculus, and that it include extensions to make it more algorithmic. We may someday reach the point where programming systems are capable of discovering good algorithms from very high-level declarative specifications, but we are not there yet.

#### DESIGN & IMPLEMENTATION

##### Alternative search strategies

Some approaches to logic programming attempt to customize the run-time search strategy in a way that is likely to satisfy goals quickly. Darlington [Dar90], for example, describes a technique in which, when an intermediate goal  $G$  fails, we try to find alternative instantiations of the variables in  $G$  that will allow it to succeed, *before* backing up to previous goals and seeing whether the alternative instantiations will work in them as well. This “failure-directed search” seems to work well for certain classes of problems. Unfortunately, no general technique is known that will automatically discover the best algorithm (or even just a “good” one) for any given problem.

**EXAMPLE 11.35**

Sorting incredibly slowly

In Section 11.2.4 (page 566), we saw that one must often consider execution order to ensure that a Prolog search will terminate. Even for searches that terminate, naive code can be *very* inefficient. Consider the problem of sorting. A natural declarative way to say that L2 is the sorted version of L1 is to say that L2 is a permutation of L1 and L2 is sorted:

```
declarative_sort(L1, L2) :- permutation(L1, L2), sorted(L2).
permutation([], []).
permutation(L, [H|T]) :- append(P, [H|S], L), append(P, S, W),
 permutation(W, T).
```

(The `append` and `sorted` predicates are defined in Section 11.2.2.) Unfortunately, Prolog’s default search strategy will take exponential time to sort a list based on these rules: it will generate permutations until it finds one that is sorted.

To obtain a more efficient sort, the Prolog programmer must adopt a less natural, “imperative” definition:

```
quicksort([], []).
quicksort([A|L1], L2) :- partition(A, L1, P1, S1),
 quicksort(P1, P2), quicksort(S1, S2), append(P2, [A|S2], L2).
partition(A, [], [], []).
partition(A, [H|T], [H|P], S) :- A >= H, partition(A, T, P, S).
partition(A, [H|T], P, [H|S]) :- A <= H, partition(A, T, P, S).
```

Even this sort is less efficient than one might hope in certain cases. When given an already-sorted list, for example, it takes quadratic time, instead of  $O(n \log n)$ . A good heuristic for quicksort is to partition the list using the median of the first, middle, and last elements. Unfortunately, Prolog provides no easy way to access the middle and final elements of a list (it has no arrays).

As we saw in Chapter 9, it can be useful to distinguish between the *specification* of a program and its *implementation*. The specification says what the program is to do; the implementation says how it is to do it. Horn clauses provide an excellent notation for specifications. When augmented with search rules (as in Prolog) they allow implementations to be expressed in the same notation.

### 11.4.3 Negation and the “Closed World” Assumption

A collection of Horn clauses, such as the facts and rules of a Prolog database, constitutes a list of things assumed to be true. It does not include any things assumed to be false. This reliance on purely “positive” logic explains why Prolog’s `not` predicate is different from logical negation. Unless the database is assumed to contain *everything* that is true (this is the *closed world assumption*), the goal `not(T)` can succeed simply because our current knowledge is insufficient to prove T. Moreover, negation in Prolog occurs *outside* any implicit existential quantifiers on the right-hand side of a rule. Thus

**EXAMPLE 11.37**

Negation as failure

```
?- not(takes(X, his201)).
```

where X is uninstantiated, means

$$? \neg \exists X[takes(X, his201)]$$

rather than

$$? \exists X[\neg takes(X, his201)]$$

If our database indicates that jane\_doe takes his201, then the goal takes(X, his201) can succeed, and not(takes(X, his201)) will fail:

```
?- not(takes(X, his201)).
no
```

If we had a way to put the negation inside the quantifier, we might hope for an implementation that would respond

```
?- not(takes(X, his201)).
X = ajit_chandra
```

or even

```
?- not(takes(X, his201)).
X != jane_doe
```

A complete characterization of the values of X for which  $\neg takes(X, his201)$  is true would require a complete exploration of the resolution tree, something that Prolog does only when all goals fail, or when repeatedly prompted with semicolons. Mechanisms to incorporate some sort of “constructive negation” into logic programming are an active topic of research. ■

#### **EXAMPLE 11.38**

#### Negation and instantiation

It is worth noting that the definition of not in terms of failure means that variable bindings are lost whenever not succeeds. For example,

```
?- takes(X, his201).
X = jane_doe
?- not(takes(X, his201)).
no
?- not(not(takes(X, his201))).
X = _395
```

When takes first succeeds, X is bound to jane\_doe. When the inner not fails, the binding is broken. Then when the outer not succeeds, a new binding is created to an uninstantiated value. Prolog provides no way to pull the binding of X out through the double negation. ■

#### **CHECK YOUR UNDERSTANDING**

9. Explain the purpose of the cut (!) in Prolog. How does it relate to not?
10. Describe three ways in which Prolog programs can depart from a pure logic programming model.

11. Describe the *generate-and-test* programming idiom.
  12. Summarize Prolog's facilities for database manipulation. Be sure to mention `assert`, `retract`, and `clause`.
  13. What sorts of logical statements cannot be captured in Horn clauses?
  14. What is the *closed world assumption*? What problems does it cause for logic programming?
- 

## 11.5 Summary and Concluding Remarks

In this chapter we have focused on the logic model of computing. Where an imperative program computes principally through iteration and side effects, and a functional program computes principally through substitution of parameters into functions, a logic program computes through the resolution of logical statements, driven by the ability to unify variables and terms.

Much of our discussion was driven by an examination of the principal logic language, Prolog, which we used to illustrate clauses and terms, resolution and unification, search/execution order, list manipulation, and high-order predicates for inspection and modification of the logic database.

Like imperative and functional programming, logic programming is related to constructive proofs. But where an imperative or functional program in some sense *is* a proof (of the ability to generate outputs from inputs), a logic program is a set of axioms from which the computer attempts to construct a proof. And where imperative and functional programming provide the full power of Turing machines and lambda calculus, respectively (ignoring hardware-imposed limits on arithmetic precision, disk and memory space, etc.), Prolog provides less than the full generality of resolution theorem proving, in the interests of time and space efficiency. At the same time, Prolog extends its formal counterpart with true arithmetic, I/O, imperative control flow, and higher-order predicates for self-inspection and modification.

Like Lisp/Scheme, Prolog makes heavy use of lists, largely because they can easily be built incrementally, without the need to allocate and then modify state as separate operations. And like Lisp/Scheme (but unlike ML and its descendants), Prolog is *homoiconic*: programs look like ordinary data structures, and can be created, modified, and executed on the fly.

As we stressed in Chapter 1, different models of computing are appealing in different ways. Imperative programs more closely mirror the underlying hardware, and can more easily be “tweaked” for high performance. Purely functional programs avoid the semantic complexity of side effects, and have proven particularly handy for the manipulation of symbolic (nonnumeric) data. Logic programs, with their highly declarative semantics and their emphasis on unification, are well-suited to problems that emphasize relationships and search. At the same

time, their de-emphasis of control flow can lead to inefficiency. At the current state of the art, computers have surpassed people in their ability to deal with low-level details (e.g., of instruction scheduling), but people are still better at inventing good algorithms.

As we also stressed in Chapter 1, the borders between language classes are often very fuzzy. The backtracking search of Prolog strongly resembles the execution of generators in Icon. Unification in Prolog resembles (but is more powerful than) the pattern matching capabilities of ML and Haskell. (Unification is also used for type checking in ML and Haskell, and for template instantiation in C++, but those are *compile-time* activities.)

There is much to be said for programming in a purely functional or logic-based style. While most Scheme and Prolog programs make some use of imperative language features, those features tend to be responsible for a disproportionate share of program bugs. At the same time, there seem to be programming tasks—graphical I/O, for example—that are almost impossible to accomplish without side effects.

## 11.6 Exercises

- 11.1 Starting with the clauses at the beginning of Example 11.17, use resolution (as illustrated in Example 11.3) to show, in two different ways, that there is a path from *a* to *e*.
- 11.2 Solve Exercise 6.18 in Prolog.
- 11.3 Write a gcd definition in Prolog. Does your definition work “backward” as well as forward? (Given integers *d* and *n*, can you use it to generate a sequence of integers *m* such that  $\text{gcd}(n, m) = d$ ?)
- 11.4 In the spirit of Example 10.23, write a Prolog program that exploits backtracking to simulate the execution of a *non*deterministic finite automaton.
- 11.5 Show that resolution is commutative and associative. Specifically, if *A*, *B*, and *C* are Horn clauses, show that  $(A \oplus B) = (B \oplus A)$  and that  $((A \oplus B) \oplus C) = (A \oplus (B \oplus C))$ , where  $\oplus$  indicates resolution. Be sure to think about what happens to variables that are instantiated as a result of unification.
- 11.6 In Example 11.8, the query `?- classmates(jane_doe, X)` will succeed three times: twice with *X* = *jane\_doe* and once with *X* = *ajit\_chandra*. Show how to modify the `classmates(X, Y)` rule so that a student is not considered a classmate of him or herself.
- 11.7 Modify Example 11.17 so that the goal `path(X, Y)`, for arbitrary already-instantiated *X* and *Y*, will succeed no more than once, even if there are multiple paths from *X* to *Y*.

- 11.8 Using only `not` (no cuts), modify the tic-tac-toe example of Section 11.2.5 so it will generate only one candidate move from a given board position. How does your solution compare to the cut-based one (Example 11.22)?
- 11.9 Prove that the tic-tac-toe strategy of Example 11.19 is optimal (wins whenever possible, draws otherwise), or give a counterexample.
- 11.10 Starting with the tic-tac-toe program of Figure 11.4, draw a directed acyclic graph in which every clause is a node and an arc from A to B indicates that it is important, either for correctness or efficiency, that A come before B in the program. (Do not draw any other arcs.) Any topological sort of your graph should constitute an equally efficient version of the program. (Is the existing program one of them?)
- 11.11 Write Prolog rules to define a version of the `member` predicate that will generate all members of a list during backtracking, but without generating duplicates. Note that the `cut` and `not` based versions of Example 11.20 will not suffice; when asked to look for an uninstantiated member, they find only the head of the list.
- 11.12 Use the `clause` predicate of Prolog to implement the `call` predicate (pretend that it isn't built-in). You needn't implement all of the built-in predicates of Prolog; in particular, you may ignore the various imperative control-flow mechanisms and database manipulators. Extend your code by making the database an explicit argument to `call`, effectively producing a meta-circular interpreter.
- 11.13 Use the `clause` predicate of Prolog to write a predicate `call_bfs` that attempts to satisfy goals breadth-first. (*Hint:* You will want to keep a queue of yet-to-be-pursued subgoals, each of which is represented by a stack that captures backtracking alternatives.)
- 11.14 Write a (list-based) *insertion sort* algorithm in Prolog. Here's what it looks like in C, using arrays:

```
void insertion_sort(int A[], int N)
{
 int i, j, t;
 for (i = 1; i < N; i++) {
 t = A[i];
 for (j = i; j > 0; j--) {
 if (t >= A[j-1]) break;
 A[j] = A[j-1];
 }
 A[j] = t;
 }
}
```

- 11.15 Quicksort works well for large lists, but has higher overhead than insertion sort for short lists. Write a sort algorithm in Prolog that uses quicksort

initially, but switches to insertion sort (as defined in the previous exercise) for sublists of fifteen or fewer elements. (*Hint:* You can count the number of elements during the `partition` operation.)

- 11.16 Write a Prolog sorting routine that is guaranteed to take  $O(n \log n)$  time in the worst case. (*Hint:* Try *merge sort*. A description can be found in almost any algorithms or data structures text.)

11.17 Consider the following interaction with a Prolog interpreter.

What is going on here? Why does the interpreter fall into an infinite loop? Can you think of any circumstances (presumably not requiring output) in which a structure like this one would be useful? If not, can you suggest how a Prolog interpreter might implement checks to forbid its creation? How expensive would those checks be? Would the cost in your opinion be justified?

© 11.18–11.20 In More Depth.

## III.7 Explorations

- 11.21 Learn about alternative search strategies for Prolog and other logic languages. How do backward chaining solvers work? What are the prospects for intelligent hybrid strategies?
  - 11.22 Between 1982 and 1992 the Japanese government invested large sums of money in logic programming. Research the *Fifth Generation* project, administered by the Japanese Ministry of International Trade and Industry (MITI). What were its goals? What was achieved? What was not? How tightly were the goals and outcomes tied to Prolog? What lessons can we learn from the project today?
  - 11.23 Read ahead to Chapter 13 and learn about XSLT, a language used to manipulate data represented in XML, the extended markup language (of which XHTML, the latest standard for web pages, is an example). XSLT is generally described as declarative. Is it logic-based? How does it compare to Prolog in expressive power, level of abstraction, and execution efficiency?
  - 11.24 Repeat the previous question for SQL, the database query language (for an introduction, type “SQL tutorial” into your favorite Internet search engine).

- 11.25 Spreadsheets like Microsoft Excel and the older VisiCalc and Lotus 1-2-3 are sometimes characterized as declarative programming. Is this fair? Ignoring extensions like Visual Basic macros, does the ability to define relationships among cells provide Turing-equivalent computing power? Compare the execution model to that of Prolog. How is the order of update for cells determined? Can data be pushed “both ways” as they can in Prolog?

⌚ 11.26–11.29 In More Depth.

## 11.8 Bibliographic Notes

Logic programming has its roots in automated theorem proving. Much of the theoretical groundwork was laid by Horn in the early 1950s [Hor51] and by Robinson in the early 1960s [Rob65]. The breakthrough for computing came in the early 1970s, when Colmerauer and Roussel at the University of Aix–Marseille in France and Kowalski and his colleagues at the University of Edinburgh in Scotland developed the initial version of Prolog. The early history of the language is recounted by Robinson [Rob83]. Theoretical foundations are covered by Lloyd [Llo87].

Prolog was originally intended for research in natural language processing, but it soon became apparent that it could serve as a general purpose language. Several versions of Prolog have since evolved. The one described here is the widely used Edinburgh dialect. The ISO standard [Int95c] is similar.

Several other logic languages have been developed, though none has yet to rival Prolog in popularity. The most widely used is probably OPS5 [BFKM86]. The more recent Gödel [HL94] includes modules, strong typing, a richer variety of logical operators, and enhanced control of execution order. Database query languages stemming from Datalog [Ull85][UW97, Secs. 4.2–4.4] are implemented using forward chaining. CLP (Constraint Logic Programming) and its variants are largely based on Prolog, but employ a more general constraint-satisfaction mechanism in place of unification [JM94]. Extensive online resources for logic programming can be found at <http://vl.fmnnet.info/logic-prog/>. There is also a *comp.lang.prolog* newsgroup.



# 12 Concurrency

**The bulk of this text has focused**, implicitly, on *sequential* programs: programs with a single active execution context. As we saw in Chapter 6, sequentiality is fundamental to imperative programming. It also tends to be implicit in declarative programming, partly because practical functional and logic languages usually include some imperative features and partly because people tend to develop imperative implementations and mental models of declarative programs (applicative order reduction, backward chaining with backtracking), even when language semantics do not require such a model.

By contrast, a program is said to be *concurrent* if it contains more than one active execution context—more than one “thread of control.” Concurrency arises for at least three important reasons.

1. To capture the logical structure of a problem. Many programs, particularly servers and graphical applications, must keep track of more than one largely independent “task” at the same time. Often the simplest and most logical way to structure such a program is to represent each task with a separate thread of control. We touched on this “multithreaded” structure when discussing coroutines (Section 8.6); we will return to it in Section 12.1.2.
2. To cope with independent physical devices. Some software is by necessity concurrent. An operating system may be interrupted by a device at almost any time. It needs one context to represent what it was doing before the interrupt and another for the interrupt itself. Likewise a system for real-time control (e.g., of a factory, or even an automobile) is likely to include a large number of processors, each connected to a separate machine or device. Each processor has its own thread(s) of control, which must interact with the threads on other processors to accomplish the overall objectives of the system. Message-routing software for the Internet is in some sense a very large concurrent program, running on thousands of servers around the world.
3. To increase performance by running on more than one processor at once. Even when concurrency is not dictated by the structure of a program or the

hardware on which it has to run, we can often increase performance by choosing to have more than one processor work on the problem simultaneously. With many processors, the resulting parallel speedup can be very large.

Section 12.1 contains a brief overview of the history of concurrent programming. It highlights major advances in parallel hardware and applications, makes the case for multithreaded programs (even on uniprocessors), and surveys the architectural features of modern multiprocessors. In Section 12.2 we survey the many ways in which parallelism may be expressed in an application. We introduce the message-passing and shared-memory approaches to communication and synchronization, and note that they can be implemented either in an explicitly concurrent programming language or in a library package intended for use with a conventional sequential language. Building on coroutines, we explain how a language or library can create and schedule threads. In the two remaining sections (12.3 and 12.4) we look at shared memory and message passing in detail. Most of the shared-memory section is devoted to synchronization.

## 12.1 Background and Motivation

Concurrency is not a new idea. Much of the theoretical groundwork for concurrent programming was laid in the 1960s, and Algol 68 includes concurrent programming features. Widespread interest in concurrency is a relatively recent phenomenon, however; it stems in part from the availability of low-cost multiprocessors and in part from the proliferation of graphical, multimedia, and web-based applications, all of which are naturally represented by concurrent threads of control.

Concurrency is an issue at many levels of a typical computer system. At the digital logic level, almost everything happens in parallel: signals propagate down thousands of connections at once. At the next level up, the pipelining and superscalar features of modern processors are designed to exploit the *instruction-level* parallelism available in well-scheduled programs. In this chapter we will focus on medium to large scale concurrency, represented by constructs that are semantically visible to the programmer, and that can be exploited by machines with many processors. In Sections 12.1.3 and 12.3.6 we will also mention an intermediate level of parallelism available on special purpose *vector* processors.

### 12.1.1 A Little History

The very first computers were single-user machines, used in *stand-alone* mode: people signed up for blocks of time, during which they enjoyed exclusive use of the hardware. Unfortunately, while single-user machines make good economic sense today, they constituted a terrible waste of resources in the late 1940s, when the cheapest computer cost millions of dollars. Rather than allow a machine to

sit idle while the user examined output or pondered the source of a bug, computer centers quickly switched to a mode of operation in which users created *jobs* (sequences of programs and their input) offline (e.g., on a keypunch machine) and then submitted them to an operator for execution. The operator would keep a *batch* of jobs constantly queued up for input on punch cards or magnetic tape. As its final operation, each program would transfer control back to a *resident monitor* program—a form of primitive operating system—which would immediately read the next program into memory for execution, from the current job or the next one, without operator intervention.

Unfortunately, this simple form of batch processing still left the processor idle much of the time, particularly on commercial applications, which tended to read a large number of data records from cards or tape, with comparatively little computation per record. To perform an I/O operation (to write results to a printer or magnetic tape, or to read a new program or input data into memory), the processor in a simple batch system would send a command to the I/O device and then *busy-wait* for completion, repeatedly testing a variable that the device would modify when done with its operation. Given a punch card device capable of reading four cards per second, a 40-kHz vacuum-tube computer would waste 10,000 instructions *per card* while waiting for input. If it performed fewer than 10,000 instructions of computation on average before reading another card, the processor would be idle more than half the time! To make use of the cycles lost to busy-waiting, researchers developed techniques to *overlap* I/O and computation. In particular, they developed *interrupt-driven I/O*, which eliminates the need to busy-wait, and *multiprogramming*, which allows more than one application program to reside in memory at once. Both of these innovations required new hardware support: the former to implement interrupts, the latter to implement memory protection, so that errors in one program could not corrupt the memory of another.

### ***Multiprogramming and Interrupt-Driven I/O***

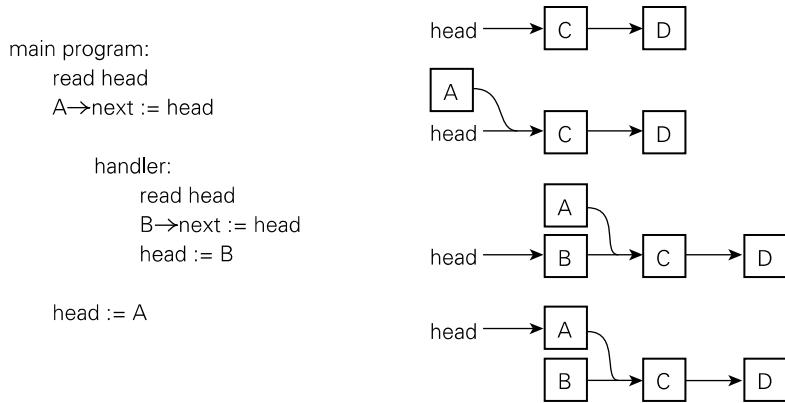
On a multiprogrammed batch system, the operating system keeps track of which programs are waiting for I/O to complete and which are currently *runnable*. To read or write a record, the currently running program transfers control to the operating system. The OS sends a command to the device to start the requested operation, and then transfers control immediately to a different program (assuming one is runnable). When the device completes its operation, it generates an interrupt, which causes the processor to transfer back into the operating system. The OS notes that the earlier program is runnable again. It then chooses a program from among those that are runnable and transfers back to it. The only time the processor is idle is when *all* of the programs that have been loaded into memory are waiting for I/O.

Interrupt-driven I/O introduced concurrency within the operating system. Because an interrupt can happen at an arbitrary time, including when control is already in the operating system, the interrupt handlers and the main bulk of the OS function as concurrent threads of control. If an interrupt occurs while the

---

#### **EXAMPLE 12.1**

A race condition in the operating system



**Figure 12.1 Example of a race condition.** Here the currently running program attempts to insert a new element into the beginning of a list. In the middle of this operation, an interrupt occurs and the interrupt handler attempts to insert a different element into the list. In the absence of synchronization, one of the elements may be lost (unreachable from the head pointer).

OS is modifying a data structure (e.g., the list of runnable programs) that may also be used by a handler, then it is possible for the handler to see that data structure in an inconsistent state (see Figure 12.1). This problem is an example of a *race condition*: the thread that corresponds to the main body of the OS and the thread that corresponds to the device are “racing” toward points in the code at which they touch some common object, and the behavior of the system depends on which thread gets there first. To ensure correct behavior, we must *synchronize* the actions of the threads: take explicit steps to control the order in which their actions occur. We discuss synchronization further in Section 12.3. It should be noted that not all race conditions are bad: sometimes any of the possible program outcomes are acceptable. The goal of synchronization is to resolve “bad” race conditions: those that might otherwise cause the program to produce incorrect results. ■

### Timesharing and Distribution

With increases in the size of physical memory, and with the development of virtual memory, it became possible to build systems with an almost arbitrary number of simultaneously loaded programs. Instead of submitting jobs offline, users could now sit at a terminal and interact with the computer directly. To provide interactive response to keystrokes, however, the OS needed to implement *preemption*. Whereas a batch system switches from one program to another only when the first one blocks for I/O, a preemptive, *timesharing* system switches several times per second as a matter of course. These *context switches* prevent a compute-bound program from hogging the machine for seconds or minutes at a time, denying access to users at keyboards.

By the early 1970s, timesharing systems were relatively common. When augmented with mechanisms to allow data sharing or other forms of communication among currently runnable programs, they introduced concurrency in user-level applications. Shortly thereafter, the emergence of computer networks introduced true parallelism in the form of *distributed* systems: programs running on physically separate machines and communicating with messages.

Most distributed systems reflect our second rationale for concurrency: they have to be concurrent in order to cope with multiple devices. A few reflect the third rationale: they are distributed in order to exploit the speedup available from multiple processors. Parallel speedup is more commonly pursued on single-chassis multiprocessors, with internal networks designed for very high bandwidth communication. Though multiprocessors have been around since the 1960s, they did not become commonplace until the 1980s. Around the turn of the century they began to appear in consumer-grade desktop machines. Given the challenge of cooling ever more complex uniprocessors, it seems likely that within the next few years most desktop machines will employ multiple simpler processors on a single chip.

### 12.1.2 The Case for Multithreaded Programs

Our first rationale for concurrency—to capture the logical structure of certain applications—has arisen several times in earlier chapters. In Section 7.9.1 we noted that interactive I/O must often interrupt the execution of the current program. In a video game, for example, we must handle keystrokes and mouse or joystick motions while continually updating the image on the screen. By far the most convenient way to structure such a program is to represent the input handlers as concurrent threads of control, which coexist with one or more threads responsible for updating the screen. In Section 8.6, we considered a screen saver program that used coroutines to interleave “sanity checks” on the file system with updates to a moving picture on the screen. We also considered discrete-event simulation, which uses coroutines to represent the active entities of some real-world system.

The semantics of discrete-event simulation require that events occur atomically at fixed points in time. Coroutines provide a natural implementation because they execute one at a time. In our other examples, however—and indeed in most “naturally concurrent” programs—there is no need for coroutine semantics. By assigning concurrent tasks to threads instead of to coroutines, we acknowledge that those tasks can proceed in parallel if more than one processor is available. We also move responsibility for figuring out which thread should run when from the programmer to the language implementation.

---

**EXAMPLE 12.2**

Multithreaded web browser

The need for multithreaded programs has become particularly apparent in recent years with the development of web-based applications. In a browser such as Firefox or Internet Explorer (see Figure 12.2), there are typically many different threads simultaneously active, each of which is likely to communicate with a

```

procedure parse_page(address : url)
 contact server, request page contents
 parse_html_header
 while current_token in {"<p>", "<h1>", "", ...,
 "<background>", "<image>", "<table>", "<frameset>, ..."}
 case current_token of
 "<p>" : break_paragraph
 "<h1>" : format_heading; match("</h1>")
 "" : format_list; match("")
 ...
 "<background>":
 a : attributes := parse_attributes
 fork render_background(a)
 "<image>": a : attributes := parse_attributes
 fork render_image(a)
 "<table>": a : attributes := parse_attributes
 scan forward for "</table>" token
 token_stream s :=... -- table contents
 fork format_table(s, a)
 "<frameset>":
 a : attributes := parse_attributes
 parse_frame_list(a)
 match("</frameset>")
 ...
 ...

procedure parse_frame_list(a1 : attributes)
 while current_token in {"<frame>", "<frameset>", "<noframes>"}
 case current_token of
 "<frame>": a2 : attributes := parse_attributes
 fork format_frame(a1, a2)
 ...

```

**Figure 12.2 Thread-based code from a hypothetical Web browser.** To first approximation, the `parse_page` subroutine is the root of a recursive-descent parser for HTML. In several cases, however, the actions associated with recognition of a construct (background, image, table, frameset) proceed concurrently with continued parsing of the page itself. In this example, concurrent threads are created with the `fork` operation. Other threads would be created automatically in response to keyboard and mouse events.

remote (and possibly very slow) server several times before completing its task. When the user clicks on a link, the browser creates a thread to request the specified document. For all but the tiniest pages, this thread will then receive a long series of message “packets.” As these packets begin to arrive the thread must format them for presentation on the screen. The formatting task is akin to typesetting: the thread must access fonts, assemble words, and break the words into lines. For many special tags within the page, the formatting thread will spawn additional threads: one for each image, one for the background if any, one to format each

table, and possibly more to handle separate frames. Each spawned thread will communicate with the server to obtain the information it needs (e.g., the contents of an image) for its particular task. The user, meanwhile, can access items in menus to create new browser windows, edit bookmarks, change preferences, and so on, all in “parallel” with the rendering of page elements. ■

The use of many threads ensures that comparatively fast operations (e.g., display of text) do not wait for slow operations (e.g., display of large images). Whenever one thread *blocks* (waits for a message or I/O), the implementation automatically switches to a different thread. In a *preemptive* thread package, the implementation switches among threads at other times as well, to make sure that none of them hogs the CPU. Any reader who remembers the early, more sequential browsers will appreciate the difference that multithreading makes in perceived performance and responsiveness.

### ***The Dispatch Loop Alternative***

---

**EXAMPLE 12.3**

Dispatch loop web browser

Without language or library support for threads, a browser must either adopt a more sequential structure, or centralize the handling of all delay-inducing events in a single *dispatch loop* (see Figure 12.3). Data structures associated with the dispatch loop keep track of all the *tasks* the browser has yet to complete. The state of a task may be quite complicated. For the high-level task of rendering a page, the state must indicate which packets have been received and which are still outstanding. It must also identify the various subtasks of the page (images, tables, frames, etc.) so that we can find them all and reclaim their state if the user clicks on a “stop” button.

To guarantee good interactive response, we must make sure that no subaction of `continue_task` takes very long to execute. Clearly we must end the current action whenever we wait for a message. We must also end it whenever we read from a file, since disk operations are slow. Finally, if any task needs to compute for longer than about a tenth of a second (the typical human perceptual threshold), then we must divide the task into pieces, between which we save state and return to the top of the loop. These considerations imply that the condition at the top of the loop must cover the full range of asynchronous events, and that evaluations of the condition must be interleaved with continued execution of any tasks that were subdivided due to lengthy computation. (In practice we would probably need a more sophisticated mechanism than simple interleaving to ensure that neither input-driven nor compute-bound tasks hog more than their share of resources.) ■

The principal problem with a dispatch loop—beyond the complexity of subdividing tasks and saving state—is that it hides the algorithmic structure of the program. Every distinct task (retrieving a page, rendering an image, walking through nested menus) could be described elegantly with standard control-flow mechanisms, if not for the fact that we must return to the top of the dispatch loop at every delay-inducing operation. In effect, the dispatch loop turns the program “inside out,” making the management of tasks explicit and the control flow

```

type task_descriptor = record
 -- fields in lieu of thread-local variables, plus control-flow information
 ...
 ready_tasks : queue of task_descriptor
 ...
procedure dispatch
loop
 -- try to do something input-driven
 if a new event E (message, keystroke, etc.) is available
 if an existing task T is waiting for E
 continue_task(T, E)
 else if E can be handled quickly, do so
 else
 allocate and initialize new task T
 continue_task(T, E)
 -- now do something compute bound
 if ready_tasks is nonempty
 continue_task(dequeue(ready_tasks), 'ok')

procedure continue_task(T : task, E : event)
if T is rendering an image
 and E is a message containing the next block of data
 continue_image_render(T, E)
else if T is formatting a page
 and E is a message containing the next block of data
 continue_page_parse(T, E)
else if T is formatting a page
 and E is 'ok' -- we're compute bound
 continue_page_parse(T, E)
else if T is reading the bookmarks file
 and E is an I/O completion event
 continue_goto_page(T, E)
else if T is formatting a frame
 and E is a push of the "stop" button
 deallocate T and all tasks dependent upon it
else if E is the "edit preferences" menu item
 edit_preferences(T, E)
else if T is already editing preferences
 and E is a newly typed keystroke
 edit_preferences(T, E)
 ...

```

**Figure 12.3 Dispatch loop from a hypothetical non-thread-based Web browser.** The clauses in `continue_task` must cover all possible combinations of task state and triggering event. The code in each clause performs the next coherent unit of work for its task, returning when (1) it must wait for an event, (2) it has consumed a significant amount of compute time, or (3) the task is complete. Prior to returning, respectively, code (1) places the task in a dictionary (used by `dispatch`) that maps awaited events to the tasks that are waiting for them, (2) enqueues the task in `ready_tasks`, or (3) deallocates the task.

within tasks implicit. The resulting complexity is similar to what we encountered when trying to enumerate a recursive set with iterator objects in Section 6.5.3, only worse. Like true iterators, a thread package turns the program “right-side out,” making the management of tasks (threads) implicit and the control flow within threads explicit.

#### **Personal Computers**

With the development of personal computers, much of the history of operating systems has repeated itself. Early PCs performed busy-wait I/O and ran one application at a time. With the development of Microsoft Windows and the Multifinder version of the MacOS, PC vendors added the ability to hold more than one program in memory at once, and to switch between them on I/O. Because a PC is a single-user machine, however, the need for preemption was not felt as keenly as in multiuser systems. For a long time it was considered acceptable for the currently running program to hog the processor: after all, that program is what the (single) user wants to run. As PCs became more sophisticated, however, users began to demand concurrent execution of threads such as those in a browser, as well as “background” threads that update windows, check for e-mail, babysit slow printers, and so on. To some extent background computation can be accommodated by requiring every program to “voluntarily” yield control of the processor at well-defined “clean points” in the computation. This sort of “cooperative multiprogramming” was found in Windows 3.1 and MacOS version 7. Unfortunately, some programs do not yield as often as they should, and the inconsistent response of cooperatively multiprogrammed systems grew increasingly annoying to users. Windows 95 added preemption for 32-bit applications. Windows NT and MacOS X add preemption for all programs, running them in separate address spaces so bugs in one program don’t damage another or cause the machine to crash.

#### **12.1.3 Multiprocessor Architecture**

Single-site (non-distributed) parallel computers can be grouped into two broad categories: those in which processors share access to common memory and those in which they must communicate with messages. Shared-memory machines are typically referred to as *multiprocessors*, though occasionally one hears that term applied to message-based machines as well. A multiprocessor typically occupies a single cabinet, in which the processors share not only memory, but also disks, power supplies, and a single copy of the operating system. Recent years have also seen a proliferation of computer *clusters*, in which uniprocessors or small multiprocessors, each physically capable of independent operation, are packed into a shared set of racks, connected by a high-speed system-area network, and administered as a single entity. Large-scale online services like Google, Amazon, or eBay are typically backed by clusters with hundreds or even thousands of processors. One will sometimes hear the term *multicomputer* applied to a single-chassis

message-based machine. Multicomputers were popular for high-end scientific and database applications of the 1980s and 1990s, but have for the most part been displaced in recent years by clusters and large multiprocessors.

Small shared-memory multiprocessors are usually *symmetric* in the sense that all memory is equally distant from all processors. Large multiprocessors usually display a *distributed memory* architecture, in which each memory bank is physically adjacent to a particular processor or small group of processors. Any processor can access the memory of any other, but local memory is faster. The small machines are sometimes called SMPs, for “symmetric multiprocessor.” Their large cousins are sometimes called NUMA machines, for “nonuniform memory access.”

Since the late 1960s, the market for high-end supercomputers has been dominated by so-called *vector processors*, which provide special instructions capable of applying the same operation to every element of an array. Vector instructions are very easy to pipeline. They are useful in many scientific programs, particularly those in which the programmer has explicitly annotated loops whose iterations can execute concurrently (we will discuss such loops in Sections 12.2.3 [Example 12.9] and 12.3.6). Traditional vector processors, however, are special purpose, multichip designs, and the inexorable advance of the general purpose microprocessor has steadily eroded their market share. At the same time, ideas from vector processors have made their way into the microprocessor world—for example, in the form of the MMX extensions to the Pentium instruction set.

From the point of view of a language or library implementor, the principal distinction between a message-based cluster and a shared-memory multiprocessor is that communication on the former requires the active participation of processors on both ends of the connection: one to send, the other to receive. On a shared-memory machine, a processor can read and write remote memory without the assistance of a remote processor. In most cases remote reads and writes use the same interface (i.e., load and store instructions) as local reads and writes.

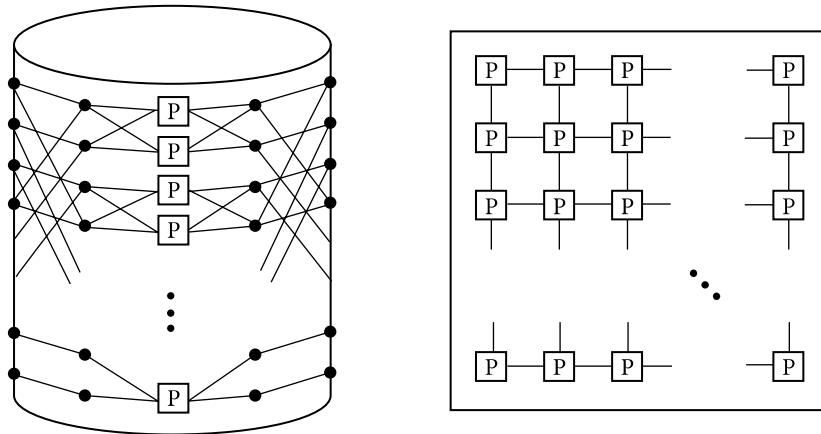
### Interconnection Networks

No matter what the communication model, every parallel computer requires some sort of interconnection network to tie its processors and memories together. Most small, symmetric machines are connected by a bus. A few are connected by a *crossbar* switch, in which every processor has a direct connection to every memory bank, forming a complete bipartite graph. Larger machines can be grouped into two camps: those with *indirect* and *direct* networks. An indirect network resembles a fishing net stretched around the outside of a cylinder (see Figure 12.4). The “knots” in the net are message-routing switches. A direct network has no internal switches: all connections run directly from one node to another. Both indirect and direct networks have many topological variants. Indirect networks are generally designed so that the distance from any node to any other is  $O(\log P)$ , where  $P$  is the total number of nodes. The distance between nodes in a direct network may be as large as  $O(\sqrt{P})$ . In practice, a hardware technique

---

**EXAMPLE 12.4**

Direct and indirect networks



**Figure 12.4 Multiprocessor network topology.** In an *indirect* network (left), processing nodes are equally distant from one another. They communicate through a log-depth switching network. In a *direct* network (right), there are no switching nodes: each processing node sends messages through a small fixed number of neighbors.

known as *wormhole routing* makes communication with distant nodes almost as fast as with neighbors. ■

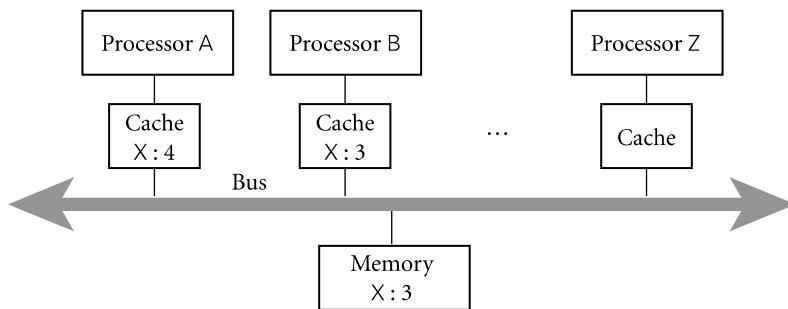
### Memory Coherence

In any machine built from modern microprocessors, performance depends critically on very fast (low latency) access to memory. To minimize delays, almost all machines depend on caches. On a message-passing machine, each processor caches its own memory. On a shared-memory machine, however, caches introduce a serious problem: unless we do something special, a processor that has cached a particular memory location will not see changes that are made to that location by other processors. This problem—how to keep cached copies of a memory location consistent with one another—is known as the *coherence* problem (see Figure 12.5). On bus-based symmetric machines the problem is relatively easy to solve: the broadcast nature of the communication medium allows cache controllers to eavesdrop (*snoop*) on the memory traffic of other processors. When another processor writes a location that is contained in the local cache, the controller can either grab the new value off the bus or, more commonly, *invalidate* the affected cache line, forcing the processor to go back to memory (or to some other processor's cache) the next time the line is needed. Bus-based cache coherence algorithms are now a standard, built-in part of most commercial microprocessors. On large machines, the lack of a broadcast bus makes cache coherence a significantly more difficult problem; commercial implementations are available, but the subject remains an active topic of research. ■

As of 2005, small bus-based SMPs are available from dozens of manufacturers, with x86, PowerPC, Sparc, AMD64 (Opteron), and IA-64 (Itanium) processors.

#### EXAMPLE 12.5

The cache coherence problem



**Figure 12.5** The cache coherence problem for shared-memory multiprocessors. Here processors A and B have both read variable X from memory. As a side effect, a copy of X has been created in the cache of each processor. If A now changes X to 4 and B reads X again, how do we ensure that the result is a 4 and not the still-cached 3? Similarly, if Z reads X into its cache, how do we ensure that it obtains the 4 from A's cache instead of the stale 3 from main memory?

Several manufacturers have recently released, or are currently developing, single-chip multiprocessors. Larger, cache-coherent shared-memory multiprocessors are available from several manufacturers, including Sun, HP, IBM, and SGI. All of these machines copy remote data to the local cache when accessed. The Cray X1, by contrast, has a shared, coherent address space, but remote locations are never cached. The field is very much in flux: several large parallel machines and manufacturers have disappeared from the market in recent years; several new machines are scheduled to appear in the near future.

### ✓ CHECK YOUR UNDERSTANDING

1. Explain the rationale for concurrency: why do people write concurrent programs? What accounts for the increased interest in concurrency in recent years?
2. Describe the evolution of computer operation from *stand-alone* mode to *batch processing*, to *multiprogramming* and *timesharing*.
3. What is *interrupt-driven I/O*? What does it have to do with concurrency?
4. What is a *race condition*?
5. What is a *context switch*? What is *preemption*?
6. What is a *dispatch loop*? What are its advantages and disadvantages?
7. Explain the distinction between a *multiprocessor* and a *cluster*.
8. Explain the *coherence problem* in the context of multiprocessor caches.
9. What is symmetric about a *symmetric multiprocessor (SMP)*?

## 12.2

## Concurrent Programming Fundamentals

We will use the word *concurrency* to characterize any program in which two or more execution contexts may be active at the same time. Under this definition, coroutines are not concurrent, because only one of them can be active at once. We will use the term *parallelism* to characterize concurrent programs in which execution is actually happening in more than one context at once. True parallelism thus requires parallel hardware. From a semantic point of view, there is no difference between true parallelism and the “quasiparallelism” of a preemptive concurrent system, which switches between execution contexts at unpredictable times: the same programming techniques apply in both situations.

Within a concurrent program, we will refer to an execution context as a *thread*. The threads of a given program are implemented on top of one or more *processes* provided by the operating system. OS designers often distinguish between a *heavyweight* process, which has its own address space, and a collection of *lightweight* processes, which may share an address space. Lightweight processes were added to most variants of Unix in the late 1980s and early 1990s, to accommodate the proliferation of shared-memory multiprocessors. Without lightweight processes, the threads of a concurrent program must run on top of more than one heavyweight process, and the language implementation must ensure that any data that is to be shared among threads is mapped into the address space of all the processes.

We will sometimes use the word *task* to refer to a well-defined unit of work that must be performed by some thread. In one common programming idiom, a collection of threads shares a common “bag of tasks”: a list of work to be done. Each thread repeatedly removes a task from the bag, performs it, and goes back for another. Sometimes the work of a task entails adding new tasks to the bag.

Unfortunately, the vocabulary of concurrent programming is not consistent across languages or authors. Several languages call their threads processes. Ada calls them tasks. Several operating systems call lightweight processes threads. The Mach OS, from which OSF Unix and MacOS X are derived, calls the address space shared by lightweight processes a task. A few systems try to avoid ambiguity by coining new words, such as “actors” or “filaments.” We will attempt to use the definitions of the preceding two paragraphs consistently, and to identify cases in which the terminology of particular languages or systems differs from this usage.

### 12.2.1 Communication and Synchronization

In any concurrent programming model, two of the most crucial issues to be addressed are *communication* and *synchronization*. Communication refers to any mechanism that allows one thread to obtain information produced by another. Communication mechanisms for imperative programs are generally based on either *shared memory* or *message passing*. In a shared-memory programming

model, some or all of a program’s variables are accessible to multiple threads. For a pair of threads to communicate, one of them writes a value to a variable and the other simply reads it. In a message-passing programming model, threads have no common state. For a pair of threads to communicate, one of them must perform an explicit *send* operation to transmit data to another.

Synchronization refers to any mechanism that allows the programmer to control the relative order in which operations occur in different threads. Synchronization is generally implicit in message-passing models: a message must be sent before it can be received. If a thread attempts to receive a message that has not yet been sent, it will wait for the sender to catch up. Synchronization is generally not implicit in shared-memory models: unless we do something special, a “receiving” thread could read the “old” value of a variable, before it has been written by the “sender.” In both shared-memory and message-based programs, synchronization can be implemented either by *spinning* (also called *busy-waiting*) or by *blocking*. In busy-wait synchronization, a thread runs a loop in which it keeps reevaluating some condition until that condition becomes true (e.g., until a message queue becomes nonempty or a shared variable attains a particular value)—presumably as a result of action in some other thread, running on some other processor. Note that busy-waiting makes no sense for synchronizing threads on a uniprocessor: we cannot expect a condition to become true while we are monopolizing a resource (the processor) required to make it true. (A thread on a uniprocessor may sometimes busy-wait for the completion of I/O, but that’s a different situation: the I/O device runs in parallel with the processor.)

In blocking synchronization (also called *scheduler-based* synchronization), the waiting thread voluntarily relinquishes its processor to some other thread. Before doing so, it leaves a note in some data structure associated with the synchronization condition. A thread that makes the condition true at some point in the future will find the note and take action to make the blocked thread run again. We will consider synchronization again briefly in Section 12.2.4, and then more thoroughly in Section 12.3.

## DESIGN & IMPLEMENTATION

### Hardware and software communication

As noted in Section 12.1.3, the distinction between shared memory and message passing applies not only to languages and libraries but also to computer hardware. It is important to note that the model of communication and synchronization provided by the language or library need not necessarily agree with that of the underlying hardware. It is easy to implement message passing on top of shared-memory hardware. With a little more effort, one can also implement shared memory on top of message-passing hardware. Systems in this latter camp are sometimes referred to as *software distributed shared memory* (S-DSM).

### 12.2.2 Languages and Libraries

Concurrency can be provided to the programmer in the form of explicitly concurrent languages, compiler-supported extensions to traditional sequential languages, or library packages outside the language proper. The latter two alternatives have historically been the most common: most parallel programs currently in use are either annotated Fortran for vector machines or C/C++ code with library calls. With the proliferation of Java and C# this situation is beginning to change, at least at the “low end.” It is likely to be some time, however, before explicitly parallel languages displace Fortran, C, and C++ for high-performance parallel applications.

Most SMP vendors provide a parallel programming library based on shared memory and threads. In the Unix world this library typically implements the POSIX *pthreads* standard [Ope96]. (Microsoft provides similar functionality for Windows.) For message-based computing, existing libraries can be grouped into two main categories: those that are intended primarily for communication among the processes of a single program and those that are intended primarily for communication across program boundaries. Packages in this latter camp usually implement one of the standard Internet protocols [PD03, Chap. 6] and bear a strong resemblance to file-based I/O (Section 7.9).

The two most popular packages for message passing within a parallel program are PVM [Sun90, GBD<sup>+</sup>94] and MPI [BDH<sup>+</sup>95, SOHL<sup>+</sup>98]. The two packages provide similar functionality in most respects. PVM is richer in the area of creating and managing processes on a heterogeneous distributed network, in which machines of different types may join and leave the computation during execution. MPI provides more control over how communication is implemented (to map it onto the primitives of particular high-performance multicomputers) and a richer set of communication primitives, especially for so-called *collective communication*: one-to-all, all-to-one, or all-to-all patterns of messages among a set of threads. Implementations of PVM and MPI are available for C, C++, and Fortran.

For communication based on requests from clients to servers, *remote procedure calls* (RPCs) provide an attractive interface to message passing. Rather than talk to a server directly, an RPC client calls a local *stub* procedure, which packages its parameters into a message, sends them to a server, and waits for a response, which it returns to the client in the form of result parameters. Several vendors provide tools that will generate stubs automatically from a formal description of the server interface. In the Unix world, Sun’s RPC [Sri95] is the de facto standard. Several generalizations of RPC, most of them based on binary *components* (page 518), are currently competing for prominence for Internet-based computing. RPC in object-oriented systems is sometimes referred to as *remote method invocation* (RMI).

In comparison to library packages, an explicitly concurrent programming language has the advantage of compiler support. It can make use of syntax other than subroutine calls, and can integrate communication and thread manage-

ment more tightly with such concepts as type checking, scoping, and exceptions. At the same time, since most programs are sequential, it is difficult for a concurrent language to gain widespread acceptance, particularly if the concurrent features make the sequential case more difficult to understand. As noted in Section 12.1, Algol 68 included concurrent features, though they were never widely used. Concurrency also appears in more recent “mainstream” languages, including Ada, Modula-3, Java, and C#. A little farther afield, but still commercially important, the Occam programming language, based on Hoare’s Communicating Sequential Processes (CSP) notation, has an active user community. Occam was the language of choice for systems built from the INMOS transputer processor, widely used in Europe in the 1980s and 90s. Andrews’s SR has been influential as a teaching language.

In the scientific community, expertise with vectorizing compilers has made its way into *parallelizing* compilers for multicomputers and multiprocessors, again exploiting annotations provided by the programmer. Several of the groups involved with this transition came together in the early 1990s to develop High Performance Fortran (HPF) [KLS<sup>+</sup>94], a *data-parallel* dialect of Fortran 90. (A data-parallel program is one in which the principal source of parallelism is the application of common operations to the members of a very large data set. A *task-parallel* program is one in which much of the parallelism stems from performing *different* operations concurrently. A data-parallel language is one whose features are designed for data-parallel programs.)

### 12.2.3 Thread Creation Syntax

One could imagine a concurrent programming system in which a fixed collection of threads was created by the language implementation, but such a static form of concurrency is generally too restrictive. Most concurrent systems allow the programmer to create new threads at run time. Syntactic and semantic details vary considerably from one language or library to another. There are at least six common options: (1) *co-begin*, (2) parallel loops, (3) launch-at-elaboration, (4) *fork* (with optional *join*), (5) implicit receipt, and (6) early reply. The first two options delimit threads with special control-flow constructs. The others declare threads with syntax resembling (or identical to) subroutines.

The SR programming language provides all six options. Algol 68 and Occam use *co-begin*. Occam also uses parallel loops, as does HPF. Ada uses both launch-at-elaboration and *fork*. Modula-3, Java, and C# use *fork/join*. Implicit receipt is the usual mechanism in RPC systems. The coroutine *detach* operation of Simula can be considered a form of early reply.

#### **Co-Begin**

##### **EXAMPLE 12.6**

Par begin in Algol 68

In Algol 68 the behavior of a *begin...end* block depends on whether the internal expressions are separated by semicolons or commas. In the former case, we have the usual sequential semantics. In the latter case, we have either nondetermin-

istic or concurrent semantics, depending on whether `begin` is preceded by the keyword `par`. The block

```
begin
 a := 3,
 b := 4
end
```

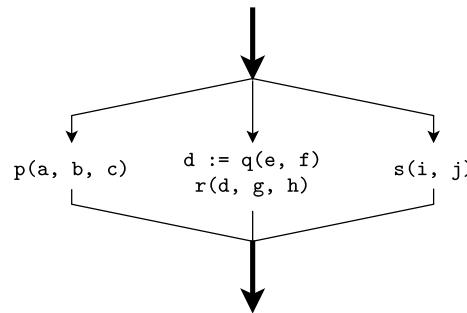
indicates that the assignments to `a` and `b` can occur in either order. The block

```
par begin
 a := 3,
 b := 4
end
```

indicates that they can occur in parallel. Of course, parallel execution makes little sense for such trivial operations as assignments; the `par begin` construct is usually used for more interesting operations:

```
par begin # concurrent #
 p(a, b, c),
 begin # sequential #
 d := q(e, f);
 r(d, g, h)
 end,
 s(i, j)
end
```

Here the executions of `p` and `s` can proceed in parallel with the sequential execution of the nested block (with the calls to `q` and `r`):



#### EXAMPLE 12.7

Par in Occam

Several other concurrent languages provide a variant of `par begin`. In Occam, which uses indentation to delimit nested control constructs, one would write

```
par
 p(a, b, c)
 seq
 d := q(e, f)
 r(d, g, h)
 s(i, j)
```

In general, a control construct whose constituent statements are meant to be executed concurrently is known as *co-begin*. ■

### **Parallel Loops**

#### **EXAMPLE 12.8**

Parallel loops in SR

```
co (i := 5 to 10) ->
 p(a, b, i) # six instances of p, each with a different i
oc
```

In Occam:

```
par i = 5 for 6
 p(a, b, i) # six instances of p, each with a different i
```

In SR it is the programmer's responsibility to make sure that concurrent execution is safe, in the sense that correctness will never depend on the outcome of race conditions. In the above example, access to global variables in the various instances of *p* would generally need to be synchronized, to make sure that those instances do not conflict with one another. In Occam, language rules prohibit conflicting accesses. The compiler checks to make sure that a variable that is written by one thread is neither read nor written by any concurrently active thread. In the code above, the Occam compiler would insist that all three parameters to *p* be passed by value (not result). Concurrently active threads in Occam communicate solely by sending messages. ■

Several parallel dialects of Fortran have provided parallel loops, with varying semantics. The *forall* loop adopted by HPF was subsequently incorporated into the 1995 revision of Fortran 90. Like the parallel loops of SR and Occam, it indicates that iterations can proceed in parallel. To resolve race conditions, however, it imposes automatic, internal synchronization on the constituent statements of the loop, each of which must be an assignment statement or a nested *forall* loop. Specifically, all reads of variables in a given assignment statement, in all iterations, must occur before any write to the left-hand side, in any iteration. The

### **DESIGN & IMPLEMENTATION**

#### **Stack frames for nested threads**

In an Algol 68 or Occam implementation, threads created by *co-begin* must share access to a common stack frame. To avoid this implementation complication, SR provides a variant of *co-begin* (delimited by *co...oc*) in which the constituent statements must all be procedure invocations, each of which begins execution in its own stack frame. In fact every new thread in SR is created in a separate subroutine, and subroutines do not nest. SR therefore has no need for the cactus stacks of Section 8.6.1.

**EXAMPLE 12.9**

Forall in Fortran 90

writes of the left-hand side in turn must occur before any reads in the following assignment statement. In the following example, the first assignment in the loop will read  $n - 1$  elements of B and  $n - 1$  elements of C, and then update  $n - 1$  elements of A. Subsequently, the second assignment statement will read all  $n$  elements of A and then update  $n - 1$  of them.

```
forall (i=1:n-1)
 A(i) = B(i) + C(i)
 A(i+1) = A(i) + A(i+1)
end forall
```

Note in particular that all of the updates to  $A(i)$  in the first assignment statement occur before any of the reads in the second assignment statement. Moreover in the second assignment statement the update to  $A(i+1)$  is *not* seen by the read of  $A(i)$  in the “subsequent” iteration: the iterations occur in parallel and each reads the variables on its right-hand side before updating its left-hand side. ■

For loops that “iterate” over the elements of an array, the `forall` semantics are ideally suited for execution on a vector machine. With a little extra effort, they can also be adapted to a more conventional multiprocessor. In HPF, an extensive set of *data distribution* and *alignment* directives allows the programmer to scatter the elements of an array across the memory associated with a large number of processors. Within a `forall` loop, the computation in a given assignment statement is usually performed by the processor that “owns” the element on the assignment’s left-hand side. In many cases an HPF or Fortran 95 compiler can prove that there are no dependences among certain (portions of) constituent statements of a `forall` loop, and can allow them to proceed without actually implementing synchronization.

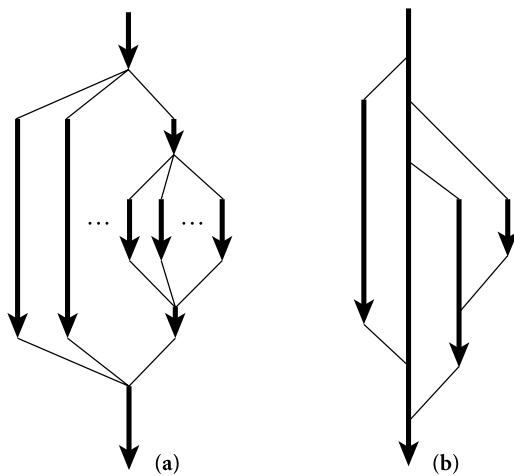
***Launch-at-Elaboration*****EXAMPLE 12.10**

Elaborated tasks in Ada

In Ada and SR (and in many other languages), the code for a thread may be declared with syntax resembling that of a subroutine with no parameters. When the declaration is elaborated, a thread is created to execute the code. In Ada (which calls its threads tasks) we may write the following.

```
procedure P is
 task T is
 ...
 end T;
begin -- P
 ...
end P;
```

Task T has its own `begin...end` block, which it begins to execute as soon as control enters procedure P. If P is recursive, there may be many instances of T at the same time, all of which execute concurrently with each other and with whatever task is executing (the current instance of) P. The main program behaves like an initial default task.



**Figure 12.6** Lifetime of concurrent threads. With `co-begin`, parallel loops, or launch-at-elaboration (a), threads are always properly nested. With `fork/join` (b), more general patterns are possible.

When control reaches the end of procedure P, it will wait for the appropriate instance of T (the one that was created at the beginning of this instance of P) to complete before returning. This rule ensures that the local variables of P (which are visible to T under the usual static scope rules) are never deallocated before T is done with them. ■

A launch-at-elaboration thread in SR is called a process.

### Fork/Join

#### EXAMPLE 12.11

Co-begin v. fork/join

Co-begin, parallel loops, and launch-at-elaboration all lead to a concurrent control-flow pattern in which thread executions are properly nested (see Figure 12.6a). With parallel loops, each thread executes the same code, using different data; with co-begin and launch-at-elaboration, the code in different threads can be different. Put another way, parallel loops are generally data-parallel; co-begin and launch-at-elaboration are task-parallel.

The `fork` operation is more general: it makes the creation of threads an explicit, executable operation. The companion `join` operation allows a thread to wait for the completion of a previously forked thread. Because `fork` and `join` are not tied to nested constructs, they can lead to arbitrary patterns of concurrent control flow (Figure 12.6b). ■

In addition to providing launch-at-elaboration tasks, Ada allows the programmer to define task types:

```
task type T is
 ...
begin
 ...
end T;
```

#### EXAMPLE 12.12

Task types in Ada

The programmer may then declare variables of type `access T` (pointer to `T`), and may create new tasks via dynamic allocation:

```
pt : access T := new T;
```

The `new` operation is a `fork`: it creates a new thread and starts it executing. There is no explicit `join` operation in Ada, though parent and child tasks can always synchronize with one another explicitly if desired (e.g., immediately before the child completes its execution). In any scope in which a task type is declared, control will wait automatically at the end of the scope for all dynamically created tasks of that type to terminate. This convention avoids creating dangling references to local variables (Ada stack frames have limited extent). ■

#### **EXAMPLE 12.13**

Fork/Join in Modula-3

```
t := Fork(c);
...
Join(t);
```

Each Modula-3 thread begins execution in a specified subroutine. The language designers could have chosen to make this subroutine the argument to `Fork`, but this choice would have forced all `Forked` subroutines to accept the same fixed set of parameters, in accordance with strong typing. To avoid this limitation, Modula-3 defines the parameter to `Fork` to be a “thread closure”<sup>1</sup> object, as described in Section 9.4.5. The object contains a reference to the thread’s initial subroutine, together with any needed start-up arguments. The `Fork` operation calls the specified subroutine, passing a single argument: a reference to the thread closure object itself. The standard thread library defines a thread closure class with nothing in it except the subroutine reference. Programmers can define derived classes that contain additional fields, which the thread’s subroutine can then access. There is no comparable mechanism to pass start-up arguments to a task in Ada; information that would be passed as thread closure fields in Modula-3 must be sent to the already-started task in Ada via messages or shared variables. ■

#### **EXAMPLE 12.14**

Forking a proc in SR

Threads may be created in SR by sending a message to a `proc`, which resembles a procedure with a separate forward declaration, called an `op`. One of the most distinctive characteristics of SR is a remarkably elegant integration of sequential and concurrent constructs, and of message passing and subroutine invocation. An SR procedure is actually defined as syntactic sugar for an `op/proc` pair that has been limited to `call` style `forks`, in which the parent thread waits

---

**I** Thread closures should not be confused with the closures used for deep binding of subroutine referencing environments, as described in Section 3.5. Modula-3 uses closures in the traditional sense of the word when passing subroutines as parameters, but because its local objects have limited extent (again, see Section 3.5), it does not allow nested subroutines to be returned from functions or assigned into subroutine-valued variables. The subroutine reference in a thread “closure” is therefore guaranteed not to require a special referencing environment; it can be implemented as just a code address.

for the child to complete before continuing execution. As in Ada, there is no explicit `join` operation in SR, though a parent and child can always synchronize with one another explicitly if desired.

**EXAMPLE 12.15**

Thread creation in Java 2

In Java one obtains a thread by constructing an object of some class derived from a predefined class called `Thread`:

```
class image_renderer extends Thread {
 ...
 image_renderer(args) {
 // constructor
 }
 public void run() {
 // code to be run by the thread
 }
}
...
image_renderer rend = new image_renderer(constructor_args);
```

Superficially, the use of `new` resembles the creation of dynamic tasks in Ada. In Java, however, the new thread does *not* begin execution when first created. To start it, the parent (or some other thread) must call the method named `start`, which is defined in `Thread`:

```
rend.start();
```

`Start` makes the thread runnable, arranges for it to execute a method named `run`, and returns to the caller. The programmer must define an appropriate `run` method in every class derived from `Thread`. The `run` method is meant to be called only by `start`; programmers should not call it directly, nor should they redefine `start`. There is also a `join` method:

```
rend.join(); // wait for completion
```

**EXAMPLE 12.16**

Thread pools in Java 5

As of Java 5 (with its `java.util.concurrent` library), programmers are discouraged from creating threads explicitly. Rather, tasks to be accomplished (objects that support the `Callable` or `Runnable` interface) may be passed to an `Executor` object, which in turn may farm them out to a managed pool of threads. By separating the concepts of task and thread, Java allows the `Executor` class to optimize the level of true concurrency and the scheduling discipline to match the characteristics of the underlying platform.

```
class image_renderer implements Runnable {
 ...
 // constructor and run() method same as before
 ...
 Executor pool = Executors.newFixedThreadPool(4);
 ...
 pool.execute(new image_renderer(constructor_args));
```

Here the argument to `newFixedThreadPool` (one of a large number of standard `Executor factories`) indicates that `pool` should manage four threads. Each task specified in a call to `pool.execute` will be run by one of these threads. Thread and thread pool facilities in C# are similar to those of Java.

### ***Implicit Receipt***

The mechanisms described in the last few paragraphs allow a program to create new threads at run time. In each case those threads run in the same address space as the existing threads. In RPC systems it is often desirable to create a new thread automatically in response to an incoming request from some *other* address space. Rather than have an existing thread execute a `receive` operation, a server can *bind* a communication channel (which may be called a link, socket, or connection) to a local thread body or subroutine. When a request comes in, a new thread springs into existence to handle it.

In effect, the `bind` operation grants remote clients the ability to perform a `fork` within the server's address space. In SR the effect of `bind` is achieved by declaring a *capability* variable, initializing it with a reference to a procedure (an `op` for which there is a `proc`), and then sending it in a message to a thread in another address space. The receiving thread can then use that capability in a `send` or `call` operation, just as it would use the name of a local `op`. When it does so, the resulting message has the effect of performing a `fork` in the original address space. In RPC stub systems designed for use with ordinary sequential languages, the creation and management of threads to handle incoming calls is often less than completely automatic; we will consider the alternatives in Section 12.4.4.

### ***Early Reply***

#### **EXAMPLE 12.17**

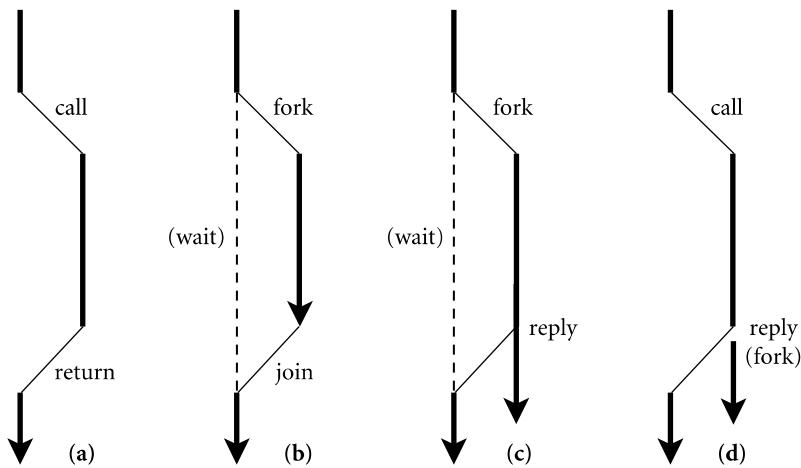
Modeling subroutines with  
`fork/join`

The similarity of `fork` and implicit receipt in SR reflects an important duality in the nature of subroutines. We normally think of sequential subroutines in terms of a single thread which saves its current context (its program counter and registers), executes the subroutine, and returns to what it was doing before (Figure 12.7a). The effect is the same, however, if we have two threads: one that executes the caller and another that executes the callee (Figure 12.7b). The caller waits for the callee to *reply* before continuing execution. The call itself is a `fork/join` pair, or a `send` and `receive` on a communication channel that has been set up for implicit receipt on the callee's end.

The two ways of thinking about subroutine calls suggest two different implementations, but either can be used to implement the other. In general, a compiler will want to avoid creating a separate thread whenever possible, in order to save time. As noted in the discussion on `fork/join` above, SR uses the two-thread model of subroutine calls. Within a single address space, however, it implements them with the usual subroutine-call mechanism whenever possible. In a similar vein, the Hermes language [SBG<sup>+</sup>91], which models subroutines in terms of threads and message passing, is able to use the usual subroutine-call implementation in the common case. If we think of subroutines in terms of separate threads for the caller and callee, there is actually no particular reason why the

#### **EXAMPLE 12.18**

Returning without  
terminating



**Figure 12.7** Threads, subroutine calls, and early reply. Conventionally, subroutine calls are conceptualized as using a single thread (a). Equivalent functionality can be achieved with separate threads (b). Early reply (c) allows a forked thread to continue execution after “returning” to the caller. To avoid creation of a callee thread in the common case, we can wait until the reply to do the fork (d).

callee should have to complete execution before it allows the caller to proceed: all it really has to do is complete the portion of its work on which result parameters depend. *Early reply* is a mechanism that allows a callee to return those results to the caller without terminating. After an early reply, the caller and callee continue execution concurrently (Figure 12.7c).

#### DESIGN & IMPLEMENTATION

##### Counterintuitive implementation

Over the course of 12 chapters we have seen numerous cases in which the implementation of a language feature may run counter to the programmer’s intuition. Early reply is but the most recent example. Others have included expression evaluation order (Section 6.1.4), subroutine inlining (Section 8.2.5), tail recursion (Section 6.6.1), nonstack allocation of activation records (for unlimited extent—Section 3.5.2), out-of-order or even noncontiguous layout of record fields (Section 7.3.2), variable lookup in a central reference table (Section 3.4.2), immutable objects under a reference model of variables (Section 6.1.2), and implementations of generics (Sections 3.6.3 and 8.4) that share code among instances with different type parameters. A compiler may, particularly at higher levels of code improvement, produce code that differs dramatically from the form and organization of its input. Unless otherwise constrained by the language definition, an implementation is free to choose any translation that is provably equivalent to the input.

If we think of subroutines in terms of a single thread for the caller and callee, then early reply can also be seen as a means of creating new threads. When the thread executing a subroutine performs an early reply, it splits itself into a pair of threads: one of these returns, the other continues execution in the callee (Figure 12.7d).

**EXAMPLE 12.19**

Early reply in SR

In SR, any subroutine can execute an early reply operation:

```
reply
```

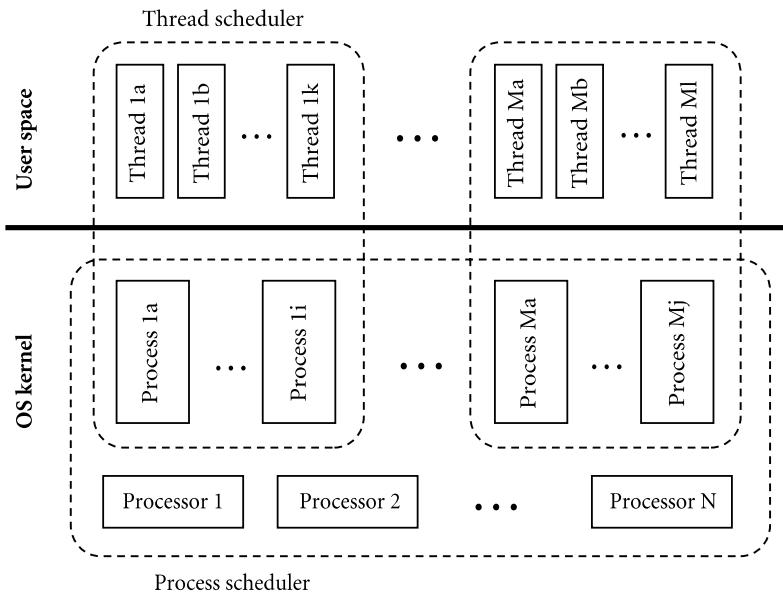
For calls within a single address space, the SR compiler waits until the `reply` before creating a new thread; a subroutine that returns without replying uses a single implementation thread for the caller and callee. Until the time of the `reply`, the stack frame of the subroutine belongs to the calling thread. To allow it to become the initial frame of a newly created thread, an SR implementation can employ a memory management scheme in which stack frames are allocated dynamically from the heap and linked together with pointers. Alternatively, the implementation can copy the current frame into the bottom of a newly allocated stack at the time of the `reply`. Early reply resembles the coroutine `detach` operation of Simula. It also appears in Lynx [Sco91].

Much of the motivation for early reply comes from applications in which the parent of a newly created thread needs to ensure that the thread has been initialized properly before it (the parent) continues execution. In a web browser, for example, the thread responsible for formatting a page will create a new child for each in-line image. The child will contact the appropriate server and begin to transfer data. The first thing the server will send is an indication of the image's size. The page-formatting thread (the parent of the image-rendering thread) needs to know this size in order to place text and other images properly on the page. Early reply allows the parent to create the child and then wait for it to reply with size information, at which point the parent and child can proceed in parallel. (We ignored this issue in Figure 12.2.)

In Java and C#, a similar purpose is served by separating thread creation from invocation of the `start` method. In our browser example, a page-formatting thread that creates a child to render an image could call a `get_size` method of the child *before* it calls the child's `start` method. `Get_size` would make the initial contact with the server and return size information to the parent. Because `get_size` is a method of the child, any data it initializes, including the size and connection-to-server information, will be stored in the thread's fields, where they will be available to the thread's `run` method.

#### 12.2.4 Implementation of Threads

As we noted near the beginning of Section 12.2, the threads of a concurrent program are usually implemented on top of one or more *processes* provided by the operating system. At one extreme, we could use a separate OS process for every thread; at the other extreme we could multiplex all of a program's threads on



**Figure 12.8** Two-level implementation of threads. A thread scheduler, implemented in a library or language run-time package, multiplexes threads on top of one or more kernel-level processes, just as the process scheduler, implemented in the operating system kernel, multiplexes processes on top of one or more physical processors.

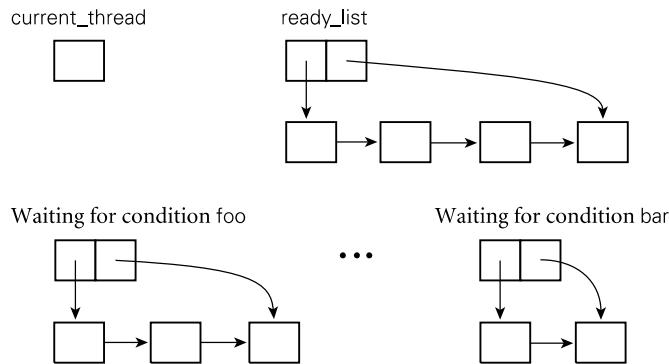
top of a single process. On a personal computer with a single address space and relatively inexpensive processes, the one-process-per-thread extreme is often acceptable. In a simple language on a uniprocessor, the all-threads-on-one-process extreme may be acceptable. Commonly, language implementations adopt an in-between approach, with a potentially large number of threads running on top of a smaller number of processes (see Figure 12.8).

#### EXAMPLE 12.21

Multiplexing threads on processes

The problem with putting every thread on a separate process is that processes (even “lightweight” ones) are simply too expensive in many operating systems. Because they are implemented in the kernel, performing any operation on them requires a system call. Because they are general purpose, they provide features that most languages do not need but have to pay for anyway. (Examples include separate address spaces, priorities, accounting information, and signal and I/O interfaces, all of which are beyond the scope of this book.) At the other extreme, there are two problems with putting all threads on top of a single process: first, it precludes parallel execution on a multiprocessor; second, if the currently running thread makes a system call that blocks (e.g., waiting for I/O), then none of the program’s other threads can run, because the single process is suspended by the OS.

In the common two-level organization of concurrency (user-level threads on top of kernel-level processes), similar code appears at both levels of the system: the language run-time system implements threads on top of one or more



**Figure 12.9 Data structures of a simple scheduler.** A designated `current_thread` is running. Threads on the `ready_list` are runnable. Other threads are blocked, waiting for various conditions to become true. If threads run on top of more than one OS-level process, each such process will have its own `current_thread` variable. If a thread makes a call into the operating system, its process may block in the kernel.

processes in much the same way that the operating system implements processes on top of one or more physical processors. A multiprocessor operating system may attempt to ensure that processes belonging to the same application run on separate processors simultaneously, in order to minimize synchronization delays (this technique is called *coscheduling*, or *gang scheduling*). Alternatively, it may give an application exclusive use of some subset of the processors (this technique is called *space sharing*, or *processor partitioning*). Such kernel-level issues are beyond the scope of this book; we concentrate here on user-level threads.

Typically, user-level threads are built on top of coroutines (Section 8.6). Recall that coroutines are a sequential control-flow mechanism, designed for implementation on top of a single OS process. The programmer can suspend the current coroutine and resume a specific alternative by calling the `transfer` operation. The argument to `transfer` is typically a pointer to the context block of the coroutine.

To turn coroutines into threads, we can proceed in a series of three steps. First, we hide the argument to `transfer` by implementing a *scheduler* that chooses which thread to run next when the current thread yields the processor. Second, we implement a *preemption* mechanism that suspends the current thread automatically on a regular basis, giving other threads a chance to run. Third, we allow the data structures that describe our collection of threads to be shared by more than one OS process, possibly on separate processors, so that threads can run on any of the processes.

### Uniprocessor Scheduling

#### EXAMPLE 12.22

Cooperative multithreading on a uniprocessor

Figure 12.9 illustrates the data structures employed by a simple scheduler. At any particular time, a thread is either *blocked* (i.e., for synchronization) or *runnable*. A runnable thread may actually be running on some processor or it may be await-

ing its chance to do so. Context blocks for threads that are runnable but not currently running reside on a queue called the *ready list*. Context blocks for threads that are blocked for scheduler-based synchronization reside in data structures (usually queues) associated with the conditions for which they are waiting. To yield the processor to another thread, a running thread calls the scheduler:

```
procedure reschedule
 t : thread := dequeue(ready_list)
 transfer(t)
```

Before calling into the scheduler, a thread that wants to run again at some point in the future must place its own context block in some appropriate data structure. If it is blocking for the sake of fairness—to give some other thread a chance to run—then it enqueues its context block on the ready list:

```
procedure yield
 enqueue(ready_list, current_thread)
 reschedule
```

To block for synchronization, a thread adds itself to a queue associated with the awaited condition:

```
procedure sleep_on(ref Q : queue of thread)
 enqueue(Q, current_thread)
 reschedule
```

When a running thread performs an operation that makes a condition true, it removes one or more threads from the associated queue and enqueues them on the ready list. ■

Fairness becomes an issue whenever a thread may run for a significant amount of time while other threads are runnable. To give the illusion of concurrent activity, even on a uniprocessor, we need to make sure that each thread gets a frequent “slice” of the processor. With *cooperative multithreading*, any long-running thread must yield the processor explicitly from time to time (e.g., at the tops of loops) to allow other threads to run. As noted in Section 12.1.2, this approach allows one improperly written thread to monopolize the system. Even with properly written threads, it leads to less than perfect fairness due to nonuniform times between yields in different threads.

### **Preemption**

Ideally, we should like to multiplex the processor fairly and at a relatively fine grain (i.e., many times per second) *without* requiring that threads call `yield` explicitly. On many systems we can do this in the language implementation by using timer signals for *preemptive multithreading*. When switching between threads we ask the operating system (which has access to the hardware clock) to deliver a signal to the currently running process at a specified time in the future. The OS delivers the signal by saving the context (registers and pc) of the process at the top of the current stack and transferring control to a previously specified *handler* routine in the language run-time system. When called, the handler modifies the

state of the currently running thread to make it appear that the thread had just executed a call to the standard `yield` routine. The handler then “returns” into `yield`, which transfers control to some other thread, as if the one that had been running had relinquished control of the process voluntarily.

Unfortunately, the fact that a signal may arrive at an arbitrary time introduces a race between voluntary calls to the scheduler and the automatic calls triggered by preemption. To illustrate the problem, suppose that a signal arrives when the currently running process has just enqueued the currently running thread onto the ready list in `yield` and is about to call `reschedule`. When the signal handler “returns” into `yield`, the process will put the current thread into the ready list a second time. If at some point in the future the thread blocks for synchronization, its second entry in the ready list may cause it to run again immediately, when it should be waiting. Even worse problems can arise if a signal occurs in the middle of an `enqueue`, at a moment when the ready list is not even a properly structured queue. To resolve the race and avoid corruption of the ready list, thread packages commonly disable signal delivery during scheduler calls:

```
procedure yield
 disable_signals
 enqueue(ready_list, current_thread)
 reschedule
 reenable_signals
```

For this convention to work, *every* fragment of code that calls `reschedule` must disable signals prior to the call, and must reenable them afterward. Because `reschedule` contains a call to `transfer`, signals may be disabled in one thread and reenabled in another. ■

#### EXAMPLE 12.24

Disabling signals during context switch

It turns out that the `sleep_on` routine must also assume that signals are disabled and enabled by the caller. To see why, suppose that a thread checks a condition, finds that it is false, and then calls `sleep_on` to suspend itself on a queue associated with the condition. Suppose further that a timer signal occurs immediately after checking the condition but before the call to `sleep_on`. Finally, suppose that the thread is allowed to run after the signal makes the condition true. Since the first thread never got a chance to put itself on the condition queue, the second thread will not find it to make it runnable. When the first thread runs again, it will immediately suspend itself, and may never be awakened. To close this *timing window*—this interval in which a concurrent event may compromise program correctness—the caller must ensure that signals are disabled before checking the condition:

```
disable_signals
if not desired_condition
 sleep_on(condition_queue)
reenable_signals
```

On a uniprocessor, disabling signals allows the check and the sleep to occur as a single, *atomic* operation: they always appear to happen “all at once” from the point of view of other threads. ■

### Multiprocessor Scheduling

A few concurrent languages (e.g., Distributed Processes [Bri78] and Lynx [Sco91]) are explicitly nonparallel: language semantics guarantee that only one thread will run in a given address space at a time, with switches among threads occurring only at well-defined points in the code. Most concurrent languages, however, permit threads to run in parallel. As we noted in Section 12.2, there is no difference from the programmer's point of view between true parallelism (on multiple processors) and the "quasiparallelism" of a system that switches between threads on timer interrupts: in both cases, threads must synchronize explicitly to cope with race conditions in the application program.

We can extend our preemptive thread package to run on top of more than one OS-provided process by arranging for the processes to share the ready list and related data structures (condition queues, etc.; note that each process must have a *separate* current\_thread variable). If the processes run on different processors of a shared-memory multiprocessor, then more than one thread will be able to run at once. If the processes share a single processor, then the program will be able to make forward progress even when all but one of the processes are blocked in the operating system. Any thread that is runnable is placed in the ready list, where it becomes a candidate for execution by any of the application's processes. When a process calls `reschedule`, the queue-based ready list we have been using in our examples will give it the longest-waiting thread. The ready list of a more elaborate scheduler might give priority to interactive or time-critical threads, or to threads that last ran on the current processor and may therefore still have data in the cache.

Just as preemption introduced a race between voluntary and automatic calls to scheduler operations, true or quasiparallelism introduces races between calls in separate OS processes. To resolve the races, we must implement additional synchronization to make scheduler operations in separate processes atomic. We will return to this subject in Section 12.3.2.

#### CHECK YOUR UNDERSTANDING

10. Explain the difference between a *coroutine*, a *thread*, a *lightweight process*, and a *heavyweight process*.
11. What is *quasiparallelism*?
12. Describe the *bag of tasks* programming model.
13. What is *busy-waiting*? What is its principal alternative?
14. Name four explicitly concurrent programming languages.
15. Why don't message-passing programs require explicit synchronization mechanisms?
16. What are the tradeoffs between language-based and library-based implementations of concurrency?

17. Explain the difference between *data parallelism* and *task parallelism*.
  18. What is *collective communication*?
  19. Describe six different syntactic constructs commonly used to create new threads of control in a concurrent program.
  20. In what sense is `fork/join` more powerful than `co-begin`?
  21. What is a *thread pool* in Java? What purpose does it serve?
  22. Why is meant by a *two-level* thread implementation?
  23. What is a *ready list*?
  24. Describe the progressive implementation of scheduling, preemption, and (true) parallelism on top of coroutines.
  25. What is *coscheduling*? What is its purpose?
- 

## 12.3 Shared Memory

As noted in Section 12.2.1, synchronization is the principal semantic challenge for shared-memory concurrent programs. One commonly sees two forms of synchronization: *mutual exclusion* and *condition synchronization*. Mutual exclusion ensures that only one thread is executing a *critical section* of code at a given point in time. Condition synchronization ensures that a given thread does not proceed until some specific condition holds (e.g., until a given variable has a given value). It is tempting to think of mutual exclusion as a form of condition synchronization (don't proceed until no other thread is in its critical section), but this sort of condition would require *consensus* among all extant threads, something that condition synchronization doesn't generally provide.

Our implementation of parallel threads, sketched at the end of Section 12.2.4, requires that *processes* (provided by the OS) use both mutual exclusion and condition synchronization to protect the ready list and related data structures. Mutual exclusion appears in the requirement that a process must never read or write the ready list while it is being modified by another process; condition synchronization appears in the requirement that a process in need of a thread to run must wait until the ready list is nonempty.

It is worth emphasizing that we do not in general want to overly synchronize programs. To do so would eliminate opportunities for parallelism, which we generally want to maximize in the interest of performance. The goal is to provide only as much synchronization as is necessary in order to eliminate “bad” race conditions: those that might otherwise cause the program to produce incorrect results.

In the first subsection below we consider busy-wait synchronization. In the second we use busy-waiting among processes to implement a parallelism-safe

thread scheduler. In subsequent subsections we use the scheduler to implement blocking synchronization for threads: semaphores, monitors, conditional critical regions, and various higher-level mechanisms in which synchronization is implicit.

### 12.3.1 Busy-Wait Synchronization

Busy-wait condition synchronization is generally easy: if we can cast a condition in the form of “location  $X$  contains value  $Y$ ,” then a thread (or process) that needs to wait for the condition can simply read  $X$  in a loop, waiting for  $Y$  to appear. All that is required from the hardware is that individual load and store instructions be atomic. Providing this atomicity is not a trivial task (memory and/or busses must serialize concurrent accesses by processors and devices), but almost every computer ever made has done it.

Busy-wait mutual exclusion is harder. We consider it under Spin Locks below. We then consider a special form of condition synchronization—namely barriers. A barrier is meant to be executed by all of the threads in a program. It guarantees that no thread will continue past a given point in a program until all threads have reached that point. Like mutual exclusion (and unlike most condition synchronization), barriers require consensus among all extant threads. Barriers are fundamental to data-parallel computing. They can be implemented either with busy-waiting or with blocking; we consider the busy-wait version here.

#### **Spin Locks**

Dekker is generally credited with finding the first two-thread mutual exclusion algorithm that requires no atomic instructions other than load and store. Dijkstra [Dij65] published a version that works for  $n$  threads in 1965. Peterson [Pet81] published a much simpler two-thread algorithm in 1981. Building on Peterson’s algorithm, one can construct a hierarchical  $n$ -thread lock, but it requires  $O(n \log n)$  space and  $O(\log n)$  time to get one thread into its critical section [YA93]. Lamport [Lam87] published an  $n$ -thread algorithm in 1987 that takes  $O(n)$  space and  $O(1)$  time in the absence of competition for the lock. Unfortunately, it requires  $O(n)$  time when multiple threads attempt to enter their critical section at once.

To achieve mutual exclusion in constant time, one needs a more powerful atomic instruction. Beginning in the 1960s, hardware designers began to equip their processors with instructions that read, modify, and write a memory location as a single atomic operation. The simplest read-modify-write instruction is known as `test_and_set`. It sets a Boolean variable to `true` and returns an indication of whether the variable was `false` previously. Given `test_and_set`, acquiring a spin lock is almost trivial:

```
while not test_and_set(L)
 -- nothing -- spin
```

---

#### **EXAMPLE 12.25**

The basic `test_and_set` lock

```

type lock = Boolean := false;

procedure acquire_lock(ref L : lock)
 while not test_and_set(L)
 while L
 -- nothing -- spin
procedure release_lock(ref L : lock)
 L := false

```

**Figure 12.10** A simple `test-and-test_and_set` lock. Waiting processes spin with ordinary read (`load`) instructions until the lock appears to be free, then use `test_and_set` to acquire it. The very first access is a `test_and_set`, for speed in the common (no competition) case.

In practice, embedding `test_and_set` in a loop tends to result in unacceptable amounts of bus traffic on a multiprocessor, as the cache coherence mechanism attempts to reconcile writes by multiple processors attempting to acquire the lock. This overdemand for hardware resources is known as *contention*, and is a major obstacle to good performance on large machines.

To reduce contention, the writers of synchronization libraries often employ a `test-and-test_and_set` lock, which spins with ordinary reads (satisfied by the cache) until it appears that the lock is free (see Figure 12.10). When a thread releases a lock there still tends to be a flurry of bus activity as waiting threads perform their `test_and_sets`, but at least this activity happens only at the boundaries of critical sections. On a large machine, bus or interconnect traffic can be further reduced by implementing a *backoff* strategy, in which a thread that is unsuccessful in attempting to acquire a lock waits for a while before trying again. ■

Many processors provide atomic instructions more powerful than `test_and_set`. Several can swap the contents of a register and a memory location atomically. A few can add a constant to a memory location atomically, returning the previous value. On the x86, most arithmetic instructions can be prefaced with a “lock” byte that causes them to update a memory location atomically. MIPS, Alpha, and PowerPC processors provide a pair of instructions called `load_linked` and `store_conditional` (LL/SC). The first of these instructions loads a memory location into a register and stores certain bookkeeping information into hidden processor registers. The second instruction stores the register back into the memory location, but only if the location has not been modified by any other processor since the `load_linked` was executed. In the time between the two instructions the processor may be limited in its ability to touch memory (implementations vary), but it can perform an almost arbitrary computation in registers, allowing the LL/SC pair to function as a *universal* atomic primitive. To add the value in register `r2` to memory location `foo`, atomically, one would execute the following instructions.

#### EXAMPLE 12.26

##### Test-and-test\_and\_set

#### EXAMPLE 12.27

##### Atomic update with LL/SC

```

start:
 r1 := load_linked(foo)
 r1 := r1 + r2
 store_conditional(r1, foo)
 if failed goto start

```

If several processors execute this code simultaneously, one of them is guaranteed to succeed the first time around the loop. The others will fail and try again. Exercise 12.6 considers another universal primitive, called `compare_and_swap`, that appears in the x86, the Sparc, the IA-64, and descendants of the IBM 370. ■

Using instructions like `atomic_add` or LL/SC, one can build spin locks that are *fair*, in the sense that threads are guaranteed to acquire the lock in the order in which they first attempt to do so. One can also build locks that work well—with no contention—on arbitrarily large machines [MCS91]. Finally, one can use universal atomic primitives to build special purpose concurrent data structures and algorithms that operate *without* locks, by modifying locations atomically in a carefully determined order. Lock-free concurrent algorithms are ideal for environments in which threads may pause (e.g., due to preemption) for arbitrary periods of time. In important special cases (e.g., queues [MS96] or memory management [Mic04]) lock-free algorithms can also be significantly faster than lock-based algorithms. Herlihy [Her91] and others [HLMS03, HF03] have developed general purpose techniques to turn sequential data structures into lock-free concurrent data structures, but these are not yet able, for most applications, to rival the performance of lock-based algorithms.

An important variant on mutual exclusion is the *reader–writer lock* [CHP71]. Reader–writer locks recognize that if several threads wish to *read* the same data structure, they can do so simultaneously without mutual interference. It is only when a thread wants to *write* the data structure that we need to prevent other threads from reading or writing simultaneously. Most busy-wait mutual exclusion locks can be extended to allow concurrent access by readers (see Exercise 12.8).

### Barriers

Barriers are common in data-parallel numeric algorithms. In *finite element analysis*, for example, a physical object such as, say, a bridge may be modeled as an enormous collection of tiny metal fragments. Each fragment imparts forces to the fragments adjacent to it. Gravity exerts a downward force on all fragments. Abutments exert an upward force on the fragments that make up base plates. The wind exerts forces on surface fragments. To evaluate stress on the bridge as a whole (e.g., to assess its stability and resistance to failures), a finite element program might divide the metal fragments among a large collection of threads (probably one per physical processor). Beginning with the external forces, the program would then proceed through a sequence of iterations. In each iteration each thread would recompute the forces on its fragments based on the forces found in the previous iteration. Between iterations, the threads would synchro-

```

shared count : integer := n
shared sense : Boolean := true
per-thread private local_sense : Boolean := true

procedure central_barrier
 local_sense := not local_sense
 -- each thread toggles its own sense
 if fetch_and_decrement(count) = 1
 -- last arriving thread
 count := n -- reinitialize for next iteration
 sense := local_sense -- allow other threads to proceed
 else
 repeat
 -- spin
 until sense = local_sense

```

**Figure 12.11** A simple “sense-reversing” barrier. Each thread has its own copy of local\_sense. Threads share a single copy of count and sense.

nize with a barrier. The program would halt when no thread found a significant change in any forces during the last iteration.

The simplest way to implement a busy-wait barrier is to use a globally shared counter, modified by an atomic `fetch_and_decrement` instruction (or equivalently by `fetch_and_add`, LL/SC, etc.). The counter begins at  $n$ , the number of threads in the program. As each thread reaches the barrier it decrements the counter. If it is not the last to arrive, the thread then spins on a Boolean flag. The final thread (the one that changes the counter from 1 to 0) flips the Boolean flag, allowing the other threads to proceed. To make it easy to reuse the barrier data structures in successive iterations (known as barrier *episodes*), threads wait for alternating values of the flag each time through. Code for this simple barrier appears in Figure 12.11. ■

Like a simple spin lock, the “sense-reversing” barrier can lead to unacceptable levels of contention on large machines. Moreover the serialization of access to the counter implies that the time to achieve an  $n$ -thread barrier is  $O(n)$ . It is possible to do better, but even the fastest software barriers require  $O(\log n)$  time to synchronize  $n$  threads [MCS91]. Several large multiprocessors, including the Thinking Machines CM-5 and the Cray T3D, T3E, and X1, have provided special hardware for near-constant-time busy-wait barriers.

### 12.3.2 Scheduler Implementation

To implement user-level threads, OS-level processes must synchronize access to the ready list and condition queues, generally by means of spinning. Code for a simple *reentrant* thread scheduler (one that can be “reentered” safely by a second process before the first one has returned) appears in Figure 12.12. As in the code

#### EXAMPLE 12.28

The “sense-reversing” barrier

#### EXAMPLE 12.29

Scheduling threads on processes

```

shared scheduler_lock : low_level_lock
shared ready_list : queue of thread
per-process private current_thread : thread

procedure reschedule
 -- assume that scheduler_lock is already held
 -- and that timer signals are disabled
 t : thread
 loop
 t := dequeue(ready_list)
 if t ≠ nil
 exit
 -- else wait for a thread to become runnable
 release_lock(scheduler_lock)
 -- window allows another thread to access ready_list
 -- (no point in reenabling signals;
 -- we're already trying to switch to a different thread)
 acquire_lock(scheduler_lock)
 transfer(t)
 -- caller must release scheduler_lock
 -- and reenable timer signals after we return

procedure yield
 disable_signals
 acquire_lock(scheduler_lock)
 enqueue(ready_list, current_thread)
 reschedule
 release_lock(scheduler_lock)
 reenable_signals

procedure sleep_on(ref Q : queue of thread)
 -- assume that caller has already disabled timer signals
 -- and acquired scheduler_lock, and will reverse
 -- these actions when we return
 enqueue(Q, current_thread)
 reschedule

```

**Figure 12.12** Pseudocode for part of a simple reentrant (parallelism-safe) scheduler. Every process has its own copy of current\_thread. There is a single shared scheduler\_lock and a single ready\_list. If processes have dedicated processors, then the low\_level\_lock can be an ordinary spin lock; otherwise it can be a “spin-then-yield” lock (Figure 12.13). The loop inside reschedule busy-waits until the ready list is nonempty. The code for sleep\_on cannot disable timer signals and acquire the scheduler lock itself, because the caller needs to test a condition and then block as a single atomic operation.

in Section 12.2.4, we disable timer signals before entering scheduler code, to protect the ready list and condition queues from concurrent access by a process and its own signal handler.

**EXAMPLE 12.30**

A race condition in thread scheduling

Our code assumes a single “low-level” lock (`scheduler_lock`) that protects the entire scheduler. Before saving its context block on a queue (e.g., in `yield` or `sleep_on`), a thread must acquire the scheduler lock. It must then release the lock after returning from `reschedule`. Of course, because `reschedule` calls `transfer`, the lock will usually be acquired by one thread (the same one that disables timer signals) and released by another (the same one that reenables timer signals). The code for `yield` can implement synchronization itself, because its work is self-contained. The code for `sleep_on`, on the other hand, cannot, because a thread must generally check a condition and block if necessary as a single atomic operation:

```
disable_signals
acquire_lock(scheduler_lock)
if not desired_condition
 sleep_on(condition_queue)
release_lock(scheduler_lock)
reenable_signals
```

If the signal and lock operations were moved inside of `sleep_on`, the following race could arise: thread *A* checks the condition and finds it to be false; thread *B* makes the condition true, but finds the condition queue to be empty; thread *A* sleeps on the condition queue forever.

A spin lock will suffice for the “low-level” lock that protects the ready list and condition queues, as long as every process runs on a different processor. As we noted in Section 12.2.1, however, it makes little sense to spin for a condition that can only be made true by some other process using the processor on which we are spinning. If we know that we’re running on a uniprocessor, then we don’t need a lock on the scheduler (just the disabled signals). If we *might* be running on a uniprocessor, however, or on a multiprocessor with fewer processors than processes, then we must be prepared to give up the processor if unable to obtain a lock. The easiest way to do this is with a “spin-then-yield” lock, first suggested by Ousterhout [Ous82]. A simple example of such a lock appears in Figure 12.13. On a multiprogrammed machine, it might also be desirable to relinquish the processor inside `reschedule` when the ready list is empty: though no other process of the current application will be able to do anything, overall system throughput may improve if we allow the operating system to give the processor to a process from another application.

On a large multiprocessor we might increase concurrency by employing a separate lock for each condition queue, and another for the ready list. We would have to be careful, however, to make sure it wasn’t possible for one process to put a thread into a condition queue (or the ready list) and for another process to attempt to transfer into that thread before the first process had finished transferring out of it (see Exercise 12.9).

**EXAMPLE 12.31**

A “spin-then-yield” lock

```

type lock = Boolean := false;

procedure acquire_lock(ref L : lock)
 while not test_and_set(L)
 count := TIMEOUT
 while L
 count -=: 1
 if count = 0
 OS_yield -- relinquish processor
 count := TIMEOUT

procedure release_lock(ref L : lock)
 L := false

```

**Figure 12.13** A simple spin-then-yield lock, designed for execution on a multiprocessor that may be multiprogrammed (i.e., on which OS-level processes may be preempted). If unable to acquire the lock in a fixed, short amount of time, a process calls the OS scheduler to yield its processor. Hopefully the lock will be available the next time the process runs.

### Scheduler-Based Synchronization

The problem with busy-wait synchronization is that it consumes processor cycles, cycles that are therefore unavailable for other computation. Busy-wait synchronization makes sense only if (1) one has nothing better to do with the current processor, or (2) the expected wait time is less than the time that would be required to switch contexts to some other thread and then switch back again. To ensure acceptable performance on a wide variety of systems, most concurrent programming languages employ scheduler-based synchronization mechanisms, which switch to a different thread when the one that was running blocks.

In the following subsection we consider the three most common forms of scheduler-based synchronization: semaphores, monitors, and conditional critical regions. In each case, scheduler-based synchronization mechanisms remove the waiting thread from the scheduler's ready list, returning it only when the awaited condition is true (or is likely to be true). By contrast, the spin-then-yield lock described in Section 12.3.2 is still a busy-wait mechanism: the currently running process relinquishes the processor but remains on the ready list. It will perform a `test_and_set` operation every time it gets a chance to run, until it finally succeeds. It is worth noting that busy-wait synchronization is generally “level-independent”—it can be thought of as synchronizing threads, processes, or processors, as desired. Scheduler-based synchronization is “level-dependent”—it is specific to threads when implemented in the language runtime system, or to processes when implemented in the operating system.

We will use a *bounded buffer* abstraction to illustrate the semantics of various scheduler-based synchronization mechanisms. A bounded buffer is a concurrent queue of limited size into which *producer* threads insert data, and from which *consumer* threads remove data. The buffer serves to even out fluctuations in the relative rates of progress of the two classes of threads, increasing system throughput. A correct implementation of a bounded buffer requires both mutual exclu-

```

type semaphore = record
 N : integer -- usually initialized to something nonnegative
 Q : queue of threads

procedure P(ref S : semaphore)
 disable_signals
 acquire_lock(scheduler_lock)
 S.N -=: 1
 if S.N < 0
 sleep_on(S.Q)
 release_lock(scheduler_lock)
 reenable_signals

procedure V(ref S : semaphore)
 disable_signals
 acquire_lock(scheduler_lock)
 S.N +=: 1
 if N ≤ 0
 -- at least one thread is waiting
 enqueue(ready_list, dequeue(S.Q))
 release_lock(scheduler_lock)
 reenable_signals

```

**Figure 12.14** Semaphore operations, for use with the scheduler code of Figure 12.12.

sion and condition synchronization: the former to ensure that no thread sees the buffer in an inconsistent state in the middle of some other thread's operation; the latter to force consumers to wait when the buffer is empty and producers to wait when the buffer is full.

### 12.3.3 Semaphores

Semaphores are the oldest of the scheduler-based synchronization mechanisms. They were described by Dijkstra in the mid-1960s [Dij68a], and appear in Algol 68. They are still heavily used today, both in library packages and in languages like SR and Modula-3.

#### EXAMPLE 12.32

Semaphore implementation

A semaphore is basically a counter with two associated operations, P and V.<sup>2</sup> A thread that calls P atomically decrements the counter and then waits until it is nonnegative. A thread that calls V atomically increments the counter and wakes up a waiting thread, if any. It is generally assumed that semaphores are fair, in the sense that threads complete P operations in the same order they start them. Implementations of P and V in terms of our scheduler operations appear in Figure 12.14.

---

**2** P and V stand for the Dutch words *passeren* ("to pass") and *vrijgeven* ("to release"). To keep them straight, speakers of English may wish to think of P as standing for "pause," since a thread will pause at a P operation if the semaphore count is negative. Algol 68 calls the P and V operations *down* and *up*, respectively.

```

shared buf : array [1..SIZE] of bdata
shared next_full, next_empty : integer := 1, 1
shared mutex : semaphore := 1
shared empty_slots, full_slots : semaphore := SIZE, 0

procedure insert(d : bdata)
 P(empty_slots)
 P(mutex)
 buf[next_empty] := d
 next_empty := next_empty mod SIZE + 1
 V(mutex)
 V(full_slots)

function remove : bdata
 P(full_slots)
 P(mutex)
 d : bdata := buf[next_full]
 next_full := next_full mod SIZE + 1
 V(mutex)
 V(empty_slots)
 return d

```

**Figure 12.15** Semaphore-based code for a bounded buffer. The mutex binary semaphore protects the data structure proper. The full\_slots and empty\_slots general semaphores ensure that no operation starts until it is safe to do so.

A semaphore whose counter is initialized to one and for which P and V operations always occur in matched pairs is known as a *binary semaphore*. It serves as a scheduler-based mutual exclusion lock: the P operation acquires the lock; V releases it. More generally, a semaphore whose counter is initialized to  $k$  can be used to arbitrate access to  $k$  copies of some resource. The value of the counter at any particular time is always  $k$  more than the difference between the number of P operations (#P) and the number of V operations (#V) that have occurred so far in the program. A P operation blocks the caller until  $#P \leq #V + k$ . Exercise 12.18 notes that binary semaphores can be used to implement general semaphores, so the two are of equal expressive power, if not of equal convenience.

Figure 12.15 shows a semaphore-based solution to the bounded buffer problem. It uses a binary semaphore for mutual exclusion, and two general (or *counting*) semaphores for condition synchronization. Exercise 12.14 considers the use of semaphores to construct an  $n$ -thread barrier.

### EXAMPLE 12.33

Bounded buffer with semaphores

#### CHECK YOUR UNDERSTANDING

26. What is a *critical section*?
27. What does it mean for an operation to be *atomic*?

28. Explain the difference between *mutual exclusion* and *condition synchronization*.
  29. Describe the behavior of a `test_and_set` instruction. Show how to use it to build a *spin lock*.
  30. Describe the behavior of the `load_linked` and `store_conditional` instructions. What advantages do they offer in comparison to `test_and_set`?
  31. Explain how a *reader-writer lock* differs from an “ordinary” lock.
  32. What is a *barrier*? In what types of programs are barriers common?
  33. What does it mean for code to be *reentrant*?
  34. Explain how to extend a preemptive uniprocessor scheduler to work correctly on a multiprocessor.
  35. What is a *spin-then-yield* lock?
  36. What is a *bounded buffer*?
  37. What is a *semaphore*? What operations does it support? How do *binary* and *general* semaphores differ?
- 

#### 12.3.4 Monitors

Though widely used, semaphores are also widely considered to be too “low-level” for well-structured, maintainable code. They suffer from two principal problems. First, their operations are simply subroutine calls, it is easy to leave one out (e.g., on a control path with several nested `if` statements). Second, unless they are hidden inside an abstraction, uses of a given semaphore tend to get scattered throughout a program, making it difficult to track them down for purposes of software maintenance.

Monitors were suggested by Dijkstra [Dij72] as a solution to these problems. They were developed more thoroughly by Brinch Hansen [Bri73] and formalized by Hoare [Hoa74] in the early 1970s. They have been incorporated into at least a score of languages, of which Concurrent Pascal [Bri75], Modula (1) [Wir77b], and Mesa [LR80] have probably been the most influential.<sup>3</sup>

---

**3** Together with Smalltalk and Interlisp, Mesa was one of three influential languages to emerge from Xerox’s Palo Alto Research Center in the 1970s. All three were developed on the Alto personal computer, which pioneered such concepts as the bitmapped display, the mouse, the graphical user interface, WYSIWYG editing, Ethernet networking, and the laser printer. The Mesa project was led by Butler Lampson (1943–), who played a key role in the later development of Euclid and Cedar as well. For his contributions to personal and distributed computing, Lampson received the ACM Turing Award in 1992.

```

monitor bounded_buf
imports bdata, SIZE
exports insert, remove

buf : array [1..SIZE] of data
next_full, next_empty : integer := 1, 1
full_slots : integer := 0
full_slot, empty_slot : condition

entry insert(d : bdata)
if full_slots = SIZE
 wait(empty_slot)
 buf[next_empty] := d
 next_empty := next_empty mod SIZE + 1
 full_slots += 1
 signal(full_slot)

entry remove : bdata
if full_slots = 0
 wait(full_slot)
 d : bdata := buf[next_full]
 next_full := next_full mod SIZE + 1
 full_slots -= 1
 signal(empty_slot)
return d

```

**Figure 12.16** Monitor-based code for a bounded buffer. Insert and remove are entry subroutines: they require exclusive access to the monitor's data. Because conditions are memory-less, both insert and remove can safely end their operation with a signal.

A monitor is a module or object with operations, internal state, and a number of *condition variables*. Only one operation of a given monitor is allowed to be active at a given point in time. A thread that calls a busy monitor is automatically delayed until the monitor is free. On behalf of its calling thread, any operation may suspend itself by *waiting* on a condition variable. An operation may also *signal* a condition variable, in which case one of the waiting threads is resumed, usually the one that waited first.

Because the operations (*entries*) of a monitor automatically exclude one another in time, the programmer is relieved of the responsibility of using P and V operations correctly. Moreover because the monitor is an abstraction, all operations on the encapsulated data, including synchronization, are collected together in one place. Figure 12.16 shows a monitor-based solution to the bounded buffer problem. It is worth emphasizing that monitor condition variables are not the same as semaphores. Specifically, they have no “memory”: if no thread is waiting on a condition at the time that a *signal* occurs, then the *signal* has no effect. Whereas a V operation on a semaphore increments the semaphore’s counter, allowing some future P operation to succeed, an un-awaited *signal* on a condition variable is lost. ■

#### EXAMPLE 12.34

##### Bounded buffer monitor

### Semantic Details

Hoare's definition of monitors employs one thread queue for every condition variable, plus two bookkeeping queues: the *entry queue* and the *urgent queue*. A thread that attempts to enter a busy monitor waits in the entry queue. When a thread executes a `signal` operation from within a monitor, and some other thread is waiting on the specified condition, then the signaling thread waits on the monitor's urgent queue, and the first thread on the appropriate condition queue obtains control of the monitor. If no thread is waiting on the signaled condition, then the `signal` operation is a no-op. When a thread leaves a monitor, either by completing its operation or by waiting on a condition, it unblocks the first thread on the urgent queue or, if the urgent queue is empty, the first thread on the entry queue, if any.

Many monitor implementations dispense with the urgent queue, or make other changes to Hoare's original definition. From the programmer's point of view, the two principal areas of variation are the semantics of the `signal` operation and the management of mutual exclusion when a thread waits inside a nested sequence of two or more monitor calls. We will return to these issues below.

Correctness for monitors depends on the notion of a *monitor invariant*. The invariant is a predicate that captures the notion that "the state of the monitor is consistent." The invariant needs to be true initially, and at monitor exit. It also needs to be true at every `wait` statement and, in a Hoare monitor, at `signal` operations as well. For our bounded buffer example, a suitable invariant would assert that `full_slots` correctly indicates the number of items in the buffer, and that those items lie in slots numbered `next_full` through `next_empty - 1` ( $\bmod \text{SIZE}$ ). Careful inspection of the code in Figure 12.16 reveals that the invariant does indeed hold initially, and that anytime we modify one of the variables mentioned in the invariant, we always modify the others accordingly before waiting, `signaling`, or returning from an entry.

Hoare defined his monitors in terms of semaphores. Conversely, it is easy to define semaphores in terms of monitors (Exercise 12.17). Together, the two definitions prove that semaphores and monitors are equally powerful: each can express all forms of synchronization expressible with the other.

### Signals as Hints and Absolutes

In general, one `signals` a condition variable when some condition on which a thread may be waiting has become true. If we want to guarantee that the condition is still true when the thread wakes up, then we need to switch to the thread as soon as the signal occurs—hence the need for the urgent queue, and the need to ensure the monitor invariant at `signal` operations. In practice, switching contexts on a `signal` tends to induce unnecessary scheduling overhead: a signaling thread seldom changes the condition associated with the `signal` during the remainder of its operation. To reduce the overhead and to eliminate the need to ensure the monitor invariant, Mesa specifies that `signals` are only *hints*: the

#### EXAMPLE 12.35

How to wait for a signal  
(hint or absolute)

language run-time system moves some waiting thread to the ready list, but the signaler retains control of the monitor, and the waiter must recheck the condition when it awakes. In effect, the standard idiom

```
if not desired_condition
 wait(condition_variable)
```

in a Hoare monitor becomes

```
while not desired_condition
 wait(condition_variable)
```

in a Mesa monitor. Modula-3 takes a similar approach. An alternative appears in Concurrent Pascal, which specifies that a `signal` operation causes an immediate return from the monitor operation in which it appears. This rule keeps overhead

## DESIGN & IMPLEMENTATION

### Monitor signal semantics

By specifying that signals are hints, instead of absolutes, Mesa and Modula-3 (and similarly Java and C#, which we consider in Section 12.3.5) avoid the need to perform an immediate context switch from a signaler to a waiting thread. They also admit simpler though less efficient implementations that lack a one-to-one correspondence between signals and thread queues, or that do not necessarily guarantee that a waiting thread will be the first to run in its monitor after the signal occurs. This approach can lead to complications, however, if we want to ensure that an appropriate thread always runs in the wake of a signal. Suppose an awakened thread rechecks its condition and discovers that it still can't run. If there may be some other thread that could run, the erroneously awakened thread may need to resignal the condition before it waits again:

```
if not desired_condition
 loop
 wait(condition_variable)
 if desired_condition
 break
 signal(condition_variable)
```

In effect, the signal “cascades” from thread to thread until some thread is able to run. (If it is possible that *no* waiting thread will be able to run, then we will need additional logic to stop the cascading when every thread has been checked.) Alternatively, the thread that makes a condition (potentially) true can use a special `broadcast` version of the `signal` operation to awaken *all* waiting threads at once. Each thread will then recheck the condition and if appropriate wait again, without the need for explicit cascading. In either case (cascading signals or broadcast), signals as hints trade potentially high overhead in the worst case for potentially low overhead in the common case and a potentially simpler implementation.

low, and also preserves invariants, but precludes algorithms in which a thread does useful work in a monitor after signaling a condition.

### Nested Monitor Calls

In most monitor languages, a `wait` in a nested sequence of monitor operations will release mutual exclusion on the innermost monitor but will leave the outer monitors locked. This situation can lead to *deadlock* if the only way for another thread to reach a corresponding `signal` operation is through the same outer monitor(s). In general, we use the term “deadlock” to describe any situation in which a collection of threads are all waiting for each other, and none of them can proceed. In this specific case, the thread that entered the outer monitor first is waiting for the second thread to execute a `signal` operation; the second thread, however, is waiting for the first to leave the monitor.

The alternative—to release exclusion on outer monitors when waiting in an inner one—was adopted by several early monitor implementations for uniprocessors, including the original implementation of Modula [Wir77a]. It has a significant semantic drawback, however: it requires that the monitor invariant hold not only at monitor exit and (perhaps) at `signal` operations, but also at any subroutine call that may result in a `wait` or (with Hoare semantics) a `signal` in a nested monitor. Such calls may not all be known to the programmer; they are certainly not syntactically distinguished in the source.

## DESIGN & IMPLEMENTATION

### The nested monitor problem

While maintaining exclusion on outer monitor(s) when waiting in an inner one may lead to deadlock with a signaling thread, releasing those outer monitors may lead to similar (if a bit more subtle) deadlocks. When a waiting thread awakens it must reacquire exclusion on both inner and outer monitors. The innermost monitor is of course available, because the matching `signal` happened there, but there is in general no way to ensure that unrelated threads will not be busy in the outer monitor(s). Moreover one of those threads may need access to the inner monitor in order to complete its work and release the outer monitor(s). If we insist that the awakened thread be the first to run in the inner monitor after the `signal`, then deadlock will result. One way to avoid this problem is to arrange for mutual exclusion across *all* the monitors of a program. This solution severely limits concurrency in multiprocessor implementations, but may be acceptable on a uniprocessor. A more general solution is addressed in Exercise 12.19.

```

buffer : record
 buf : array [1..SIZE] of data
 next_full, next_empty : integer := 1, 1
 full_slots : integer := 0

procedure insert(d : bdata)
 region buffer when full_slots < SIZE
 buf[next_empty] := d
 next_empty := next_empty mod SIZE + 1
 full_slots := 1

function remove : bdata
 region buffer when full_slots > 0
 d : bdata := buf[next_full]
 next_full := next_full mod SIZE + 1
 full_slots := 1
 return d

```

**Figure 12.17** Conditional critical regions for a bounded buffer. Boolean conditions on the region statements eliminate the need for explicit condition variables.

### 12.3.5 Conditional Critical Regions

#### EXAMPLE 12.36

Original CCR syntax

Conditional critical regions (CCRs) are another alternative to semaphores, proposed by Brinch Hansen at about the same time as monitors [Bri73]. A critical region is a syntactically delimited critical section in which code is permitted to access a *protected* variable. A *conditional* critical region also specifies a Boolean condition, which must be true before control will enter the region:

```

region protected_variable when Boolean_condition do
 ...
end region

```

No thread can access a protected variable except within a `region` statement for that variable, and any thread that reaches a `region` statement waits until the condition is true and no other thread is currently in a region for the same variable. Regions can nest, though as with nested monitor calls, the programmer needs to worry about deadlock. Figure 12.17 uses CCRs to implement a bounded buffer. ■

Conditional critical regions appear in the concurrent language Edison [Bri81], and also seem to have influenced the synchronization mechanisms of Ada 95 and Java/C#. These later languages might be said to blend the features of monitors and CCRs, albeit in different ways.

#### Synchronization in Ada 95

The principal mechanism for synchronization in Ada, introduced in Ada 83, is based on message passing; we will describe it in Section 12.4. Ada 95 augments this mechanism with a notion of *protected object*. A protected object can have

three types of methods: functions, procedures, and *entries*. Functions can only read the fields of the object; procedures and entries can read and write them. An implicit reader–writer lock on the protected object ensures that potentially conflicting operations exclude one another in time: a procedure or entry obtains exclusive access to the object; a function can operate concurrently with other functions but not with a procedure or entry.

Procedures and entries differ from one another in two important ways. First, an entry can have a Boolean expression *guard*, for which the calling task (thread) will wait before beginning execution (much as it would for the condition of a CCR). Second, an entry supports three special forms of call: *timed* calls, which abort after waiting for a specified amount of time; *conditional* calls, which execute alternative code if the call cannot proceed immediately; and *asynchronous* calls, which begin executing alternative code immediately but abort it if the call is able to proceed before the alternative completes.

In comparison to the conditions of CCRs, the guards on entries of protected objects in Ada 95 admit a more efficient implementation, because they do not have to be evaluated in the context of the calling thread. Moreover, because all guards are gathered together in the definition of the protected object, the compiler can generate code to test them as a group as efficiently as possible, in a manner suggested by Kessels [Kes77]. Though an Ada task cannot wait on a condition in the middle of an entry (only at the beginning), it can *requeue* itself on another entry, achieving much the same effect. Ada 95 code for a bounded buffer would closely resemble the pseudocode of Figure 12.17; we leave the details to Exercise 12.21.

### Synchronization in Java

#### EXAMPLE 12.37

Synchronized statement  
in Java

### DESIGN & IMPLEMENTATION

#### Conditional critical regions

Conditional critical regions avoid the question of signal semantics because they use explicit Boolean conditions instead of condition variables and because conditions can be awaited only at the beginning of critical regions. At the same time, they introduce potentially significant inefficiency. In the general case, the code used to exit a conditional critical region must tentatively resume each waiting thread, allowing that thread to recheck its condition in its own referencing environment. Optimizations are possible in certain special cases (e.g., for conditions that depend only on global variables, or that consist of only a single Boolean variable), but in the worst case it may be necessary to perform context switches in and out of every waiting thread on every exit from a region.

```

synchronized (my_shared_obj) {
 ...
 // critical section
}

```

All executions of `synchronized` statements that refer to the same shared object exclude one another in time. Synchronized statements that refer to different objects may proceed concurrently. As a form of syntactic sugar, a method of a class may be prefixed with the `synchronized` keyword, in which case the body of the method is considered to have been surrounded by an implicit `synchronized (this)` statement. Invocations of nonsynchronized methods of a shared object—and direct accesses to public fields—can proceed concurrently with each other, or with a `synchronized` statement or method.

Within a `synchronized` statement or method, a thread can suspend itself by calling the predefined method `wait`. `Wait` has no arguments in Java: the core language does not distinguish among the different reasons why threads may be suspended on a given object (the `java.util.concurrent` library, which became standard with Java 5, does provide a mechanism for multiple conditions; more on this below). Like Mesa, Java allows a thread to be awoken for spurious reasons; programs must therefore embed the use of `wait` within a condition-testing loop:

```

while (!condition) {
 wait();
}

```

A thread that calls the `wait` method of an object releases the object's lock. With nested `synchronized` statements, however, or with nested calls to `synchronized` methods, the thread does *not* release locks on any other objects.

To resume a thread that is suspended on a given object, some other thread must execute the predefined method `notify` from within a `synchronized` statement or method that refers to the same object. Like `wait`, `notify` has no arguments. In response to a `notify` call, the language run-time system picks an arbitrary thread suspended on the object and makes it runnable. If there are no such threads then the `notify` is a no-op. As in Mesa, it may sometimes be appropriate to awaken *all* threads waiting in a given object. Java provides a built-in `notifyAll` method for this purpose.

If threads are waiting for more than one condition (i.e., if their waits are embedded in dissimilar loops), there is no guarantee that the “right” thread will awaken. To ensure that an appropriate thread does wake up, the programmer may choose to use `notifyAll` instead of `notify`. To ensure that only *one* thread continues after wakeup, the first thread to discover that its condition has been satisfied must modify the state of the object in such a way that other awakened threads, when they get to run, will simply go back to sleep. Unfortunately, since all waiting threads will end up reevaluating their conditions every time one of them can run, this “solution” to the multiple-condition problem can be prohibitively expensive.

#### EXAMPLE 12.38

Notify as hint in Java

The mechanisms for synchronization in C# are similar to the Java mechanisms just described. The C# `lock` statement is similar to Java's `synchronized`. It cannot be used to label a method, but a similar effect can be achieved (a bit more clumsily) by specifying a `Synchronized attribute` for the method. The methods `Pulse` and `PulseAll` are used instead of `signal` and `signalAll`.

**EXAMPLE 12.39**

Lock variables in Java 5

**Lock Variables** In C# and in versions of Java prior to Java 5, programmers concerned with efficiency must generally look for algorithms in which threads are never waiting for more than one condition within a given object at a given time. The `java.util.concurrent` package, released in 2004, provides a more general solution. As an alternative to `synchronized` statements and methods, the programmer may now create explicit Lock variables. Code that might once have been written

```
synchronized (my_shared_obj) {
 ... // critical section
}
```

may now be written

```
Lock l = new ReentrantLock();
l.lock();
try {
 ... // critical section
} finally {
 l.unlock();
}
```

**DESIGN & IMPLEMENTATION****Condition variables in Java**

As illustrated by Mesa and Java, the distinction between monitors and CCRs is somewhat blurry. It turns out to be possible (see Exercise 12.20) to solve completely general synchronization problems in such a way that for every protected object there is only one Boolean condition on which threads ever spin. The solutions, however, may not be pretty: they amount to low-level use of semaphores, without the implicit mutual exclusion of `synchronized` statements and methods. For programs that are naturally expressed with multiple conditions, Java's basic synchronization mechanism (and the similar mechanism in C#) may force the programmer to choose between elegance and efficiency. The concurrency enhancements of Java 5 are a deliberate attempt to lessen this dilemma: Lock variables retain the distinction between mutual exclusion and condition synchronization characteristic of both monitors and CCRs, while allowing the programmer to partition waiting threads into equivalence classes that can be awoken independently. By varying the fineness of the partition the programmer can choose essentially any point of the spectrum between the simplicity of CCRs and the efficiency of Hoare-style monitors. Exercises 12.22 through 12.24 explore this issue further using bounded buffers as a running example.

**EXAMPLE 12.40**

Multiple Conditions in Java 5

A similar interface supports reader–writer locks.

Like semaphores, Java Lock variables lack the implicit release at end of scope associated with `synchronized` statements and methods. The need for an explicit release introduces a potential source of bugs, but allows programmers to create algorithms in which locks are acquired and released in non-LIFO order (see Example 12.12). In a manner reminiscent of the `timed entry` calls of Ada 95, Java Lock variables also support a `tryLock` method, which acquires the lock only if it is available immediately or within an optionally specified timeout interval (a Boolean return value indicates whether the attempt was successful). Finally, a Lock variable may have an arbitrary number of associated Condition variables, making it easy to write algorithms in which threads wait for multiple conditions, without resorting to `notifyAll`:

```
Condition c1 = l.newCondition();
Condition c2 = l.newCondition();
...
c1.await();
...
c2.signal();
```

Java objects that use only `synchronized` methods (no locks or `synchronized` statements) closely resemble Mesa monitors in which there is a limit of one condition variable per monitor. By the same token, a `synchronized` statement in Java that begins with a `wait` in a loop resembles a CCR in which the retesting of conditions has been made explicit. Because `notify` also is explicit, a Java implementation need not reevaluate conditions on every exit from a critical section—only those in which a `notify` occurs.

### 12.3.6 Implicit Synchronization

In several shared-memory languages, the operations that threads can perform on shared data are restricted in such a way that synchronization can be implicit in the operations themselves, rather than appearing as separate, explicit operations. We have seen one example of implicit synchronization already: the `forall` loop of HPF and Fortran 95 (Example 12.9). Separate iterations of a `forall` loop proceed concurrently, semantically in lock-step with each other: each iteration reads all data used in its instance of the first assignment statement before any iteration updates its instance of the left-hand side. The left-hand side updates in turn occur before any iteration reads the data used in its instance of the second assignment statement, and so on. Compilation of `forall` loops for vector machines, while far from trivial, is more or less straightforward. On a more conventional multiprocessor, however, good performance usually depends on high-quality *dependence analysis*, which allows the compiler to identify situations in which statements within a loop do not in fact depend on one another, and can proceed without synchronization.

Dependence analysis plays a crucial role in other languages as well. In Section 6.6.1 we mentioned the purely functional languages Sisal and pH (recall that iterative constructs in these languages are syntactic sugar for tail recursion). Because Sisal and pH are side-effect-free, their constructs can be evaluated in any order, or concurrently, as long as no construct attempts to use a value that has yet to be computed. The Sisal implementation developed at Lawrence Livermore National Lab uses extensive compiler analysis to identify promising constructs for parallel execution. It also employs tags on data objects that indicate whether the object's value has been computed yet. When the compiler is unable to guarantee that a value will have been computed by the time it is needed at run time, the generated code uses tag bits for synchronization, spinning or blocking until they are properly set. Sisal's developers claim [Can92] that their language and compiler rival parallel Fortran in performance.

**EXAMPLE 12.41**

Future construct in Multilisp

In a less ambitious vein, the Multilisp [Hal85, MKH91] dialect of Scheme allows the programmer to enclose any function evaluation in a special `future` construct:

```
(future (my-function my-args))
```

In a purely functional program, `future` is semantically neutral: program behavior will be exactly the same as if `(my-function my-args)` had appeared without the surrounding call. In the implementation, however, `future` arranges for the embedded function to be evaluated by a separate thread of control. The parent thread continues to execute until it actually tries to use the return value of `my-function`, at which point it waits for execution of the `future` to complete. If two or more arguments to a function are enclosed in `futures`, then evaluation of the arguments can proceed in parallel:

```
(parent-func (future (child-1 args-1)) (future (child-2 args-2)))
```

There are no additional synchronization mechanisms: `future` itself is Multilisp's only addition to Scheme.

Multilisp, Sisal, and pH employ the same basic idea: concurrent evaluation of functions in a language that is (at least mostly) side-effect-free. The Sisal and pH compilers attempt to find code fragments that can profitably be executed in parallel; the Multilisp programmer must identify them explicitly. In some ways the `future` construct of Multilisp resembles the built-in `delay` and `force` of Scheme (Section 6.6.2). Where `future` supports concurrency, however, `delay` supports lazy evaluation: it defers evaluation of its embedded function until the return value is known to be needed. Any use of a `delayed` expression in Scheme must be surrounded by `force`. By contrast, synchronization on a `future` is implicit: there is no analog of `force`.

Several researchers have noted that the backtracking search of logic languages such as Prolog is also amenable to parallelization. Two strategies are possible. The first is to pursue in parallel the subgoals found in the right-hand side of a rule. This strategy is known as *AND parallelism*. The fact that variables in logic, once initialized, are never subsequently modified ensures that parallel

branches of an AND cannot interfere with one another. The second strategy is known as *OR parallelism*; it pursues alternative resolutions in parallel. Because they will generally employ different unifications, branches of an OR must use separate copies of their variables. In a search tree such as that of Figure 11.1 (page 567), AND parallelism and OR parallelism create new threads at alternating levels.

OR parallelism is *speculative*: since success is required on only one branch, work performed on other branches is in some sense wasted. OR parallelism works well, however, when a goal cannot be satisfied (in which case the entire tree must be searched) or when there is high variance in the amount of execution time required to satisfy a goal in different ways (in which case exploring several branches at once reduces the expected time to find the first solution). Both AND and OR parallelism are problematic in Prolog, because they fail to adhere to the deterministic search order required by language semantics.

Some of the ideas embodied in concurrent functional languages can be adapted to imperative languages as well. CC++ [Fos95], for example, is a concurrent extension to C++ in which synchronization is implicit in the use of *single-assignment* variables. To declare a single-assignment variable, the CC++ programmer prepends the keyword `synch` to an ordinary variable declaration. The value of a `synch` variable is initially undefined. A thread that attempts to read the variable will wait until it is assigned a value by some other thread. It is a runtime error for any thread to attempt to assign to a `synch` variable that already has a value.

## DESIGN & IMPLEMENTATION

### Side-effect freedom and implicit synchronization

In a partially imperative Multilisp program, the programmer must take care to make sure that concurrent execution of futures will not compromise program correctness. The expression `(f1 (future (f2 args2)) (future (f3 args3)))` may produce unpredictable behavior if the evaluations of `f2` and `f3` depend on one another, or if the evaluation of `f1` depends on any aspect of `f2` and `f3` other than their return values. Such behavior may be very difficult to debug. Sisal and pH avoid the problem by permitting only side-effect-free programs.

In a key sense, Sisal and pH are ideally suited to parallel execution: they eliminate all artificial connections—all anti- and output dependences (Section 15.6)—among expressions: all that remains is the actual *data flow*. Two principal barriers to performance remain: (1) the standard challenges of efficient code generation for functional programs (Section 10.7), and (2) the need to identify which potentially parallel code fragments are large enough and independent enough to merit the overhead of thread creation and implicit synchronization.

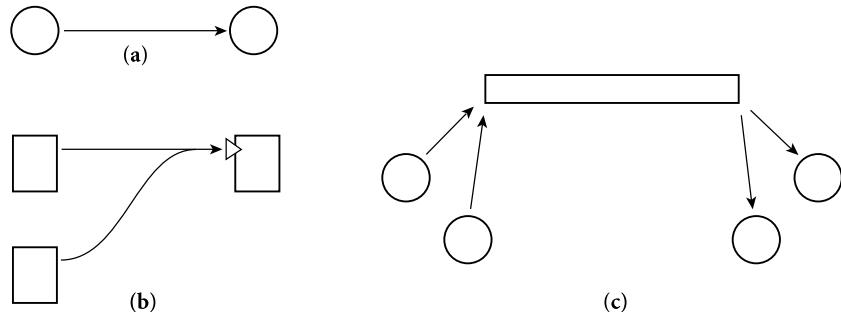
In a similar vein, Linda [ACG86] is a set of concurrent programming mechanisms that can be embedded into almost any imperative language. It consists of a set of subroutines that manipulate a shared abstraction called the *tuple space*. The elements of tuple space resemble the tuples of ML (Example ⑩ 7.111) and Python (Example 13.73), except that they have single assignment semantics, and are accessed associatively by content, rather than by name. The `in` procedure adds a tuple to the tuple space. The `out` procedure extracts a tuple that matches a specified *pattern*, waiting if no such tuple currently exists. The `read` procedure is a nondestructive `out`. A special form of `in` forks a concurrent thread to calculate the value to be inserted, much like a `future` in Multilisp. All three subroutines can be supported as ordinary library calls, but performance is substantially better when using a specially designed compiler that generates optimized code for commonly occurring patterns of tuple space operations.

A few multiprocessors, including the Denelcor HEP [Jor85] and the Tera machine [ACC<sup>+</sup>90], provide hardware support for single-assignment variables in the form of so-called *full-empty* bits. Each memory location contains a bit that indicates whether the variable in that location has been initialized. Any attempt to access an uninitialized variable stalls the current processor, causing it to switch contexts (in hardware) to another thread of control.

### CHECK YOUR UNDERSTANDING

---

38. What is a *monitor*? How do monitor *condition variables* differ from semaphores?
39. Explain the difference between treating monitor signals as *hints* and treating them as *absolutes*.
40. What is a *monitor invariant*? Under what circumstances must it be guaranteed to hold?
41. Describe the *nested monitor problem* and some potential solutions.
42. What is *deadlock*?
43. What is a *conditional critical region*? How does it differ from a monitor?
44. Summarize the synchronization mechanisms of Ada 95, Java, and C#. Contrast them with one another, and with monitors and conditional critical regions. Be sure to explain the features added to Java 5.
45. Describe the semantics of the HPF/Fortran 95 `forall` loop.
46. Why might pure functional languages be said to provide a particularly attractive notation for concurrent programming?



**Figure 12.18** Three common schemes to name communication partners. In (a), processes name each other explicitly. In (b), senders name an *input port* of a receiver. The port may be called an *entry* or an *operation*. The receiver is typically a module with one or more threads inside. In (c), senders and receivers both name an independent *channel abstraction*, which may be called a *connection* or a *mailbox*.

47. Explain the difference between AND *parallelism* and OR *parallelism* in Prolog.
48. What are *single-assignment variables*? In what languages do they appear?

## 12.4 Message Passing

While shared-memory concurrent programming is common on small-scale multiprocessors, most concurrent programming on large multicompilers and networks is currently based on messages. In Sections 12.4.1 through 12.4.3 we consider three principal issues in message-based computing: naming, sending, and receiving. In Section 12.4.4 we look more closely at one particular combination of send and receive semantics, namely remote procedure call. Most of our examples will be drawn from the Ada, Occam, and SR programming languages, the Java network library, and the PVM and MPI library packages.

### 12.4.1 Naming Communication Partners

#### EXAMPLE 12.42

Naming processes, ports, and entries

To send or receive a message, one must generally specify where to send it to or where to receive it from: communication partners need names for (or references to) one another. Names may refer directly to a thread or process. Alternatively, they may refer to an *entry* or *port* of a module, or to some sort of *socket* or *channel* abstraction. We illustrate these options in Figure 12.18. ■

The first naming option—addressing messages to processes—appears in Hoare’s original CSP proposal and in PVM and MPI. Each PVM or MPI process has a unique *id* (an integer), and each *send* or *receive* operation specifies the *id* of the communication partner. MPI implementations are required to be

reentrant; a process can safely be divided into multiple threads, each of which can send or receive messages on the process's behalf. PVM has hidden state variables that are not automatically synchronized, making threaded PVM programs problematic.

**EXAMPLE 12.43**

Entry calls in Ada

The second naming option—addressing messages to ports—appears in Ada. An Ada *entry call* of the form `t.foo(args)` sends a message to the entry named `foo` in task (thread) `t` (`t` may be either a task name or the name of a variable whose value is a pointer to a task). As we saw in Section 12.2.3, an Ada task resembles a module; its entries resemble subroutine headers nested directly inside the task. A task receives a message that has been sent to one of its entries by executing an `accept` statement (to be discussed in Section 12.4.3). Every entry belongs to exactly one task; all messages sent to the same entry must be received by that one task.

The third naming option—addressing messages to channels—appears in Occam (though not in CSP). Channel declarations are supported with the built-in `CHAN` and `CALL` types:

```
CHAN OF BYTE stream :
CALL lookup(RESULT [36]BYTE name, VAL INT ssn) :
```

These declarations specify a one-directional channel named `stream` that carries messages of type `BYTE` and a two-directional channel named `lookup` that carries requests containing an integer named `ssn` and replies containing a 36-byte string named `name`. `CALL` channels are syntactic sugar for a pair of `CHAN` channels, one in each direction. To send a message on a `CHAN` channel, an Occam thread uses a special “exclamation point” operator:

```
stream ! 'x'
```

To send a message (and receive a reply) on a `CALL` channel, a thread uses syntax that resembles a subroutine call:

```
lookup(name, 123456789)
```

We noted in our coverage of parallel loops (page 606) that language rules in Occam prohibit concurrent threads from making conflicting accesses to the same variable. For channels, the basic rule is that exactly one thread may send to a channel, and exactly one may receive from it. (For `CALL` channels, exactly one thread may send requests, and exactly one may accept them and send replies.) These rules are relaxed in Occam 3 to permit `SHARED` channels, which provide a mutual exclusion mechanism. Only one thread may accept requests over a `SHARED CALL` channel, but multiple threads may send them. In a similar vein, multiple threads may `CLAIM` a set of `CHAN` channels for exclusive use in a critical section, but only one thread may `GRANT` those channels; it serves as the other party for every message sent or received.

In SR and the Internet libraries of Java we see combinations of our naming options. An SR program executes on a collection of one or more *virtual machines*, each of which has a separate address space, and may be implemented on a

separate node of a network. Within a virtual machine, messages are sent to (and received from) a channel-like abstraction called an op. Unlike an Occam channel, an SR op has no restrictions on the number or identity of sending and receiving threads: any thread that can see an op under the usual lexical scoping rules can send to it or receive from it. A receive operation must name its op explicitly; a send operation may do so also, or it may use a *capability* variable. A capability in SR is like a pointer to an op, except that pointers work only within a given virtual machine, while capabilities work across the boundaries between them. Aside from start-up parameters and possibly I/O, capabilities provide the *only* means of communicating among separate virtual machines. At the outermost level, then, an SR program can be seen as having a port-like naming scheme: messages are sent (via capabilities) to ops of virtual machines, within which they may potentially be received by any local thread.

Java's standard `java.net` library provides two styles of message passing, corresponding to the UDP and TCP Internet protocols. UDP is the simpler of the two. It is a *datagram* protocol, meaning that each message is sent to its destination independently and unreliably. The network software will attempt to deliver it but makes no guarantees. Moreover two messages sent to the same destination (assuming they both arrive) may arrive in either order. UDP messages use port-based naming (Figure 12.18b): each message is sent to a specific *Internet address* and *port number*.<sup>4</sup> The TCP protocol also uses port-based naming, but only for the purpose of establishing *connections* (Figure 12.18c), which it then uses for all subsequent communication. Connections deliver messages reliably and in order.

#### EXAMPLE 12.45

##### Datagram messages in Java

To send or receive UDP messages, a Java thread must create a *datagram socket*:

```
DatagramSocket my_socket = new DatagramSocket(port_id);
```

The parameter of the `DatagramSocket` constructor is optional; if it is not specified, the operating system will choose an available port. Typically servers specify a port and clients allow the OS to choose. To send a UDP message, a thread says

```
DatagramPacket my_msg = new DatagramPacket(buf, len, addr, port);
... // initialize message
my_socket.send(my_msg);
```

The parameters to the `DatagramPacket` constructor specify an array of bytes `buf`, its length `len`, and the Internet address and port of the receiver. Receiving is symmetric:

```
my_socket.receive(my_msg);
... // parse content of my_msg
```

---

<sup>4</sup> Every publicly visible machine on the Internet has its own unique address. Though a transition to 128-bit addresses has been underway for some time, as of 2005 most addresses are still 32-bit integers, usually printed as four period-separated fields (e.g., 192.5.54.209). Internet name servers translate symbolic names (e.g., `gate.cs.rochester.edu`) into numeric addresses. Port numbers are also integers, but are local to a given Internet address. Ports 1024 through 4999 are generally available for application programs; larger and smaller numbers are reserved for servers.

**EXAMPLE 12.46**

Connection-based  
messages in Java

For TCP communication, a server typically “listens” on a port to which clients send requests to establish a connection:

```
ServerSocket my_server_socket = new ServerSocket(port_id);
Socket client_connection = my_server_socket.accept();
```

The `accept` operation blocks until the server receives a connection request from a client. Typically a server will immediately fork a new thread to communicate with the client; the parent thread loops back to wait for another connection with `accept`.

A client sends a connection request by passing the server’s symbolic name and port number to the `Socket` constructor:

```
Socket server_connection = new Socket(host_name, port_id);
```

Once a connection has been created, a client and server in Java typically call methods of the `Socket` class to create input and output streams, which support all of the standard Java mechanisms for text I/O (Section ⑩ 7.9.3):

```
BufferedReader in = new BufferedReader(
 new InputStreamReader(client_connection.getInputStream()));
PrintStream out =
 new PrintStream(client_connection.getOutputStream());
// This is in the server; the client would make streams out
// of server_connection.
...
String s = in.readLine();
out.println("Hi, Mom\n");
```

Among all the message-passing mechanisms we have considered, datagrams are the only one that does not provide some sort of *ordering* constraint. In general, most message-passing systems guarantee that messages sent over the same “communication path” arrive in order. When naming processes explicitly, a path links a single sender to a single receiver. All messages from that sender to that receiver arrive in the order sent. When naming ports, a path links an arbitrary number of senders to a single receiver (though as we saw in SR, if a receiver is a complex entity like a virtual machine, it may have many threads inside). Messages that arrive at a port in a given order will be seen by receivers in that order. Note, however, that while messages from the same sender will arrive at a port in order, messages from *different* senders may arrive in different orders.<sup>5</sup> When naming channels, a path links all the senders that can use the channel to all the receivers

---

**5** Suppose, for example, that process *A* sends a message to port *p* of process *B* and then sends a message to process *C*, while process *C* first receives the message from *A* and then sends its own message to port *p* of *B*. If messages are sent over a network with internal delays, and if *A* is allowed to send its message to *C* before its first message has reached port *p*, then it is possible for *B* to hear from *C* before it hears from *A*. This apparent reversal of ordering could easily happen on the Internet, for example, if the message from *A* to *B* traverses a satellite link, while the messages from *A* to *C* and from *C* to *B* use ocean-floor cables.

that can use it. A Java TCP connection has a single OS process at each end, but there may be many threads inside, each of which can use its process's end of the connection. An SR op can be used by any thread to which it is visible. In both cases, the channel functions as a queue: send (enqueue) and receive (dequeue) operations are ordered, so that everything is received in the order it was sent.

### 12.4.2 Sending

One of the most important issues to be addressed when designing a send operation is the extent to which it may block the caller: once a thread has initiated a send operation, when is it allowed to continue execution? Blocking can serve at least three purposes.

*resource management:* A sending thread should not modify outgoing data until the underlying system has copied the old values to a safe location. Most systems block the sender until a point at which it can safely modify its data without danger of corrupting the outgoing message.

*failure semantics:* Particularly when communicating over a long-distance network, message passing is more error-prone than most other aspects of computing. Many systems block a sender until they are able to guarantee that the message will be delivered without error.

*return parameters:* In many cases a message constitutes a *request*, for which a *reply* is expected. Many systems block a sender until a reply has been received.

When deciding how long to block, we must consider synchronization semantics, buffering requirements, and the reporting of run-time errors.

#### Synchronization Semantics

On its way from a sender to a receiver, a message may pass through many intermediate steps, particularly if traversing the Internet. It first descends through several layers of software on the sender's machine, then through a potentially large number of intermediate machines, and finally up through several layers of software on the receiver's machine. We could imagine unblocking the sender after any of these steps, but most of the options would be indistinguishable in terms of user-level program behavior. If we assume for the moment that a message-passing system can always find buffer space to hold an outgoing message, then our three rationales for delay suggest three principal semantic options.

#### EXAMPLE 12.47

Three main options for send semantics

*no-wait send:* The sender does not block for more than a small, bounded period of time. The message-passing implementation copies the message to a safe location and takes responsibility for its delivery.

*synchronization send:* The sender waits until its message has been received.

*remote-invocation send:* The sender waits until it receives a reply.

These three alternatives are illustrated in Figure 12.19.

No-wait send appears in SR and in the Java Internet library. Synchronization send appears in Occam. Remote-invocation send appears in SR, Occam, and Ada. PVM and MPI provide an implementation-oriented hybrid of no-wait send and synchronization send: a send operation blocks until the data in the outgoing message can safely be modified. In implementations that do their own internal buffering, this rule amounts to no-wait send. In other implementations, it amounts to synchronization send. PVM programs must be written to cope with the latter, more restrictive option. In MPI, the programmer has the option, if desired, to insist on no-wait send or synchronization send; performance may suffer on some systems if the request is different from the default.

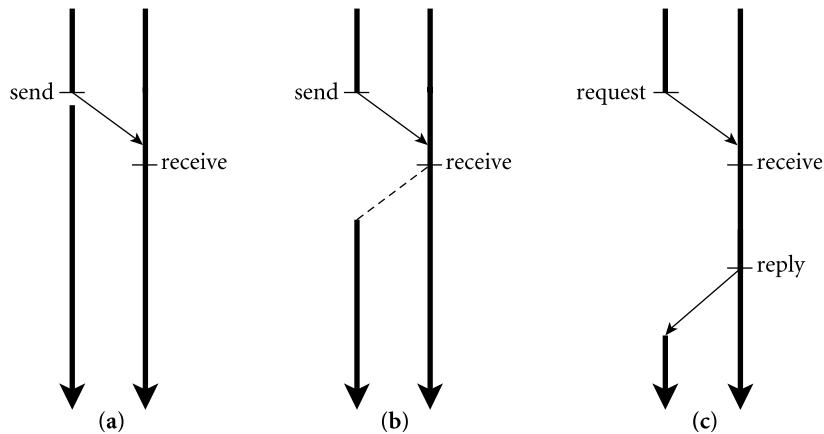
### **Buffering**

In practice, unfortunately, no message-passing system can provide a version of send that never waits (unless of course it simply throws some messages away). If we imagine a thread that sits in a loop sending messages to a thread that never receives them, we quickly see that unlimited amounts of buffer space would be required. At some point, any implementation must be prepared to block an overactive sender, to keep it from overwhelming the system. Such blocking is a form of *backpressure*. Milder backpressure can also be applied by reducing a thread's scheduling priority or by changing parameters of the underlying message delivery mechanism.

## **DESIGN & IMPLEMENTATION**

### **The semantic impact of implementation issues**

The inability to buffer unlimited amounts of data, or to report errors synchronously to a sender that has continued execution, are only the most recent of the many examples we have seen in which pragmatic implementation issues may restrict the language semantics available to the programmer. Other examples include limitations on the length of source lines or variable names (Section 2.1.1); limits on the memory available for data (whether global, stack, or heap allocated) and for recursive function evaluation (Section 3.2); the lack of ranges in case statement labels (Section 6.4.2); in reverse, downto, and constant step sizes for for loops (Section 6.5.1); limits on set universe size (to accommodate bit vectors—Section 7.6); limited procedure nesting (to accommodate displays—Section 8.1); the fixed size requirement for opaque exports in Modula-2 (Section 9.2.1); and the lack of nested threads or of unrestricted arms on a cobegin statement (to avoid the need for cactus stacks—Section 8.6.1 and the sidebar on page 606). Some of these limitations are reflected in the formal semantics of the language. Others (generally those that vary most from one implementation to another) restrict the set of semantically valid programs that the system will run correctly.



**Figure 12.19** Synchronization semantics for the `send` operation: no-wait `send` (a), synchronization `send` (b), and remote-invocation `send` (c). In each diagram we have assumed that the original message arrives before the receiver executes its `receive` operation; this need not in general be the case.

#### EXAMPLE 12.48

Buffering-dependent deadlock

For any fixed amount of buffer space, it is possible to design a program that requires a larger amount of space to run correctly. Imagine, for example, that the message-passing system is able to buffer  $n$  messages on a given communication path. Now imagine a program in which  $A$  sends  $n + 1$  messages to  $B$ , followed by one message to  $C$ .  $C$  then sends one message to  $B$  on a different communication path. Finally,  $B$  insists on receiving the message from  $C$  before receiving the messages from  $A$ . If  $A$  blocks after message  $n$ , implementation-dependent deadlock will result. The best that an implementation can do is to provide a sufficiently large amount of space that realistic applications are unlikely to find the limit to be a problem. ■

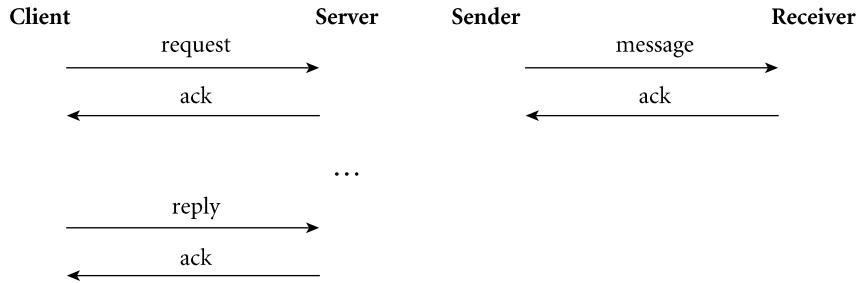
For synchronization `send` and remote-invocation `send`, buffer space is not generally a problem: the total amount of space required for messages is bounded by the number of threads, and there are already likely to be limits on how many threads a program can create. A thread that sends a reply message can always be permitted to proceed: we know that we shall be able to reuse the buffer space quickly, because the thread that sent the request is already waiting for the reply.

#### Error Reporting

#### EXAMPLE 12.49

Acknowledgments

If the underlying message-passing system is unreliable, a language or library will typically employ *acknowledgment* messages to verify successful transmission (Figure 12.20). If an acknowledgment is not received within a reasonable amount of time, the implementation will typically resend. If several attempts fail to elicit an acknowledgment, an error will be reported. ■



**Figure 12.20 Acknowledgment messages for error detection.** In the absence of piggybacking, remote-invocation `send` (left) may require four underlying messages; synchronization `send` (right) may require two.

As long as the sender of a message is blocked, errors that occur in attempting to deliver a message can be reflected back as exceptions, or as status information in result parameters or global variables. Once a sender has continued, there is no obvious way in which to report any problems that arise. Like limits on message buffering, this dilemma poses semantic problems for no-wait `send`. For UDP, the solution is to state that messages are unreliable: if something goes wrong, the message is simply lost, silently. For TCP, the “solution” is to state that only “catastrophic” errors will cause a message to be lost, in which case the connection will become unusable and future calls will fail immediately. An even more drastic approach is taken in MPI: certain implementation-specific errors may be detected and handled at run time, but in general if a message cannot be delivered then the program as a whole is considered to have failed. PVM provides a *notification* mechanism that will send a message to a previously designated process in the event of a node or process failure. The designated process can then perform cleanup actions such as aborting any related, dependent processes, or starting new processes to pick up the work of those that failed.

#### ***Emulation of Alternatives***

All three varieties of `send` can be emulated by the others. To obtain the effect of remote-invocation `send`, a thread can follow a no-wait `send` of a request with a `receive` of the reply. Similar code will allow us to emulate remote-invocation `send` using synchronization `send`. To obtain the effect of synchronization `send`, a thread can follow a no-wait `send` with a `receive` of a high-level acknowledgment, which the receiver will send immediately upon receipt of the original message. To obtain the effect of synchronization `send` using remote-invocation `send`, a thread that receives a request can simply reply immediately, with no return parameters.

To obtain the effect of no-wait `send` using synchronization `send` or remote-invocation `send`, we must interpose a buffer process (the message-passing analogue of our shared-memory bounded buffer) that replies immediately to “senders” or “receivers” whenever possible. The space available in the buffer

process makes explicit the resource limitations that are always present below the surface in implementations of no-wait send.

### Syntax and Language Integration

In the preceding emulation examples we assumed a library-based implementation of message passing. Because `send`, `receive`, `accept`, and so on are ordinary subroutines in such an implementation, they take a fixed, static number of parameters, two of which typically specify the location and size of the message to be sent. To send a message containing values held in more than one program variable, the programmer must explicitly *gather*, or *marshal*, those values into the fields of a record. On the receiving end, the programmer must *scatter* (*unmarshal*) the values back into program variables. By contrast, a concurrent programming language can provide message-passing operations whose “argument” lists can include an arbitrary number of values to be sent. Moreover, the compiler can arrange to perform type checking on those values, using techniques similar to those employed for subroutine linkage across compilation units (to be described in Section 14.6.2). Finally, as we shall see in Section 12.4.3, an explicitly concurrent language can employ non-procedure-call syntax—for example, to couple a remote-invocation `accept` and `reply` in such a way that the `reply` doesn’t have to explicitly identify the `accept` to which it corresponds.

## DESIGN & IMPLEMENTATION

### Emulation and efficiency

Unfortunately, user-level emulations of alternative `send` semantics are seldom as efficient as optimized implementations using the underlying primitives. Suppose for example that we wish to use remote-invocation `send` to emulate synchronization `send`. Suppose further that our implementation of remote-invocation `send` is built on top of network software that needs acknowledgments to verify message delivery. After sending a reply, the server’s run-time system will wait for an acknowledgment from the client. If a server thread can work for an arbitrary amount of time before sending a reply, then the run-time system will need to send separate acknowledgments for the request and the reply. If a programmer uses this implementation of remote-invocation `send` to emulate synchronization `send`, then the underlying network may end up transmitting a total of four messages (more if there are any transmission errors). By contrast, a “native” implementation of synchronization `send` would require only two underlying messages. In some cases the run-time system for remote-invocation `send` may be able to delay transmission of the first acknowledgment long enough to “piggyback” it on the subsequent reply if there is one; in this case an emulation of synchronization `send` may transmit three underlying messages instead of only two. We consider the efficiency of emulations further in Exercise 12.33 and Exploration 12.38.

### 12.4.3 Receiving

Probably the most important dimension on which to categorize mechanisms for receiving messages is the distinction between explicit `receive` operations and the *implicit* receipt described in Section 12.2.3 (page 611). Among the languages and systems we have been using as examples, only SR provides implicit receipt (some RPC systems also provide it, as we shall see in Section 12.4.4).

With implicit receipt, every message that arrives at a given port (or over a given channel) will create a new thread of control, subject to resource limitations (any implementation will have to stall incoming requests when the number of threads grows too large). With explicit receipt, a message must be queued until some already-existing thread indicates a willingness to receive it. At any given point in time there may be a potentially large number of messages waiting to be received. Most languages and libraries with explicit receipt allow a thread to exercise some sort of *selectivity* with respect to which messages it wants to consider.

In PVM and MPI, every message includes the `id` of the process that sent it, together with an integer `tag` specified by the sender. A `receive` operation specifies a desired sender `id` and message tag. Only matching messages will be received. In many cases receivers specify “wild cards” for the sender `id` and/or message tag, allowing any of a variety of messages to be received. Special versions of `receive` also allow a process to test (without blocking) to see if a message of a particular type is currently available (this operation is known as *polling*) or to “time out” and continue if a matching message cannot be received within a specified interval of time.

Because they are languages instead of library packages, Ada, Occam, and SR are able to use special, non-procedure-call syntax for selective message receipt. Moreover because messages are built into the naming and typing system, these languages are able to receive selectively on the basis of port/channel names and parameters, rather than the more primitive notion of tags. In all three languages, the selective `receive` construct is a special form of *guarded command*, as described in Section ⑩ 6.7.

Figure 12.21 contains code for a bounded buffer in Ada 83. Here an active “manager” thread executes a `select` statement inside a loop. (Recall that it is also possible to write a bounded buffer in Ada using *protected objects*, without a manager thread, as described in Section 12.3.2.) The Ada `accept` statement receives the `in` and `in out` parameters (Section 8.3.1) of a remote invocation request. At the matching `end`, `accept` returns the `in out` and `out` parameters as a reply message. A client task would communicate with the bounded buffer using an *entry call*:

```
-- producer: -- consumer:
buffer.insert(3); buffer.remove(x);
```

---

**EXAMPLE 12.50**

Bounded buffer in Ada 83

```

task buffer is
 entry insert(d : in bdata);
 entry remove(d : out bdata);
end buffer;

task body buffer is
 SIZE : constant integer := 10;
 subtype index is integer range 1..SIZE;
 buf : array (index) of bdata;
 next_empty, next_full : index := 1;
 full_slots : integer range 0..SIZE := 0;
begin
 loop
 select
 when full_slots < SIZE =>
 accept insert(d : in bdata) do
 buf(next_empty) := d;
 end;
 next_empty := next_empty mod SIZE + 1;
 full_slots := full_slots + 1;
 or
 when full_slots > 0 =>
 accept remove(d : out bdata) do
 d := buf(next_full);
 end;
 next_full := next_full mod SIZE + 1;
 full_slots := full_slots - 1;
 end select;
 end loop;
end buffer;

```

**Figure 12.21** Bounded buffer in Ada, with an explicit manager task.

The `select` statement in our buffer example has two arms. The first arm may be selected when the buffer is not full and there is an available `insert` request; the second arm may be selected when the buffer is not empty and there is an available `remove` request. Selection among arms is a two-step process: first the guards (when expressions) are evaluated, then for any that are true the subsequent `accept` statements are considered to see if a message is available. (The guard in front of an `accept` is optional; if missing it behaves like `when true =>`.) If both of the guards in our example are true (the buffer is partly full) and both kinds of messages are available, then either arm of the statement may be executed, at the discretion of the implementation. (For a discussion of issues of *fairness* in the choice among true guards, see the sidebar on page 76-CD.) ■

#### EXAMPLE 12.51

Timeout and distributed termination

Every `select` statement must have at least one arm beginning with `accept` (and optionally `when`). In addition, it may have three other types of arms:

```

when condition => delay how_long
other_statements
...
or when condition => terminate
...
else ...

```

A `delay` arm may be selected if no other arm becomes selectable within `how_long` seconds. (Ada implementations are required to support delays as long as one day or as short as 20 ms.) A `terminate` arm may be selected only if all potential communication partners have already terminated or are likewise stuck in `select` statements with `terminate` arms. Selection of the arm causes the task that was executing the `select` statement to terminate. An `else` arm, if present, will be selected when none of the guards are true or when no `accept` statement can be executed immediately. A `select` statement with an `else` arm is not permitted to have any `delay` arms. In practice, one would probably want to include a `terminate` arm in the `select` statement of a manager-style bounded buffer.

Occam's equivalent of `select` is known as `ALT`. As in Ada, the choice among arms can be based both on Boolean conditions and on the availability of messages. (One minor difference: Occam semantics specify a one-step evaluation process; message availability is considered part of the guard.) The body of our bounded buffer example is shown in Figure 12.22. Recall that Occam uses indentation to delimit control-flow constructs. Also note that Occam has no `mod` operator.

The question-mark operator (?) is Occam's `receive`; the exclamation-mark operator (!) is its `send`. As in Ada, an active manager thread must embed the `ALT` statement in a loop. As written here, the `ALT` statement has two guards. The first guard is true when `full_slots < SIZE` and a message is available on the channel named `producer`; the second guard is true when `full_slots > 0` and a message is available on the channel named `request`.

Because we are using synchronization `send` in this example, there is an asymmetry between the treatment of producers and consumers: the former need only send the manager data; the latter must send it a dummy argument and then wait for the manager to send the data back:

```

BDATA x :

-- producer: -- consumer:
producer ! x request ! TRUE
 consumer ? x

```

The asymmetry could be removed by using remote invocation on `CALL` channels:

```

-- channel declarations:
CALL insert(VAL BDATA d) :
CALL remove(RESULT BDATA d) :

```

```

-- channel declarations:
CHAN OF BDATA producer, consumer :
CHAN OF BOOL request :
-- buffer manager:
... -- (data declarations omitted)
WHILE TRUE
ALT
 full_slots < SIZE & producer ? d
 SEQ
 buf[next_empty] := d
 IF
 next_empty = SIZE
 next_empty := 1
 next_empty < SIZE
 next_empty := next_empty + 1
 full_slots := full_slots + 1
 full_slots > 0 & request ? t
 SEQ
 consumer ! buf[next_full]
 IF
 next_full = SIZE
 next_full := 1
 next_full < SIZE
 next_full := next_full + 1
 full_slots := full_slots - 1

```

Figure 12.22 Bounded buffer as an active Occam process.

```

-- buffer manager:
WHILE TRUE
ALT
 full_slots < SIZE & ACCEPT insert(VAL BDATA d)
 buf[next_empty] := d
 IF -- increment next_empty, etc.
 ...
 full_slots > 0 & ACCEPT remove(RESULT BDATA d)
 d := buf[next_full]
 IF -- increment next_full, etc.
 ...

```

Client code now looks like this:

```

-- producer: -- consumer:
insert(x) remove(x)

```

In the code of the buffer manager, the body of the ACCEPT is the single subsequent statement (the one that accesses buf). Updates to next\_empty, next\_full, and full\_slots occur after replying to the client. █

The effect of an Ada delay can be achieved in Occam by an ALT arm that “receives” from a *timer* pseudo-process:

#### EXAMPLE 12.54

Timeout in Occam receipt

```

resource buffer
 op insert(d : bdata)
 op remove() returns d : bdata
body buffer()
 const SIZE := 10;
 var buf[0:SIZE-1] : bdata
 var full_slots := 0, next_empty := 0, next_full := 0
process manager
 do true ->
 in insert(d) st full_slots < SIZE ->
 buf[next_empty] := d
 next_empty := (next_empty + 1) % SIZE
 full_slots++
 [] remove() returns d st full_slots > 0 ->
 d := buf[next_full]
 next_full := (next_full + 1) % SIZE
 full_slots--
 ni
 od
 end # manager
end # buffer

```

**Figure 12.23** Bounded buffer as an active SR process.

clock ? AFTER quit\_time

An arm can also be selected on the basis of a Boolean condition alone, without attempting to receive:

a > b & SKIP -- do nothing

Occam's ALT has no equivalent of the Ada `terminate`, nor is there an `else` (a similar effect can be achieved with a very short delay). ■

## DESIGN & IMPLEMENTATION

### Peeking inside messages

The ability of guards and scheduling expressions to “peek inside” a message in SR requires that all pending messages be visible to the language run-time system. An SR implementation must therefore be prepared to accept (and buffer) an arbitrary number of messages; it cannot rely on the operating system or other underlying software to provide the buffering for it. Moreover the fact that buffer space can never be truly unlimited means that guards and scheduling expressions will be unable to see messages whose delivery has been delayed by backpressure.

**EXAMPLE 12.55**

Bounded buffer in SR

In SR, selective receipt is again based on guarded commands; code appears in Figure 12.23. The `st` stands for “such that”; it introduces the Boolean half of a guard. Client code looks like this:

```
producer: # consumer:
call insert(x) x := remove()
```

If desired, an explicit `reply` to the client could be inserted between the access to `buf` and the updates of `next_empty`, `next_full`, and `full_slots` in each arm of the `in`.

**EXAMPLE 12.56**

Peeking at messages in SR

In a significant departure from Ada and Occam, SR arranges for the parameters of a potential message to be in the scope of the `st` condition, allowing a receiver to “peek inside” a message before deciding whether to receive it:

```
in insert(d) st d % 2 = 1 -> # only accept odd numbers
```

A receiver can also accept messages on a given port (i.e., of a given `op`) out-of-order by specifying a *scheduling expression*:

```
in insert(d) st d % 2 = 1 by -d ->
 # only accept odd numbers, and pick the largest one first
```

Like an Ada `select`, an SR `in` statement can end with an `else` guard; this guard will be selected if no message is immediately available. There is no equivalent of `delay` or `terminate`.

#### 12.4.4 Remote Procedure Call

Any of the three principal forms of `send` (no-wait, synchronization, remote-invocation) can be paired with either of the principal forms of `receive` (explicit or implicit). The combination of remote-invocation `send` with explicit receipt (e.g., as in Ada) is sometimes known as *rendezvous*. The combination of remote-invocation `send` with implicit receipt is usually known as *remote procedure call*. RPC is available in several concurrent languages (SR obviously among them). It is also supported on many systems by augmenting a sequential language with a *stub compiler*. The stub compiler is independent of the language’s regular compiler. It accepts as input a formal description of the subroutines that are to be called remotely. The description is roughly equivalent to the subroutine headers and declarations of the types of all parameters. Based on this input the stub compiler generates source code for *client* and *server stubs*. A client stub for a given subroutine marshals request parameters and an indication of the desired operation into a message buffer, sends the message to the server, waits for a reply message, and unmarshals that message into result parameters. A server stub takes a message buffer as parameter, unmarshals request parameters, calls the appropriate local subroutine, marshals return parameters into a reply message, and sends that message back to the appropriate client. Invocation of a client stub is relatively straightforward. Invocation of server stubs is discussed under “Implementation” below.

### **Semantics**

A principal goal of most RPC systems is to make the remote nature of calls as *transparent* as possible; that is, to make remote calls look as much like local calls as possible [BN84]. In a stub compiler system, a client stub should have the same interface as the remote procedure for which it acts as proxy; the programmer should usually be able to call the routine without knowing or caring whether it is local or remote.

Several issues make it difficult to achieve transparency in practice.

*parameter modes:* It is difficult to implement call-by-reference parameters across a network, since actual parameters will not be in the address space of the called routine. (Access to global variables is similarly difficult.)

*performance:* There is no escaping the fact that remote procedures may take a long time to return. In the face of network delays, one cannot use them casually.

*failure semantics:* Remote procedures are much more likely to fail than are local procedures. It is generally acceptable in the local case to assume that a called procedure will either run exactly once or else the entire program will fail. Such an assumption is overly restrictive in the remote case.

We can use value/result parameters in place of reference parameters as long as program correctness does not rely on the aliasing created by reference parameters. As noted in Section 8.3.1, Ada declares that a program is *erroneous* if it can tell the difference between pass-by-reference and pass-by-value/result implementations of `in out` parameters. If absolutely necessary, reference parameters and global variables can be implemented with message-passing thunks in a manner reminiscent of call-by-name parameters (Section 8.3.2), but only at very high cost. As noted in Section 7.10, a few languages and systems perform deep copies of linked data structures passed to remote routines.

Performance differences between local and remote calls can only be hidden by artificially slowing down the local case. Such an option is clearly unacceptable.

Exactly-once failure semantics can be provided by aborting the caller in the event of failure or, in highly reliable systems, by delaying the caller until the operating system or language run-time system is able to rebuild the failed computation using information previously dumped to disk. (Failure recovery techniques are beyond the scope of this text.) An attractive alternative is to accept “at-most-once” semantics with notification of failure. The implementation retransmits requests for remote invocations as necessary in an attempt to recover from lost messages. It guarantees that retransmissions will never cause an invocation to happen more than once, but it admits that in the presence of communication failures the invocation may not happen at all. If the programming language provides exceptions, then the implementation can use them to make communication failures look like any other kind of run-time error.

### **Implementation**

At the level of the kernel interface, `receive` is usually an explicit operation. To make `receive` appear implicit to the application programmer, the code produced by an RPC stub compiler (or the run-time system of a language like SR) must bridge this explicit-to-implicit gap. We describe the implementation here in terms of stub compilers; in a concurrent language with implicit receipt the regular compiler does essentially the same work.

#### **EXAMPLE 12.57**

An RPC server system

Figure 12.24 illustrates the layers of a typical RPC system. Code above the upper horizontal line is written by the application programmer. Code in the middle is a combination of library routines and code produced by the RPC stub generator. To initialize the RPC system, the application makes a pair of calls into the run-time system. The first provides the system with pointers to the stub routines produced by the stub compiler; the second starts a *message dispatcher*. What happens after this second call depends on whether the server is concurrent and, if so, whether its threads are implemented on top of one OS process or several.

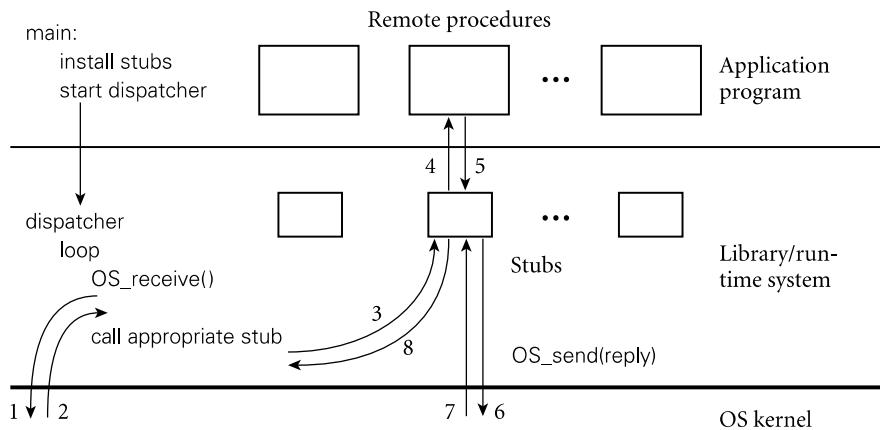
In the simplest case—a single-threaded server on a single OS process—the dispatcher runs a loop that calls into the kernel to receive a message. When a message arrives, the dispatcher calls the appropriate RPC stub, which unmarshals request parameters and calls the appropriate application-level procedure. When that procedure returns, the stub marshals return parameters into a reply message, calls into the kernel to send the message back to the caller, and then returns to the dispatcher. ■

### **DESIGN & IMPLEMENTATION**

#### **Parameters to remote procedures**

Ada's comparatively high-level semantics for parameter modes allows the same set of modes to be used for both subroutines and entries (rendezvous). An Ada compiler will generally pass a large argument to a subroutine by reference whenever possible, to avoid the expense of copying. If tasks are on separate processors of a multicomputer or cluster, however, the compiler will generally pass the same argument to an entry by value-result.

A few concurrent languages provide parameter modes specifically designed with remote invocation in mind. In Emerald [JLHB88], for example, every parameter is a reference to an object. References to remote objects are implemented transparently via message passing. To minimize the frequency of such references, objects passed to remote procedures often *migrate* with the call: they are packaged with the request message, sent to the remote site (where they can be accessed locally), and returned to the caller in the reply. Emerald calls this *call by move*. In Hermes [SBG<sup>+</sup>91], parameter-passing is *destructive*: arguments become uninitialized from the caller's point of view and can therefore migrate to a remote callee without danger of inducing remote references.



**Figure 12.24 Implementation of a remote procedure call server.** Application code initializes the RPC system by installing stubs generated by the stub compiler (not shown). It then calls into the run-time system to enable incoming calls. Depending on details of the particular system in use, the dispatcher may use the main program's single process (in which case the call to start the dispatcher never returns), or it may create a pool of processes that handle incoming requests.

This simple organization works well as long as each remote request can be handled quickly, without ever needing to block. If remote requests must sometimes wait for user-level synchronization, then the server's process must manage a ready list of threads, as described in Section 12.2.4, but with the dispatcher integrated into the usual thread scheduler. When the current thread blocks (in application code), the scheduler/dispatcher will grab a new thread from the ready list. If the ready list is empty, the scheduler/dispatcher will call into the kernel to receive a message, fork a new thread to handle it, and then continue to execute runnable threads until the list is empty again.

In a multiprocess server, the call to start the dispatcher will generally ask the kernel to fork a “pool” of processes to service remote requests. Each of these processes will then perform the operations described in the previous paragraphs. In a language or library with a one-one correspondence between threads and processes, each process will repeatedly receive a message from the kernel and then call the appropriate stub. With a more general thread package, each process will run threads from the ready list until the list is empty, at which point it (the process) will call into the kernel for another message. As long as the number of runnable threads is greater than or equal to the number of processes, no new messages will be received. When the number of runnable threads drops below the number of processes, then the extra processes will call into the kernel, where they will block until requests arrive.

#### **CHECK YOUR UNDERSTANDING**

49. Describe three ways in which processes commonly name their communication partners.

50. What is a *datagram*?
  51. Why, in general, might a `send` operation need to block?
  52. What are the three principal synchronization options for the sender of a message? What are the tradeoffs among them?
  53. What are *gather* and *scatter* operations in a message-passing program? What is *marshaling* and *unmarshaling*?
  54. Describe the tradeoffs between *explicit* and *implicit* message receipt.
  55. What is a *remote procedure call* (RPC)? What is a *stub compiler*?
  56. What are the obstacles to *transparency* in an RPC system?
  57. What is a *rendezvous*? How does it differ from a remote procedure call?
  58. Explain the purpose of a `select` statement in Ada (or, equivalently, of ALT in Occam).
  59. What semantic and pragmatic challenges are introduced by the ability to “peek” inside messages before they are received?
- 

## 12.5 Summary and Concluding Remarks

Concurrency and parallelism have become ubiquitous in modern computer systems. It is probably safe to say that most computer research and development today involves concurrency in one form or another. High-end computer systems are almost always parallel, and multiprocessor PCs are likely to become the norm within the next few years. With the explosion over the past decade in multimedia and Internet-based applications, multithreaded and message-passing programs have become central to day-to-day computing even on uniprocessors.

In this chapter we have provided an introduction to concurrent programming with an emphasis on programming language issues. We began with a quick synopsis of the history of concurrency, the motivation for multithreaded programs, and the architecture of modern multiprocessors. We then surveyed the fundamentals of concurrent software, including communication, synchronization, and the creation and management of threads. We distinguished between shared-memory and message-passing models of communication and synchronization, and between language and library-based implementations of concurrency.

Our survey of thread creation and management described some six different constructs for creating threads: `co-begin`, parallel loops, launch-at-elaboration, `fork/join`, implicit receipt, and early reply. Of these `fork/join` is the most common; it is found in Ada, Java, C#, Modula-3, SR, and library-based packages

such as PVM and MPI. RPC systems usually use `fork/join` internally to implement implicit receipt. Regardless of thread creation mechanism, most concurrent programming systems implement their language or library-level threads on top of a collection of OS-level processes, which the operating system implements in a similar manner on top of a collection of hardware processors. We built our sample implementation in stages, beginning with routines on a uniprocessor, then adding a ready list and scheduler, then timers for preemption, and finally parallel scheduling on multiple processors.

Our section on shared memory focused primarily on synchronization. We distinguished between mutual exclusion and condition synchronization, and between busy-wait and scheduler-based implementations. Among busy-wait mechanisms we looked in particular at spin locks and barriers. Among scheduler-based mechanisms we looked at semaphores, monitors, and conditional critical regions. Of the three, semaphores are the simplest and most common. Monitors and conditional critical regions provide a better degree of encapsulation and abstraction but are not amenable to implementation in a library. Conditional critical regions might be argued to provide the most pleasant programming model but cannot in general be implemented as efficiently as monitors. We also considered the implicit synchronization found in the loops of High Performance Fortran, the functional constructs of Sisal and pH, and the `future`-like constructs of Multilisp, Linda, and CC++.

Our section on message passing examined four principal issues: how to name communication partners, how long to block when sending a message, whether to receive explicitly or implicitly, and how to select among messages that may be available for receipt simultaneously. We noted that any of the three principal `send` mechanisms (no-wait, synchronization, remote-invocation) can be paired with either of the principal `receive` mechanisms (explicit, implicit). Remote-invocation `send` with explicit receipt is sometimes known as *rendezvous*. Remote-invocation `send` with implicit receipt is generally known as *remote procedure call*.

As in previous chapters, we saw many cases in which language design and language implementation influence one another. Some mechanisms (cactus stacks, conditional critical regions, content-based message screening) are sufficiently complex that many language designers have chosen not to provide them. Other mechanisms (Ada-style parameter modes) have been developed specifically to facilitate an efficient implementation technique. And in still other cases (the semantics of no-wait send, blocking inside a monitor) implementation issues play a major role in some larger set of tradeoffs.

Despite the very large number of concurrent languages that have been designed to date, much concurrent programming continues to employ conventional sequential languages augmented with library packages. As of 2005, HPF and other concurrent languages for large-scale clusters have yet to seriously undermine the dominance of MPI (though OpenMP, a shared-memory system that combines a library package with modest C or Fortran compiler support, is making inroads [Exploration 12.41]). For smaller-scale shared-memory computing,

many programmers continue to rely on library packages in C and C++, though Java and, more recently, C# are challenging that state of affairs. Java's suitability for network-based computing and its extreme portability across platforms have earned it a very strong base of support. Microsoft clearly hopes that C# will prove equally popular, though only time will tell whether it will be successful beyond the x86/Windows platform.

## 12.6 Exercises

- 12.1 Give an example of a “benign” race condition: one whose outcome affects program behavior but not correctness.
- 12.2 We have defined the *ready list* of a thread package to contain all threads that are runnable but not running, with a separate variable to identify the currently running thread. Could we just as easily have defined the ready list to contain *all* runnable threads, with the understanding that the one at the head of the list is running? (*Hint:* Think about multiprocessors.)
- 12.3 Imagine you are writing the code to manage a hash table that will be shared among several concurrent threads. Assume that operations on the table need to be atomic. You could use a single mutual exclusion lock to protect the entire table, or you could devise a scheme with one lock per hash-table bucket. Which approach is likely to work better, under what circumstances? Why?
- 12.4 The typical spin lock holds only one bit of data but requires a full word of storage, because only full words can be read, modified, and written atomically in hardware. Consider, however, the hash table of the previous exercise. If we choose to employ a separate lock for each bucket of the table, explain how to implement a “two-level” locking scheme that couples a conventional spin lock for the table as a whole with a *single bit* of locking information for each bucket. Explain why such a scheme might be desirable, particularly in a table with external chaining. (*Hint:* See the paper by Stumm et al. [UKGS94].)
- 12.5 Many of the most compute-intensive scientific applications are “dusty-deck” Fortran programs, generally very old and very complex. Years of effort may sometimes be required to rewrite a dusty-deck program to run on a parallel machine. An attractive alternative would be to develop a compiler that could “parallelize” old programs automatically. Explain why this is not an easy task.
- 12.6 The `load_linked` and `store_conditional` (LL/SC) instructions of Section 12.3.1 resemble an earlier universal atomic operation known as `compare-and-swap` (CAS). CAS was introduced by the IBM 370 architecture. It also appears in the x86, IA-64, and Sparc V9 instruction sets.

It takes three operands: the location to be modified, a value that the location is expected to contain, and a new value to be placed there if (and only if) the expected value is found. Like `store_conditional`, CAS returns an indication of whether it succeeded. The atomic add instruction sequence shown for `load_linked/store_conditional` in Example 12.27 would be written as follows with CAS.

```
start:
 r1 := foo
 r3 := r1 + r2
 CAS(foo, r1, r3)
 if failed goto start
```

Discuss the relative advantages of LL/SC and CAS. Consider how they might be implemented on a cache-coherent multiprocessor. Are there situations in which one would work but the other would not? (*Hints:* Consider algorithms in which a thread may need to touch more than one memory location. Also consider algorithms in which the contents of a memory location might be changed and then restored.)

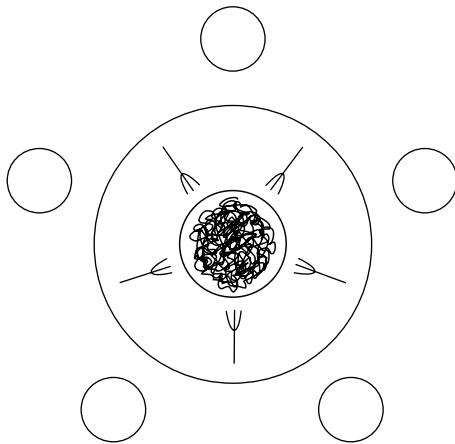
- 12.7** On most machines, a SC instruction can fail for any of several reasons, including the occurrence of an interrupt in the time since the matching LL. What steps must a programmer take to make sure that algorithms work correctly in the face of such “spurious” SC failures?
- 12.8** Starting with the test-and-test\_and\_set lock of Figure 12.10, implement busy-wait code that will allow readers to access a data structure concurrently. Writers will still need to lock out both readers and other writers. You may use any reasonable atomic instruction(s) (e.g., LL/SC). Consider the issue of fairness. In particular, if there are *always* readers interested in accessing the data structure, your algorithm should ensure that writers are not locked out forever.
- 12.9** The mechanism used in Figure 12.12 (page 624) to make scheduler code reentrant employs a single OS-provided lock for all the scheduling data structures of the application. Among other things, this mechanism prevents threads on separate processors from performing P or V operations on unrelated semaphores, even when none of the operations needs to block. Can you devise another synchronization mechanism for scheduler-related operations that admits a higher degree of concurrency but that is still correct?
- 12.10** We have seen how the scheduler for a thread package that runs on top of more than one OS-provided process must both disable timer signals *and* acquire a spin lock to safeguard the integrity of the ready list and condition queues. To implement processes within the operating system, the kernel still uses spin locks, but with processors instead of processes, and hardware interrupts instead of signals. Unfortunately, the kernel can-

not afford to disable interrupts for more than a small, bounded period of time, or devices may not work correctly. A straightforward adaptation of the code in Figure 12.12 will not suffice because it would attempt to acquire a spin lock (an unbounded operation) while interrupts were disabled. Similarly, the kernel cannot afford to acquire a spin lock and then disable interrupts because, if an interrupt occurs in between these two operations, other processors may be forced to spin for a very long time. How would you solve this problem? (*Hint:* Look carefully at the loop in the middle of `reschedule`, and consider a hybrid technique that disables interrupts and acquires a spin lock as a single operation.)

- 12.11 Mohit Aron and Peter Druschel of Rice University have proposed a mechanism they call a *soft timer* [AD00]. Soft timers use the hardware cycle counter available in many modern processors. This counter is a special register, readable with a nonprivileged instruction, whose contents are automatically incremented at some high frequency, typically once per microsecond. Among other things, soft timers can be used to implement preemption in a thread package. Instead of asking the OS kernel to deliver a signal at some specified future time, the thread scheduler can arrange to inspect the cycle counter every once in a while and perform a context switch if it has exceeded some previously computed value. Discuss the advantages and disadvantages of soft timers in comparison to signal-based timers.
- 12.12 Show how to implement a concurrent set as a singly linked sorted list. Your implementation should support insert, find, and remove operations, and should permit operations on separate portions of the list to occur concurrently (so a single lock for the entire list will not suffice). (*Hint:* You will want to use a “walking lock” idiom in which acquire and release operations are interleaved in non-LIFO order.)
- 12.13 To make spin locks useful on a multiprogrammed multiprocessor, one might want to ensure that no process is ever preempted in the middle of a critical section. That way it would always be safe to spin in user space, because the process holding the lock would be guaranteed to be running on some other processor, rather than preempted and possibly in need of the current processor. Explain why an operating system designer might not want to give user processes the ability to disable preemption arbitrarily. (*Hint:* Think about fairness and multiple users.) Can you suggest a way to get around the problem? (References to several possible solutions can be found in the paper by Kontothanassis, Wisniewski, and Scott [KWS97].)
- 12.14 Show how to use semaphores to construct an  $n$ -thread barrier.
- 12.15 Would it ever make sense to declare a semaphore with an initially negative count? Why or why not?
- 12.16 Without looking at Hoare’s definition, show how to implement monitors with semaphores.

- |2.17 Using monitors, show how to implement semaphores. What is your monitor invariant?
- |2.18 Show how to use binary semaphores to implement general semaphores.
- |2.19 Suppose that every monitor has a separate mutual exclusion lock, so that different threads can run in different monitors concurrently, and that we want to release exclusion on both inner and outer monitors when a thread waits in a nested call. When the thread awakens it will need to reacquire the outer locks. How can we ensure its ability to do so? (*Hint:* Think about the order in which to acquire locks, and be prepared to abandon Hoare semantics. For further hints, see Wettstein [Wet78].)
- |2.20 Show how general semaphores can be implemented with conditional critical regions in which all threads wait for the same condition, thereby avoiding the overhead of unproductive wakeups.
- |2.21 Write code for a bounded buffer using the protected object mechanism of Ada 95.
- |2.22 Repeat the previous exercise in Java using synchronized statements or methods. Try to make your solution as simple and conceptually clear as possible. You will probably want to use notifyAll.
- |2.23 Give a more efficient solution to the previous exercise that avoids the use of notifyAll. (*Warning:* It is tempting to observe that the buffer can never be both full and empty at the same time, and to assume therefore that waiting threads are either all producers or all consumers. This need not, however, be the case: if the buffer ever becomes even a temporary performance bottleneck, there may be an arbitrary number of waiting threads, including both producers and consumers.)
- |2.24 Repeat the previous exercise using Java Lock variables.
- |2.25 Explain how *escape analysis*, mentioned briefly in the sidebar on page 492, could be used to reduce the cost of certain synchronized statements and methods in Java.
- |2.26 The *dining philosophers problem* [Dij72] is a classic exercise in synchronization (Figure 12.25). Five philosophers sit around a circular table. In the center is a large communal plate of spaghetti. Each philosopher repeatedly thinks for a while and then eats for a while, at intervals of his or her own choosing. On the table between each pair of adjacent philosophers is a single fork. To eat, a philosopher requires both adjacent forks: the one on the left and the one on the right. Because they share a fork, adjacent philosophers cannot eat simultaneously.

Write a solution to the dining philosophers problem in which each philosopher is represented by a process and the forks are represented by shared data. Synchronize access to the forks using semaphores, monitors, or conditional critical regions. Try to maximize concurrency.



**Figure 12.25** The dining philosophers. Hungry philosophers must contend for the forks to their left and right in order to eat.

- 12.27** In the previous exercise you may have noticed that the dining philosophers are prone to deadlock. One has to worry about the possibility that all five of them will pick up their right-hand forks simultaneously and then wait forever for their left-hand neighbors to finish eating.

Discuss as many strategies as you can think of to address the deadlock problem. Can you describe a solution in which it is provably impossible for any philosopher to go hungry forever? Can you describe a solution that is fair in a strong sense of the word (i.e., in which no one philosopher gets more chance to eat than some other over the long term)? For a particularly elegant solution, see the paper by Chandy and Misra [CM84].

- 12.28** In some concurrent programming systems, global variables are shared by all threads. In others, each newly created thread has a separate copy of the global variables, commonly initialized to the values of the globals of the creating thread. Under this private globals approach, shared data must be allocated from a special heap. In still other programming systems, the programmer can specify which global variables are to be private and which are to be shared.

Discuss the tradeoffs between private and shared global variables. Which would you prefer to have available, for which sorts of programs? How would you implement each? Are some options harder to implement than others? To what extent do your answers depend on the nature of processes provided by the operating system?

- 12.29** AND parallelism in logic languages is analogous to the parallel evaluation of arguments in a functional language (e.g., Multilisp). Does OR parallelism have a similar analog? (*Hint:* Think about special forms [Section 10.4].) Can you suggest a way to obtain the effect of OR parallelism in Multilisp?

- 12.30 In Section 12.3.6 we claimed that both AND parallelism and OR parallelism were problematic in Prolog because they failed to adhere to the deterministic search order required by language semantics. Elaborate on this claim. What specifically can go wrong?
- 12.31 In Section 12.3.4 we cast monitors as a mechanism for synchronizing access to shared memory, and we described their implementation in terms of semaphores. It is also possible to think of a monitor as a module inhabited by a single process, which accepts request messages from other processes, performs appropriate operations, and replies. Give the details of a monitor implementation consistent with this conceptual model. Be sure to include condition variables. (*Hint:* See the discussion of early reply in Section 12.2.3, page 611.)
- 12.32 Show how shared memory can be used to implement message passing. Specifically, choose a set of message-passing operations (e.g., no-wait send and explicit message receipt), and show how to implement them in your favorite shared-memory notation.
- 12.33 When implementing reliable messages on top of unreliable messages, a sender can wait for an acknowledgment message, and retransmit if it doesn't receive it within a bounded period of time. But how does the receiver know that its acknowledgment has been received? Why doesn't the sender have to acknowledge the acknowledgment (and the receiver acknowledge the acknowledgment of the acknowledgment ...)? (For more information on the design of fast, reliable protocols, you might want to consult a text on computer networks [Tan02, PD03].)
- 12.34 An arm of an Occam ALT statement may include an *input guard*—a receive (?) operation—in which case the arm can be chosen only if a potential partner is trying to send a matching message. One could imagine allowing *output guards* as well: send (!) operations that would allow their arm to be chosen only if a potential partner were trying to receive a matching message. Neither Occam nor CSP (as originally defined) permits output guards. Can you guess why? Suppose you wished to provide them. How would the implementation work? (*Hint:* For ideas, see the articles of Bernstein [Ber80], Buckley and Silberschatz [BS83b], Bagrodia [Bag86], or Ramesh [Ram87].)
- 12.35 In Section 12.4.3 we described the semantics of a `terminate` arm on an Ada `select` statement: this arm may be selected if and only if all potential communication partners have terminated, or are likewise stuck in `select` statements with `terminate` arms. Occam and SR have no similar facility, though the original CSP proposal does. How would you implement `terminate` arms in Ada? Why do you suppose they were left out of Occam and SR? (*Hint:* For ideas, see the work of Apt and Francez [Fra80, AF84].)

## 12.7 Explorations

- 12.36 In Section 12.3.1 (page 622) we alluded to the design of *nonblocking* concurrent data structures, which work correctly without locks. Learn more about this topic. How hard is it to write correct nonblocking code? How does the performance compare to that of lock-based code? You might want to start with the work of Michael [MS98] and Sundell [Sun04].
- 12.37 Learn about the *software transactional memory* systems of Herlihy et al. [HLMS03] and Harris and Fraser [HF03]. To what extent do these constitute a general purpose mechanism for the construction of nonblocking concurrent data structures? Under what circumstances might they be preferable to locks?
- 12.38 Find out how message-passing is implemented in some locally available concurrent language or library. Does this system provide no-wait send, synchronization send, remote-invocation send, or some related hybrid? If you wanted to emulate the other options using the one available, how expensive would emulation be in terms of low-level operations performed by the underlying system? How would this overhead compare to what could be achieved on the same underlying system by a language or library that provided an optimized implementation of the other varieties of `send`?
- 12.39 Throughout Section 12.3 we assumed, implicitly, that memory shared between processors is *sequentially consistent*—that all memory operations occur in some global total order that is consistent with the actions of each individual processor. Many multiprocessors, however, implement *relaxed* memory models that provide significantly weaker consistency guarantees.  
Learn about relaxed memory models (see, for example, the tutorial by Adve and Gharachorloo [AG96]). Explain how the implementor of a language run-time system might use so-called *fence* instructions to ensure correct program behavior.
- 12.40 In light of the issues raised in the previous exploration, several programming languages, including Ada, Java, and C#, explicitly define a required memory model in the language definition. Learn the rules in each of these. Compare and contrast them. How efficiently can each be implemented on various real machines? What are the challenges for implementors? Note in particular the controversy that arose around the memory model in the original definition of Java (fixed in Java 5—see the paper by Pugh [Pug00] for a discussion).
- 12.41 OpenMP is a combination of compiler directives and run-time system for shared-memory programming on large multiprocessors and clusters. Its supporters promote it as an easier-to-use alternative to MPI. Learn about OpenMP (visit [openmp.org](http://openmp.org) or see the book by Chandra et al. [CMD<sup>+</sup>01]). Describe its programming model. What mechanisms does it provide to

- create and synchronize threads? How do these mechanisms compare to the ones discussed in Section 12.3? What compromises, if any, have the designers made for the sake of high performance on very large machines?
- 12.42 Learn about the `shmem` library package, originally developed by Robert Numrich of Cray, Inc., for the T3D supercomputer. `Shmem` is widely used for parallel programming on both large-scale multiprocessors and clusters. It has been characterized as a cross between shared memory and message passing. Is this a fair characterization? Under what circumstances might a `shmem` program be expected to outperform solutions in MPI or OpenMP? (Note: As of this writing, `shmem` has not been standardized, so implementations may differ some across platforms. The Cray `man` pages are available at [docs.cray.com/books/S-2383-23/S-2383-23-manual.pdf](http://docs.cray.com/books/S-2383-23/S-2383-23-manual.pdf).)
- 12.43 In the spirit of the previous two questions, investigate co-array Fortran ([www.co-array.org](http://www.co-array.org)), UPC ([upc.gwu.edu](http://upc.gwu.edu)), and/or Titanium ([www.cs.berkeley.edu/projects/titanium/](http://www.cs.berkeley.edu/projects/titanium/)). These are extensions to Fortran, C, and Java, respectively, designed for nonuniform shared-memory computing, in which the physical location of data is under explicit program control. How do these language extensions compare to OpenMP and `shmem`? How easy are they to use? What compromises, if any, have been made for the sake of efficiency?

## 12.8 Bibliographic Notes

Much of the early study of concurrency stems from a pair of articles by Dijkstra [Dij68a, Dij72]. Andrews and Schneider [AS83] provide an excellent survey of concurrent programming notations. A more recent book by Andrews [And91] extends this survey with extensive discussion of axiomatic semantics for concurrent programs and algorithmic paradigms for distributed computing. Holt et al. [HGLS78] is a useful reference for many of the classic problems in concurrency and synchronization. Anderson [ALL89] discusses thread package implementation details and their implications for performance. The July 1989 issue of *IEEE Software* and the September 1989 issue of *ACM Computing Surveys* contain survey articles and descriptions of many concurrent languages. References for monitors appear in Section 12.3.4.

Peterson's two-process synchronization algorithm appears in a remarkably elegant and readable two-page paper [Pet81]. Lamport's 1978 article on "Time, Clocks, and the Ordering of Events in a Distributed System" [Lam78] argued convincingly that the notion of global time cannot be well defined, and that distributed algorithms must therefore be based on causal *happens before* relationships among individual processes. Reader-writer locks are due to Courtois, Heymans, and Parnas [CHP71]. Mellor-Crummey and Scott [MCS91] survey the principal busy-wait synchronization algorithms and introduce locks and barriers that scale without contention to very large machines. The seminal paper on lock-free

synchronization is that of Herlihy [Her91]. Promising new general purpose techniques for *software transactional memory* have recently been reported by Herlihy, Luchangco, Moir, and Scherer [HLMS03] and by Harris and Fraser [HF03].

Concurrent logic languages are surveyed by Shapiro [Sha89], Tick [Tic91], and Ciancarini [Cia92]. Parallel Lisp dialects include Multilisp [Hal85, MKH91] (Section 12.3.6), Qlisp [GG89], and Spur Lisp [ZHL<sup>+</sup>89].

Remote procedure call received increasing attention in the wake of Nelson's doctoral research [Nel81, BN84]. Schroeder and Burrows [SB90] discuss the efficient implementation of RPC on a network of workstations. Bershad [BALL90] discusses its implementation across address spaces within a single machine.

Almasi and Gottlieb [AG94] describe the principal classes of parallel computers and the styles of algorithms and languages that work well on each. The leading texts on computer networks are by Tanenbaum [Tan02] and Peterson and Davie [PD03]. The text of Culler, Singh, and Gupta [CS98] contains a wealth of information on parallel programming and multiprocessor architecture. PVM [Sun90, GBD<sup>+</sup>94] and MPI [BDH<sup>+</sup>95, SOHL<sup>+</sup>98] are documented in a variety of articles and books. Sun RPC is documented in Internet RFC number 1831 [Sri95].

Software distributed shared memory (S-DSM) was originally proposed by Li as part of his doctoral research [LH89]. Stumm and Zhou [SZ90] and Nitzberg and Lo [NL91] provide early surveys of the field. The TreadMarks system from Rice University is widely considered the best of the more recent implementations [ACD<sup>+</sup>96].

# 13 Scripting Languages

**Traditional programming languages are intended** primarily for the construction of self-contained applications: programs that accept some sort of input, manipulate it in some well understood way, and generate appropriate output. But most actual *uses* of computers require the coordination of multiple programs. A large institutional payroll system, for example, must process time-reporting data from card readers, scanned paper forms, and manual (keyboard) entry; execute thousands of database queries; enforce hundreds of legal and institutional rules; create an extensive “paper trail” for record keeping, auditing, and tax preparation purposes; print paychecks; and communicate with servers around the world for online direct deposit, tax withholding, retirement accumulation, medical insurance, and so on. These tasks are likely to involve dozens or hundreds of separately executable programs. Coordination among these programs is certain to require tests and conditionals, loops, variables and types, subroutines and abstractions—the same sorts of logical tools that a conventional language provides *inside* an application.

On a much smaller scale, a graphic artist or photojournalist may routinely download pictures from a digital camera; convert them to a favorite format; rotate the pictures that were shot in vertical orientation; down-sample them to create browsable thumbnail versions; index them by date, subject, and color histogram; back them up to a remote archive; and then reinitialize the camera’s memory. Performing these steps at hand is likely to be both tedious and error-prone. In a similar vein, the creation of a dynamic web page may require authentication and authorization, database lookup, image manipulation, remote communication, and the reading and writing of HTML text. All these scenarios suggest a need for programs that coordinate other programs.

It is of course possible to write coordination code in Java, C, or some other conventional language, but it isn’t always easy. Conventional languages tend to stress efficiency, maintainability, portability, and the static detection of errors. Their type systems tend to be built around such hardware-level concepts as fixed-

size integers, floating-point numbers, characters, and arrays. By contrast *scripting languages* tend to stress flexibility, rapid development, local customization, and dynamic (run-time) checking. Their type systems, likewise, tend to embrace such high level concepts as tables, patterns, lists, and files.

General purpose scripting languages like Perl and Python are sometimes called *glue languages*, because they were originally designed to “glue” existing programs together to build a larger system. With the growth of the World Wide Web, scripting languages have gained new prominence in the generation of dynamic content. They are also widely used as *extension languages*, which allow the user to customize or extend the functionality of “scriptable” tools.

We consider the history and nature of scripting in more detail in Section 13.1. We then turn in Section 13.2 to some of the problem domains in which scripting is widely used. These include command interpretation (shells), text processing and report generation, mathematics and statistics, general purpose program coordination, and configuration and extension. In Section 13.3 we consider several forms of scripting used on the World Wide Web, including CGI scripts, server- and client-side processing of scripts embedded in web pages, Java applets, and XSLT. Finally, in Section 13.4, we consider some of the more interesting language features, common to many scripting languages, that distinguish them from their more traditional “mainstream” cousins. We look in particular at naming, scoping, and typing; string and pattern manipulation; and high-level structured data. We will not provide a detailed introduction to any one scripting language, though we will consider concrete examples in several. As in most of this book, the emphasis will be on underlying concepts.

## 13.1 What Is a Scripting Language?

Modern scripting languages have two principal sets of ancestors. In one set are the command interpreters or “shells” of traditional batch and “terminal” (command-line) computing. In the other set are various tools for text processing and report generation. Examples in the first set include IBM’s JCL, the MS-DOS command interpreter, and the Unix sh and csh shell families. Examples in the second set include IBM’s RPG, and Unix’s sed and awk. From these evolved Rexx, IBM’s “Restructured Extended Executor,” which dates from 1979, and Perl, originally devised by Larry Wall in the late 1980s and now the most widely used general purpose scripting language. Other general purpose scripting languages include Tcl (“tickle”), Python, Ruby, VBScript (for Windows) and AppleScript (for the Mac).

With the growth of the World Wide Web in the late 1990s, Perl was widely adopted for “server side” web scripting, in which a web server executes a program (on the server’s machine) to generate the content of a page. One early web scripting enthusiast was Rasmus Lerdorf, who created a collection of scripts to track access to his personal home page. Originally written in Perl but soon re-

designed as a full-fledged and independent language, these scripts evolved into PHP, now the most popular platform for server-side web scripting. PHP competitors include JSP (Java Server Pages) and, on Microsoft platforms, VBScript. For scripting on the client computer, all major browsers implement JavaScript, a language developed by Netscape Corporation in the mid-1990s and standardized by ECMA (the European standards body) in 1999 [ECM99].

In his classic paper on scripting [Ous98], John Ousterhout, the creator of Tcl, notes that “Scripting languages assume that a collection of useful components already exist in other languages. They are intended not for writing applications from scratch but rather for combining components.” Ousterhout envisions a fu-

## DESIGN & IMPLEMENTATION

### Scripting on Microsoft platforms

As in several other aspects of computing, Microsoft tends to rely on internally developed technology in the area of scripting languages. Most of its scripting applications are based on VBScript, a dialect of Visual Basic. At the same time, Microsoft has developed a very general scripting interface (Windows Script) that is implemented uniformly by the operating system (Windows Script Host [WSH]), the web server (Active Server Pages [ASP]), and the Internet Explorer browser. A Windows Script implementation of JScript, the company’s version of JavaScript, comes preinstalled on Windows machines, but languages like Perl and Python can be installed as well, and used to drive the same interface. Many other Microsoft applications, including the entire Office suite, use VBScript as an extension language, but for these the implementation framework (Visual Basic for Applications [VBA]) does not make it easy to use other languages instead.

Given Microsoft’s share of the desktop computing market, VBScript is one of the most widely used scripting languages. It is almost never used on other platforms, however, while Perl, Tcl, Python, PHP, and others see significant use on Windows. For server-side web scripting, PHP currently predominates: as of February 2005, some 69% of the 59 million Internet web sites surveyed by Netcraft LTD were running the open source Apache web server,<sup>1</sup> and of them most of the ones with active content were using PHP. Microsoft’s Internet Information Server (IIS) was second to Apache, with 21% of the sites, and many of those had PHP installed as well.<sup>2</sup> For client-side scripting, where Internet Explorer controls about 70% of the browser market,<sup>3</sup> most web site administrators need their content to be visible to the other 30%. Explorer supports JavaScript (JScript), but other browsers do not support VBScript.

<sup>1</sup> [news.netcraft.com/archives/web\\_server\\_survey.html](http://news.netcraft.com/archives/web_server_survey.html)

<sup>2</sup> [news.netcraft.com/archives/2003/08/30/php\\_growing\\_surprisingly\\_strongly\\_on\\_windows.html](http://news.netcraft.com/archives/2003/08/30/php_growing_surprisingly_strongly_on_windows.html)

<sup>3</sup> [www.w3schools.com/browsers/browsers\\_stats.asp](http://www.w3schools.com/browsers/browsers_stats.asp)

ture in which programmers increasingly rely on scripting languages for the top-level structure of their systems, where clarity, reusability, and ease of development are crucial. Traditional “systems languages” like C, C++, or Java, he argues, will be used for self-contained, reusable system components, which emphasize complex algorithms or execution speed. As a general rule of thumb, he suggests that code can be developed 5 to 10 times faster in a scripting language but will run 10 to 20 times faster in a traditional systems language.

Some authors reserve the term “scripting” for the glue languages used to coordinate multiple programs. In common usage, however, scripting is a broader and vaguer concept. It clearly includes web scripting. For most authors it also includes *extension languages*.

Many readers will be familiar with the Visual Basic “macros” of Microsoft Office and related applications. Others may be familiar with the Lisp-based extension language of the `emacs` text editor. Several languages, including Tcl, Rexx, Python, and the Guile and Elk dialects of Scheme, have implementations designed in such a way that they can be incorporated into a larger program and used to extend its features. Extension was in fact the original purpose of Tcl. In a similar vein, several widely used commercial applications provide their own proprietary extension languages. For graphical user interface (GUI) programming, the Tk toolkit, originally designed for use with Tcl, has been incorporated into several scripting languages, including Perl, Python, and Ruby.

One can also view XSLT (extensible stylesheet language transformations) as a scripting language, albeit somewhat different from the others considered in this chapter. XSLT is part of the growing family of XML (extensible markup language) tools. We consider it further in Section 13.3.5.

### 13.1.1 Common Characteristics

While it is difficult to define scripting languages precisely, there are several characteristics that they tend to have in common.

*Both batch and interactive use.* A few scripting languages (notably Perl) use a just-in-time compiler that insists on reading the entire source program before it produces any output. Most other languages, however, are willing to compile or interpret their input line-by-line. Rexx, Python, Tcl, Guile, and (with a short helper script) Ruby will all accept commands from the keyboard.

*Economy of expression.* To support both rapid development and interactive use, scripting languages tend to require a minimum of “boilerplate.” Some make heavy use of punctuation and very short identifiers (Perl is notorious for this), while others (e.g., Rexx, Tcl, and AppleScript) tend to be more “English-like,” with lots of words and not much punctuation. All attempt to avoid the extensive declarations and top-level structure common to conventional languages. Where a trivial program looks like this in Java:

#### EXAMPLE 13.1

Trivial programs in conventional and scripting languages

```

class Hello {
 public static void main(String[] args) {
 System.out.println("Hello, world!");
 }
}

```

and like this in Ada:

```

with ada.text_IO; use ada.text_IO;
procedure hello is
begin
 put_line("Hello, world!");
end hello;

```

in Perl, Python, or Ruby it is simply:

```
print "Hello, world!\n"
```

## DESIGN & IMPLEMENTATION

### Compiling interpreted languages

Several times in this chapter we will make reference to “the compiler” for a scripting language. As we saw in Examples 1.6 and 1.7, interpreters almost never work with source code; a front-end translator first replaces that source with some sort of intermediate form. For most implementations of most of the languages described in this chapter, the front end is sufficiently complex to deserve the name “compiler.” Intermediate forms are typically “byte code” representations reminiscent of those of Java.

## DESIGN & IMPLEMENTATION

### Canonical implementations

Because they are implemented with interpreters, scripting languages tend to be easy to port from one machine to another—substantially easier than compilers for which one must write a new code generator. Given a native compiler for the language in which the interpreter is written, the only difficult part (and it may indeed be difficult) is to implement any necessary modifications to the part of the interpreter that provides the interface to the operating system.

At the same time, the ease of porting an interpreter means that several scripting languages, including Perl, Python, Tcl, and Ruby, have a single widely used implementation, which serves as the de facto language definition. Reading a book on Perl, it can be difficult to tell how a subtle program will behave. When in doubt, one may need to “try it out.” Rexx and JavaScript appear to be unique among widely used scripting languages in having a formal definition codified by an international standards body and independent of any one implementation. (Sed, awk, and sh have also been standardized by POSIX [Int03b], but none of these has the complexity of Perl, Python, Tcl, or Ruby.)

*Lack of declarations; simple scoping rules.* Most scripting languages dispense with declarations, and provide simple rules to govern the scope of names. In some languages (e.g., Perl) everything is global by default; optional declarations can be used to limit a variable to a nested scope. In other languages (e.g., PHP and Tcl), everything is local by default; globals must be explicitly imported. Python adopts the interesting rule that any variable that is assigned a value is local to the block in which the assignment appears. Special syntax is required to assign to a variable in a surrounding scope.

*Flexible dynamic typing.* In keeping with the lack of declarations, most scripting languages are dynamically typed. In some (e.g., PHP, Python, Ruby, and Scheme), the type of a variable is checked immediately prior to use. In others (e.g. Rexx, Perl, and Tcl), a variable will be interpreted differently in different contexts. In Perl, for example, the program

#### EXAMPLE 13.2

##### Coercion in Perl

```
$a = "4";
print $a . 3 . "\n"; # '.' is concatenation
print $a + 3 . "\n"; # '+' is addition
```

will print

43

7

This contextual interpretation is similar to coercion, except that there isn't necessarily a notion of "natural" type from which an object must be converted; the various possible interpretations may all be equally "natural." We shall have more to say about context in Perl in Section 13.4.3. ■

*Easy access to other programs.* Most programming languages provide a way to ask the underlying operating system to run another program, or to perform some operation directly. In scripting languages, however, these requests are much more fundamental, and have much more direct support. Perl, for one, provides well over 100 built-in commands that access operating system functions for input and output, file and directory manipulation, process management, database access, sockets, interprocess communication and synchronization, protection and authorization, time-of-day clock, and network communication. These built-in commands are generally a good bit easier to use than corresponding library calls in languages like C.

*Sophisticated pattern matching and string manipulation.* In keeping with their text processing and report generation ancestry, and to facilitate the manipulation of textual input and output for external programs, scripting languages tend to have extraordinarily rich facilities for pattern matching, search, and string manipulation. Typically these are based on *extended regular expressions*. We discuss these further in Section 13.4.2.

*High-level data types.* High-level data types like sets, bags, dictionaries, lists, and tuples are increasingly common in the standard library packages of conventional programming languages. A few languages (notably C++) allow users

to redefine standard infix operators to make these types as easy to use as more primitive, hardware-centric types. Scripting languages go one step further by building high-level types into the syntax and semantics of the language itself. In most scripting languages, for example, it is commonplace to have an “array” that is indexed by character strings, with an underlying implementation based on hash tables. Storage is invariably garbage collected.

Much of the most rapid change in programming languages today is occurring in scripting languages. This can be attributed to several causes, including the continued growth of the web, the dynamism of the open-source community, and the comparatively low investment required to create a new scripting language. Where a compiled, industrial quality language like Java or C# requires a multiyear investment by a very large programming team, a single talented designer, working alone, can create a usable implementation of a new scripting language in only a year or two.

Due in part to this rapid change, newer scripting languages have been able to incorporate some of the most innovative concepts in language design. Ruby, for example, has a uniform object model (much like Smalltalk), true iterators (like Clu), array slices (like Fortran 90), structured exception handling, multiway assignment, and reflection. Python also provides several of these features, together with anonymous first-class functions and Haskell-like list comprehensions.

## 13.2 Problem Domains

Some general purpose languages—Scheme and Visual Basic in particular—are widely used for scripting. Conversely, some scripting languages, including Perl, Python, and Ruby, are intended by their designers for general purpose use, with features intended to support “programming in the large”: modules, separate compilation, reflection, program development environments, and so on. For the most part, however, scripting languages tend to see their principal use in well-defined problem domains. We consider some of these in the following subsections.

### 13.2.1 Shell (Command) Languages

In the days of punch card computing, simple command languages allowed the user to “script” the processing of a card deck. A control card at the front of the deck, for example, might indicate that the upcoming cards represented a program to be compiled, or perhaps assembly language for the compiler itself, or input for a program already compiled and stored on disk. A control card embedded later in the deck might test the exit status of the most recently executed program and choose what to do next based on whether that program completed successfully. Given the linear nature of a card deck, however (one can’t in general back up),

command languages for batch processing tend not to be very sophisticated. JCL, for example, has no iteration constructs.

With the development of interactive timesharing in the 1960s and early 1970s, command languages became much more sophisticated. Louis Pouzin wrote a simple command interpreter for CTSS, the Compatible Time Sharing System at MIT, in 1963 and 1964. When work began on the groundbreaking Multics system in 1964, Pouzin sketched the design of an extended command language, with quoting and argument-passing mechanisms, for which he coined the term “shell.” The subsequent implementation served as inspiration for Ken Thompson in the design of the original Unix shell in 1973. In the mid-1970s Stephen Bourne and John Mashey separately extended the Thompson shell with control flow and variables; Bourne’s design was adopted as the Unix standard, taking the place (and the name) of the Thompson shell, sh.

In the late 1970s Bill Joy developed the so-called “C shell” (csh), inspired at least in part by Mashey’s syntax, and introducing significant enhancements for interactive use, including history, aliases, and job control. The tcsh version of csh adds command line editing and command completion. David Korn incorporated these mechanisms into a direct descendant of the Bourne shell, ksh, which is very similar to the standard POSIX shell [Int03b]. The popular “Bourne again” shell, bash, is an open source version of ksh. While tcsh is still popular in some quarters, ksh/bash/POSIX sh is substantially better for writing shell scripts, and comparable for interactive use.

In addition to features designed for interactive use, which we will not consider further here, shell languages provide a wealth of mechanisms to manipulate file names, arguments, and commands, and to glue together other programs. Most of these features are retained by more general scripting languages. We consider a few of them here, using bash syntax. The discussion is of necessity heavily simplified; full details can be found in the bash *man* page, or in various online tutorials.

### ***Filename and Variable Expansion***

---

#### **EXAMPLE 13.3**

“Wildcards” and  
“globbing”

Most users of a Unix shell are familiar with “wildcard” expansion of file names. The following command will list all files in the current directory whose names end in .pdf.

```
ls *.pdf
```

The shell expands the pattern \*.pdf into a list of all matching names. If there are three of them (say fig1.pdf, fig2.pdf, and fig3.pdf), the result is equivalent to

```
ls fig1.pdf fig2.pdf fig3.pdf
```

Filename expansion is sometimes called “globbing,” after the original Unix glob command that implemented it. In addition to \* wildcards, one can usually specify “don’t care” or alternative characters or substrings. The pattern fig?.pdf will match (expand to) any file(s) with a single character be-

tween the `g` and the dot. The pattern `fig[0-9].pdf` will require that character to be a digit. The pattern `fig3.{eps, pdf}` will match both `fig3.eps` and `fig3.pdf`.

Filename expansion is particularly useful in loops. Such loops may be typed directly from the keyboard, or embedded in scripts intended for later execution. Suppose, for example, that we wish to create PDF versions of all our EPS figures:<sup>4</sup>

```
for fig in *.eps
do
 ps2pdf $fig
done
```

The `for` construct arranges for the shell variable `fig` to take on the names in the expansion of `*.eps`, one at a time, in consecutive iterations of the loop. The dollar sign in line 3 causes the value of `fig` to be expanded into the `ps2pdf` command before it is executed. (Interestingly, `ps2pdf` is itself a shell script that calls the `gs` PostScript interpreter.) Optional braces can be used to separate a variable name from following characters, as in `cp $foo ${foo}_backup`.

Multiple commands can be entered on a single line if they are separated by semicolons. The following, for example, is equivalent to the loop in the previous example.

```
for fig in *.eps; do ps2pdf $fig; done
```

### **Tests, Queries, and Conditions**

The loop of the preceding example will execute `ps2pdf` for every EPS file in the current directory. Suppose, however, that we already have some PDF files, and only want to create the ones that are missing.

```
for fig in *.eps
do
 target=${fig%.eps}.pdf
 if [$fig -nt $target]
 then
 ps2pdf $fig
 fi
done
```

The third line of this script is a variable assignment. The expression  `${fig%.eps}` within the right-hand side expands to the value of `fig` with any trailing `.eps` re-

---

**4** PostScript is a programming language developed at Adobe Systems, Inc. for the description of images and documents (we consider it again in the sidebar on page 767). Encapsulated PostScript (EPS) is a restricted form of PostScript intended for figures that are to be embedded in other documents. Portable Document Format (PDF, also by Adobe) is a self-contained file format that combines a subset of PostScript with font embedding and compression mechanisms. It is strictly less powerful than PostScript from a computational perspective, but much more portable, and faster and easier to render.

moved. Similar special expansions can be used to test or modify the value of a variable in many different ways. The square brackets in line four delimit a conditional test. The `-nt` operator checks to see whether the file named by its left operand is newer than the file named by its right operand (or if the left operand exists but the right does not). Similar *file query* operators can be used to check many other properties of files. Additional operators can be used for arithmetic or string comparisons.

### Pipes and Redirection

#### EXAMPLE 13.7

Pipes

One of the principal innovations of Unix was the ability to chain commands together, “piping” the output of one to the input of the next. Like most shells, bash uses the vertical bar character (`|`) to indicate a pipe. To count the number of figures in our directory, without distinguishing between EPS and PDF versions, we might type

```
for fig in *; do echo ${fig%.*}; done | sort -u | wc -l
```

Here the first command, a `for` loop, prints the names of all files with extensions (dot-suffixes) removed. The `echo` command inside the loop simply prints its arguments. The `sort -u` command after the loop removes duplicates, and the `wc -l` command counts lines.

Like most shells, bash also allows output to be directed to a file, or input read from a file. To create a list of figures, we might type

```
for fig in *; do echo ${fig%.*}; done | sort -u > all_figs
```

#### EXAMPLE 13.8

Output redirection

## DESIGN & IMPLEMENTATION

### Built-in commands in the shell

Commands in the shell generally take the form of a sequence of *words*, the first of which is the name of the command. Most commands are executable programs, found in directories on the shell’s *search path*. A large number, however (about 50 in bash) are *built-ins*—commands that the shell recognizes and executes itself, rather than starting an external program. Interestingly, several commands that are available as separate programs are duplicated as built-ins, either for the sake of efficiency or to provide additional semantics. Conditional tests, for example, were originally supported by the external `test` command (for which square brackets are syntactic sugar), but these occur sufficiently often in scripts that execution speed improved significantly when a built-in version was added. By contrast, while the `kill` command is not used very often, the built-in version allows processes to be identified by small integer or symbolic names from the shell’s *job control* mechanism. The external version supports only the longer and comparatively unintuitive process identifiers supplied by the operating system.

The “greater than” sign indicates output redirection. If doubled (`sort -u >> all_figs`) it causes output to be appended to the specified file, rather than overwriting the previous contents.

In a similar vein, the “less than” sign indicates input redirection. Suppose we want to print our list of figures all on one line, separated by spaces, instead of on multiple lines. On a Unix system we can type

```
tr '\n' ' ' < all_figs
```

This invocation of the standard `tr` command converts all newline characters to spaces. Because `tr` was written as a simple filter, it does not accept a list of files on the command line; it only reads standard input. ■

For any executing Unix program, the operating system keeps track of a list of open files. By convention, *standard input* and *standard output* (`stdin` and `stdout`) are files numbers 0 and 1. File number 2 is by convention *standard error* (`stderr`), to which programs are supposed to print diagnostic error messages. One of the advantages of the `sh` family of shells over the `csh` family is the ability to redirect `stderr` and other open files independent of `stdin` and `stdout`. Consider, for example, the `ps2pdf` script. Under normal circumstances this script works silently. If it encounters an error, however, it prints a message to `stdout` and quits. This violation of convention (the message should go to `stderr`) is harmless when the command is invoked from the keyboard. If it is embedded in a script, however, and the output of the script is directed to a file, the error message may end up in the file instead of on the screen, and go unnoticed by the user. With `bash` we can type

```
ps2pdf my_fig.eps 1>&2
```

Here `1>&2` means “make `ps2pdf` send file 1 (`stdout`) to the same place that the surrounding context would normally send file 2 (`stderr`).” ■

Finally, like most shells, `bash` allows the user to provide the input to a command in-line:

```
tr '\n' ' ' <<END
list
of
input
lines
END
```

The `<<END` indicates that subsequent input lines, up to a line containing only `END`, are to be supplied as input to `tr`. Such in-line input (traditionally called a “here document”) is seldom used interactively, but is highly useful in shell scripts. ■

### **Quoting and Expansion**

Shells typically provide several *quoting* mechanisms to group words together into strings. Single (forward) quotes inhibit filename and variable expansion in the quoted text, and cause it to be treated as a single word, even if it contains white

#### **EXAMPLE 13.9**

Redirection of `stderr` and `stdout`

#### **EXAMPLE 13.10**

Heredocs (in-line input)

#### **EXAMPLE 13.11**

Single and double quotes

space. Double quotes also cause the contents to be treated as a single word, but do not inhibit expansion. Thus

```
foo=bar
single='$foo'
double="$foo"
echo $single $double
```

will print “\$foo bar”.

#### **EXAMPLE 13.12**

##### Subshells

Several other bracketing constructs in bash group the text inside, for various purposes. Command lists enclosed in parentheses are passed to a subshell for evaluation. If the opening parenthesis is preceded by a dollar sign, the output of the nested command list is expanded into the surrounding context:

```
for fig in $(cat my_figs); do ps2pdf ${fig}.eps; done
```

Here `cat` is the standard command to print the content of a file. Most shells use backward single quotes for the same purpose (‘`cat my_figs`’); bash supports this syntax as well, for backward compatibility.

Command lists enclosed in braces are treated by bash as a single unit. They can be used, for example, to redirect the output of a sequence of commands:

```
{ date; ls; } >> file_list
```

Unlike parenthesized lists, commands enclosed in braces are executed by the current shell. From a programming languages perspective, parentheses and braces behave “backward” from the way they do in C: parentheses introduce a nested dynamic scope in bash, while braces are purely for grouping. In particular, variables that are assigned new values within a parenthesized command list will revert to their previous values once the list has completed execution.

#### **EXAMPLE 13.14**

##### Pattern-based list generation

When not surrounded by white space, braces perform pattern-based list generation, in a manner similar to filename expansion but without the connection to the file system. For example, `echo abc{12,34,56}xyz` prints `abc12xyz abc34xyz abc56xyz`. Also, as we have seen, braces serve to delimit variable names when the opening brace is preceded by a dollar sign.

In Example 13.6 we used square brackets to enclose a conditional expression. Double square brackets serve a similar purpose, but with more C-like expression syntax, and without filename expansion. Double parentheses are used to enclose arithmetic computations, again with C-like syntax.

The interpolation of commands in `$()` or backquotes, patterns in `{ }`, and arithmetic expressions in `(( ))` are all considered forms of expansion, analogous to filename expansion and variable expansion. The splitting of strings into words is also considered a form of expansion, as is the replacement, in certain contexts, of tilde (~) characters with the name of the user’s home directory. All told, these give us seven different kinds of expansion in bash.

All of the various bracketing constructs have rules governing which kinds of expansion are performed within. The rules are intended to be as intuitive as possible, but they are not uniform across constructs. Filename expansion, for exam-

ple, does not occur within `[[ ]]]`-bracketed conditions. Similarly, a double quote character may appear inside a double-quoted string if escaped with a backslash, but a single quote character may not appear inside a single-quoted string.

### **Functions**

#### **EXAMPLE 13.15**

User-defined shell functions

Users can define functions in `bash` that then work like built-in commands. Many users, for example, define `ll` as a shortcut for `ls -l`, which lists files in the current directory in “long format.”

```
function ll () {
 ls -l "$@"
}
```

Within the function, `$1` represents the first parameter, `$2` represents the second, and so on. In the definition of `ll`, `$@` represents the entire parameter list. Functions can be arbitrarily complex. In particular, `bash` supports both local variables and recursion. Shells in the `csh` family provide a more primitive `alias` mechanism that works via macro expansion. ■

### **The #! Convention**

#### **EXAMPLE 13.16**

The `#!` convention in script files

As noted above, shell commands can be read from a *script* file. To execute them in the current shell, one uses the “dot” command:

```
. my_script
```

where `my_script` is the name of the file. Many operating systems, including most versions of Unix, allow one to make a script function as an executable program, so that users can simply type

### **DESIGN & IMPLEMENTATION**

#### **Magic numbers**

When the Unix kernel is asked to execute a file (via the `execve` system call), it checks the first few bytes of the file for a “magic number” that indicates the file’s type. Some values correspond to directly executable object file formats. Under Linux, for example, the first four bytes of an object file are `0x7f45_4c46` ((`del`) ELF in ASCII). Under MacOS X they are `0xfeed_face`. If the first two bytes are `0x2321` (`#!` in ASCII), the kernel assumes that the file is a script, and reads subsequent characters to find the name of the interpreter.

The `#!` convention in Unix is the main reason that most scripting languages use `#` as the opening comment delimiter. Early versions of `sh` used the no-op command `( : )` as a way to introduce comments. Joy’s C shell introduced `#`, whereupon some versions of `sh` were modified to launch `csh` when asked to execute a script that appeared to begin with a C shell comment. This mechanism evolved into the more general mechanism used in many (though not all) variants of Unix today.

```
my_script
```

Two steps are required. First, the file must be marked *executable* in the eyes of the operating system. On Unix one types `chmod +x my_script`. Second, the file must be self-descriptive in a way that allows the operating system to tell which shell (or other interpreter) will understand the contents. Under Unix, the file must begin with the characters `#!`, followed by the name of the shell. The typical bash script thus begins with

```
#!/bin/bash
```

Specifying the full path name is a safety feature: it anticipates the possibility that the user may have a search path for commands on which some other program named `bash` appears before the shell. (Unfortunately, the requirement for full path names makes `#!` lines nonportable, since shells and other interpreters may be installed in different places on different machines.) ■

#### **CHECK YOUR UNDERSTANDING**

1. Give a plausible one-sentence definition of “scripting language.”
2. List the principal ways in which scripting languages differ from conventional “systems” languages.
3. From what two principal sets of ancestors are modern scripting languages descended?
4. What IBM creation is generally considered the first general purpose scripting language?
5. What is the most popular language for server-side web scripting?
6. How does the notion of *context* in Perl differ from coercion?
7. What is *globbing*? What is a *wildcard*?
8. What is a *pipe* in Unix? What is *redirection*?
9. Describe the three standard I/O streams provided to every Unix process.
10. Explain the significance of the `#!` convention in Unix shell scripts.

#### **13.2.2 Text Processing and Report Generation**

Shell languages tend to be heavily string-oriented. Commands are strings, parsed into lists of words. Variables are string-valued. Variable expansion mechanisms allow the user to extract prefixes, suffixes, or arbitrary substrings. Concatenation is indicated by simple juxtaposition. There are elaborate quoting conventions. Few more conventional languages have similar support for strings.

```

label (target for branch):
:top
/ <[hH] [123]>.*</[hH] [123]>/ {
 h
 s/\(<\/[hH] [123]>\).*$/\1/
 s/^.*\(<[hH] [123]>\)\1/
 p
 g
 s/<\/[hH] [123]>//
 b top
}
/ <[hH] [123]> / {
 N
 b top
}
d
;
```

**Figure 13.1** Script in sed to extract headers from an HTML file. The script assumes that opening and closing tags are properly matched, and that headers do not nest.

At the same time, shell languages are clearly not intended for the sort of text manipulation commonly performed in editors like emacs or vi. Search and substitution, in particular, are missing, and many other tasks that editors accomplish with a single keystroke—insertion, deletion, replacement, bracket-matching, forward and backward motion—would be awkward to implement, or simply make no sense, in the context of the shell. For repetitive text manipulation it is natural to want to automate the editing process. Tools to accomplish this task constitute the second principal class of ancestors for modern scripting languages.

### Sed

#### EXAMPLE 13.17

Extracting HTML headers with sed

As a simple text processing example, consider the problem of extracting all headers from a web page (an HTML file). These are strings delimited by `<H1> ... </H1>`, `<H2> ... </H2>`, and `<H3> ... </H3>` tags. Accomplishing this task in an editor like emacs, vi, or even Microsoft Word is straightforward but tedious: one must search for an opening tag, delete preceding text, search for a closing tag, mark the current position (as the starting point for the next deletion), and repeat. A program to perform these tasks in sed, the Unix “stream editor,” appears in Figure 13.1. The code consists of a label and three commands, the first two of which are compound. The first compound command prints the first header, if any, found in the portion of the input currently being examined (what sed calls the *pattern space*). The second compound command appends a new line to the pattern space whenever it already contains a header-opening tag. Both compound commands, and several of the subcommands, use regular expression patterns, delimited by slashes. We will discuss these patterns further in Section 13.4.2. The third command (the lone d) simply deletes the current line. Because each compound command ends with a branch back to the top of

the script, the second will execute only if the first does not, and the delete will execute only if neither compound does.

The editor heritage of `sed` is clear in this example. Commands are generally one character long, and there are no variables—no state of any kind beyond the program counter and text that is being edited. These limitations make `sed` best suited to “one-line programs,” typically entered verbatim from the keyboard with the `-e` command-line switch. The following, for example, will read from standard input, delete blank lines, and (implicitly) print the nonblank lines to standard output.

```
sed -e '/^[:space:]*$/d'
```

Here `^` represents the beginning of the line and `$` represents the end. The `[[:space:]]` expression matches any white-space character in the local character set, to be repeated an arbitrary number of times, as indicated by the Kleene star (`*`). The `d` indicates deletion. Nondeleted lines are printed by default.

### **Awk**

In an attempt to address the limitations of `sed`, Alfred Aho, Peter Weinberger, and Brian Kernighan designed `awk` in 1977 (the name is based on the initial letters of their last names). `Awk` is in some sense an evolutionary link between stream editors like `sed` and full-fledged scripting languages. It retains `sed`'s line-at-a-time filter model of computation, but allows the user to escape this model when desired, and replaces single-character editing commands with syntax reminiscent of C. `Awk` provides (typeless) variables and a variety of control flow constructs, including subroutines.

An `awk` program consists of a sequence of *patterns*, each of which has an associated *action*. For every line of input, the interpreter executes, in order, the actions whose patterns evaluate to true. An example with a single pattern-action pair appears in Figure 13.2. It performs essentially the same task as the `sed` script of Figure 13.1. Lines that contain no opening tag are ignored. In a line with an opening tag, we delete any text that precedes the header. We then print lines until we find the closing tag, and repeat if there is another opening tag on the same line. We fall back into the interpreter's main loop when we're cleanly outside any header.

Several conventions can be seen in this example. The current input line is available in the pseudo-variable `$0`. The `getline` function reads into this variable by default. The `substr(s, a, b)` function extracts the portion of string `s` starting at position `a` and with length `b`. If `b` is omitted, the extracted portion runs to the end of `s`. Conditions, like patterns, can use regular expressions; we can see an example in the `do ... while` loop. By default, regular expressions match against `$0`.

Perhaps the two most important innovations of `awk` are *fields* and *associative arrays*, neither of which appears in Figure 13.2. Like the shell, `awk` parses each input line into a series of words (fields). By default these are delimited by white space, though the user can change this behavior dynamically by assign-

### **EXAMPLE 13.18**

#### One-line scripts in `sed`

### **EXAMPLE 13.19**

#### Extracting HTML headers with `awk`

```

/<[hH] [123]>/ {
 # execute this block if line contains an opening tag
 do {
 open_tag = match($0, /<[hH] [123]>/)
 $0 = substr($0, open_tag) # delete text before opening tag
 # $0 is the current input line
 while (!/<\/[hH] [123]>/) { # print interior lines
 print # in their entirety
 if (getline != 1) exit
 }
 close_tag = match($0, /<\/[hH] [123]>/) + 4

 print substr($0, 0, close_tag) # print through closing tag
 $0 = substr($0, close_tag + 1) # delete through closing tag
 } while (/<[hH] [123]>/) # repeat if more opening tags
}

```

**Figure 13.2** Script in `awk` to extract headers from an HTML file. Unlike the `sed` script, this version prints interior lines incrementally. It again assumes that the input is well formed.

#### EXAMPLE 13.20

Fields in `awk`

#### EXAMPLE 13.21

Capitalizing a title in `awk`

ing a regular expression to the built-in variable `FS` (field separator). The fields of the current input line are available in the pseudo-variables `$1`, `$2`, .... The built-in variable `NR` gives the total number of fields. Awk is frequently used for field-based one-line programs. The following, for example, will print the second word of every line of standard input.

```
awk 'print $2'
```

Associative arrays will be considered in more detail in Section 13.4.3. Briefly, they combine the functionality of hash tables with the syntax of arrays. We can illustrate both fields and associative arrays with an example script (Figure 13.3) that capitalizes each line of its input as if it were a title. The script declines to modify “noise” words (articles, conjunctions, and short prepositions) unless they are the first word of the title or of a subtitle, where a subtitle follows a word ending with a colon or a dash. The script also declines to modify words in which any letter other than the first is already capitalized.

#### Perl

Perl was originally developed by Larry Wall in 1987, while he was working at the National Security Agency. The original version was, to first approximation, an attempt to combine the best features of `sed`, `awk`, and `sh`. It was a Unix-only tool, meant primarily for text processing (the name stands for “practical extraction and report language”). Over the years Perl has grown into a large and complex language, with an enormous user community. Though it is hard to judge such things, Perl is almost certainly the most popular and widely used scripting language. It is also fast enough for much general purpose use, and includes separate compilation, modularization, and dynamic library mechanisms appropriate for large-scale projects. It has been ported to almost every known operating system.

```

BEGIN { # "noise" words
 nw["a"] = 1; nw["an"] = 1; nw["and"] = 1; nw["but"] = 1
 nw["by"] = 1; nw["for"] = 1; nw["from"] = 1; nw["in"] = 1
 nw["into"] = 1; nw["nor"] = 1; nw["of"] = 1; nw["on"] = 1
 nw["or"] = 1; nw["over"] = 1; nw["the"] = 1; nw["to"] = 1
 nw["via"] = 1; nw["with"] = 1
}
{
 for (i=1; i <= NF; i++) {
 if ((!nw[$i] || i == 0 || $(i-1) ~ /[[:]-$/)) && ($i !~ /.+[A-Z]/)) {
 # capitalize
 $i = toupper(substr($i, 1, 1)) substr($i, 2)
 }
 printf $i " "; # don't add trailing line feed
 }
 printf "\n";
}

```

**Figure 13.3** Script in awk to capitalize a title. The BEGIN block is executed before reading any input lines. The main block has no explicit pattern, so it is applied to every input line.

Perl consists of a relatively simple language core, augmented with an enormous number of built-in library functions and an equally enormous number of shortcuts and special cases. A hint at this richness of expression can be found on page 622 of the standard language reference [WCO00], which lists (only) the 97 built-in functions “whose behavior varies the most across platforms.” The cover of the book is emblazoned with the language motto: “There’s more than one way to do it.”

#### EXAMPLE 13.22

##### Extracting HTML headers with Perl

We will return to Perl several times in this chapter, notably in Sections 13.2.4 and 13.4. For the moment we content ourselves with a simple text processing example, again to extract headers from an HTML file (Figure 13.4). We can see several Perl shortcuts in this figure, most of which help to make the code shorter than the equivalent programs in sed (Figure 13.1) and awk (Figure 13.2). Angle brackets (<>) are the “readline” operator, used for text file input. Normally they surround a *file handle* variable name, but as a special case, empty angle brackets generate as input the concatenation of all files specified on the command line when the script was first invoked (or standard input, if there were no such files). When a readline operator appears by itself in the control expression of a *while* loop (but nowhere else in the language), it generates its input a line at a time into the pseudo-variable `$_`. Several other operators work on `$_` by default. Regular expressions, for example, can be used to search within arbitrary strings, but when none is specified, `$_` is assumed.

The `next` statement is similar to `continue` in C or Fortran: it jumps to the bottom of the innermost loop and begins the next iteration. The `redo` statement also skips the remainder of the current iteration, but returns to the top of the loop, *without* reevaluating the control expression. In our example program,

```

while (<>) { # iterate over lines of input
 next if !/<[hH] [123]>/; # jump to next iteration
 while (!/<\/[hH] [123]>/) { $_[.= <>]; } # append next line to $_[0]
 s/.?(<[hH] [123]>.*?<\/[hH] [123]>)/$1;
 # perform minimal matching; capture parenthesized expression in $1
 print $1, "\n";
 redo unless eof; # continue without reading next line of input
}

```

**Figure 13.4** Script in Perl to extract headers from an HTML file. For simplicity we have again adopted the strategy of buffering entire headers, rather than printing them incrementally.

`redo` allows us to append additional input to the current line, rather than reading a new line. Because end-of-file is normally detected by an undefined return value from `<>`, and because that failure will happen only once per file, we must explicitly test for `eof` when using `redo` here. Note that `if` and its symmetric opposite, `unless`, can be used as either a prefix or a postfix test.

Readers familiar with Perl may have noticed two subtle but key innovations in the substitution command of line 4 of the script. First, where the expression `.*` (in `sed`, `awk`, and Perl) matches the longest possible string of characters that permits subsequent portions of the match to succeed, the expression `.*?` in Perl matches the *shortest* possible such string. This distinction allows us to easily isolate the first header in a given line. Second, much as `sed` allows later portions of a regular expression to refer back to earlier, parenthesized portions (line 4 of Figure 13.1), Perl allows such *captured* strings to be used *outside* the regular expression. We have leveraged this feature to print matched headers in line 6 of Figure 13.4. In general, the regular expressions of Perl are significantly more powerful than those of `sed` and `awk`; we will return to this subject in more detail in Section 13.4.2. ■

### 13.2.3 Mathematics and Statistics

As we noted in our discussions of `sed` and `awk`, one of the distinguishing characteristics of text processing and report generation is the frequent use of “one-line programs” and other simple scripts. Anyone who owns a programmable calculator realizes that similar needs arise in mathematics and statistics. And just as shell and report generation tools have evolved into powerful languages for general purpose computing, so too have notations and tools for mathematical and statistical computing.

In Section 7.4.1 we mentioned APL, one of the more unusual languages of the 1960s. Originally conceived as a pen-and-paper notation for teaching applied mathematics, APL retained its emphasis on the concise, elegant expression of mathematical algorithms when it evolved into a programming language. Though it lacks both easy access to other programs and sophisticated string manipulation, APL displays all the other characteristics of scripting described in Section 13.1.1, and one sometimes finds it listed as a scripting language.

The modern successors to APL include a trio of commercial packages for mathematical computing: Maple, Mathematica, and Matlab. Though their design philosophies differ, each provides extensive support for numerical methods, symbolic mathematics (formula manipulation), data visualization, and mathematical modeling. All three provide powerful scripting languages, with a heavy orientation toward scientific and engineering applications.

As the “3 Ms” are to mathematical computing, so the S and R languages are to statistical computing. Originally developed at Bell Labs by John Chambers and colleagues in the late 1970s, S is a commercial package widely used in the statistics community and in quantitative branches of the social and behavioral sciences. R is an Open Source alternative to S that is largely though not entirely compatible with its commercial cousin. Among other things, R supports multidimensional array and list types, array slice operations, user-defined infix operators, call-by-need parameters, first-class functions, and unlimited extent.

### 13.2.4 “Glue” Languages and General Purpose Scripting

From their text processing ancestors, scripting languages inherit a rich set of pattern matching and string manipulation mechanisms. From command interpreter shells they inherit a wide variety of additional features including simple syntax; flexible typing; easy creation and management of subprograms, with I/O redirection and access to completion status; file queries; easy interactive and file-based I/O; easy access to command-line arguments, environment strings, process identifiers, time-of-day clock, and so on; and automatic interpreter start-up (the `#!` convention). As noted in Section 13.1.1, many scripting languages have interpreters that will accept commands interactively.

#### EXAMPLE 13.23

“Force quit” script in Perl

The combination of shell and text processing mechanisms allows a scripting language to prepare input to, and parse output from, subsidiary processes. As a simple example, consider the (Unix-specific) “force quit” Perl script shown in Figure 13.5. Invoked with a regular expression as argument, the script identifies all of the user’s currently running processes whose name, process id, or command line arguments match that regular expression. It prints the information for each, and prompts the user for an indication of whether the process should be killed.

The second line of the code starts a subsidiary process to execute the Unix `ps` command. The command-line arguments cause `ps` to print the process id and name of all processes owned by the current user, together with their full command-line arguments. The pipe symbol (`|`) at the end of the command indicates that the output of `ps` is to be fed to the script through the `PS` file handle. The main `while` loop then iterates over the lines of this output. Within the loop, the `if` condition matches each line against `$ARGV[0]`, the regular expression provided on the script’s command line. It also compares the first word of the line (the process id) against `$$`, the id of the Perl interpreter currently running the script.

Scalar variables (which in Perl include strings) begin with a dollar sign (`$`). Arrays begin with an at sign (`@`). In the first line of the `while` loop in Figure 13.5,

```

$#ARGV == 0 || die "usage: $0 pattern\n";
open(PS, "ps -w -w -x -o'pid,command' |"); # 'process status' command
<PS>; # discard header line
while (<PS>) {
 @words = split; # parse line into space-separated words
 if (/ARGV[0]/i && $words[0] ne $$) {
 chomp; # delete trailing newline
 print;
 do {
 print "? ";
 $answer = <STDIN>;
 } until $answer =~ /^[yn]/i;
 if ($answer =~ /y/i) {
 kill 9, $words[0]; # signal 9 in Unix is always fatal
 sleep 1; # wait for 'kill' to take effect
 die "unsuccessful; sorry\n" if kill 0, $words[0];
 }
 # kill 0 tests for process existence
 }
}

```

**Figure 13.5** Script in Perl to “force quit” errant processes. Perl’s text processing features allow us to parse the output of `ps`, rather than filtering it through an external tool like `sed` or `awk`.

the input line (`$_`, implicitly) is split into space-separated words, which are then assigned into the array `@words`. In the following line, `$words[0]` refers to the first element of this array, a scalar. A single variable name may have different values when interpreted as a scalar, an array, a hash table, a subroutine, or a file handle. The choice of interpretation depends on the leading punctuation mark and on the *context* in which the name appears. We shall have more to say about context in Perl in Section 13.4.3.

Beyond the combination of shell and text processing mechanisms, the typical glue language provides an extensive library of built-in operations to access features of the underlying operating system, including files, directories, and I/O; processes and process groups; protection and authorization; interprocess communication and synchronization; timing and signals; and sockets, name service, and network communication. Just as text processing mechanisms minimize the need to employ external tools like `sed`, `awk`, and `grep`, operating system built-ins minimize the need for other external tools.

At the same time, scripting languages have, over time, developed a rich set of features for internal computation. Most have significantly better support for mathematics than is typically found in a shell. Several, including Scheme, Python, and Ruby, support arbitrary precision arithmetic. Most provide extensive support for higher level types, including arrays, strings, tuples, lists, and hashes (associative arrays). Several support classes and object orientation. Some support iterators, continuations, threads, reflection, and first-class and higher-order functions. Some, including Perl, Tcl, Python, and Ruby, support modules and

dynamic loading, for “programming in the large.” These features serve to maximize the amount of code that can be written in the scripting language itself, and to minimize the need to escape to a more traditional, compiled language.

In summary, the philosophy of general purpose scripting is make it as easy as possible to construct the overall framework of a program, escaping to external tools only for special purpose tasks, and to compiled languages only when performance is at a premium.

### Tcl

Tcl was originally developed in the late 1980s by Professor John Ousterhout of the University of California, Berkeley. Over the previous several years his group had developed a suite of VLSI design automation tools, each of which had its own idiosyncratic command language. The initial motivation for Tcl (“tool command language”) was the desire for an *extension language* that could be embedded in all the tools, providing them with uniform command syntax and reducing the complexity of development and maintenance. Tk, a set of extensions for graphical user interface programming, was added to Tcl early in its development, and both Tcl and Tk were made available to other researchers starting in 1990. The user community grew rapidly in the 1990s, and Tcl quickly evolved beyond its emphasis on command extension to encompass “glue” applications as well. Ousterhout joined Sun Microsystems in 1994, where for three years he led a multiperson team devoted to Tcl development. In 1997 he launched a startup company specializing in Tcl applications and tools.

In comparison to Perl, Tcl is somewhat more verbose. It makes less use of punctuation and has fewer special cases. Everything in the language, including control flow constructs, takes the form of a (possibly quoted) *command* (an identifier) followed by a series of arguments. In the spirit of Unix command-line invocation, the first few, optional arguments typically begin with a minus sign (-) and are known as “switches.”

A simple Tcl script, equivalent to the Perl script of Figure 13.5, appears in Figure 13.6. The `set` command is an assignment; it copies the value of its second argument into the variable named by the first argument. In most other contexts a variable name needs to be preceded by a dollar sign (\$); as in shell languages, this indicates that the value of the variable should be expanded in-line. (Note the contrast to Perl, in which the dollar sign indicates scalar type and must appear even when the variable is used as an l-value.) As in most scripting languages, variables in Tcl need not be declared.

Double quote marks (as in `"$line? "`) behave in the familiar way: variable references inside are expanded before the string is used. Braces (`{ }`) work much as the single quotes of shell languages or Perl: they inhibit internal expansion. Brackets (`[]`) are a bit like traditional backquotes, but instead of interpreting the enclosed string as a program name and arguments, they interpret that string as a Tcl script, whose output should be expanded in place of the bracketed string. In the header of the `while` loop of Figure 13.6, the `eof` command returns a 1 or

---

#### EXAMPLE 13.24

“Force quit” script in Tcl

```

if {$argc != 1} {puts stderr "usage: $argv0 pattern"; exit 1}
set PS [open "|/bin/ps -w -w -x -opid,command" r]

gets $PS ;# discard header line
while {! [eof $PS]} {
 set line [gets $PS] ;# returns blank line at eof
 regexp {[0-9]+} $line proc
 if {[regexp [lindex $argv 0] $line] && [expr $proc != [pid]]} {
 puts -nonewline "$line? "
 flush stdout ;# force prompt out to screen
 set answer [gets stdin]
 while {! [regexp -nocase {^yn} $answer]} {
 puts -nonewline "? "
 flush stdout
 set answer [gets stdin]
 }
 if {[regexp -nocase {^y} $answer]} {
 set stat [catch {exec kill -9 $proc}]
 exec sleep 1
 if {$stat || [exec ps -p $proc | wc -l] > 1} {
 puts stderr "unsuccessful; sorry"; exit 1
 }
 }
 }
}

```

**Figure 13.6** Script in Tcl to “force quit” errant processes. Compare to the Perl script of Figure 13.5.

a 0, which is then interpreted as true or false. Like \$-prefixed variable names, bracketed expressions are expanded inside double quotes and brackets, but not inside braces.

In the third line of the `while` loop there are two pairs of nested brackets. The expression `[lindex $argv 0]` returns the first element of the list `$argv` (the one with index zero). This is the pattern specified on the command line of the script. It is passed as the first argument to the `regexp` command, along with the current line of output from the `ps` program. The `regexp` command in turn returns a 1 or a 0, depending on whether the pattern could be found within the line. The `expr` command interprets its remaining arguments as an arithmetic/logical expression with infix operators. The `pid` command returns the process id of the Tcl interpreter currently running the script. To facilitate the use of infix notation in conditions, the first argument to the `if` and `while` commands is automatically passed to `expr`.

Multiple Tcl commands can be written on a single line, as long as they are separated by semicolons. A newline character terminates the current command unless it is escaped with a backslash (\) or appears within a brace-quoted string. Control structures like `if` and `while` can thus span multiple lines so long as

the nested commands are enclosed in braces, and the opening brace appears on the same line as the condition. All variables and arguments, including nested bracketed scripts, are represented internally as character strings. Moreover arguments are expanded and evaluated lazily, so `if` and `while` behave as one would expect. The sharp character (#) introduces a comment, but as in `sed` (and in contrast to most programming languages) this is permitted only where a command might otherwise appear. In particular, a comment that follows a command on the same line of the script must be separated from the command by a semicolon.

The `exec` command interprets its remaining arguments as the name and arguments of an external program; it executes that program and returns its output. Many functions that are built into Perl must be invoked as external programs in Tcl; the `kill` and `sleep` functions of Figures 13.5 and 13.6 are two examples. The `catch` command executes the nested `exec` in a protected environment that produces no error messages but returns a status code that can be inspected later (nonzero indicates error). The external pipe `ps -p $proc | wc -l` counts the number of lines (including header) generated by a request to list the (hopefully now nonexistent) process `proc`. ■

### **Python**

As noted in Section 13.1, Rexx is generally considered the first of the general purpose scripting languages, predating Perl and Tcl by almost a decade. Perl and Tcl are roughly contemporaneous: both were initially developed in the late 1980s. Perl was originally intended for glue and text processing applications. Tcl was originally an extension language, but soon grew into glue applications as well. As the popularity of scripting grew in the 1990s, users were motivated to develop additional languages, provide additional features, address the needs of specific application domains (more on this in subsequent sections), or support a style of programming more in keeping with the personal taste of their designers.

Python was originally developed by Guido van Rossum at CWI in Amsterdam, the Netherlands, in the early 1990s. He continued his work at CNRI in Reston, Virginia, beginning in 1995. In 2000 the Python team moved to BeOpen.com, and to Digital Creations (now Zope Corp.) shortly thereafter. Recent versions of the language are owned by the Python Software Foundation, of which Zope is a member. All releases are Open Source.

#### **EXAMPLE 13.25**

“Force quit” script in Python

Figure 13.7 presents a Python version of our “force quit” program. Reflecting the maturation of programming language design, Python was from the beginning an object-oriented language.<sup>5</sup> It includes a standard library as rich as that of Perl, but partitioned into a collection of namespaces reminiscent of those of C++, Java, or C#. The first line of our script imports symbols from the `sys`, `os`, `re`, and

---

**5** Rexx and Tcl have object-oriented extensions, named Object Rexx and Incr Tcl, respectively. Perl 5 includes some (rather awkward) object-oriented features; Perl 6 will have more uniform object support.

```

import sys, os, re, time
if len(sys.argv) != 2:
 sys.stderr.write('usage: ' + sys.argv[0] + ' pattern\n')
 sys.exit(1)

PS = os.popen("/bin/ps -w -w -x -o'pid,command'")
line = PS.readline() # discard header line
line = PS.readline().rstrip() # prime pump
while line != "":
 proc = int(re.search('\S+', line).group())
 if re.search(sys.argv[1], line) and proc != os.getpid():
 print line + '? ',
 answer = sys.stdin.readline()
 while not re.search('^[yn]', answer, re.I):
 print '? ', # trailing comma inhibits newline
 answer = sys.stdin.readline()
 if re.search('^y', answer, re.I):
 os.kill(proc, 9)
 time.sleep(1)
 try: # expect exception if process
 os.kill(proc, 0) # no longer exists
 sys.stderr.write("unsuccessful; sorry\n"); sys.exit(1)
 except: pass # do nothing
 sys.stdout.write('') # inhibit prepended blank on next print
 line = PS.readline().rstrip()

```

**Figure 13.7** Script in Python to “force quit” errant processes. Compare to Figures 13.5 and 13.6.

time library modules. The fifth line launches ps as an external program and ties its output to the file object PS. In standard object-oriented style, readline is then invoked as a method of this object.

Perhaps the most distinctive feature of Python, though hardly the most important, is its reliance on indentation for syntactic grouping. We have already seen that Tcl uses line breaks to separate commands. Python does so also, and further specifies that the body of a structured statement consists of precisely those subsequent statements that are indented one more tab stop. Like the “more than one way to do it” philosophy of Perl, Python’s use of indentation tends to arouse strong feelings among users: some strongly positive, some strongly negative.

The regular expression (re) library has all of the power available in Perl but employs the somewhat more verbose syntax of method calls, rather than the built-in notation of Perl. The search routine returns a “match object” that captures, lazily, the places in the string at which the pattern appears. If no match is found, search returns None, the empty object, instead. In a condition, None is interpreted as false, while a true match object is interpreted as true. The match object in turn supports a variety of methods, including group, which returns the

substring corresponding to the first match. The `re.I` flag to `search` indicates case insensitivity. Note that `group` returns a string. Unlike Perl and Tcl, Python will not coerce this to an integer—hence the need for the explicit type conversion on the first line of the body of the `while` loop.

As in Perl (and in contrast to Tcl), the `readline` method does not remove the newline character at the end of an input line; we use the `rstrip` method to do this. The `print` routine adds a newline to the end of its argument list unless that list ends with a trailing comma. The `print` routine also prepends a space to its output unless a set of well-defined heuristics indicate that the output will appear at the beginning of a line. The `write` of a null string at the bottom of the `while` loop serves to defeat these heuristics in the wake of the user’s input, avoiding a spurious blank at the beginning of the next process prompt.

The `sleep` and `kill` routines are built into Python, much as they are in Perl. When given a signal number of 0, `kill` tests for process existence. Instead of returning a status code, however, as it does in Perl, the Python `kill` throws an exception if the process does not exist. We use a `try` block to catch this exception in the expected case. ■

While our “force quit” program may convey, at least in part, the “feel” of various languages, it cannot capture the breadth of their features. Python includes many of the more interesting features discussed in earlier chapters, including nested functions with static scoping, lambda expressions and higher-order functions, true iterators, list comprehensions, array slice operations, reflection, structured exception handling, multiple inheritance, and modules and dynamic loading.

### Ruby

Ruby is the newest of the widely used glue languages. It was developed in Japan in the early 1990s by Yukihiro “Matz” Matsumoto. Matz writes that he “wanted a language more powerful than Perl, and more object-oriented than Python” [TH04, Foreword]. The first public release was made available in 1995, and quickly gained widespread popularity in Japan. With the more recent publication of English-language documentation, Ruby has spread rapidly elsewhere as well.

#### **EXAMPLE 13.26**

##### Method call syntax in Ruby

In keeping with Matz’s original motivation, Ruby is a pure object-oriented language, in the sense of Smalltalk: everything—even instances of built-in types—is an object. Integers have more than 25 built-in methods. Strings have more than 75. Smalltalk-like syntax is even supported: `2 * 4 + 5` is syntactic sugar for `(2.*(4)).+(5)`, which is in turn equivalent to `(2.send('*','4')).send('+', 5)`.<sup>6</sup> ■

---

**6** Parentheses here are significant. Infix arithmetic follows conventional precedence rules, but method invocation proceeds from left to right. Likewise, parentheses can be omitted around argument lists, but the method-selecting dot (`.`) groups more tightly than the argument-separating comma (`,`), so `2.send '*', 4.send '+', 5` evaluates to 18, not 13.

```

ARGV.length() == 1 or begin
 $stderr.print("usage: #{$0} pattern\n"); exit(1)
end

pat = Regexp.new(ARGV[0])
IO.popen("ps -w -w -x -o'pid,command'") {|PS|
 PS.gets # discard header line
 PS.each {|line|
 proc = line.split[0].to_i
 if line =~ "pat" and proc != Process.pid then
 print line.chomp
 begin
 print "? "
 answer = $stdin.gets
 end until answer =~ /^[yn]/i
 if answer =~ /^y/i then
 Process.kill(9, proc)
 sleep(1)
 begin # expect exception (process gone)
 Process.kill(0, proc)
 $stderr.print("unsuccessful; sorry\n"); exit(1)
 rescue # handler -- do nothing
 end
 end
 end
 }
}
}

```

**Figure 13.8** Script in Ruby to “force quit” errant processes. Compare to Figures 13.5, 13.6, and 13.7.

#### EXAMPLE 13.27

##### “Force quit” script in Ruby

Figure 13.8 presents a Ruby version of our “force quit” program. As in Tcl, a newline character serves to end the current statement, but indentation is not significant. A dollar sign (\$) at the beginning of an identifier indicates a global name. Though it doesn’t appear in this example, an at sign (@) indicates an instance variable of the current object. Double at signs (@@) indicate an instance variable of the current *class*.

Probably the most distinctive feature of Figure 13.8 is its use of *blocks* and *iterators*. The IO.popen class method takes as argument a string that specifies the name and arguments of an external program. The method also accepts, in a manner reminiscent of Smalltalk, an *associated block*, specified as a multiline fragment of Ruby code delimited with curly braces. This block is invoked by *popen*, passing as parameter a file handle (an object of class IO) that represents the output of the external command. The |PS| at the beginning of the block specifies the name by which this handle is known within the block. In a similar vein, the *each* method of object PS is an iterator that invokes the associated block (the code in braces

beginning with `|line|`) once for every line of data. For those more comfortable with traditional `for` loop syntax, the iterator can also be written

```
for line in PS
 ...
end
```

In addition to (true) iterators, Ruby provides continuations and first-class and higher-order functions. Its *module* mechanism supports an extended form of mix-in inheritance. Though a class cannot inherit data members from a module, it *can* inherit code. Run-time type checking makes such inheritance more or less straightforward. Methods of modules that have not been explicitly included into the current class can be accessed as qualified names; `Process.kill` is an example in Figure 13.8. Methods `sleep` and `exit` belong to module `Kernel`, which is included by class `Object`, and is thus available everywhere without qualification. Like `popen`, they are class methods rather than instance methods; they have no notion of “current object.” Variables `stdin` and `stderr` refer to global objects of class `IO`.

Regular expression operations in Ruby are methods of class `RegExp`, and can be invoked with standard object-oriented syntax. For convenience, Perl-like notation is also supported as syntactic sugar; we have used this notation in Figure 13.8.

The `rescue` clause of the innermost `begin ... end` block is an exception handler. As in the Python code of Figure 13.7, it allows us to determine whether the `kill` operation has succeeded by catching the (expected) exception that arises when we attempt to refer to a process after it has died. ■

### 13.2.5 Extension Languages

Most applications accept some sort of *commands*, which tell them what to do. Sometimes these commands are entered textually; more often they are triggered by user interface events such as mouse clicks, menu selections, and keystrokes. Commands in a graphical drawing program might save or load a drawing; select, insert, delete, or modify its parts; choose a line style, weight, or color; zoom or rotate the display; or modify user preferences.

An *extension language* serves to increase the usefulness of an application by allowing the user to create new commands, generally using the existing commands as primitives. Extension languages are increasingly seen as an essential feature of sophisticated tools. Adobe’s graphics suite (Illustrator, Photoshop, InDesign, etc.) can be extended (scripted) using JavaScript, Visual Basic (on Windows), or AppleScript (on the Mac). AOLserver, an open-source web server from America Online, can be scripted using Tcl. Disney and Industrial Light and Magic use Python to extend their internal (proprietary) tools. Many commercially available packages, including AutoCAD, Maya, Director, and Flash have their own unique scripting languages. This list barely scratches the surface.

To admit extension, a tool must

- incorporate, or communicate with, an interpreter for a scripting language.
- provide hooks that allow scripts to call the tool's existing commands.
- allow the user to tie newly defined commands to user interface events.

With care, these mechanisms can be made independent of any particular scripting language. As we noted in the sidebar on page 673, Microsoft's Windows Script interface allows arbitrary languages to be used to script the operating system, web server, and browser. GIMP, the widely used GNU Image Manipulation Program, has a comparably general interface and can be scripted in Scheme, Tcl, Python, and Perl, among others. There is a tendency, of course, for user communities to converge on a favorite language, to facilitate sharing of code. Microsoft tools are usually scripted with Visual Basic. GIMP is usually scripted with the SIOD dialect of Scheme. Adobe tools are usually scripted with Visual Basic on the PC or AppleScript on the Mac.

One of the oldest existing extension mechanisms is that of the `emacs` text editor, used to write this book. An enormous number of extension packages have been created for `emacs`; many of them are installed by default in the standard distribution. In fact much of what users consider the editor's core functionality is actually provided by extensions; the truly built-in parts are comparatively small.

The extension language for `emacs` is a dialect of Lisp called Emacs Lisp. An example script appears in Figure 13.9. It assumes that the user has used the standard *marking* mechanism to select a region of text. It then inserts a line number at the beginning of every line in the region. The first line is numbered 1 by default, but an alternative starting number can be specified with an optional parameter. Line numbers are bracketed with a prefix and suffix that are “ ” (empty) and “`)`” by default, but can be changed by the user if desired. To maintain existing alignment, small numbers are padded on the left with enough spaces to match the width of the number on the final line.

Many features of Emacs Lisp can be seen in this example. The `setq-default` command is an assignment that is visible in the current buffer (editing session) and in any concurrent buffers that haven't explicitly overridden the previous value. The `defun` command defines a new command. Its arguments are, in order, the command name, formal parameter list, documentation string, interactive specification, and body. The argument list for `number-region` includes the start and end locations of the currently marked region, and the optional initial line number. The documentation string is automatically incorporated into the online help system. The interactive specification controls how arguments are passed when the command is invoked through the user interface. (The command can also be called from other scripts, in which case arguments are passed in the conventional way.) The “`*`” raises an exception if the buffer is read-only. The “`r`” represents the beginning and end of the currently marked region. The “`\n`” separates the “`r`” from the following “`p`,” which indicates an optional nu-

---

**EXAMPLE 13.28**

Numbering lines with  
Emacs Lisp

```
(setq-default line-number-prefix "")
(setq-default line-number-suffix ") ")
(defun number-region (start end &optional initial)
 "Add line numbers to all lines in region.
With optional prefix argument, start numbering at num.
Line number is bracketed by strings line-number-prefix
and line-number-suffix (default \"\" and \") \")."
 (interactive "*r\np") ; how to parse args when invoked from keyboard
 (let* ((i (or initial 1))
 (num-lines (+ -1 initial (count-lines start end)))
 (fmt (format "%%" dd" (length (number-to-string num-lines))))))
 ; yields "%1d", "%2d", etc. as appropriate
 (finish (set-marker (make-marker) end)))
 (save-excursion
 (goto-char start)
 (beginning-of-line)
 (while (< (point) finish)
 (insert line-number-prefix (format fmt i) line-number-suffix)
 (setq i (1+ i))
 (forward-line 1))
 (set-marker finish nil))))
```

**Figure 13.9** Emacs Lisp function to number the lines in a selected region of text.

meric *prefix argument*. When the command is bound to a keystroke, a prefix argument of, say, 10 can be specified by preceding the keystroke with “C-u 10” (control-U 10).

As usual in Lisp, the `let*` command introduces a set of local variables in which later entries in the list (`fmt`) can refer to earlier entries (`num-lines`). A *marker* is an index into the buffer that is automatically updated to maintain its position when text is inserted in front of it. We create the `finish` marker so that newly inserted line numbers do not alter our notion of where the to-be-numbered region ends. We set `finish` to `nil` at the end of the script to relieve `emacs` of the need to keep updating the marker between now and whenever the garbage collector gets around to reclaiming it.

The `format` command is similar to `sprintf` in C. We have used it, once in the declaration of `fmt` and again in the call to `insert`, to pad all line numbers out to an appropriate length. The `save-excursion` command is roughly equivalent to an exception handler (e.g., a Java `try` block) with a `finally` clause that restores the current focus of attention (`(point)`) and the borders of the marked region.

Our script can be supplied to `emacs` by including it in a personal startup file (usually `~/.emacs`), by using the interactive `load-file` command to read some other file in which it resides, or by loading it into a buffer, placing the focus of attention immediately after it, and executing the interactive `eval-last-sexp` command. Once any of these has been done, we can invoke our command interactively by typing `M-x number-region <RET>` (meta-X, followed by the com-

mand name and the return key). Alternatively, we can *bind* our command to a keyboard shortcut:

```
(define-key global-map "\C-c#" 'number-region)
```

This one-line script, executed in any of the ways described above, binds our `number-region` command to the two-character sequence “`C-c #`” (control-C #).

### CHECK YOUR UNDERSTANDING

---

11. What is the most widely used scripting language?
  12. List the principal limitations of `sed`.
  13. What is meant by the *pattern space* in `sed`?
  14. Briefly describe the *fields* and *associative arrays* of `awk`.
  15. What is the Perl motto?
  16. Explain the special relationship between `while` loops and file handles in Perl. What is the meaning of the empty file handle, `<>`?
  17. Name three widely used commercial packages for mathematical computing.
  18. List several distinctive features of the R statistical scripting language.
  19. Explain the meaning of the `$` and `@` characters at the beginning of variable names in Perl. Explain the different meaning for the `$` sign in `Tcl`, and the still different meanings of `$`, `@`, and `@@` in Ruby.
  20. Describe the semantics of braces (`{ }`) and square brackets (`[ ]`) in `Tcl`.
  21. Which of the languages described in Section 13.2.4 uses indentation to control syntactic grouping?
  22. List several distinctive features of Python.
  23. Describe, briefly, how Ruby uses *blocks* and *iterators*.
  24. What capabilities must a scripting language provide in order to be used for extension?
  25. Name several commercial tools that use extension languages.
- 

## 13.3

### Scripting the World Wide Web

Much of the content of the World Wide Web—particularly the content that is visible to search engines—is static: pages that seldom, if ever, change. But hypertext, the abstract notion on which the web is based, was always conceived as a way

to represent “the complex, the changing, and the indeterminate” [Nel65]. Much of the power of the web today lies in its ability to deliver pages that move, play sounds, respond to user actions, or—perhaps most important—contain information created or formatted on demand, in response to the page fetch request.

From a programming languages point of view, simple playback of recorded audio or video is not particularly interesting. We therefore focus our attention here on content that is generated on the fly by a program—a script—associated with an Internet URI (uniform resource identifier).<sup>7</sup> Suppose we type a URI into a browser on a client machine, and the browser sends a request to the appropriate web server. If the content is dynamically created, an obvious first question is: does the script that creates it run on the server or the client machine? These options are known as *server-side* and *client-side* web scripting, respectively.

Server-side scripts are typically used when the service provider wants to retain complete control over the content of the page but can’t (or doesn’t want to) create the content in advance. Examples include the pages returned by search engines, Internet retailers, auction sites, and any organization that provides its clients with online access to personal accounts. Client-side scripts are typically used for tasks that don’t need access to proprietary information, and are more efficient if executed on the client’s machine. Examples include interactive animation, error-checking of fill-in forms, and a wide variety of other self-contained calculations.

### 13.3.1 CGI Scripts

The original mechanism for server-side web scripting is the Common Gateway Interface (CGI). A CGI script is an executable program residing in a special directory known to the web server program. When a client requests the URI corresponding to such a program, the server executes the program and sends its output back to the client. Naturally, this output needs to be something that the browser will understand: typically HTML.

CGI scripts may be written in any language available on the server’s machine, though Perl is particularly popular: its string-handling and “glue” mechanisms are ideally suited to generating HTML, and it was already widely available during the early years of the web. As a simple if somewhat artificial example, suppose we would like to be able to monitor the status of a server machine shared by some community of users. The Perl script in Figure 13.10 creates a web page titled by the name of the server machine and containing the output of the `uptime` and `who` commands (two simple sources of status information). The script’s initial `print` command produces an HTTP message header, indicating that what

---

**EXAMPLE 13.29**

Remote monitoring with a CGI script

---

<sup>7</sup> The term “URI” is often used interchangeably with “URL” (uniform resource locator), but the World Wide Web Consortium distinguishes between the two. All URIs are hierarchical (multi-part) names. URLs are one kind of URIs; they use a naming scheme that indicates where to find the resource. Other URIs can use other naming schemes.

```
#!/usr/bin/perl

print "Content-type: text/html\n\n";

$host = 'hostname'; chop $host;
print "<HTML>\n<HEAD>\n<TITLE>Status of ", $host,
 "</TITLE>\n</HEAD>\n<BODY>\n";
print "<H1>", $host, "</H1>\n";
print "<PRE>\n", 'uptime', "\n", 'who';
print "</PRE>\n</BODY>\n</HTML>\n";
```

**Figure 13.10** A simple CGI script in Perl. If this script is named `status.perl` and is installed in the server's `cgi-bin` directory, then a user anywhere on the Internet can obtain summary statistics and a list of users currently logged into the server by typing `hostname/cgi-bin/status.perl` into a browser window.

follows is HTML. Sample output from executing the script appears in Figure 13.11.

#### EXAMPLE 13.30

Adder web form with a CGI script

CGI scripts are commonly used to process online forms. A simple example appears in Figure 13.12. The `FORM` element in the HTML file specifies the URI of the CGI script, which is invoked when the user hits the Submit button. Values previously entered into the `INPUT` fields are passed to the script either as a trailing part of the URI (for a `get` type form) or on the standard input stream (for a `post` type form, shown here).<sup>8</sup> With either method, we can access the values using the `param` routine of the standard CGI Perl library, loaded at the beginning of our script.

### 13.3.2 Embedded Server-Side Scripts

Though widely used, CGI scripts have several disadvantages.

- The web server must launch each script as a separate program, with potentially significant overhead (though a CGI script compiled to native code can be very fast once running).
- Because the server has little control over the behavior of a script, scripts must generally be installed in a trusted directory by trusted system administrators; they cannot reside in arbitrary locations as ordinary pages do.
- The name of the script appears in the URI, typically prefixed with the name of the trusted directory, so static and dynamic pages look different to end users.

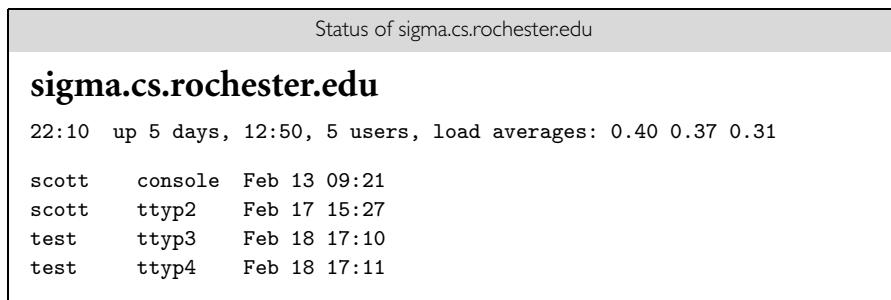
**8** One typically uses `post` type forms for one-time requests. A `get` type form appears a little clumsier, because arguments are visibly embedded in the URI, but this gives it the advantage of repeatability: it can be “bookmarked” by client browsers.

```

<HTML>
<HEAD>
<TITLE>Status of sigma.cs.rochester.edu</TITLE>
</HEAD>
<BODY>
<H1>sigma.cs.rochester.edu</H1>
<PRE>
22:10 up 5 days, 12:50, 5 users, load averages: 0.40 0.37 0.31

scott console Feb 13 09:21
scott tttyp2 Feb 17 15:27
test tttyp3 Feb 18 17:10
test tttyp4 Feb 18 17:11
</PRE>
</BODY>
</HTML>

```



**Figure 13.11** Sample output from the script of Figure 13.10. HTML source appears at top; the rendered page is below.

- Each script must generate not only dynamic content, but also the HTML tags that are needed to format and display it. This extra “boilerplate” makes scripts more difficult to write.

To address these disadvantages, most web servers now provide a “module loading” mechanism that allows interpreters for one or more scripting languages to be incorporated into the server itself. Scripts in the supported language(s) can then be embedded in “ordinary” web pages. The web server interprets such scripts directly, without launching an external program. It then replaces the scripts with the output they produce, before sending the page to the client. Clients have no way to even know that the scripts exist.

Embedable server-side scripting languages include PHP, Visual Basic (in Microsoft Active Server Pages), Cold Fusion (from Macromedia Corp.), and Java (via “Servlets” running in Java Server Pages). The most common of these is PHP. Though descended from Perl, PHP has been extensively customized for its target domain, with built-in support for (among other things) e-mail and MIME encoding, all the standard Internet communication protocols, authentication and

```
<HTML>
<HEAD>
<TITLE>Adder</TITLE>
</HEAD>
<BODY>
<FORM action="/cgi-bin/add.perl" method="post">
<P><INPUT name="argA" size=3>First addend

 <INPUT name="argB" size=3>Second addend
<P><INPUT type="submit">
</FORM>
</BODY>
</HTML>
```

Adder

12	First addend
34	Second addend

---

```
#!/usr/bin/perl

use CGI qw(:standard); # provides access to CGI input fields
$argA = param("argA"); $argB = param("argB"); $sum = $argA + $argB;

print "Content-type: text/html\n\n";

print "<HTML>\n<HEAD>\n<TITLE>Sum</TITLE>\n</HEAD>\n<BODY>\n";
print "<P>$argA plus $argB is $sum";
print "</BODY>\n</HTML>\n";
```

---

```
<HTML>
<HEAD>
<TITLE>Sum</TITLE>
</HEAD>
<BODY>
<P>12 plus 34 is 46</BODY>
</HTML>
```

Sum

12 plus 34 is 46

**Figure 13.12** An interactive CGI form. Source for the original web page is shown at the upper left, with the rendered page to the right. The user has entered 12 and 34 in the text fields. When the Submit button is pressed, the client browser sends a request to the server for URI /cgi-bin/add.perl. The values 12 and 13 are contained within the request. The Perl script, shown in the middle, uses these values to generate a new web page, shown in HTML at the bottom left, with the rendered page to the right.

security, HTML and URI manipulation, and interaction with dozens of database systems.

### EXAMPLE 13.31

Remote monitoring with a PHP script

The PHP equivalent of Figure 13.10 appears in Figure 13.13. Most of the text in this figure is standard HTML. PHP code is embedded between <?php and ?> delimiters. These delimiters are not themselves HTML; rather they identify the portions of the page that need to be executed by the PHP interpreter

```
<HTML>
<HEAD>
<TITLE>Status of <?php echo $host = chop('hostname') ?></TITLE>
</HEAD>
<BODY>
<H1><?php echo $host ?></H1>
<PRE>
<?php echo 'uptime', "\n", 'who' ?>
</PRE>
</BODY>
</HTML>
```

**Figure 13.13** A simple PHP script embedded in a web page. When served by a PHP-enabled host, this page performs the equivalent of the CGI script of Figure 13.10.

```
<HTML><BODY><P>
<?php
 for ($i = 0; $i < 20; $i++) {
 if ($i % 2) { ?>
<?php
 echo " $i"; ?>
<?php
 } else echo " $i";
 }
?>
</BODY></HTML>
```

**Figure 13.14** A fragmented PHP script. The `if` and `for` statements work as one might expect, despite the intervening raw HTML. When requested by a browser, this page displays the numbers from 0 to 19, with odd numbers written in bold.

to generate replacement text. The “boilerplate” parts of the page can thus appear verbatim; they need not be generated by `print` (Perl) or `echo` (PHP) commands. Note that the separate script fragments are part of a single program. The `$host` variable, for example, is set in the first fragment and used again in the second. ■

PHP scripts can even be broken into fragments in the middle of structured statements. Figure 13.14 contains a script in which `if` and `for` statements span fragments. In effect, the HTML text between the end of one script fragment and the beginning of the next behaves as if it had been output by an `echo` command. Web designers are free to use whichever approach (`echo` or escape to raw HTML) seems most convenient for the task at hand. ■

### ***Self-Posting Forms***

---

#### **EXAMPLE 13.33**

##### A fragmented PHP script

By changing the `action` attribute of the `FORM` element, we can arrange for the Adder page of Figure 13.12 to invoke a PHP script instead of a CGI script:

Adder web form with a  
PHP script

```

<HTML><HEAD><TITLE>Sum</TITLE></HEAD><BODY><P>
<?php
 $argA = $_REQUEST['argA']; $argB = $_REQUEST['argB'];
 $sum = $argA + $argB;
 echo "$argA plus $argB is $sum\n";
?>
</BODY></HTML>

<?php
 $argA = $_REQUEST['argA']; $argB = $_REQUEST['argB'];
 if (!isset($_REQUEST['argA']) || $argA == "" || $argB == "") {
 # form has not been posted, or arguments are incomplete
?>
 <HTML><HEAD><TITLE>Adder</TITLE></HEAD><BODY>
 <FORM action="adder.php" method="post">
 <P>First addend: <INPUT name="argA" size=3>
 Second addend: <INPUT name="argB" size=3>
 <P><INPUT type="submit">
 </FORM></BODY></HTML>
<?php
 } else { # form is complete; return results
?>
 <HTML><HEAD><TITLE>Sum</TITLE></HEAD><BODY><P>
<?php
 $sum = $argA + $argB;
 echo "$argA plus $argB is $sum\n";
?>
 </BODY></HTML>
<?php
}
?>

```

**Figure 13.15** An interactive PHP web page. The script at top could be used in place of the script in the middle of Figure 13.12. The lower script in the current figure replaces both the web page at the top and the script in the middle of Figure 13.12. It checks to see if it has received a full set of arguments. If it hasn't, it displays the fill-in form; if it has, it displays results.

```
<FORM action="add.php" method="post">
```

The PHP script itself is shown in the top half of Figure 13.15. Form values are made available to the script in an associative array (hash table) named `_REQUEST`. No special library is required. ■

Because our PHP script is executed directly by the web server, it can safely reside in an arbitrary web directory, including the one in which the Adder page resides. In fact, by checking to see how a page was requested, we can merge the form and the script into a single page, and let it service its own requests! We illustrate this option in the bottom half of Figure 13.15. ■

#### EXAMPLE 13.34

Self-posting Adder web form

### 13.3.3 Client-Side Scripts

While embedded server-side scripts are generally faster than CGI scripts, at least when startup cost predominates, communication across the Internet is still too slow for truly interactive pages. If we want the behavior or appearance of the page to change as the user moves the mouse, clicks, types, or hides or exposes windows, we really need to execute some sort of script on the client's machine.

Because they run on the web designer's site, CGI scripts and, to a lesser extent, embedable server-side scripts can be written in many different languages. All the client ever sees is standard HTML. Client-side scripts, by contrast, require an interpreter on the client's machine. As a result, there is a powerful incentive for convergence in client-side scripting languages: most designers want their pages to be viewable by as wide an audience as possible. While Visual Basic is widely used within specific organizations, where all the clients of interest are known to run Internet Explorer, pages intended for the general public almost always use JavaScript for interactive features.

**EXAMPLE 13.35**

Adder web form in  
JavaScript

Figure 13.16 shows a page with embedded JavaScript that imitates (on the client) the behavior of the Adder scripts of Figures 13.12 and 13.15. Function `doAdd` is defined in the header of the page so it is available throughout. In particular, it will be invoked when the user clicks on the Calculate button. By default the input values are character strings; we use the `parseInt` function to convert them to integers. The parentheses around `(argA + argB)` in the final assignment statement then force the use of integer addition. The other occurrences of `+` are string concatenation. To disable the usual mechanism whereby input data are submitted to the server when the user hits the enter or return key, we have specified a dummy behavior for the `onsubmit` attribute of the form.

Rather than replace the page with output text, as our CGI and PHP scripts did, we have chosen in our JavaScript version to append the output at the bottom. The HTML SPAN element provides a named place in the document where this output can be inserted, and the `getElementById` JavaScript method provides us with a reference to this element. The HTML *Document Object Model* (DOM), standardized by the World Wide Web Consortium, specifies a very large number of other elements, attributes, and user actions, all of which are accessible in JavaScript. Through them scripts can, at appropriate times, inspect or alter almost any aspect of the content, structure, or style of a page. ■

### 13.3.4 Java Applets

An applet is a program designed to run inside some other program. The term is most often used for Java programs that display their output in (a portion of) a web page. To support the execution of applets, most modern browsers contain a Java virtual machine.

```

<HTML>
<HEAD>
<TITLE>Adder</TITLE>
<SCRIPT type="text/javascript">
function doAdd() {
 argA = parseInt(document.adder.argA.value)
 argB = parseInt(document.adder.argB.value)
 x = document.getElementById('sum')
 while (x.hasChildNodes())
 x.removeChild(x.lastChild) // delete old content
 t = document.createTextNode(argA + " plus "
 + argB + " is " + (argA + argB))
 x.appendChild(t)
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="adder" onsubmit="return false">
<P><INPUT name="argA" size=3> First addend

 <INPUT name="argB" size=3> Second addend
<P><INPUT type="button" onclick="doAdd()" value="Calculate">
</FORM>
<P>
</BODY>
</HTML>

```

Adder

12	First addend
34	Second addend

Calculate

12 plus 34 is 46

**Figure 13.16** An interactive JavaScript web page. Source appears at left. The rendered version on the right shows the appearance of the page after the user has entered two values and hit the Calculate button, causing the output message to appear. By entering new values and clicking again, the user can calculate as many sums as desired. Each new calculation will replace the output message.

Like JavaScript, Java applets can be used to create animated or interactive pages. Together with the similarity in language names, the fact that many tasks can be accomplished with either mechanism has created a great deal of confusion between the two (see sidebar on page 710). In fact, however, they are very different.

#### EXAMPLE 13.36

Embedding an applet in a web page

To embed an applet in a web page, one would traditionally use an APPLET tag:

```
<APPLET width=150 height=150 code="Clock.class">
```

Seeing this element embedded in the page, the client browser would request the URI *Clock.class* from the server. Assuming the server returned an applet, it would run this applet and display the output on the page. ■

Unlike a JavaScript script, an applet does not produce HTML output for the browser to render. Rather it directly controls a portion of the page's real estate, in which it uses routines from one of Java's graphical user interface (GUI) libraries (typically AWT or Swing) to display whatever it wants. The *width* and *height*

attributes of the APPLET element tell the browser how big the applet's portion of the page should be.

In effect, applets allow the web designer to escape from HTML entirely, and to create a very precise “look and feel,” independent of any design choices embodied by the browser. Images, of course, provide another way to escape from HTML, with static or simple animated content, as do embedded objects of other kinds (movies in Flash or QuickTime format are popular examples). Most modern browsers provide a “plug-in” mechanism that allows the installation of interpreters for arbitrary formats. In support of these, the HTML 4.0 standard provides a generic OBJECT element that is meant to be used for any embedded content not rendered by the browser itself. The APPLET element is now officially deprecated: one is supposed to use the following instead.

```
<P><OBJECT codetype="application/java" classid="java:Clock.class"
 width=150 height=150>
```

Applets are subject to certain restrictions intended to prevent them from damaging the client’s machine. For the most part, however, they can make use of the entire Java language, and it is usually a simple task to convert an applet to a stand-alone program or vice versa. The typical applet has no significant interaction with the browser or any other program. For this reason, applets are generally *not* considered a scripting mechanism.

## DESIGN & IMPLEMENTATION

### JavaScript and Java

Despite its name, JavaScript has no connection to Java beyond some superficial syntactic similarity. The language was originally developed by Brendan Eich at Netscape Corp. in 1995. Eich called his creation *LiveScript*, but the company chose to rename it as part of a joint marketing agreement with Sun Microsystems, prior to its public release. Trademark on the JavaScript name is actually owned by Sun.

Netscape’s browser was still the market leader in 1995, and JavaScript usage grew extremely fast. To remain competitive, developers at Microsoft added JavaScript support to Internet Explorer, but they used the name *JScript* instead, and they introduced a number of incompatibilities with the Netscape version of the language. A common version was standardized as *ECMAScript* by the European standards body in 1997, but major incompatibilities remained in the Document Object Models provided by different browsers. These have been gradually resolved through a series of standards from the World Wide Web Consortium, but legacy pages and legacy browsers continue to plague web developers.

 **CHECK YOUR UNDERSTANDING**

26. Explain the distinction between *server-side* and *client-side* web scripting.
27. List the tradeoffs between CGI scripts and embedded PHP.
28. Why are CGI scripts usually installed only in a special directory?
29. Explain how a PHP page can service its own requests.
30. Why might we prefer to execute a web script on the server rather than the client? Why might we sometimes prefer the client instead?
31. What is the HTML *Document Object Model*? What is its significance for client-side scripting?
32. What is the relationship between JavaScript and Java?

**DESIGN & IMPLEMENTATION****Sandboxing**

Security becomes an issue whenever code is executed using someone else's resources. Web servers are usually installed with very limited access rights and with only a limited view of the file system of the server machine. This generally limits the set of pages they can serve to a well-defined subset of what would be visible to users logged into the server machine directly. Because they are separate executable programs, CGI scripts can be designed to run with the privileges of whoever installed them. To prevent users on the server machine from accidentally or intentionally passing their privileges to arbitrary users on the Internet, most system administrators configure their servers so that CGI scripts must reside in a special directory, and be installed by a trusted user. Embedded server-side scripts can reside in any file because they are guaranteed to run with the (limited) rights of the server.

A larger risk is posed by code downloaded over the Internet and executed on a client machine. Because such code is in general untrusted, it must be executed in a carefully controlled environment, sometimes called a *sandbox*, to prevent it from doing any damage. As a general rule, JavaScript scripts cannot access the local file system, memory management system, or network, nor can they manipulate documents from other sites. Java applets, likewise, have only limited ability to access external resources. Reality is a bit more complicated, of course: sometimes a script needs access to, say, a temporary file of limited size, or a network connection to a trusted server. Mechanisms exist to certify sites as *trusted*, or to allow a trusted site to certify the trustworthiness of pages from other sites. Scripts on pages obtained through a trusted mechanism may then be given extended rights. Such mechanisms must be used with care. Finding the right balance between security and functionality remains one of the central challenges of the Web, and of distributed computing in general.

- 
33. What is an *applet*? Why are applets usually not considered a form of scripting?
- 

### 13.3.5 XSLT

Most readers will undoubtedly have had the opportunity to write, or at least to read, the HTML (hypertext markup language) used to compose web pages. HTML has, for the most part, a nested structure in which fragments of documents (*elements*) are delimited by *tags* that indicate their purpose or appearance. We saw in Section 13.2.2, for example, that top-level headings are delimited with `<H1>` and `</H1>`. HTML is inspired by an older standard known as SGML (standard generalized markup language), widely used in the business world to represent structured data. Because of the informal way in which the web evolved, and the sometimes incompatible and ad hoc extensions made by competing vendors, standardization of HTML has been a long and complicated process. Incompatibilities between browsers continue to frustrate web designers, and several features of HTML that have been *deprecated*<sup>9</sup> in the most recent standards are nonetheless still widely used. Other features, while not deprecated, are widely regarded in hindsight to have been mistakes.

---

**EXAMPLE 13.38**

---

Content versus  
appearance in HTML

Probably the biggest problem with HTML is that it does not adequately distinguish between the *content* and the *appearance* of a document. As a trivial example, web designers frequently use `<I> ... </I>` tags to request that text be set in an italic font, when `<EM> ... </EM>` (emphasis) would be more appropriate. A browser for the visually impaired might choose to emphasize text with something other than italics, and might render book titles (also often specified with `<I> ... </I>`) in some entirely different fashion. More significantly, many web designers use tables (`<TABLE> ... </TABLE>`) to control the relative positioning of elements on a page, when the content isn't tabular at all. As more and more vendors work to bring web content to cell phones, televisions, handheld computers, and audio-only devices, the need to distinguish between content and appearance (presentation) is becoming increasingly critical. SGML has always made this distinction, but it is widely seen as overkill—far too complex for use on the web.

---

This is where XML steps in. XML (extensible markup language) is a deliberately streamlined descendant of SGML with at least three important advantages over HTML: (1) its syntax and semantics are more regular and consistent, and more consistently implemented across platforms; (2) it is *extensible*, meaning that users can define new tags; and (3) it specifies content only, leaving presentation to a companion standard known as XSL (extensible stylesheet language). XSLT is a portion of XSL devoted to *transforming* XML: selecting, reorganizing, and

---

**9** A *deprecated* feature is one whose use is officially discouraged, but permitted on a temporary basis to ease the transition to new and presumably better alternatives.

modifying tags and the elements they delimit—in effect, scripting the processing of data represented in XML.

### **Internet Alphabet Soup**

Learning about web standards can be a daunting task: there is an enormous number of buzzwords, standards, and multiletter abbreviations. It helps to remember the three families of markup languages—SGML, HTML, and XML—and to know that each has a corresponding *stylesheet language*: DSSSL, CSS, and XSL, respectively. A stylesheet language is used to control the presentation of a document, separate from its content. Stylesheet languages are essential for SGML and XML; without them there is no way to know whether a <RECORD> represents a database entry, an antique phonograph album, or an Olympic achievement, much less how to display it. HTML is less dependent on stylesheets, but web sites increasingly use CSS to create a uniform “look and feel” across a collection of pages without embedding redundant information in every page.

SGML and DSSSL remain important in the business world but are little used on the web. HTML is likely to persist for a very long time, but its lack of extensibility and its mix of content and presentation are increasingly perceived as fundamental limitations. XML is widely viewed as the notation of the future. Even for documents that remain in HTML, designers are likely to migrate toward XHTML (extensible hypertext markup language), an almost (but not quite) backward compatible variant of HTML that conforms to the XML standard.

### **XML and XHTML**

#### **EXAMPLE 13.39**

Well-formed XHTML

An XML document must be *well formed*: tags must either constitute properly nested, matched pairs or be explicit singletons, which end with a “>” delimiter. The following fragment, for example, is well formed (though incomplete) XHTML.

```
<q>I defy the tyranny of precedent</q>
(Clara Barton).
```

Here the quotation element (<q> ... </q>) is nested inside the emphasis element (<em> ... </em>). Moreover the anchor element (<a ... />), which can serve as the target of a link, is explicitly a singleton; it has a slash before its closing “>” delimiter. (To avoid confusing certain legacy browsers, one sometimes needs a space in front of the slash.) The example fragment would be malformed if the slash were missing or if the opening <em><q> tags were reversed (<q><em>). ■

Well-formedness is a simple syntactic rule, like the requirement that parentheses be balanced in Lisp. It makes XML (and thus XHTML) much easier than plain HTML to parse and to process automatically. The careful reader may also have noticed that we used lowercase letters for tags in XHTML, where previous HTML examples were all in uppercase. HTML is case-insensitive; either style is accepted, though uppercase has been the convention in standards documents. XML is case-sensitive, so <em> and <EM> are different. The XHTML designers had to pick one.

Going against the existing convention (but not the existing rules) preserves backward compatibility while helping the reader identify documents that are likely to conform to the newer standard.

The set of tags to be used in an XML document is specified by either a *document type definition* (DTD) or an *XML Schema*. DTDs are inherited from SGML. They indicate which tags are allowed, whether they are pairs or singletons, whether they permit attributes (name-value pairs like the `id="favorite-quote"` in Example 13.39), and whether any attributes are mandatory. The rules of the DTD take the form of XML *declarations*, which look like elements beginning with a “`<!`” delimiter. These can be included directly in the XML document. More often they are kept in an external document with its own URI, and the XML document begins with a `<!DOCTYPE ...>` declaration that specifies that URI. (Comments also look like declarations: `<!-- ignored -->`.) If an XML document has no explicit DTD (neither in-line nor external), it is said to define a DTD *implicitly* by virtue of which tags are actually used.

XML Schemas are a newer mechanism, meant to replace DTDs. They are written in XSD, the XML Schema Definition language, which is itself an example of well-formed XML, defined by a DTD. Because they are written in XSD, XML Schemas can be created using XML-aware editors, parsed with XML parsers, and transformed with XSLT. In comparison to DTDs, XSD provides a significantly richer vocabulary for specifying syntactic rules. Among other things, it allows the designer to specify the data types of elements and attributes in considerable detail, providing a level of automatic checking not possible with DTDs. XSD also supports inheritance, so one XML Schema can be defined as an extension of another. Unfortunately, as of this writing DTDs are still more common than XML Schemas. In particular, XHTML is officially defined by a set of DTDs; the corresponding XML Schemas are still a work in progress. We will rely on DTDs in the remainder of this section.

---

**EXAMPLE 13.40**

XHTML to display a favorite quote

Because tags must nest in XML, a document has a natural tree-based structure. Figure 13.17 shows the source for a small but complete XHTML document together with the tree it represents. There are three kinds of nodes in the tree: elements (delimited by tags in the source), text, and attributes. The internal (non-leaf) nodes are all elements. Everything nested between the beginning and ending tags of an element is an attribute or child of that element in the tree.

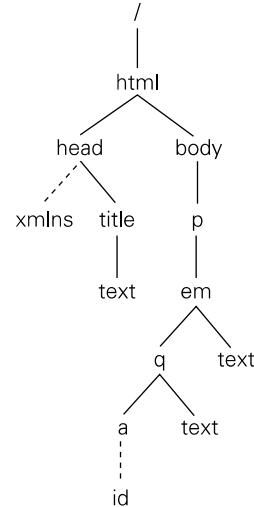
Our document begins with an `<?xml ... ?>` *processing directive*. This directive indicates the version of XML and the character encoding used in the rest of the document. The directive is included for the benefit of tools that process the document; it isn’t part of the XML source itself. (Note that we’ve seen processing directives before, in Section 13.3.2, where they provided input to the PHP interpreter.)

The second line of our document is a `<!DOCTYPE ...>` declaration that names an XHTML DTD at the World Wide Web Consortium. The remainder of the document is data. The root, named “`/`”, has one child: the `html` element. This in turn has two children: the `head` and the `body`. The `head` has a `title` child

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Favorite Quote</title>
</head>
<body>
<p>
<q>
I defy the tyranny of precedent</q>
(Clara Barton).
</p>
</body>
</html>

```



**Figure 13.17** A complete XHTML document and its corresponding tree. Child relationships are shown with solid lines, attributes with dashed lines.

and an `xmlns` attribute. The latter declares `xhtml` to be the default *namespace* for the document. Namespaces in XML are similar to the namespaces of C99 or the packages of Java (Section 3.7); they allow us to give tag names a disambiguating prefix: `xhtml:table` versus `furniture:table`. With the value we have specified for the `xmlns` attribute, any tag in the document that doesn't have a prefix will automatically be interpreted as being in the `xhtml` namespace. ■

### XSLT, XPath, and XSL-FO

XSL (extensible stylesheet language) can be thought of as a language for specifying what to *do* with an XML document. It has three sublanguages, called XSLT, XPath, and XSL-FO. XSLT is a scripting language that takes XML as input and produces textual output—often transformed XML or HTML but potentially other formats as well.

XPath is a language used to name things in XML files. XPath names frequently appear in the attributes of XSLT elements. Returning to Figure 13.17, the quotation element of our document could be named in XPath as `/html/body/p/em/q`. The quotation element and its text-node sibling, together, could be named as `/html/body/p/em/*`. XPath includes a rich set of naming mechanisms, including absolute (from the root) and relative (from the current node) navigation; wildcards; predicates; substring and regular expression manipulation; and counting and arithmetic functions. We will see some of these in the extended example below. ■

XSL-FO (XSL formatting objects) is a set of tags to specify the *layout* (appearance) of a document, in terms of pages, regions (e.g., header, body, footer), blocks (paragraph, table, list), lines, and in-line elements (character, image). An XSLT

#### EXAMPLE 13.41

XPath names for XHTML elements

```

<?xml version="1.0" encoding="UTF-8"?>
<?xmlstylesheet type="text/xsl" href="bib.xsl"?>
<bibliography>
 <book>
 <author>Guido van Rossum</author>
 <editor>Fred L. Drake, Jr.</editor>
 <title>The Python Language Reference Manual</title>
 <publisher>Network Theory, Ltd.</publisher>
 <address>Bristol, UK</address>
 <year>2003</year>
 <note>Available at <uri>http://www.network-theory.co.uk/docs/pylang/</uri></note>
 </book>
 <article>
 <author>John K. Ousterhout</author>
 <title>Scripting: Higher-Level Programming for the 21st Century</title>
 <journal>Computer</journal>
 <volume>31</volume>
 <number>3</number>
 <month>March</month>
 <year>1998</year>
 <pages>23–30</pages>
 </article>
 <inproceedings>
 <author>Theodor Holm Nelson</author>
 <title>Complex Information Processing: A File Structure for the
 Complex, the Changing, and the Indeterminate</title>
 <booktitle>Proceedings of the Twentieth ACM National Conference</booktitle>
 <month>August</month>
 <year>1965</year>
 <address>Cleveland, OH</address>
 <pages>84–100</pages>
 </inproceedings>
 <inproceedings>
 <author>Stephan Kepser</author>
 <title>A Simple Proof for the Turing-Completeness of XSLT and
 XQuery</title>
 <booktitle>Proceedings, Extreme Markup Languages 2004</booktitle>
 <address>Montréal, Canada</address>
 <year>2004</year>
 <month>August</month>
 <note>Available at <uri>http://www.mulberrytech.com/Extreme/Proceedings/html
 /2004/Kepser01/EML2004Kepser01.html</uri></note>
 </inproceedings>

```

**Figure 13.18** A bibliography in XML. References (two books, a journal article, and three conference papers) appear in arbitrary order. The Kepser URI has been wrapped to fit on the printed page. (*continued*)

```
<inproceedings>
 <author>David G. Korn</author>
 <title><code>ksh</code>: An Extensible High Level Language</title>
 <booktitle>Proceedings of the USENIX Very High Level Languages
 Symposium</booktitle>
 <address>Santa Fe, NM</address>
 <year>1994</year>
 <month>October</month>
 <pages>129–146</pages>
</inproceedings>
<book>
 <author>Larry Wall</author>
 <author>Tom Christiansen</author>
 <author>Jon Orwant</author>
 <title>Programming Perl</title>
 <edition>third</edition>
 <publisher>O’Reilly and Associates</publisher>
 <address>Cambridge, MA</address>
 <year>2000</year>
</book>
</bibliography>
```

Figure 13.18 (continued)

script might be used to add XSL-FO tags to an XML document, or to transform a document that already has XSL-FO tags in it—perhaps to split a long single-page document intended for the web into a multipage document intended for printing on paper. For the sake of simplicity, we will not use XSL-FO in any of our examples. Rather we will format XML documents by using XSLT to turn them into HTML.

An XML document can explicitly specify an XSLT script that should be used to transform or format it. This is a standard but somewhat restrictive way to go about things: by tying a single stylesheet to the XML file we compromise the separation between content and presentation that was a principal motivation for creating XML in the first place. An alternative is to use client-side JavaScript or server-side PHP to invoke the XSLT processor, passing the XML document and the XSLT script as arguments. Unfortunately, as of this writing the details vary across both server and client platforms.

#### **Extended Example: Bibliographic Formatting**

---

##### **EXAMPLE 13.42**

Creating a reference list  
with XSLT

As an example of a task for which we might realistically use XSLT, consider the creation of a bibliographic reference list. Figure 13.18 contains XML source for such a list. (Field names have been borrowed from BIBTeX [Lam94, Appendix B].) The document begins with a pair of processing directives: one to specify the XML version and character encoding, the other to specify the XSL stylesheet to be used to format the file.

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html><head><title>Bibliography</title></head><body><h1>Bibliography</h1>
<xsl:for-each select="bibliography/*"><xsl:sort select="title"/>
<xsl:apply-templates select=". "/>
</xsl:for-each>
</body></html>
</xsl:template>

<xsl:template match="bibliography/article">
<q><xsl:apply-templates select="title/node()"/>, </q>
by <xsl:call-template name="author-list"/>.
<xsl:apply-templates select="journal/node()"/>
<xsl:text> </xsl:text><xsl:apply-templates select="volume/node()"/>
:<xsl:apply-templates select="number/node()"/>
(<xsl:apply-templates select="month/node()"/><xsl:text> </xsl:text>
<xsl:apply-templates select="year/node()"/>,
pages <xsl:apply-templates select="pages/node()"/>.
<xsl:if test="note"><xsl:apply-templates select="note/node()"/>.</xsl:if>
</xsl:template>

<xsl:template match="bibliography/book">
<xsl:apply-templates select="title/node()"/>,
by <xsl:call-template name="author-list"/>.
<xsl:apply-templates select="publisher/node()"/>,
<xsl:apply-templates select="address/node()"/>,
<xsl:if test="edition">
<xsl:apply-templates select="edition/node()"/> edition, </xsl:if>
<xsl:apply-templates select="year/node()"/>.
<xsl:if test="note"><xsl:apply-templates select="note/node()"/>.</xsl:if>
</xsl:template>

<xsl:template match="bibliography/inproceedings">
<q><xsl:apply-templates select="title/node()"/>, </q>
by <xsl:call-template name="author-list"/>.
In <xsl:apply-templates select="booktitle/node()"/>
<xsl:if test="pages">, pages <xsl:apply-templates select="pages/node()"/></xsl:if>
<xsl:if test="address">, <xsl:apply-templates select="address/node()"/></xsl:if>
<xsl:if test="month">, <xsl:apply-templates select="month/node()"/></xsl:if>
<xsl:if test="year">, <xsl:apply-templates select="year/node()"/></xsl:if>.
<xsl:if test="note"><xsl:apply-templates select="note/node()"/>.</xsl:if>
</xsl:template>

```

**Figure 13.19** Bibliography stylesheet in XSL. This script will generate HTML when applied to a bibliography like that of Figure 13.18. (continued)

```

<xsl:template name="author-list"> <!-- format author list -->
 <xsl:for-each select="author|editor">
 <xsl:if test="last() > 1 and position() = last()"> and </xsl:if>
 <xsl:apply-templates select=".//node()"/>
 <xsl:if test="self::editor"> (editor)</xsl:if>
 <xsl:if test="last() > 2 and last() > position()">, </xsl:if>
 </xsl:for-each>
</xsl:template>

<xsl:template match="uri"> <!-- format link -->
 <a><xsl:attribute name="href"><xsl:value-of select=".//node()"/></xsl:attribute>
 <xsl:value-of select="substring-after(., 'http://')"/>
</xsl:template>

<xsl:template match="@*|node()"> <!-- default: copy content -->
 <xsl:copy><xsl:apply-templates select="@*|node()"/></xsl:copy>
</xsl:template>

</xsl:stylesheet>

```

**Figure 13.19 (continued)**

At the top level, the `bibliography` element consists of a series of `book`, `article`, and `inproceedings` elements, each of which may contain elements for author and editor names, title, publisher, date and address, and so on. Some elements may contain nested `uri` elements, which specify online links. Characters that cannot be represented in ASCII are shown as Unicode *character entities*, as described in the sidebar on page 313.

Figure 13.19 contains an XSLT stylesheet (script) to format the bibliography as HTML, which may then be rendered in a browser. This script was named at the beginning of the XML document (Figure 13.18). Like the XML document, the script begins with a pair of processing directives. The first specifies the XML version and character encoding; the second specifies the XSL version and namespace. The remainder of the script contains a mix of XSL and HTML elements. The XSL tags all specify the `xsl:` namespace explicitly. They are recognized by the XSLT processor. Elements from other namespaces are treated as ordinary text, to be copied through to the output when encountered.

The fundamental construct in XSLT is the `template`, which specifies a set of *instructions* to be applied to nodes in an XML source tree. Templates are typically invoked by executing an `apply-templates` or `call-template` instruction in some other template. Each invocation has a concept of *current node*. The execution as a whole begins by invoking an initial template with the root of the source tree (`/`) as current node. In our bibliographic example, the initial template is the one at the top of the script, because its `match` attribute is the XPath expression `"//"`. The body of the initial template begins with a string of HTML elements and text. This string is copied directly to the output. The `for-each` element, however, is an XSLT instruction, so it is executed.

The `select` attribute of the `for-each` uses an XPath expression (`"bibliography/*"`) to build a *node set* consisting of all top-level entries in our bibliography. Other expressions could have been used if we wanted to be selective: `"bibliography/*[year>=2000]"` would match only recent entries; `"bibliography/*[note]"` would match only entries with `note` elements; `"bibliography/article|bibliography/book"` would match only articles and books.

The nested `sort` instruction forces the selected node set to be ordered alphabetically by title. The body of the `for-each` is then executed with each entry in turn selected as current node. The body contains a recursive invocation of `apply-templates`, bracketed by HTML list tags (`<li>...</li>`). These tags are copied to the output, with the result of the recursive call nested in between.

So how does the recursive call work? Its `select` attribute, like that of `for-each`, uses XPath to build a node set. In this case it is the trivial node set containing only `". "`, the current node of the current iteration of `for-each`. The XSLT processor searches for a template that matches this node. We have created three appropriate candidates, one for each kind of bibliographic entry. When it finds the matching template, the processor invokes it, with an updated notion of current node.

Each of our three main templates contains a set of instructions to format its kind of entry (article, book, conference paper). Most of the instructions use additional invocations of `apply-templates` to format individual portions of an entry (author, title, publisher, etc.). Interspersed in these instructions are snippets of text and HTML elements. In several cases we use an `if` instruction to generate output only when a given XML element is present in the source. In most of these the recursive call uses the XPath `node()` function to select all children of the element in question.

White space is ignored when it comes between the end of one instruction and the beginning of the next. To force white space into the output in this case, we must delimit it with `<text> ... </text>` tags. Extra white space (e.g., after the ends of sentences) is specified with the “nonbreaking space” character entity, `&#160;`.

Three extra templates end our script. The most interesting of these serves to format author lists. It has a `name` attribute rather than a `match` attribute, and is invoked with `call-template` rather than `apply-templates`. A called template always takes the current node of the caller—in this case the node that represents a bibliographic entry. Internally, the author list template executes a `for-each` instruction that selects all child nodes representing authors or editors. The `for-each`, in turn, uses the XPath `last()` and `position()` functions to determine how many names there are, and where each name falls in the list. It inserts the word “and” between the final two names, and puts commas after all names but the last in lists of three or more.

The template with `match="uri"` serves to format URIs that appear anywhere in the XML source. It creates an HTML link in the output, but uses the XPath

substring-after function to strip the leading *http://* off the visible text. XPath provides a variety of similar functions for string and regular expression manipulation. The value-of instruction copies the contents of the selected node to the output, as a character string.

Our final template serves as a default case. The XPath expression "@\*|node()" will match any attribute or other node in the XML source. Inside, the copy instruction copies the node's tags, if any, to the output, with the result of a recursive call to apply-templates in between. The "@\*|node()" on the recursive call selects a node set consisting of all the current node's attributes and children. The end result is that any XML elements in the source that are delimited by tags for which we do not have special templates will be regenerated in the output just as they appear in the source. The recursion stops at text nodes and attributes, which are the leaves of the XML tree.

HTML output from our script appears in Figure 13.20. The rendered web page appears in Figure 13.21.

While lengthy by the standards of this text, our example illustrates only a fraction of the capabilities of XSLT. In the standard categorization of programming languages, the notation is strongly declarative: values may have names, but there are no mutable variables and no side effects. There is a limited looping mechanism (*for-each*), but the real power comes from recursion, and from recursive traversal of XML trees in particular. ■



#### CHECK YOUR UNDERSTANDING

34. Explain the relationships among SGML, HTML, and XML. What are their corresponding stylesheet languages?
35. Why does XML work so hard to distinguish between *content* and *appearance*?
36. What are the three main components of XSL? What are their respective purposes?
37. What is XHTML? How does it differ from HTML?
38. Explain the correspondence between XML documents and trees.
39. What does it mean for an XML document to be *well formed*?
40. What is a *document type definition* (DTD)? An *XML Schema*? Briefly, how do they compare?
41. Explain the distinctions (syntactic and semantic) among *elements*, *declarations*, and *processing directives* in XML. Also explain the distinctions among *elements*, *tags*, and *attributes*.
42. Summarize the execution model of XSLT. In a nutshell, how does it work?
43. Explain the difference between *applying* templates and *calling* them in XSLT.

```

<html><head><title>Bibliography</title></head>
<body><h1>Bibliography</h1>

 <q>A Simple Proof for the Turing-Completeness of XSLT and XQuery,</q>
 by Stephan Kepser. In Proceedings, Extreme Markup Languages
 2004, Montr´al, Canada, August, 2004. Available at
 <a href="http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01
 /EML2004Kepser01.html">www.mulberrytech.com/Extreme/Proceedings/html/2004
 /Kepser01/EML2004Kepser01.html.

 <q>Complex Information Processing: A File Structure for the Complex,
 the Changing, and the Indeterminate,</q> by Theodor Holm Nelson.
 In Proceedings of the Twentieth ACM National Conference,
 pages 84–100, Cleveland, OH, August, 1965.

 <q><code>ksh</code>: An Extensible High Level Language,</q> by David
 G. Korn. In Proceedings of the USENIX Very High Level Languages
 Symposium, pages 129–146, Santa Fe, NM, October, 1994.

 Programming Perl, by Larry Wall, Tom Christiansen, and Jon
 Orwant. O’Reilly and Associates, Cambridge, MA, third edition,
 2000.

 <q>Scripting: Higher-Level Programming for the 21st Century,</q> by
 John K. Ousterhout. Computer 31:3 (March 1998), pages
 23–30.

 The Python Language Reference Manual, by Guido van Rossum and
 Fred L. Drake, Jr. (editor). Network Theory, Ltd., Bristol, UK, 2003.
 Available at www.network-
 theory.co.uk/docs/pylang/.

</body></html>

```

Figure 13.20 Result of applying the stylesheet of Figure 13.19 to the bibliography of Figure 13.18.

## 13.4 Innovative Features

In Section 13.1.1 we listed several common characteristics of scripting languages.

1. Both batch and interactive use
2. Economy of expression
3. Lack of declarations; simple scoping rules
4. Flexible dynamic typing
5. Easy access to other programs

Bibliography

## Bibliography

1. “A Simple Proof for the Turing-Completeness of XSLT and XQuery,” by Stephan Kepser. In *Proceedings, Extreme Markup Languages 2004*, Montréal, Canada, August, 2004. Available at [www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01/EML2004Kepser01.html](http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01/EML2004Kepser01.html).
2. “Complex Information Processing: A File Structure for the Complex, the Changing, and the Indeterminate,” by Theodor Holm Nelson. In *Proceedings of the Twentieth ACM National Conference*, pages 84–100, Cleveland, OH, August, 1965.
3. *ksh: An Extensible High Level Language*, by David G. Korn. In *Proceedings of the USENIX Very High Level Languages Symposium*, pages 129–146, Santa Fe, NM, October, 1994.
4. *Programming Perl*, by Larry Wall, Tom Christiansen, and Jon Orwant. O’Reilly and Associates, Cambridge, MA, third edition, 2000.
5. “Scripting: Higher-Level Programming for the 21st Century,” by John K. Ousterhout. *Computer* 31:3 (March 1998), pages 23–30.
6. *The Python Language Reference Manual*, by Guido van Rossum and Fred L. Drake, Jr. (editor). Network Theory, Ltd., Bristol, UK, 2003. Available at [www.network-theory.co.uk/docs/pylang/](http://www.network-theory.co.uk/docs/pylang/).

**Figure 13.21** Rendered version of the HTML in Figure 13.20.

6. Sophisticated pattern matching and string manipulation
7. High-level data types

Several of these are discussed in more detail in the subsections below. Specifically, Section 13.4.1 considers naming and scoping in scripting languages; Section 13.4.2 discusses string and pattern manipulation; and Section 13.4.3 considers data types. Items (1), (2), and (5) in our list, while important, are not particularly difficult or subtle, and will not be considered further here.

### 13.4.1 Names and Scopes

Most scripting languages (Scheme is the obvious exception) do not require variables to be declared. A few languages, notably Perl and JavaScript, permit optional declarations, primarily as a sort of compiler-checked documentation. Perl can be run in a mode (`use strict 'vars'`) that requires declarations.

With or without declarations, most scripting languages use dynamic typing. Values are generally self-descriptive, so the interpreter can perform type checking at run time, or coerce values when appropriate. Tcl is unusual in that all values—even lists—are represented internally as strings, which are parsed as appropriate to support arithmetic, indexing, and other operations.

Nesting and scoping conventions vary quite a bit. Scheme, Python, JavaScript, and R provide the classic combination of nested subroutines and static (lexi-

cal) scope. Tcl allows subroutines to nest but uses dynamic scope (more on this below). Named subroutines (methods) do not nest in PHP or Ruby, and they only sort of nest in Perl (more on this below as well), but Perl and Ruby join Scheme, Python, JavaScript, and R in providing first-class anonymous local subroutines. Nested blocks are statically scoped in Perl. In Ruby they are part of the named scope in which they appear. Scheme, Perl, Python, and R provide unlimited extent for variables captured in closures. Ruby and JavaScript do not. PHP, R, and the major glue languages (Perl, Tcl, Python, Ruby) all have sophisticated namespace mechanisms for information hiding and the selective import of names from separate modules.

### ***What Is the Scope of an Undeclared Variable?***

In languages with static scope, the lack of declarations raises an interesting question: when we access a variable *x*, how do we know if it is local, global, or (if scopes can nest) something in between? Existing languages take several different approaches. In Perl all variables are global unless explicitly declared. In PHP they are local unless explicitly imported (and all imports are global, since scopes do not nest). Ruby, too, has only two real levels of scoping, but as we saw in Section 13.2.4 it distinguishes between them using prefix characters on names: *foo* is a local variable; *\$foo* is a global variable; *@foo* is an instance variable of the current object (the one whose method is currently executing); *@@foo* is an instance variable of the current object's *class* (shared by all sibling instances). (Note: As we shall see in Section 13.4.3, Perl uses similar prefix characters to indicate *type*. These very different uses are a potential source of confusion for programmers who switch between the two languages.)

Perhaps the most interesting scope resolution rule is that of Python and R. In these languages a variable that is written is assumed to be local, unless it is explicitly imported. A variable that is only read in a given scope is found in the closest enclosing scope that contains a defining write. Consider, for example, the Python program of Figure 13.22. Here we have a set of nested subroutines, as indicated by indentation level. The main program calls *outer*, which calls *middle*, which in turn calls *inner*. Before its call, the main program writes both *i* and *j*. *Outer* reads *j* (to pass it to *middle*) but does not write it. It does, however, write *i*. Consequently *outer* reads the global *j*, but has its own *i*, different from the global one. *Middle* reads both *i* and *j*, but it does not write either, so it must find them in surrounding scopes. It finds *i* in *outer*, and *j* at the global level. *Inner*, for its part, also writes the global *i*. When executed the program prints

```
(2, 3, 3)
4 3
```

Note that while the tuple returned from *middle* (forwarded on by *outer*, and printed by the main program) has a 2 as its first element, the global *i* still contains the 4 that was written by *inner*. Note also that while the write to *i* in *outer* appears textually after the read of *i* in *middle*, its scope extends over all of *outer*, including the body of *middle*.

---

#### **EXAMPLE 13.43**

##### Scoping rules in Python

```

i = 1; j = 3
def outer():
 def middle(k):
 def inner():
 global i # from main program, not outer
 i = 4
 inner()
 return i, j, k # 3-element tuple
 i = 2 # new local i
 return middle(j) # old (global) j

print outer()
print i, j

```

**Figure 13.22** A program to illustrate scope rules in Python. There is one instance each of `j` and `k`, but two of `i`: one global and one local to `outer`. The scope of the latter is all of `outer`, not just the portion after the assignment. The `global` statement provides `inner` with access to the outermost `i`, so it can write it without defining a new instance.

**EXAMPLE 13.44**

Super-assignment in R

Interestingly, there is no way in Python for a nested routine to write a variable that belongs to a surrounding but nonglobal scope. In Figure 13.22, `inner` could not be modified to access `outer`'s `i`. R provides an alternative mechanism that does provide this functionality. Rather than declare `i` to be `global`, R uses a “super-assignment” operator. Where a normal assignment `i <- 4` assigns the value 4 into a local variable `i`, the super-assignment `i <<- 4` assigns 4 into whatever `i` would be found under the normal rules of static (lexical) scoping. ■

**EXAMPLE 13.45**

Scoping rules in Tcl

In a completely different vein, Tcl makes the unusual choice not only of employing dynamic scope, but of implementing that choice in an unusual way. Variables in calling scopes are never accessed automatically. The programmer must ask for them explicitly, as shown in Figure 13.23. The `upvar` and `uplevel` commands take an optional first argument that specifies a frame on the dynamic chain, either as an absolute value prefaced with a sharp sign (#) or, as in the call to `uplevel` shown in our example, as a distance below the current frame. If omitted, as in our call to `upvar`, the argument defaults to 1. The `upvar` command accesses a variable in the specified frame, and gives it a local name. The `uplevel` command provides a nested Tcl script, which is executed in the context of the specified frame in a manner reminiscent of call-by-name parameters. In our example we use `upvar` to obtain a local name for `foo`'s `i`, and `uplevel` to execute a command that uses the global `a` and `b`. The program prints a 5 and a 3. Note that the usual behavior of dynamic scoping, in which we automatically obtain the most recently created variable of a given name regardless of the scope that created it, is not available in Tcl. ■

```

proc bar {} {
 upvar i j ;# j is local name for caller's i
 puts "$j"
 uplevel 2 { puts [expr $a + $b] }
 # execute 'puts' two scopes up the dynamic chain
}

proc foo { i } {
 bar
}

set a 1; set b 2; foo 5

```

**Figure 13.23** A program to illustrate scope rules in Tcl. The `upvar` command allows `bar` to access variable `i` in its caller's scope, using the name `j`. The `uplevel` command allows `bar` to execute a nested Tcl script (the `puts` command) in its caller's caller's scope.

### Scoping in Perl

Perl has evolved over the years. At first there were only global variables. Locals were soon added for the sake of modularity, so a subroutine with a variable named `i` wouldn't have to worry about modifying a global `i` that was needed elsewhere in the code. Unfortunately, locals were originally defined in terms of dynamic scope, and the need for backward compatibility required that this behavior be retained when static scoping was added in Perl 5. Consequently, the language provides both mechanisms.

Any variable that is not declared is global in Perl by default. Variables declared with the `local` operator are dynamically scoped. Variables declared with the `my` operator are statically scoped. The difference can be seen in Figure 13.24, in which subroutine `outer` declares two local variables, `lex` and `dyn`. The former is statically scoped; the latter is dynamically scoped. Both are initialized to be a copy of `foo`'s first parameter. (Parameters are passed in the pseudo-variable `@_`. The first element of this array is `$_[0]`.)

Two lexically identical anonymous subroutines are nested inside `outer`, one before and one after the redeclarations of `$lex` and `$dyn`. References to these are stored in local variables `sub_A` and `sub_B`. Because static scopes in Perl extend from a declaration to the end of its block, `sub_A` sees the global `$lex`, while `sub_B` sees `outer`'s `$lex`. In contrast, because the declaration of local `$dyn` occurs before either `sub_A` or `sub_B` is called, both see this local version. Our program prints

```

main 1, 1
outer 2, 2
sub_A 1, 2
sub_B 2, 2
main 1, 1

```

---

#### EXAMPLE 13.46

##### Static and dynamic scope in Perl

```

sub outer($) { # must be called with scalar arg
 $sub_A = sub {
 print "sub_A $lex, $dyn\n";
 };
 my $lex = $_[0]; # static local initialized to first arg
 local $dyn = $_[0]; # dynamic local initialized to first arg
 $sub_B = sub {
 print "sub_B $lex, $dyn\n";
 };
 print "outer $lex, $dyn\n";
 $sub_A->();
 $sub_B->();
}
}

$lex = 1; $dyn = 1;
print "main $lex, $dyn\n";
outer(2);
print "main $lex, $dyn\n";

```

**Figure 13.24** A program to illustrate scope rules in Perl. The `my` operator creates a statically scoped local variable; the `local` operator creates a new dynamically scoped instance of a global variable. Static scope extends from the point of declaration to the lexical end of the block; dynamic scope extends from elaboration to the end of the block's execution.

#### EXAMPLE 13.47

##### Accessing globals in Perl

In cases where static scoping would normally access a variable at an in-between level of nesting, Perl allows the programmer to force the use of a global variable with the `our` operator, whose name is intended to contrast with `my`:

#### DESIGN & IMPLEMENTATION

##### Thinking about dynamic scope

In Section 3.3.6 we described dynamic scope rules as introducing a new meaning for a name that remains visible, wherever we are in the program, until control leaves the scope in which the new meaning was created. This conceptual model mirrors the association list implementation described in Section ⑩ 3.4.2 and, as described in the sidebar on page 133, probably accounts for the use of dynamic scoping in early dialects of Lisp.

Documentation for Perl suggests a semantically equivalent but conceptually different model. Rather than saying that a `local` declaration introduces a new variable whose name hides previous declarations, Perl says that there is a *single* variable, at the global level, whose previous *value* is saved when the new declaration is encountered, and then automatically restored when control leaves the new declaration's scope. This model mirrors the underlying implementation in Perl, which uses a central reference table (also described in Section ⑩ 3.4.2). In keeping with this model and implementation, Perl does not allow a `local` operator to create a dynamic instance of a variable that is not global.

```

($x, $y, $z) = (1, 1, 1); # global scope
{
 my ($x, $y) = (2, 2); # middle scope
 local $z = 3;
 {
 # inner scope
 our ($x, $z); # use globals
 print "$x, $y, $z\n";
 }
}

```

Here there is one lexical instance of `z` and two of `x` and `y`: one global, one in the middle scope. There is also a dynamic `z` in the middle scope. When it executes its `print` statement, the inner scope finds the `y` from the middle scope. It finds the global `x`, however, because of the `our` operator on line 6. Now what about `z`? The rules require us to start with static scope, ignoring `local` operators. According, then, to the `our` operator in the inner scope, we are using the global `z`. Once we know this, we look to see whether a dynamic (`local`) redeclaration of `z` is in effect. In this case indeed it is, and our program prints 1, 2, 3. As it turns out, the `our` declaration in the inner scope had no effect on this program. If only `x` had been declared `our`, we would still have used the global `z` and then found the dynamic instance from the middle scope. ■

### 13.4.2 String and Pattern Manipulation

When we first considered regular expressions, in Section 2.1.1, we noted that many scripting languages and related tools employ extended versions of the notation. Some extensions are simply a matter of convenience. Others increase the expressive power of the notation, allowing us to generate (match) nonregular sets of strings. Still other extensions serve to tie the notation to other language features.

We have already seen examples of extended regular expressions in `sed` (Figure 13.1), `awk` (Figures 13.2 and 13.3), Perl (Figures 13.4 and 13.5), Tcl (Figure 13.6), Python (Figure 13.7), and Ruby (Figure 13.8). We've also made note of `grep`, the stand-alone Unix pattern-matching tool (see sidebar on page 729).

While there are many different implementations of extended regular expressions (“REs” for short), with slightly different syntax, most fall into two main groups. The first group includes `awk`, `egrep` (the most widely used of several different versions of `grep`), the `regex` routines of the C standard library, and older versions of Tcl. These implement REs as defined in the POSIX standard [Int03b]. Languages in the second group follow the lead of Perl, which provides a large set of extensions, sometimes referred to as “advanced REs.” Perl-like advanced REs appear in PHP, Python, Ruby, JavaScript, Emacs Lisp, Java, C#, and recent versions of Tcl. They can also be found in third-party packages for C++ and other languages. A few tools, including `sed`, classic `grep`, and older Unix editors, provide so-called “basic” REs, less capable than those of `egrep`.

In certain languages and tools—notably `sed`, `awk`, Perl, PHP, Ruby, and JavaScript—regular expressions are tightly integrated into the rest of the language, with special syntax and built-in operators. In these languages an RE is typically delimited with slash characters, though other delimiters may be accepted in some cases (and Perl in fact provides slightly different semantics for a few alternative delimiters). In most other languages, REs are expressed as ordinary character strings and are manipulated by passing them to library routines. Over the next few pages we will consider POSIX and advanced REs in more detail. Following Perl, we will use slashes as delimiters. Our coverage will of necessity be incomplete. The chapter on REs in the Perl book [WCO00, Chapter 5] is nearly 80 pages long. The corresponding Unix `man` page runs to more than 20 pages.

## DESIGN & IMPLEMENTATION

### Automata for regular expressions

POSIX regular expressions are typically implemented using the constructions described in Section 2.2.1, which transform the RE into an NFA and then a DFA. Advanced REs of the sort provided by Perl are typically implemented via backtracking search in the obvious NFA. The NFA-to-DFA construction is usually not employed, because it fails to preserve some of the advanced RE extensions (notably the *capture* mechanism described in Examples 13.62–13.65) [WCO00, pages 197–202]. Some implementations use a DFA first to determine whether there *is* a match, and then an NFA or backtracking search to actually effect the match. This strategy pays the price of the slower automaton only when it's sure to be worthwhile.

## DESIGN & IMPLEMENTATION

### The `grep` command and the birth of Unix tools

Historically, regular expression tools have their roots in the pattern matching mechanism of the `ed` line editor, which dates from the earliest days of Unix. In 1973, Doug McIlroy, head of the department where Unix was born, was working on a project in computerized voice synthesis. As part of this project he was using the editor to search for potentially challenging words in an online dictionary. The process was both tedious and slow. At McIlroy's request, Ken Thompson extracted the pattern matcher from `ed` and made it a stand-alone tool. He named his creation `grep`, after the `g/re/p` command sequence in the editor: `g` for “global”; `/` to search for a regular expression (*re*); `p` to print [HH97a, Chapter 9].

Thompson's creation was one of the first in a large suite of stream-based Unix tools. As described in Section 13.2.1 (page 680), such tools are frequently combined with pipes to perform a variety of filtering, transforming, and formatting operations.

### POSIX Regular Expressions

#### EXAMPLE 13.48

Basic operations in POSIX REs

#### EXAMPLE 13.49

Extra quantifiers in POSIX REs

#### EXAMPLE 13.50

Zero-length assertions

#### EXAMPLE 13.51

Character classes

#### EXAMPLE 13.52

The dot (.) character

#### EXAMPLE 13.53

Negation and quoting in character classes

Like the “true” regular expressions of formal language theory, extended REs support concatenation, alternation, and Kleene closure. Parentheses are used for grouping.

`/ab(cd|ef)g*/` matches abcd, abcdg, abefg, abefgg, abcdggg, etc.

Several other *quantifiers* (generalizations of Kleene closure) are also available: ? indicates zero or one repetitions, + indicates one or more repetitions, {n} indicates exactly *n* repetitions, {n,} indicates at least *n* repetitions, and {n, m} indicates *n*–*m* repetitions.

<code>/a(bc)*/</code>	matches a, abc, abcbc, abcbcabc, etc.
<code>/a(bc)?/</code>	matches a or abc
<code>/a(bc)+/</code>	matches abc, abcbc, abcbcabc, etc.
<code>/a(bc){3}/</code>	matches abcbcabc only
<code>/a(bc){2,}/</code>	matches abcbc, abcbcabc, etc.
<code>/a(bc){1,3}/</code>	matches abc, abcbc, and abcbcabc (only)

Two *zero-length assertions*, ^ and \$, match only at the beginning and end, respectively, of a target string. Thus while /abe/ will match abe, abet, babe, and label, /^abe/ will match only the first two of these, /abe\$/ will match only the first and the third, and /^abe\$/ will match only the first.

As an abbreviation for /a|b|c|d/, extended REs permit *character classes* to be specified with square brackets:

`/b[aeiou]d/` matches bad, bed, bid, bod, and bud

Ranges are also permitted:

`/0x[0-9a-fA-F]+/` matches any hexadecimal integer

Outside a character class, a dot (.) matches any character other than a newline. The expression /b.d/, for example, matches not only bad, bbd, bcd, and so on, but also b:d, b7d, and many, many others, including sequences in which the middle character isn’t printable. In a Unicode-enabled version of Perl, there are tens of thousands of options.

A caret (^) at the beginning of a character class indicates negation: the class expression matches anything *other* than the characters inside. Thus /b[^aq]d/ matches anything matched by /b.d/ except for bad and bqd. A caret, right bracket, or hyphen can be specified inside a character class by preceding it with a backslash. A backslash will similarly protect any of the special characters | ( ) [ ] { } \$ . \* + ? outside a character class.<sup>10</sup> To match a literal backslash, use two of them in a row:

`/a\\b/` matches a\b

**10** Strictly speaking, ] and } don’t require a protective backslash unless there is a preceding unmatched (and unprotected) [ or {, respectively.

**EXAMPLE 13.54**

Predefined POSIX character classes

Several character classes' expressions are predefined in the POSIX standard. As we saw in Example 13.18, the expression `[:space:]` can be used to capture white space. For punctuation there is `[:punct:]`. The exact definition of these classes depends on the local character set and language. Note, too, that these expressions must be used *inside* a built-up character class; they aren't classes by themselves. A variable name in C, for example, might be matched by `/[[[:alpha:]_][[:alpha:][:digit:]]*/` or, a bit more simply, `/[[[:alpha:]_][[:alnum:]]*/`. Additional syntax, not described here, allows character classes to capture Unicode *collating elements* (multibyte sequences such as a character and associated accents) that collate (sort) as if they were single elements. Perl provides less cumbersome versions of most of these special classes.

**Perl Extensions****EXAMPLE 13.55**

RE matching in Perl

```
$foo = "albatross";
if ($foo =~ /ba.*s+/) ... # true
if ($foo =~ /^ba.*s+/) ... # false (no match at start of string)
```

The string to be matched against can also be left unspecified, in which case Perl uses the pseudo-variable `$_` by default:

```
$_ = "albatross";
if (/ba.*s+/) ... # true
if (/^ba.*s+/) ... # false
```

Recall that (as we noted in Section 13.2.2 [page 687]), `$_` is set automatically when iterating over the lines of a file. It is also the default index variable in `for` loops.

The `!~` operator returns true when a pattern *does not* match:

```
if ("albatross" !~ /^ba.*s+/) ... # true
```

For substitution, the binary “mixfix” operator `s///` replaces whatever lies between the first and second slashes with whatever lies between the second and the third:

```
$foo = "albatross";
$foo =~ s/lbat/c/; # "across"
```

Again, if a left-hand side is not specified, `s///` matches and modifies `$_`.

**Modifiers and Escape Sequences**

Both matches and substitutions can be *modified* by adding one or more characters after the closing delimiter. A trailing `i`, for example, makes the match case-insensitive:

```
$foo = "Albatross";
if ($foo =~ /^al/i) ... # true
```

**EXAMPLE 13.58**

Trailing modifiers on RE matches

Escape	Meaning
\0	NUL character
\a	alarm (BEL) character
\b	backspace (within character class)
\e	escape (ESC) character
\f	form-feed (FF) character
\n	newline
\r	return
\t	tab
\NNN	character given by NNN in octal
\x{abcd}	character given by abcd in hexadecimal
\b	word boundary (outside character classes)
\B	not a word boundary
\A	beginning of string
\z	end of string
\Z	prior to final newline, or end of string if none
\d	digit (decimal)
\D	not a digit
\s	white space (space, tab, newline, return, form feed)
\S	not white space
\w	word character (letter, digit, underscore)
\W	not a word character

**Figure 13.25** Regular expression escape sequences in Perl. Sequences in the top portion of the table represent individual characters. Sequences in the middle are zero-width assertions. Sequences at the bottom are built-in character classes.

A trailing g on a substitution replaces *all* occurrences of the regular expression:

```
$foo = "albatross";
$foo =~ s/[aeiou]/-/g; # "-lb-tr-ss"
```

For matching in multiline strings, a trailing s allows a dot (.) to match an embedded newline (which it normally cannot). A trailing m allows \$ and ^ to match immediately before and after such a newline, respectively. A trailing x causes Perl to ignore both comments and embedded white space in the pattern, so that particularly complicated expressions can be broken across multiple lines, documented, and indented.

In the tradition of C and its relatives (Example 7.73, page 366), Perl allows nonprinting characters to be specified in REs using backslash *escape sequences*. These are summarized in the top portion of Figure 13.25. Perl also provides several zero-width assertions, in addition to the standard ^ and \$. These are shown in the middle of the figure. The \A and \Z escapes differ from ^ and \$ in that they continue to match only at the beginning and end of the string, respectively, even in multiline searches that use the modifier m. Finally, Perl provides several built-in character classes, shown at the bottom of the figure. These can be used

both inside and outside user-defined (i.e., bracket-delimited) classes. Note that \b has *different* meanings inside and outside such classes.

### **Greedy and Minimal Matches**

The usual rule for matching in REs is sometimes called “leftmost longest”: when a pattern can match at more than one place within a string, the chosen match will be the one that starts at the earliest possible position within the string, and then extends as far as possible. In the string abc<sub>bc</sub>c<sub>bcd</sub>e, for example, the pattern /(bc)+/ can match in six different ways:

```
abcbccbcde
abcbccbcde
abcbccbcde
abcbccbcde
abcbccbcde
abcbccbcde
```

The third of these is “leftmost longest,” also known as *greedy*. In some cases, however, it may be desirable to obtain a “leftmost shortest” or *minimal* match. This corresponds to the first alternative above. ■

We saw a more realistic example in Example 13.22 (Figure 13.4), which contains the following substitution.

```
s/.+?(<[hH] [123]>.+?<\/[hH] [123]>)//s;
```

Assuming that the HTML input is well formed and that headers do not nest, this substitution deletes everything between the beginning of the string (implicitly \$\_) and the end of the first embedded header. It does so by using the \*? quantifier instead of the usual \*. Without the question marks, the pattern would match through (and the substitution would delete through) the end of the *last* header in the string. Recall that the trailing s modifier allows our headers to span lines.

In general, \*? matches the smallest number of instances of the preceding subexpression that will allow the overall match to succeed. Similarly, +? matches at least one instance, but no more than necessary to allow the overall match to succeed, and ?? matches either zero or one instances, with a preference for zero. ■

### **Variable Interpolation and Capture**

Like double-quoted strings, regular expressions in Perl support *variable interpolation*. Any dollar sign that does not immediately precede a vertical bar, closing parenthesis, or end of string is assumed to introduce the name of a Perl variable, whose value as a string is expanded prior to passing the pattern to the regular expression evaluator. This allows us to write code that generates patterns at run time:

```
$prefix = ...
$suffix = ...
if ($foo =~ /^$prefix.*$suffix$/) ...
```

---

### **EXAMPLE 13.61**

Variable interpolation in extended REs

---

### **EXAMPLE 13.59**

Greedy and minimal matching

---

### **EXAMPLE 13.60**

Minimal matching of HTML headers

**EXAMPLE 13.62**

Variable capture in extended REs

Note the two different roles played by \$ in this example.

The flow of information can go the other way as well: we can pull the values of variables out of regular expressions. We saw a simple example in the sed script of Figure 13.1:

```
s/^.*\(<[hH] [123]>\)/\1/ ;# delete text before opening tag
```

The equivalent in Perl would look something like this:

```
$line =~ s/^.*\(<[hH] [123]>\)/\1/;
```

Every parenthesized fragment of a Perl RE is said to *capture* the text that it matches. The captured strings may be referenced in the right-hand side of the substitution as \1, \2, and so on. Outside the expression they remain available (until the next substitution is executed) as \$1, \$2, and so on:

```
print "Opening tag: ", $1, "\n";
```

One can even use a captured string later in the RE itself. Such a string is called a *backreference*:

```
if (/.*?\(<[hH] ([123])>.*?<\/[hH]\2>/) {
 print "header: $1\n";
}
```

Here we have used \2 to insist that the closing tag of an HTML header match the opening tag.

**EXAMPLE 13.63**

Dissecting a floating-point literal

One can, of course capture multiple strings:

```
if (/^([+-]?)(\d+)\.(\d*)(\d+)(e([+-]?\d+))?\$/) {
 # floating point number
 print "sign: ", $1, "\n";
 print "integer: ", $3, $4, "\n";
 print "fraction: ", $5, "\n";
 print "mantissa: ", $2, "\n";
 print "exponent: ", $7, "\n";
}
```

As in the previous example, the numbering corresponds to the occurrence of left parentheses, read from left to right. With input -123.45e-6 we see

```
sign: -
integer: 123
fraction: 45
mantissa: 123.45
exponent: -6
```

Note that because of alternation, exactly one of \$3 and \$4 is guaranteed to be set. Note also that while we need the sixth set of parentheses for grouping (it has a ? quantifier), we don't really need it for capture.

**EXAMPLE 13.65**

Implicit capture of prefix, match, and suffix

For simple matches, Perl also provides pseudo-variables named \$', \$\$, and \$. These name the portions of the string before, in, and after the most recent match, respectively:

```
$line = <>;
chop $line; # delete trailing newline
$line =~ /is/;
print "prefix('$') match($&) suffix('$')\n";
```

With input “now is the time”, this code prints

```
prefix(now) match(is) suffix(the time)
```

### CHECK YOUR UNDERSTANDING

44. What popular scripting language uses dynamic scope?
45. Summarize the strategies used in Perl, PHP, Ruby, and Python to determine the scope of variables that are not declared.
46. Describe the conceptual model for dynamically scoped variables in Perl.
47. List the principal features found in POSIX regular expressions, but not in the regular expressions of formal language theory (Section 2.1.1).
48. List the principal features found in Perl REs, but not in those of POSIX.

### DESIGN & IMPLEMENTATION

#### Compiling regular expressions

Before it can be used as the basis of a search, a regular expression must be compiled into a deterministic or nondeterministic (backtracking) automaton. Patterns that are clearly constant can be compiled once, either when the program is loaded or when they are first encountered. Patterns that contain interpolated strings, however, must in the general case be recompiled whenever they are encountered, at potentially significant run-time cost. A programmer who knows that interpolated variables will never change can inhibit recompilation by attaching a trailing `o` modifier to the regular expression, in which case the expression will be compiled the first time it is encountered and never thereafter. For expressions that must sometimes but not always be recompiled, the programmer can use the `qr` operator to force recompilation of a pattern, yielding a result that can be used repeatedly and efficiently:

```
for (@patterns) { # iterate over patterns
 my $pat = qr($_); # compile to automaton
 for (@strings) { # iterate over strings
 if (/^$pat/) { # no recompilation required
 print; # print all strings that match
 print "\n";
 }
 }
 print "\n";
}
```

49. Explain the purpose of search *modifiers* (characters following the final delimiter) in Perl-type regular expressions.
  50. Describe the three different categories of *escape sequences* in Perl-type regular expressions.
  51. Explain the difference between *greedy* and *minimal* matches.
  52. Describe the notion of *capture* in regular expressions.
- 

### 13.4.3 Data Types

As we have seen, scripting languages don't generally require (or even permit) the declaration of types for variables. Most perform extensive run-time checks to make sure that values are never used in inappropriate ways. Some languages (e.g., Scheme, Python, and Ruby) are relatively strict about this checking; the programmer who wants to convert from one type to another must say so explicitly. If we type the following in Ruby

```
a = "4"
print a + 3, "\n"
```

we get the following message at run time: “In '+': failed to convert Fixnum into String (TypeError).” Perl is much more forgiving. As we saw in Example 13.2, the program

```
$a = "4";
print $a . 3 . "\n"; # '.' is concatenation
print $a + 3 . "\n"; # '+' is addition
```

prints 43 and 7. ■

In general, Perl (and likewise Rexx and Tcl) takes the position that programmers should check for the errors they care about, and in the absence of such checks the program should do something reasonable. Perl is willing, for example, to accept the following (though it prints a warning if run with the `-w` compile-time switch):

```
$a[3] = "1"; # (array @a was previously undefined)
print $a[3] + $a[4], "\n";
```

Here `$a[4]` is uninitialized and hence has value `undef`. In a numeric context (as an operand of `+`) the string “1” evaluates to 1, and `undef` evaluates to 0. Added together, these yield 1, which is converted to a string and printed. ■

A comparable code fragment in Ruby requires a bit more care. Before we can subscript `a` we must make sure that it refers to an array:

```
a = []
a[3] = "1" # empty array assignment
```

#### EXAMPLE 13.67

Coercion and context in Perl

#### EXAMPLE 13.68

Explicit conversion in Ruby

If the first line were not present (and `a` had not been initialized in any other way), the second line would have generated an “undefined local variable” error. After these assignments, `a[3]` is a string, but other elements of `a` are `nil`. We cannot concatenate a string and `nil`; neither can we add them (both operators are specified in Ruby using the operator `+`). If we want concatenation, and `a[4]` may be `nil`, we must say

```
print a[3] + String(a[4]), "\n"
```

If we want addition, we must say

```
print Integer(a[3]) + Integer(a[4]), "\n"
```

As these examples suggest, Perl (and likewise Tcl) uses a value model of variables. Scheme, Python, and Ruby use a reference model. PHP and JavaScript, like Java, use a value model for variables of primitive type and a reference model for variables of object type. The distinction is less important in PHP and JavaScript than it is in Java, because the same variable can hold a primitive value at one point in time and an object reference at another.

### Numeric Types

As we have seen in Section 13.4.2, scripting languages generally provide a very rich set of mechanisms for string and pattern manipulation. Syntax and interpolation conventions vary, but the underlying functionality is remarkably consistent, and heavily influenced by Perl. The underlying support for numeric types shows a bit more variation across languages, but the programming model is again remarkably consistent: users are, to first approximation, encouraged to think of numeric values as “simply numbers,” and not to worry about the distinction between fixed and floating point or about the limits of available precision.

Internally, numbers in JavaScript are always double precision floating point. In Tcl they are strings, converted to integers or floating-point numbers (and back again) when arithmetic is needed. PHP uses integers (guaranteed to be at least 32 bits wide), plus double-precision floating point. To these Perl and Ruby add arbitrary precision (multiword) integers, sometimes known as *bignums*. Python has bignums too, plus support for complex numbers. Scheme has all of the above, plus precise rationals, maintained as `(numerator, denominator)` pairs. In all cases the interpreter “up-converts” as necessary when doing arithmetic on values with different representations, or when overflow would otherwise occur.

Perl is scrupulous about hiding the distinctions among different numeric representations. Most other languages allow the user to determine which is being used, though this is seldom necessary. Ruby is perhaps the most explicit about the existence of different representations: classes `Fixnum`, `Bignum`, and `Float` (double-precision floating point) have overlapping but not identical sets of built-in methods. In particular, integers have iterator methods, which floating-point numbers do not, and floating-point numbers have rounding and error checking methods, which integers do not. `Fixnum` and `Bignum` are both descendants of `Integer`.

### Composite Types

The type constructors of compiled languages like C, Fortran, and Ada were chosen largely for the sake of efficient implementation. Arrays and records, in particular, have straightforward time- and space-efficient implementations, which we studied in Chapter 7. Efficiency, however, is less important in scripting languages. Designers have felt free to choose type constructors oriented more toward ease of understanding than pure run-time performance. In particular, most scripting languages place a heavy emphasis on *mappings*, sometimes called *dictionaries*, *hashes*, or *associative arrays*. As might be guessed from the third of these names, a mapping is typically implemented with a hash table. Access time for a hash remains  $O(1)$ , but with a significantly higher constant than is typical for a compiled array or record.

Perl, the oldest of the widely used scripting languages, inherits its principal composite types—the array and the hash—from awk. It also uses prefix characters on variable names as an indication of type: \$foo is a scalar (a number, Boolean, string, or pointer [which Perl calls a “reference”]); @foo is an array; %foo is a hash; &foo is a subroutine; and plain foo is a filehandle or an I/O format, depending on context.

#### EXAMPLE 13.69

Perl arrays

Ordinary arrays in Perl are indexed using square brackets and integers starting with 0:

```
@colors = ("red", "green", blue"); # initializer syntax
print $colors[2]; # green
```

Note that we use the @ prefix when referring to the array as a whole and the \$ prefix when referring to one of its (scalar) elements. Arrays are self-expanding: assignment to an out-of-bounds element simply makes the array larger (at the cost of dynamic memory allocation and copying). Uninitialized elements have the value `undef` by default.

#### EXAMPLE 13.70

Perl hashes

Hashes are indexed using curly braces and character string names:

```
%complements = {"red" => "cyan",
 "green" => "magenta", "blue" => "yellow");
print $complements{"blue"}; # yellow
```

These, too, are self-expanding.

Records and objects are typically built from hashes. Where the C programmer would write `fred.age = 19`, the Perl programmer writes `$fred{"age"} = 19`. In object-oriented code, `$fred` is more likely to be a reference, in which case we have `$fred->{"age"} = 19`.

#### EXAMPLE 13.71

Arrays and hashes in Python and Ruby

Python and Ruby, like Perl, provide both conventional arrays and hashes. They use square brackets for indexing in both cases, and distinguish between array and hash initializers (aggregates) using bracket and brace delimiters, respectively:

```
colors = ["red", "green", "blue"]
complements = {"red" => "cyan",
 "green" => "magenta", "blue" => "yellow"}
print colors[2], complements["blue"]
```

**EXAMPLE 13.72**

Array access methods in Ruby

(This is Ruby syntax; Python uses : in place of =>.)

As a purely object-oriented language, Ruby defines subscripting as syntactic sugar for invocations of the [] (get) and []= (put) methods:

```
c = colors[2] # same as c = colors.[](2)
colors[2] = c # same as colors.[](2,c)
```

**EXAMPLE 13.73**

Tuples in Python

In addition to arrays (which it calls *lists*) and hashes (which it calls *ictionaries*), Python provides two other composite types: tuples and sets. A tuple is essentially an immutable list (array). The initializer syntax uses parentheses rather than brackets:

```
crimson = (0xdc, 0x14, 0x3c) # R,G,B components
```

Tuples are more efficient to access than arrays: their immutability eliminates the need for most bounds and resizing checks. They also form the basis of multiway assignment:

```
a, b = b, a # swap
```

Parentheses can be omitted in this example: the comma groups more tightly than the assignment operator.

## DESIGN & IMPLEMENTATION

### Typeglobs in Perl

It turns out that a global name in Perl can have multiple independent meanings. It is possible, for example, to use \$foo, @foo, %foo, &foo and two different meanings of foo, all in the same program. To keep track of these multiple meanings, Perl interposes a level of indirection between the symbol table entry for foo and the various values foo may have. The intermediate structure is called a *typeglob*. It has one slot for each of foo's meanings. It also has a name of its own: \*foo. By manipulating typeglobs, the expert Perl programmer can actually modify the table used by the interpreter to look up names at run time. The simplest use is to create an alias:

```
*a = *b;
```

After executing this statement, a and b are indistinguishable; they both refer to the same typeglob, and changes made to (any meaning of) one of them will be visible through the other. Perl also supports *selective aliasing*, in which *one slot* of a typeglob is made to point to a value from a different typeglob:

```
*a = \&b;
```

The backslash operator (\) in Perl is used to create a pointer. After executing this statement, &a (the meaning of a as a function) will be the same as &b, but all other meanings of a will remain the same. Selective aliasing is used, among other things, to implement the mechanism that imports names from libraries in Perl.

**EXAMPLE 13.74**

Sets in Python

Python sets are like dictionaries that don't map to anything of interest but simply serve to indicate whether elements are present or absent. Unlike dictionaries, they also support union, intersection, and difference operations:

```
X = set(['a', 'b', 'c', 'd']) # set constructor
Y = set(['c', 'd', 'e', 'f']) # takes array as parameter
U = X | Y # ([‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’])
I = X & Y # ([‘c’, ‘d’])
D = X - Y # ([‘a’, ‘b’])
O = X ^ Y # ([‘a’, ‘b’, ‘e’, ‘f’])
‘c’ in I # True
```

**EXAMPLE 13.75**Conflated types in PHP,  
Tcl, and JavaScript

PHP and Tcl have simpler composite types: they eliminate the distinction between arrays and hashes. An array is simply a hash for which the programmer chooses to use numeric keys. JavaScript employs a similar simplification, unifying arrays, hashes, and objects. The usual `obj.attr` notation to access a member of an object (what JavaScript calls a *property*) is simply syntactic sugar for `obj["attr"]`. So objects are hashes, and arrays are objects with integer property names.

Higher dimensional types are straightforward to create in most scripting languages: one can define arrays of (references to) hashes, hashes of (references to) arrays, and so on. Alternatively, one can create a “flattened” implementation by using composite objects as keys in a hash. Tuples in Python work particularly well:

```
matrix = {} # empty dictionary (hash)
matrix[2, 3] = 4 # key is (2, 3)
```

This idiom provides the appearance and functionality of multidimensional arrays, though not their efficiency. There exist extension libraries for Python that provide more efficient homogeneous arrays, with only slightly more awkward syntax. Numeric and statistical scripting languages, such as Maple, Mathematica, Matlab, and R, have much more extensive support for multidimensional arrays.

**Context**

In Section 7.2.2 we defined the notion of *type compatibility*, which determines, in a statically typed language, which types can be used in which *contexts*. In this definition the term “context” refers to information about how a value will be used. In C, for example, one might say that in the declaration

```
double d = 3;
```

the 3 on the right-hand side occurs in a context that expects a floating-point number. The C compiler *coerces* the 3 to make it a `double` instead of an `int`.

In Section 7.2.3 we went on to define the notion of *type inference*, which allows a compiler to determine the type of an expression based on the types of its constituent parts and, in some cases, the context in which it appears. We saw an

extreme example in ML and its descendants, which use a sophisticated form of inference to determine types for most objects without the need for declarations.

In both of these cases—compatibility and inference—contextual information is used at compile time only. Perl extends the notion of context to drive decisions made at run time. More specifically, each operator in Perl determines, at compile time, and for each of its arguments, whether that argument should be interpreted as a *scalar* or a *list*. Conversely each argument (which may itself be a nested operator) is able to tell, at run time, which kind of context it occupies, and can consequently exhibit different behavior.

#### **EXAMPLE 13.77**

Scalar and list context in Perl

As a simple example, the assignment operator (=) provides a scalar or list context to its right-hand side based on the type of its left-hand side. This type is always known at compile time, and is usually obvious to the casual reader, because the left-hand side is a name and its prefix character is either a dollar sign (\$), implying a scalar context, or an at (@) or percent (%) sign, implying a list context. If we write

```
$time = gmtime();
```

Perl's standard gmtime() library function will return the time as a character string, along the lines of "Tue Mar 15 21:09:39 2005". On the other hand, if we write

```
@time_ary = gmtime();
```

the same function will return (39, 09, 21, 15, 2, 105, 2, 73), an 8-element array indicating seconds, minutes, hours, day of month, month of year (with January = 0), year (counting from 1900), day of week (with Sunday = 0), and day of year. ■

#### **EXAMPLE 13.78**

Using wantarray to determine calling context

So how does gmtime know what to do? By calling the built-in function wantarray. This returns true if the current function was called in a list context, and false if it was called in a scalar context. By convention, functions typically indicate an error by returning the empty array when called in a list context, and the undefined value (undef) when called in a scalar context:

```
if (something went wrong) {
 return wantarray ? () : undef;
}
```

### 13.4.4 Object Orientation

Though not an object-oriented language, Perl 5 has features that allow one to program in an object-oriented style.<sup>11</sup> PHP and JavaScript have cleaner, more conventional-looking object-oriented features, but both allow the programmer to use a more traditional imperative style as well. Python and Ruby are explicitly and uniformly object-oriented.

---

<sup>11</sup> More extensive features, currently under design for Perl 6, will not be covered here.

Perl uses a value model for variables; objects are always accessed via pointers. In PHP and JavaScript, a variable can hold either a value of a primitive type or a reference to an object of composite type. In contrast to Perl, however, these languages provide no way to speak of the reference itself, only the object to which it refers. Python and Ruby use a uniform reference model.

Classes are themselves objects in Python and Ruby, much as they are in Smalltalk. They are merely types in PHP, much as they are in C++, Java, or C#. Classes in Perl are simply an alternative way of looking at packages (namespaces). JavaScript, remarkably, has objects but no classes; its inheritance is based on a concept known as *prototypes*.

### Perl 5

Object support in Perl 5 boils down to two main things: (1) a *blessing* mechanism that associates a reference with a package and (2) special syntax for method calls that automatically passes an object reference or package name as the initial argument to a function. While any reference can in principle be blessed, the usual convention is to use a hash so that fields can be named as shown in Example 13.70.

#### EXAMPLE 13.79

##### A simple class in Perl

As a very simple example, consider the Perl code of Figure 13.26. Here we have defined a package, Integer, that plays the role of a class. It has three functions, one of which (*new*) is intended to be used as a constructor, and two of which (*set* and *get*) are intended to be used as accessors. Given this definition we can write

```
$c1 = Integer->new(2); # Integer::new("Integer", 2)
$c2 = new Integer(3); # alternative syntax
$c3 = new Integer; # no initial value specified
```

Both *Integer->new* and *new Integer* are syntactic sugar for calls to *Integer::new* with an additional first argument that contains the name of the package (class) as a character string. In the first line of function *new* we assign this string into the variable *\$class*. (The *shift* operator returns the first element of pseudo-variable *@\_* [the function's arguments], and shifts the remaining arguments, if any, so they will be seen if *shift* is used again.) We then create a reference to a new hash, store it in local variable *\$self*, and invoke the *bless* operator to associate it with the appropriate class. With a second call to *shift* we retrieve the initial value for our integer, if any. (The “or” expression *[||]* allows us to use 0 instead if no explicit argument was present.) We assign this initial value into the *val* field of *\$self* using the usual Perl syntax to dereference a pointer and subscript a hash. Finally we return a reference to the newly created object. ■

#### EXAMPLE 13.80

##### Invoking methods in Perl

Once a reference has been blessed, Perl allows it to be used with method invocation syntax: *c1->get()* and *get c1()* are syntactic sugar for *Integer::get(\$c1)*. Note that this call passes a reference as the additional first parameter, rather than the name of a package. Given the declarations of *\$c1*, *\$c2*, and *\$c3* above, the following code

```
{
 package Integer;

 sub new {
 my $class = shift; # probably "Integer"
 my $self = {};
 bless($self, $class);
 $self->{val} = (shift || 0);
 return $self;
 }
 sub set {
 my $self = shift;
 $self->{val} = shift;
 }
 sub get {
 my $self = shift;
 return $self->{val};
 }
}
```

**Figure 13.26** Object-oriented programming in Perl. Blessing a reference (object) into package `Integer` allows `Integer`'s functions to serve as the object's methods.

```

print $c1->get, " ", $c2->get, " ", $c3->get, " ", "\n";
$c1->set(4); $c2->set(5); $c3->set(6);
print $c1->get, " ", $c2->get, " ", $c3->get, " ", "\n";
will print
2 3 0
4 5 6

```

As usual in Perl, if an argument list is empty, the parentheses can be omitted. ■

Inheritance in Perl is obtained by means of the `@ISA` array, initialized at the global level of a package. Extending the previous example, we might define a `Tally` class that inherits from `Integer`:

```

{
 package Tally;
 @ISA = ("Integer");

 sub inc {
 my $self = shift;
 $self->{val}++;
 }
}
...
$t1 = new Tally(3);
$t1->inc;
$t1->inc;
print $t1->get, "\n"; # prints 5

```

#### EXAMPLE 13.81

##### Inheritance in Perl

The `inc` method of `t1` works as one might expect. However when Perl sees a call to `Tally::new` or `Tally::get` (neither of which is actually in the package), it uses the `@ISA` array to locate additional package(s) in which these methods may be found. We can list as many packages as we like in the `@ISA` array; Perl supports multiple inheritance. The possibility that `new` may be called through `Tally` rather than `Integer` explains the use of `shift` to obtain the class name in Figure 13.26. If we had used "`Integer`" explicitly we would not have obtained the desired behavior when creating a `Tally` object. ■

**EXAMPLE 13.82**Inheritance via `use base`

Most often packages (and thus classes) in Perl are declared in separate modules (files). In this case, one must import the module corresponding to a superclass in addition to modifying `@ISA`. The standard `base` module provides convenient syntax for this combined operation, and is the preferred way to specify inheritance relationships:

```
{ package Tally;
 use base ("Integer");
 ...
}
```

***PHP and JavaScript***

While Perl's mechanisms suffice to create object-oriented programs, dynamic lookup makes them slower than equivalent imperative programs, and it seems fair to characterize the syntax as less than elegant. Both PHP and JavaScript are more explicitly object-oriented.

PHP 4 provided a variety of object-oriented features, which were heavily revised in PHP 5. The newer version of the language provides a reference model of (class typed) variables, interfaces and mix-in inheritance, abstract methods and classes, final methods and classes, static and constant members, and access control specifiers (`public`, `protected`, and `private`) reminiscent of those of Java, C#, and C++. In contrast to all other languages discussed in this subsection, class declarations in PHP must include declarations of all members (fields and methods), and the set of members in a given class cannot subsequently change (though one can of course declare derived classes with additional members).

JavaScript takes the unusual approach of providing objects—with inheritance and dynamic method dispatch—without providing classes. Functions are first-class entities in JavaScript—objects, in fact. A method is simply a function that is referred to by a *property* (member) of an object. When we call `o.m`, the keyword `this` will refer to `o` during the execution of the function referred to by `m`. Likewise when we call `new f`, `this` will refer to a newly created (initially empty) object during the execution of `f`. A constructor in JavaScript is thus a function whose purpose is to assign values into properties (fields and methods) of a newly created object.

Associated with every constructor `f` is an object `f.prototype`. If object `o` was constructed by `f`, then JavaScript will look in `f.prototype` whenever we attempt to use a property of `o` that `o` itself does not provide. In effect, `o` inherits

```

function Integer(n) {
 this.val = n || 0; // use 0 if n is missing (undefined)
}
function Integer_set(n) {
 this.val = n;
}
function Integer_get() {
 return this.val;
}
Integer.prototype.set = Integer_set;
Integer.prototype.get = Integer_get;

```

**Figure 13.27** Object-oriented programming in JavaScript. The `Integer` function is used as a constructor. Assignments to members of its prototype object serve to establish methods. These will be available to any object created by `Integer` that doesn't have corresponding members of its own.

from `f.prototype` anything that it does not override. Prototype properties are commonly used to hold methods. They can also be used for constants or for what other languages would call “class variables.”

Figure 13.27 illustrates the use of prototypes. It is roughly equivalent to the Perl code of Figure 13.26. Function `Integer` serves as a constructor. Assignments to properties of `Integer.prototype` serve to establish methods for objects constructed by `Integer`. Using the code in the figure, we can write

```

c2 = new Integer(3);
c3 = new Integer();

document.write(c2.get() + " " + c3.get() + "
");
c2.set(4); c3.set(5);
document.write(c2.get() + " " + c3.get() + "
");

```

This code will print

```

3 0
4 5

```

#### EXAMPLE 13.84

##### Overriding instance methods in JavaScript

Interestingly, the lack of a formal notion of class means that we can override methods and fields on an object-by-object basis:

```

c2.set = new Function("n", "this.val = n * n;");
// anonymous function constructor
c2.set(3); c3.set(4); // these call different methods!
document.write(c2.get() + " " + c3.get() + "
");

```

If nothing else has changed since the previous example, this code will print

```

9 4

```

**EXAMPLE 13.85**

## Inheritance in JavaScript

To obtain the effect of inheritance, we can write

```
function Tally(n) {
 this.base(n); // call to base constructor
}
function Tally_inc() {
 this.val++;
}
Tally.prototype = new Integer; // inherit methods
Tally.prototype.base = Integer; // make base constructor available
Tally.prototype.inc = Tally_inc; // new method
...
t1 = new Tally(3);
t1.inc(); t1.inc();
document.write(t1.get() + "
");
```

This code will print a 5.

**Python and Ruby**

As we have noted, both Python and Ruby are explicitly object-oriented. Both employ a uniform reference model for variables. Like Smalltalk, both incorporate an object hierarchy in which classes themselves are represented by objects. The root class in Python is called `object`; in Ruby it is `Object`.

**EXAMPLE 13.86**

## Constructors in Python and Ruby

In both Python and Ruby, each class has a single distinguished constructor, which cannot be overloaded. In Python it is `__init__`; in Ruby it is `initialize`. To create a new object in Python one says `my_object = My_class(args)`; in Ruby one says `my_object = My_class.new(args)`. In each case the `args` are passed to the constructor. To achieve the effect of overloading, with different numbers or types of arguments, one must arrange for the single constructor to inspect its arguments explicitly. We employed a similar idiom in Perl (in the `new` routine of Figure 13.26) and JavaScript (in the `Integer` function of Figure 13.27).

Both Python and Ruby are more flexible than PHP or more traditional object-oriented languages regarding the contents (members) of a class. New fields can be added to a Python object simply by assigning to them: `my_object.new_field = value`. The set of methods, however, is fixed when the class is first defined. In Ruby only methods are visible outside a class (“put” and “get” methods must be used to access fields), and all methods must be explicitly declared. It is possible, however, to modify an existing class declaration, adding or overriding methods. One can even do this on an object-by-object basis. As a result, two objects of the same class may not display the same behavior.

**EXAMPLE 13.87**

## Naming class members in Python and Ruby

Python and Ruby differ in many other ways. The initial parameter to methods is explicit in Python; by convention it is usually named `self`. In Ruby `self` is a keyword, and the parameter it represents is invisible. Any variable beginning with a single @ sign in Ruby is a field of the current object. Within a Python method, uses of object members must name the object explicitly. One must, for example, write `self.print()`; just `print()` will not suffice.

Ruby methods may be `public`, `protected`, or `private`.<sup>12</sup> Access control in Python is purely a matter of convention; both methods and fields are universally accessible. Finally, Python has multiple inheritance. Ruby has mix-in inheritance: a class cannot obtain data from more than one ancestor. Unlike most other languages, however, Ruby allows an interface (mix-in) to define not only the signatures of methods, but also their implementation (code).

### CHECK YOUR UNDERSTANDING

- 53. Contrast the philosophies of Perl and Ruby with regard to error checking and reporting.
- 54. Compare the numeric types of popular scripting languages to those of compiled languages like C or Fortran.
- 55. What are *bignums*? Which languages support them?

### DESIGN & IMPLEMENTATION

#### Executable class declarations

Both Python and Ruby take the interesting position that class declarations are executable code. Elaboration of a declaration executes the code inside. Among other things, we can use this mechanism to achieve the effect of conditional compilation:

```
class My_class # Ruby code
 def initialize(a, b)
 @a = a; @b = b;
 end
 if expensive_function()
 def get()
 return @a
 end
 else
 def get()
 return @b
 end
 end
end
```

Instead of computing the expensive function inside `get`, on every invocation, we compute it once, ahead of time, and define an appropriate specialized version of `get`.

---

**12** The meanings of `private` and `protected` in Ruby are different from those in C++, Java, or C#: `private` methods in Ruby are available only to the current instance of an object; `protected` methods are available to any instance of the current class or its descendants.

56. What are *associative arrays*? By what other names are they sometimes known?
  57. Why don't most scripting languages provide direct support for records?
  58. What is a *typeglob* in Perl? What purpose does it serve?
  59. Describe the *tuple* and *set* types of Python.
  60. Explain the unification of arrays and hashes in PHP and Tcl.
  61. Explain the unification of arrays and objects in JavaScript.
  62. Explain how tuples and hashes can be used to emulate multidimensional arrays in Python.
  63. Explain the concept of *context* in Perl. How is it related to type compatibility and type inference? What are the two principal contexts defined by the language's operators?
  64. Compare the approaches to object orientation taken by Perl 5, PHP 5, JavaScript, Python, and Ruby.
  65. What is meant by the *blessing* of a reference in Perl?
  66. What are *prototypes* in JavaScript? What purpose do they serve?
- 

## 13.5 Summary and Concluding Remarks

Scripting languages serve primarily to control and coordinate other software components. Though their roots go back to interpreted languages of the 1960s, they have received relatively little attention from academic computer science. With an increasing emphasis on programmer productivity, however, and with the birth of the World Wide Web, scripting languages have seen enormous growth in interest and popularity, both in industry and in academia. Many significant advances have been made by commercial developers and by the Open Source community. Scripting languages may well come to dominate programming in the 21st century, with traditional compiled languages more and more seen as special purpose tools.

In comparison to their traditional cousins, scripting languages emphasize flexibility and richness of expression over sheer run-time performance. Common characteristics include both batch and interactive use, economy of expression, lack of declarations, simple scoping rules, flexible dynamic typing, easy access to other programs, sophisticated pattern matching and string manipulation, and high-level data types.

We began our chapter by tracing the historical development of scripting, starting with the command interpreter, or *shell* programs of the mid-1970s, and the text processing and report generation tools that followed soon thereafter. We

looked in particular at the “Bourne again” shell, `bash`, and the Unix tools `sed` and `awk`. We also mentioned such special purpose domains as mathematics and statistics, where scripting languages are widely used for data analysis, visualization, modeling, and simulation. We then turned to the three domains that dominate scripting today: “glue” (coordination) applications, configuration and extension, and scripting of the World Wide Web.

In terms of “market share,” Perl is almost certainly the most popular of the general purpose scripting languages, widely used for report generation, glue, and server-side (CGI) web scripting. Python and Ruby both appear to be growing in popularity, and Tcl retains a strong core of support. Several scripting languages, including Scheme, Python, and Tcl, are widely used to extend the functionality of complex applications. In addition, many commercial packages have their own proprietary extension languages. Visual Basic has historically been the language of choice for scripting on Microsoft platforms but will probably give way over time to C# and the various cross-platform options.

Web scripting comes in many forms. On the server side of an HTTP connection, the Common Gateway Interface (CGI) standard allows a URI to name a program that will be used to generate dynamic content. Alternatively, web-page-embedded scripts, often written in PHP, can be used to create dynamic content in a way that is invisible to users. To reduce the load on servers, and to improve interactive responsiveness, scripts can also be executed within the client browser. JavaScript is the dominant notation in this domain; it uses the HTML Document Object Model (DOM) to manipulate web page elements. For more demanding

## DESIGN & IMPLEMENTATION

### Worse Is Better

Any discussion of the relative merits of scripting and “systems” languages invariably ends up addressing the tradeoffs between expressiveness and flexibility on the one hand and compile-time safety and performance on the other. It may also digress into questions of “quick and dirty” versus “polished” applications. An interesting take on this debate can be found in the widely circulated essays of Richard Gabriel ([www.dreamsongs.com/WorseIsBetter.html](http://www.dreamsongs.com/WorseIsBetter.html)). While working for Lucid Corp. in 1989, Gabriel found himself asking why Unix and C had been so successful at attracting users, while Common Lisp (Lucid’s principal focus) had not. His explanation contrasts “The Right Thing,” as exemplified by Common Lisp, with a “Worse Is Better” philosophy, as exemplified by C and Unix. “The Right Thing” emphasizes complete, correct, consistent, and elegant design. “Worse Is Better” emphasizes the rapid development of software that does most of what users need most of the time, and can be tuned and improved incrementally, based on field experience. Much of scripting, and Perl in particular, fits the “Worse Is Better” philosophy (Ruby and Scheme enthusiasts might beg to disagree). Gabriel, for his part, says he still hasn’t made up his mind; his essays argue both points of view.

tasks, most browsers can be directed to run a Java *applet*, which takes full responsibility for some portion of the “screen real estate.” With the continued evolution of the web, XML is likely to become the standard vehicle for storing and transmitting structured data. XSL, the Extensible Stylesheet Language, will then play a major role in transforming and formatting dynamic content.

Because of their rapid evolution, scripting languages have been able to take advantage of many of the most powerful and elegant mechanisms described in previous chapters, including first class and higher-order functions, garbage collection, unlimited extent, iterators, list comprehensions, and object orientation—not to mention extended regular expressions and such high-level data types as dictionaries, sets, and tuples. Given current technological trends, scripting languages are likely to become increasingly ubiquitous and to remain a principal focus of language innovation.

## 13.6 Exercises

- 13.1 Does filename “globbing” provide the expressive power of standard regular expressions? Explain.
- 13.2 Write shell scripts to
  - (a) Replace blanks with underscores in the names of all files in the current directory.
  - (b) Rename every file in the current directory by prepending to its name an ASCII representation of its modification date.
  - (c) Find all `eps` files in the file hierarchy below the current directory, and create any corresponding `pdf` files that are missing or out of date.
  - (d) Print the names of all files in the file hierarchy below the current directory for which a given predicate evaluates to true. Your (quoted) predicate should be specified on the command line using the syntax of the Unix `test` command, with one or more at signs (@) standing in for the name of the candidate file.
- 13.3 In Example 13.15 we used “\$@” to refer to the parameters passed to `11`. What would happen if we removed the quote marks? (*Hint:* Try this for files whose names contain spaces!) Read the `man` page for `bash` and learn the difference between `$@` and `$*`. Create versions of `11` that use `$*` or “`$*`” instead of “`$@`”. Explain what’s going on.
- 13.4 (a) Extend the code in Figures 13.5, 13.6, 13.7, or 13.8 to try to kill processes more gently. You’ll want to read the `man` page for the standard `kill` command. Use a `TERM` signal first. If that doesn’t work, ask the user if you should resort to `KILL`.

- (b) Extend your solution to part (a) so that the script accepts an optional argument specifying the signal to be used. Alternatives to TERM and KILL include HUP, INT, QUIT, and ABRT.
- 13.5 Write a Perl, Python, or Ruby script that creates a simple *concordance*: a sorted list of significant words appearing in an input document, with a sublist for each that indicates the lines on which the word occurs, with up to six words of surrounding context. Exclude from your list all common articles, conjunctions, prepositions, and pronouns.
- 13.6 Write Emacs Lisp scripts to perform the following tasks.
- Insert today's date into the current buffer at the insertion point (current cursor location).
  - Place quote marks (" ") around the word surrounding the insertion point.
  - Fix end-of-sentence spaces in the current buffer. Use the following heuristic: if a period, question mark, or exclamation point is followed by a single space (possibly with closing quote marks, parentheses, brackets, or braces in between), then add an extra space, unless the character preceding the period, question mark, or exclamation point is a capital letter (in which case we assume it is an abbreviation).
  - Run the contents of the current buffer through your favorite spell checker, and create a new buffer containing a list of misspelled words.
  - Delete one misspelled word from the buffer created in (d), and place the cursor (insertion point) on top of the first occurrence of that misspelled word in the current buffer.
- 13.7 Explain the circumstances under which it makes sense to realize an interactive task on the Web as a CGI script, an embedded server-side script, or a client-side script. For each of these implementation choices, give three examples of tasks for which it is clearly the preferred approach.
- 13.8 (a) Write a web page with embedded PHP to print the first 10 rows of Pascal's triangle (see Example ⑩ 15.10 if you don't know what this is). When rendered, your output should look like Figure 13.28.
- (b) Modify your page to create a self-posting form that accepts the number of desired rows in an input field.
- (c) Rewrite your page in JavaScript.
- 13.9 Create a fill-in web form that uses a JavaScript implementation of the Luhn formula (Exercise 4.9) to check for typos in credit card numbers. (But don't use real credit card numbers; homework exercises don't tend to be very secure!)

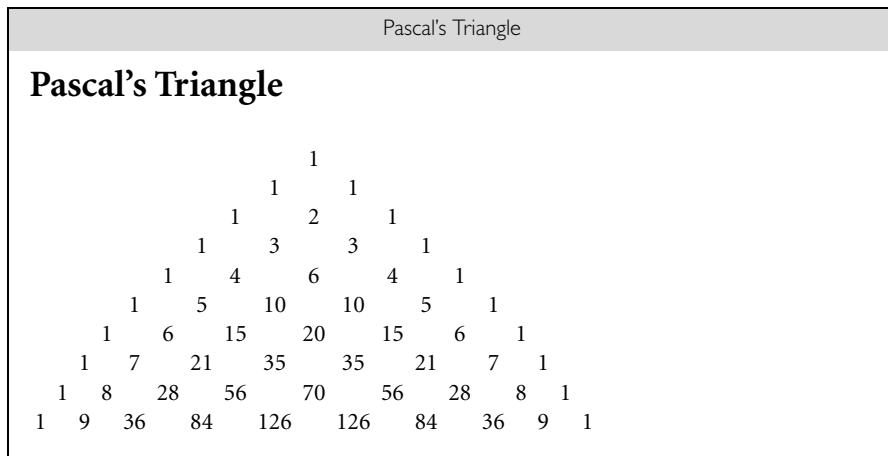


Figure 13.28 Pascal's triangle rendered in a web page (Exercise 13.8).

- 13.10 (a) Modify the code of Figure 13.16 (Example 13.35) so that it replaces the form with its output, as the CGI and PHP versions of Figures 13.12 and 13.15 do.
- (b) Modify the CGI and PHP scripts of Figures 13.12 and 13.15 (Examples 13.30 and 13.34) so they appear to append their output to the bottom of the form, as the JavaScript version of Figure 13.16 does.
- 13.11 Run the following program in Perl.

```
sub foo {
 my $lex = $_[0];
 sub bar {
 print "$lex\n";
 }
 bar();
}

foo(2); foo(3);
```

You may be surprised by the output. Perl 5 allows named subroutines to nest but does not create closures for them properly. Rewrite the code above to create a reference to an anonymous local subroutine and verify that it does create closures correctly. Add the line `use diagnostics;` to the beginning of the original version and run it again. Based on the explanation this will give you, speculate as to how nested named subroutines are implemented in Perl 5.

- 13.12 Modify the XSLT of Figure 13.19 to do one or more of the following.
- (a) Alter the titles of conference papers so that only first words, words that follow a dash or colon (and thus begin a subtitle), and proper

nouns are capitalized. You will need to adopt a convention by which the creator of the document can identify proper nouns.

- (b) Sort entries by the last name of the first author or editor. You will need to adopt a convention by which the creator of the document can identify compound last names (“von Neumann,” for example, should be alphabetized under ‘v’).
  - (c) Allow bibliographic entries to contain an `abstract` element, which when formatted appears as an indented block of text in a smaller font.
  - (d) In addition to the `book`, `article`, and `inproceedings` elements, add support for other kinds of entries, such as manuals, technical reports, theses, newspaper articles, web sites, and so on. You may want to draw inspiration from the categories supported by BibTeX [Lam94, Appendix B].
  - (e) Format entries according to some standard style convention (e.g., that of the *Chicago Manual of Style* [Uni03] or the ACM Transactions [[www.acm.org/pubs/submissions/latex\\_style/index.htm](http://www.acm.org/pubs/submissions/latex_style/index.htm)]).
- 13.13** Suppose bibliographic entries in Figure 13.18 contain a mandatory `key` element, and that other documents can contain matching `cite` elements. Create an XSLT script that imitates the work of BibTeX. Your script should
- (a) read an XML document, find all the `cite` elements, collect the keys they contain, and replace them with `bibref` elements that contain small integers instead.
  - (b) read a separate XML bibliography document, extract the entries with matching keys, and write them, in sorted order, to a new (and probably smaller) bibliography.
- The small numbers in the `bibref` elements of the new document from (a) should match the corresponding numbered entries in the new bibliography from (b).
- 13.14** Write a program that will read an XHTML file and print an outline of its contents, by extracting all `<title>`, `<h1>`, `<h2>`, and `<h3>` elements, and printing them at varying levels of indentation. Write
- (a) in C or Java.
  - (b) in `sed` or `awk`.
  - (c) in Perl, Python, Tcl, or Ruby.
  - (d) in XSLT.
- Compare and contrast your solutions.
- 13.15** Write a program that will map the web pages stored in the file hierarchy below the current directory. Your output should itself be a web page containing the names of all directories and `.html` files, printed at levels of

indentation corresponding to their level in the file hierarchy. Each .html file name should be a live link to a file of the form described in the previous exercise. Use whatever language(s) seem most appropriate to the task.

- 13.16** In Section 13.4.1 we claimed that nested blocks in Ruby were part of the named scope in which they appear. Verify this claim by running the following Ruby script and explaining its output.

```
def foo(x)
 y = 2
 bar = proc {
 print x, "\n"
 y = 3
 }
 bar.call()
 print y, "\n"
end

foo(3)
```

Now comment out the second line (`y = 2`) and run the script again. Explain what happens. Restate our claim about scoping more carefully and precisely.

- 13.17** Write a Perl script to translate English measurements (in, ft, yd, mi) into metric equivalents (cm, m, km). You may want to learn about the `e` modifier on regular expressions, which allows the right-hand side of an `s///e` expression to contain executable code.
- 13.18** Write a Perl script to find, for each input line, the longest substring that appears at least twice within the line, without overlapping. (*Hint:* This is harder than it sounds. Remember that by default Perl searches for a *left-most longest* match.)
- 13.19** Perl provides an alternative `(?:...)` form of parentheses that supports grouping in regular expressions without performing capture. Using this syntax, Example 13.64 could have been written as follows.

```
if (/^([+-]?)((\d+)\.|(\d*)\.(.\d+))(:e([+-]?\d+))?$/ {
 # floating point number
 print "sign: ", $1, "\n";
 print "integer: ", $3, $4, "\n";
 print "fraction: ", $5, "\n";
 print "mantissa: ", $2, "\n";
 print "exponent: ", $6, "\n"; # not $7
}
```

What purpose does this extra notation serve? Why might the code here be preferable to that of Example 13.64?

- 13.20** Consider again the `sed` code of Figure 13.1. It is tempting to write the first of the compound statements as follows (note the differences in the three substitution commands).

```
/<[hH] [123]>.*<\/[hH] [123]>/ {
 h ;# match whole heading
 s/^.*\(<[hH] [123]>\)/\1/ ;# save copy of pattern space
 s/\(<\/[hH] [123]>\).*/\1/ ;# delete text before opening tag
 p ;# delete text after closing tag
 g ;# print what remains
 s/^.*<\/[hH] [123]>// ;# retrieve saved pattern space
 b top ;# delete through closing tag
```

Explain why this doesn't work. (*Hint:* Remember the difference between *greedy* and *minimal* matches [Example 13.60]. `Sed` lacks the latter.)

- 13.21** Consider the following regular expression in Perl: `/^(?:((?:ab)+|a(?:ba)*))$/`. Describe, in English, the set of strings it will match. Show a natural NFA for this set, together with the minimal DFA. Describe the substrings that should be captured in each matching string. Based on this example, discuss the practicality of using DFAs to match strings in Perl.

## 13.7 Explorations

- 13.22** Learn about the Scheme shell, `sccsh`. Compare it to `sh/bash`. Which would you rather use from the keyboard? Which would you rather use for scripting?
- 13.23** Research the security mechanisms of JavaScript and/or Java applets. What exactly are programs allowed to do and why? What potentially useful features have not been provided because they can't be made secure? What potential security holes remain in the features that *are* provided?
- 13.24** Learn about *web crawlers*: programs that explore the World Wide Web. Build a crawler that searches for something of interest. What language features or tools seem most useful for the task? **Warning:** Automated web crawling is a public activity, subject to strict rules of etiquette. Before creating a crawler, do a web search and learn the rules, and test your code *very* carefully before letting it outside your local subnet (or even your own machine). In particular, be aware that rapid-fire requests to the same server constitute a *denial of service attack*, a potentially criminal offense.
- 13.25** Learn about *taint mode* in Perl and Ruby. How does it compare to the notion of *sandboxing* (as described in the sidebar on page 711)? What sorts of security problems does it catch? What sorts of problems does it *not* catch?

- 13.26** In the sidebar on page 729 we noted that the “extended” REs of `awk` and `egrep` are typically implemented by translating first to an NFA and then to a DFA, while those of Perl et al. are typically implemented via backtracking search. Some tools, including GNU `ggrep`, use a variant of the Boyer-Moore-Gosper algorithm [BM77, KMP77] for faster deterministic search. Find out how this algorithm works. What are its advantages? Could it be used in languages like Perl?
- 13.27** In the sidebar on page 735 we noted that nonconstant patterns must generally be recompiled whenever they are used. Perl programmers who wish to reduce the resulting overhead can inhibit recompilation using the `o` trailing modifier or the `qr` quoting operator. Investigate the impact of these mechanisms on performance. Also speculate as to the extent to which it might be possible for the language implementation to determine, automatically and efficiently, when recompilation should occur.
- 13.28** Our coverage of Perl REs in Section 13.4.2 was incomplete. Features not covered include look-ahead and look-behind (context) assertions, comments, incremental enabling and disabling of modifiers, embedded code, conditionals, Unicode support, non-slash delimiters, and the transliteration (`tr///`) operator. Learn how these work. Explain if (and how) they extend the expressive power of the notation. How could each be emulated (possibly with surrounding Perl code) if it were not available?
- 13.29** Investigate the details of RE support in PHP, Tcl, Python, Ruby, JavaScript, Emacs Lisp, Java, and C#. Write a paper that documents, as concisely as possible, the differences among these, using Perl as a reference for comparison.
- 13.30** Do a web search for Perl 6 (currently under development as of early 2005). Write a report that summarizes the changes with respect to Perl 5. What do you think of these changes? If you were in charge of the revision, what would you do differently?

## 13.8 Bibliographic Notes

Most of the major scripting languages are described in books by the language designers or their close associates: `awk` [AKW88], Perl [WCO00], PHP [LT02], Tcl [Ous94, WJH03], Python [vRD03], and Ruby [TH04]. Several of these have versions available online. Most of the languages are also described in a variety of other texts, and most have dedicated web sites: `perl.com`, `php.net`, `tcl.tk`, `python.org`, `ruby-lang.org`. Extensive documentation for Perl is available online at many sites; type `man perl` for an index.

Rexx [Ame96a] has been standardized by ANSI, the American National Standards Institute. JavaScript [ECM99] has been standardized by ECMA, the European standards body. Scheme implementations intended for scripting include

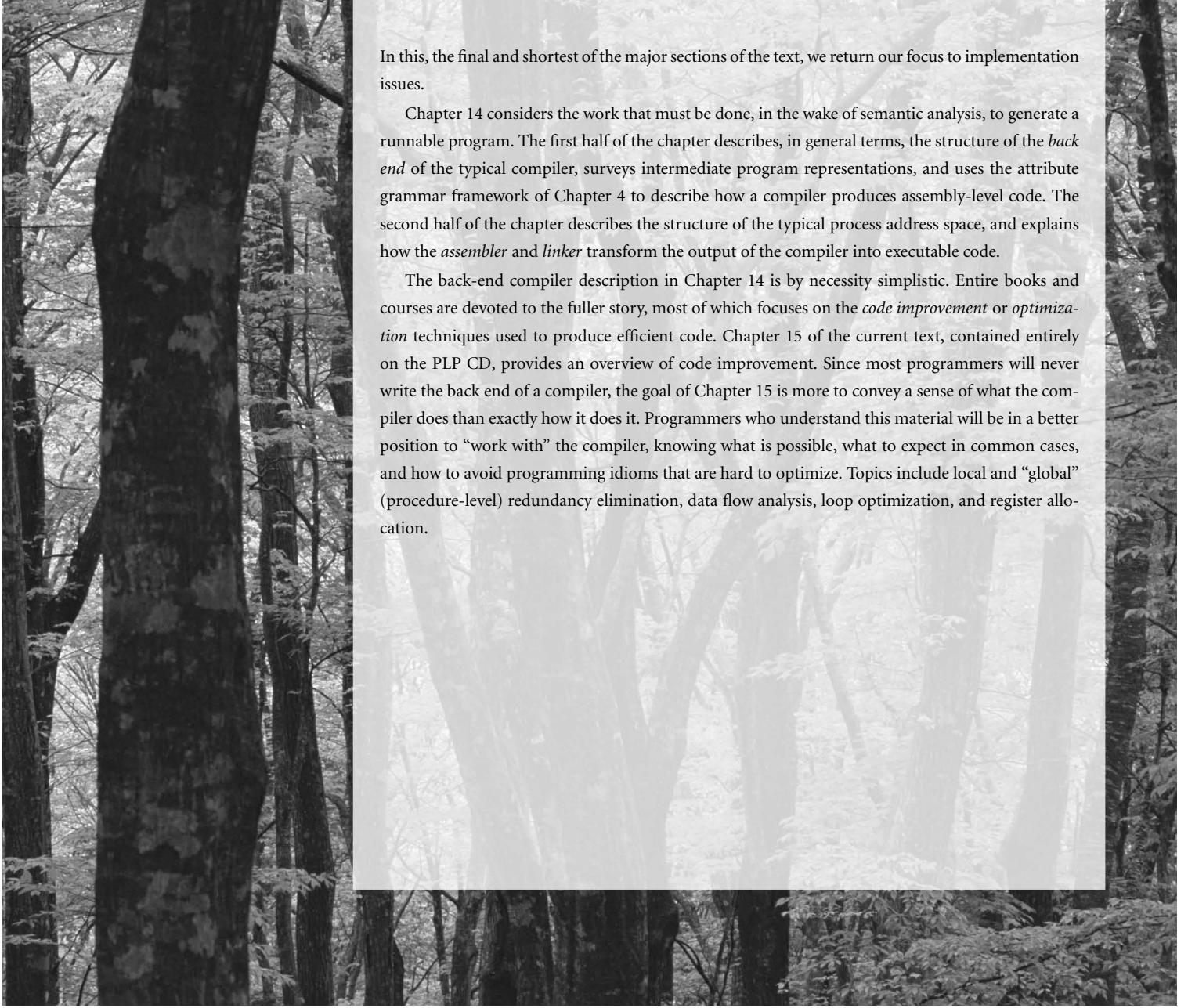
Elk ([www-rn.informatik.uni-bremen.de/software/elk/](http://www-rn.informatik.uni-bremen.de/software/elk/), [sam.zoy.org/projects/elk/](http://sam.zoy.org/projects/elk/)), Guile ([gnu.org/software/guile/](http://gnu.org/software/guile/)), and SIOD (Scheme in One Defun) ([people.delphiforums.com/gjc/siod.html](http://people.delphiforums.com/gjc/siod.html)). Standards for the World Wide Web, including HTML, XML, XSL, XPath, and XHTML, are promulgated by the World Wide Web Consortium: [www.w3.org](http://www.w3.org). For those experimenting with the conversion to XHTML, the validation service at [validator.w3.org](http://validator.w3.org) is particularly useful. High-quality tutorials on many web-related topics can be found at [w3schools.com](http://w3schools.com).

Hauben and Hauben [HH97a] describe the historical roots of the Internet, including early work on Unix. Original articles on the various Unix shell languages include those of Mashey [Mas76], Bourne [Bou78], and Korn [Kor94]. Information on the Scheme shell, *scsh*, is available at [scsh.net](http://scsh.net). The original reference on APL is by Iverson [Ive62]. Ousterhout [Ous98] makes the case for scripting languages in general and Tcl in particular. Chonacky and Winch [CW05] compare and contrast Maple, Mathematica, and Matlab. Richard Gabriel's collection of "Worse Is Better" papers can be found at [www.dreamsongs.com/WorseIsBetter.html](http://www.dreamsongs.com/WorseIsBetter.html). A similar comparison of Tcl and Scheme can be found in the introductory chapter of Abelson, Greenspun, and Sandon's on-line *Tcl for Web Nerds* guide ([philip.greenspun.com/tcl/index.adp](http://philip.greenspun.com/tcl/index.adp)).



# IV

## A Closer Look at Implementation



In this, the final and shortest of the major sections of the text, we return our focus to implementation issues.

Chapter 14 considers the work that must be done, in the wake of semantic analysis, to generate a runnable program. The first half of the chapter describes, in general terms, the structure of the *back end* of the typical compiler, surveys intermediate program representations, and uses the attribute grammar framework of Chapter 4 to describe how a compiler produces assembly-level code. The second half of the chapter describes the structure of the typical process address space, and explains how the *assembler* and *linker* transform the output of the compiler into executable code.

The back-end compiler description in Chapter 14 is by necessity simplistic. Entire books and courses are devoted to the fuller story, most of which focuses on the *code improvement* or *optimization* techniques used to produce efficient code. Chapter 15 of the current text, contained entirely on the PLP CD, provides an overview of code improvement. Since most programmers will never write the back end of a compiler, the goal of Chapter 15 is more to convey a sense of what the compiler does than exactly how it does it. Programmers who understand this material will be in a better position to “work with” the compiler, knowing what is possible, what to expect in common cases, and how to avoid programming idioms that are hard to optimize. Topics include local and “global” (procedure-level) redundancy elimination, data flow analysis, loop optimization, and register allocation.



# Building a Runnable Program

**As noted in Section 1.6,** the various phases of compilation are commonly grouped into a *front end* responsible for the analysis of source code and a *back end* responsible for the synthesis of target code. Chapters 2 and 4 discussed the work of the front end, culminating in the construction of a syntax tree. The current chapter turns to the work of the back end, and specifically to code generation, assembly, and linking. We will continue with code improvement in Chapter 15.

In Chapters 6 through 9, we often discussed the code that a compiler would generate to implement various imperative language features. Now we will look at how the compiler produces that code from a syntax tree, and how it combines the output of multiple compilations to produce a runnable program. We begin in Section 14.1 with a more detailed overview of the work of program synthesis than was possible in Chapter 1. We focus in particular on one of several plausible ways of dividing that work into phases. In Section 14.2 we then consider the many possible forms of intermediate code passed between these phases. On the PLP CD we provide a bit more detail on two concrete examples: Diana, commonly used by Ada compilers, and RTL, used by the GNU compilers.

In Section 14.3 we discuss the generation of assembly code from an abstract syntax tree, using attribute grammars as a formal framework. In Section 14.4 we discuss the internal organization of binary object files and the layout of programs in memory. Section 14.5 describes assembly. Section 14.6 considers linking.

## 4.1 Back-End Compiler Structure

As we noted in Chapter 4, there is less uniformity in back-end compiler structure than there is in front-end structure. Even such unconventional compilers as text processors, source-to-source translators, and VLSI layout tools must scan, parse, and analyze the semantics of their input. When it comes to the back end, how-

ever, even compilers for the same language on the same machine can have very different internal structure.

As we shall see in Section 14.2, different compilers may use different intermediate forms to represent a program internally. Depending on the preferences of the programmers building a compiler, the constraints under which those programmers are working, and the expected user community, compilers may also differ dramatically in the forms of code improvement they perform. A simple compiler, or one designed for speed of compilation rather than speed of target code execution (a “just-in-time” compiler, for example) may not do much improvement at all. A just-in-time or “load-and-go” compiler (one that compiles and then executes a program as a single high level operation, without writing the target code to a file) may not use a separate linker. In many compilers, much or all of the code generator may be written automatically by a tool (a “code generator generator”) that takes a formal description of the target machine as input [GFH82].

### 14.1.1 A Plausible Set of Phases

#### EXAMPLE 14.1

Phases of compilation

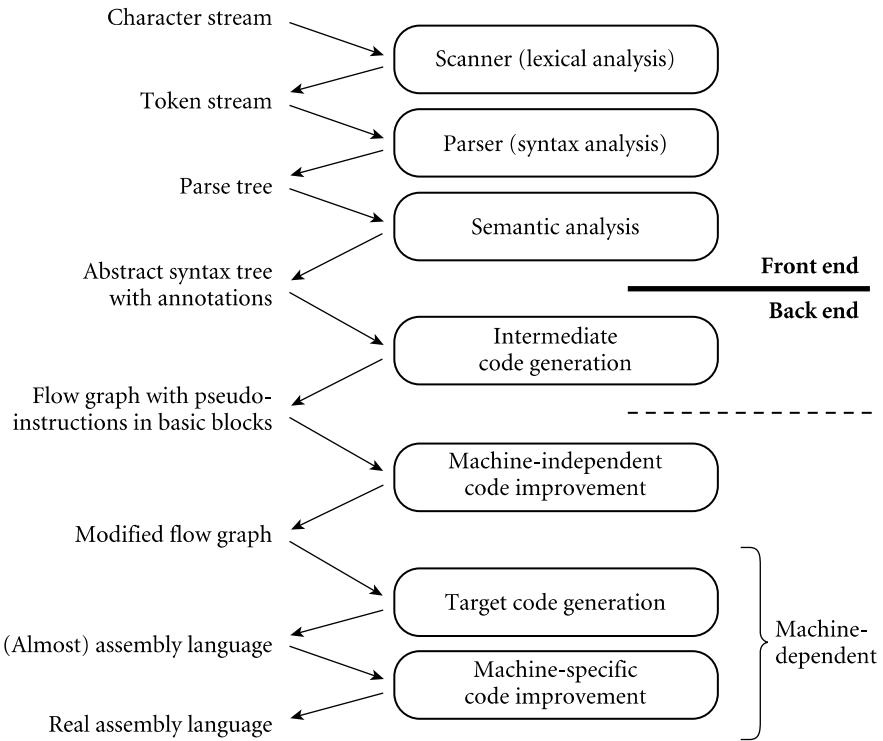
Figure 14.1 illustrates a plausible seven-phase structure for a conventional compiler. The first three phases (scanning, parsing, and semantic analysis) are language-dependent; the last two (target code generation and machine-specific code improvement) are machine-dependent, and the middle two (intermediate code generation and machine-independent code improvement) are (to first approximation) dependent on neither the language nor the machine. The scanner and parser drive a set of action routines that build a syntax tree. The semantic analyzer traverses the tree, performing all static semantic checks and initializing various attributes (mainly symbol table pointers and indications of the need for dynamic checks) of use to the back end. ■

While certain code improvements can be performed on syntax trees, a less hierarchical representation of the program makes most code improvement easier. Our example compiler therefore includes an explicit phase for intermediate code generation. The code generator begins by grouping the nodes of the tree into *basic blocks*, each of which consists of a maximal-length set of operations that should execute sequentially at run time, with no branches in or out. It then creates a *control flow graph* in which the nodes are basic blocks and the arcs represent interblock control flow. Within each basic block, operations are represented as instructions for an idealized RISC machine with an unlimited number of registers. We will call these *virtual registers*. By allocating a new one for every computed value, the compiler can avoid creating artificial connections between otherwise independent computations too early in the compilation process.

#### EXAMPLE 14.2

GCD program abstract  
syntax tree (reprise)

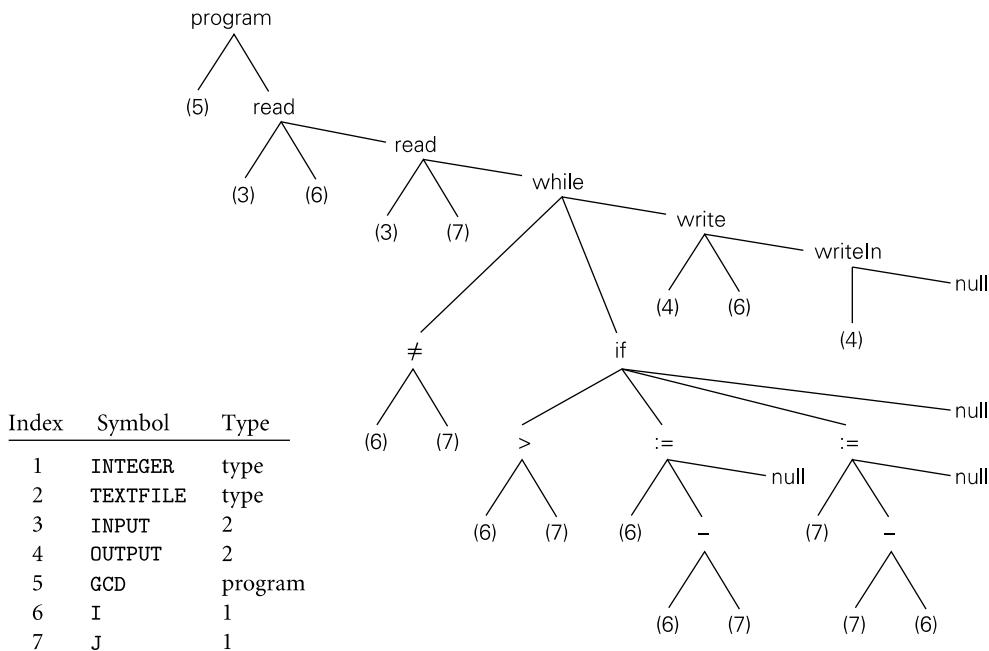
In Section 1.6 we used a simple greatest common divisor (GCD) program to illustrate the phases of compilation. The syntax tree for this program appeared in Figure 1.4; it is reproduced here (in slightly altered form) as Figure 14.2. A corresponding control flow graph appears in Figure 14.3. We will discuss techniques



**Figure 14.1** A plausible structure for the compiler back end. Here we have shown a sharper separation between semantic analysis and intermediate code generation than we considered in Chapter 1 (see Figure 1.2, page 23). Machine-independent code improvement employs an intermediate form that resembles the assembly language for an idealized machine with an unlimited number of registers. Machine-specific code improvement—register allocation and instruction scheduling in particular—employs the assembly language of the target machine. The dashed line shows a common alternative “break point” between the front end and back end of a two-pass compiler.

to generate this graph in Section 14.3 and Exercise 14.6. Additional examples of control flow graphs will appear in Chapter 15. ■

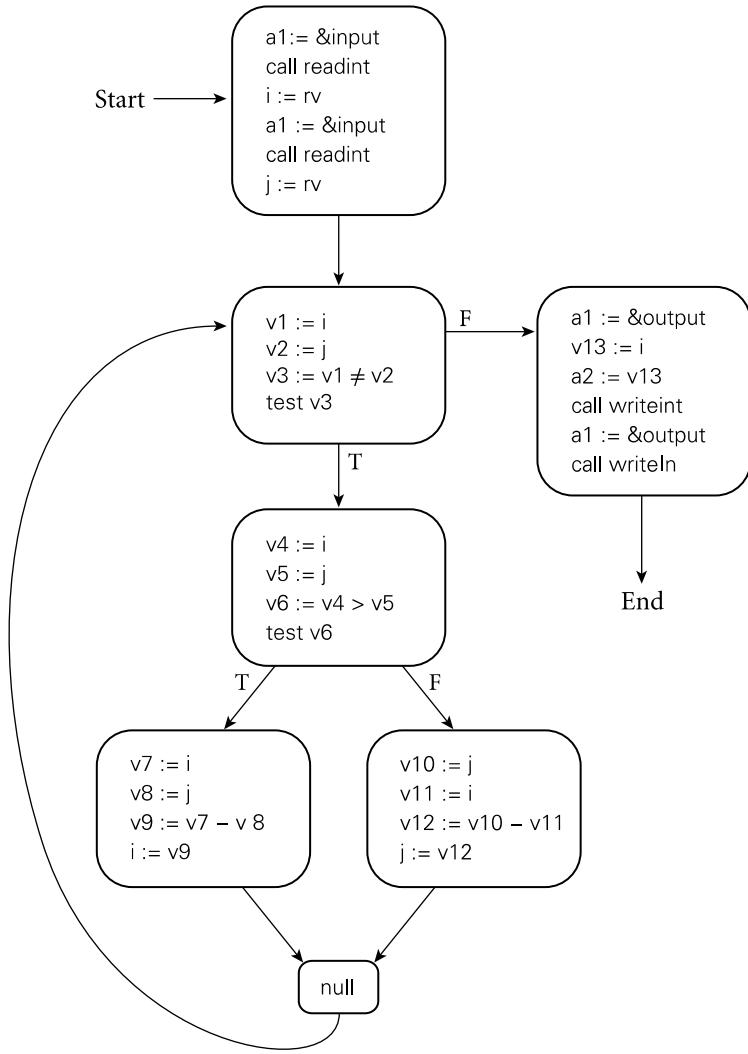
The second phase of the back end, machine-independent code improvement, performs a variety of transformations on the control flow graph. It modifies the instruction sequence within each basic block to eliminate redundant loads, stores, and arithmetic computations; this is *local code improvement*. It also identifies and removes a variety of redundancies across the boundaries between basic blocks within a subroutine; this is *global code improvement*. As an example of the latter, an expression whose value is computed immediately before an `if` statement need not be recomputed within the code that follows the `else`. Likewise an expression that appears within the body of a loop need only be evaluated once if its value will not change in subsequent iterations. Some global improvements change the number of basic blocks and/or the arcs among them.



**Figure 14.2** Syntax tree and symbol table for the GCD program. The only difference from Figure 1.4 is the addition of explicit null nodes to terminate statement lists.

It is worth noting that “global” code improvement typically considers only the current subroutine, not the program as a whole. Much recent research in compiler technology has been aimed at “truly global” techniques, known as *interprocedural code improvement*. Since programmers are generally unwilling to give up separate compilation (recompiling hundreds of thousands of lines of code is a very time-consuming operation), a practical interprocedural code improver must do much of its work at link time. One of the (many) challenges to be overcome is to develop a division of labor and an intermediate representation that allow the compiler to do as much work as possible during (separate) compilation but leave enough of the details undecided that the link-time code improver is able to do its job.

Following machine-independent code improvement, the next phase of compilation is target code generation. This phase strings the basic blocks together into a linear program, translating each block into the instruction set of the target machine and generating branch instructions (or “fall-throughs”) that correspond to the arcs of the control flow graph. The output of this phase differs from real assembly language primarily in its continued reliance on virtual registers. As long as the pseudoinstructions of the intermediate form are reasonably close to those of the target machine, this phase of compilation, though tedious, is more or less straightforward.



**Figure 14.3 Control flow graph for the GCD program.** Code within basic blocks is shown in the pseudo-assembly notation of Section 5.4.5, with a different virtual register (here named  $v1 \dots v13$ ) for every computed value. Registers  $a1$ ,  $a2$ , and  $rv$  are used to pass values to and from subroutines.

To reduce programmer effort and increase the ease with which a compiler can be ported to a new target machine, target code generators are often generated automatically from a formal description of the machine. Automatically generated code generators all rely on some sort of pattern-matching algorithm to replace sequences of intermediate code instructions with equivalent sequences of target machine instructions. References to several such algorithms can be found in the

Bibliographic Notes at the end of this chapter; details are beyond the scope of this book.

The final phase of our example compiler structure consists of register allocation and instruction scheduling, both of which can be thought of as machine-specific code improvement. Register allocation requires that we map the unlimited virtual registers employed in earlier phases onto the bounded set of architectural registers available in the target machine. If there aren't enough architectural registers to go around, we may need to generate additional loads and stores to multiplex a given architectural register among two or more virtual registers. As described in Section 5.5, instruction scheduling consists of reordering the instructions of each basic block in an attempt to fill the pipeline(s) of the target machine.

#### 14.1.2 Phases and Passes

In Section 1.6 we defined a *pass* of compilation as a phase or sequence of phases that is serialized with respect to the rest of compilation: it does not start until previous phases have completed, and it finishes before any subsequent phases start. If desired, a pass may be written as a separate program, reading its input from a file and writing its output to a file. Two-pass compilers are particularly common. They may be divided between the front end and the back end (i.e., between semantic analysis and intermediate code generation) or between intermediate code generation and global code improvement. In the latter case, the first pass is still commonly referred to as the front end and the second pass as the back end.

Like most compilers, our example generates symbolic assembly language as its output (a few compilers, including those written by IBM for the PowerPC, generate binary machine code directly). The assembler (not shown in Figure 14.1) behaves as an extra pass, assigning addresses to fragments of data and code, and translating symbolic operations into their binary encodings. In most cases, the input to the compiler will have consisted of source code for a single compilation unit. After assembly, the output will need to be *linked* to other fragments of the application, and to various preexisting subroutine libraries. Some of the work of linking may be delayed until load time (immediately prior to program execution) or even until run time (during program execution). We will discuss assembly and linking in Sections 14.5 through 14.7.

## 14.2 Intermediate Forms

An *intermediate form* (IF) provides the connection between the front end and the back end of the compiler, and continues to represent the program during the various back-end phases.

IFs can be classified in terms of their *level*, or degree of machine dependence. High-level IFs are often based on trees or directed acyclic graphs (DAGs) that

directly capture the hierarchical structure of modern programming languages. A high-level IF facilitates certain kinds of machine-independent code improvement, incremental program updates (e.g., in a language-based editor), direct interpretation, and other operations based strongly on the structure of the source. Because the permissible structure of a tree can be described formally by a set of productions (as described in Section 4.6), manipulations of tree-based forms can be written as attribute grammars.

*Stack-based* languages are another common type of high-level IF. We saw two examples of these languages in Section 1.4: the P-code generated by many early Pascal compilers and the byte code used by Java. Stack-based IFs are both simple and compact. Though written in linear form, they closely resemble the result of enumerating tree nodes in post-order. Operations in a stack-based language obtain their operands from, and return their result to, a common implicit stack.

The most common medium-level IFs consist of three-address instructions for a simple idealized machine, typically one with an unlimited number of registers.

#### DESIGN & IMPLEMENTATION

##### Stack-based IFs

For Pascal, the simplicity of P-code interpreters was a major contributing factor to the language's popularity: Pascal was easy to port to a wide variety of machines. For Java, the compactness of byte code helps reduce the time required to send program fragments (*applets*) over low-bandwidth Internet links. Unfortunately, stack-based languages do not lend themselves well to many important code improvements, especially for modern machines. As a result they tend not to be used in most conventional compilers.

#### DESIGN & IMPLEMENTATION

##### Postscript

Perhaps the most important use of stack-based languages today occurs in document preparation. Many document compilers (TeX, troff, Microsoft Word, etc.) generate Postscript as their target language (most employ some special purpose intermediate language as well, and have multiple back ends, so they can also generate other target languages). Postscript is stack-based. It is portable, compact, and easy to generate. It is also written in ASCII, so it can be read (albeit with some difficulty) by human beings. Postscript interpreters are embedded in most professional-quality printers. Issues of code improvement are relatively unimportant: most of the time required for printing is consumed by network delays, mechanical paper transport, and data manipulations embedded in (optimized) library routines; interpretation time is seldom a bottleneck. Compactness, on the other hand, is crucial, because it contributes to network delays.

Since the typical instruction specifies two operands, an operator, and a destination, three-address instructions are sometimes called *quadruples*. In older compilers, one may sometimes find an intermediate form consisting of *triples* or *indirect triples* in which destinations are specified implicitly: the index of a triple in the instruction stream serves as the name of the result, and an operand is generally named by specifying the index of the triple that produced it.

Different compilers use different IFs. Many compilers use more than one IF internally, though in the common two-pass organization one of these is distinguished as “the” intermediate form by virtue of being the externally visible connection between the front end and the back end. In the example of Section 14.1.1, the syntax trees passed from semantic analysis to intermediate code generation constitute a high-level IF. Control flow graphs containing pseudo-assembly language (passed in and out of machine-independent code improvement) are a medium-level IF. The assembly language of the target machine (initially with virtual registers; later with architectural registers) serves as a low-level IF.

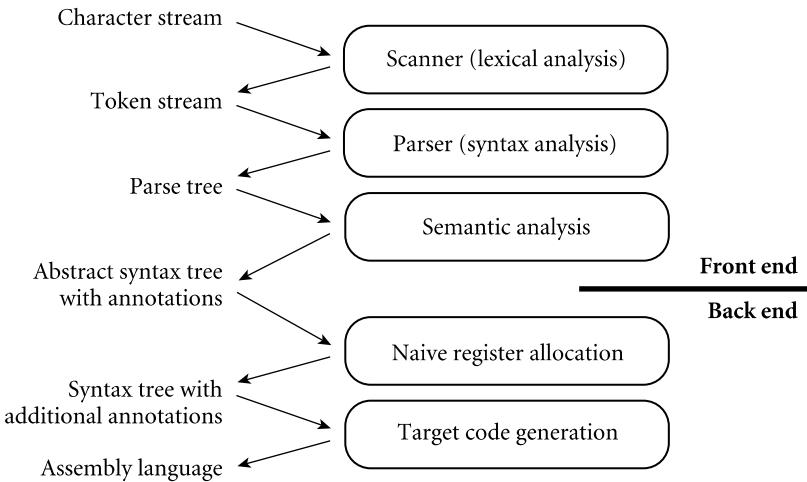
Compilers that have back ends for several different target architectures tend to do as much work as possible on a high- or medium-level IF, so that the machine-independent parts of the code improver can be shared by different back ends. By contrast, some (but not all) compilers that generate code for a single architecture perform most code improvement on a comparatively low-level IF, closely modeled after the assembly language of the target machine.

In a multilanguage compiler family, an IF that is independent of both source language and target machine allows a software vendor who wishes to sell compilers for  $n$  languages on  $m$  machines to build just  $n$  front ends and  $m$  back ends, rather than  $n \times m$  integrated compilers. Even in a single-language compiler family, a common, possibly language-dependent IF simplifies the task of porting to a new machine by isolating the code that needs to be changed. In a rich program development environment, there may be a variety of tools in addition to the passes of the compiler that understand and operate on the IF. Examples include editors, assemblers, linkers, debuggers, pretty-printers, and version-management software. In a language system capable of interprocedural (whole-program) code improvement, separately compiled modules and libraries may be compiled only to the IF, rather than the target language, leaving the final stages of compilation to the linker.

To be stored in a file, an IF requires a linear representation. Sequences of quadruples are naturally linear. Tree-based IFs can be linearized via ordered traversal. Structures like control flow graphs can be linearized by replacing pointers with indices relative to the beginning of the file.

#### IN MORE DEPTH

On the PLP CD we consider a pair of widely used IFs. The first is a high-level tree-based form called Diana [GWEB83], used by most Ada compilers. The second is a medium-level IF called RTL (Register Transfer Language), used by gcc



**Figure 14.4** A simpler, nonoptimizing compiler structure, assumed in Section 14.3. The target code generation phase closely resembles the intermediate code generation phase of Figure 14.1.

and most of the Free Software Foundation’s other GNU compilers (including the Ada 95 Translator, gnat).

## 14.3 Code Generation

### EXAMPLE 14.3

Simpler compiler structure

The back end of Figure 14.1 is too complex to present in any detail in a single chapter. To limit the scope of our discussion, we will content ourselves in this chapter with producing correct but naive code. This choice will allow us to consider a significantly simpler back end. Starting with the structure of Figure 14.1, we drop the machine-independent code improver and then merge intermediate and target code generation into a single phase. This merged phase generates pure, linear assembly language; because we are not performing code improvements that alter the program’s control flow, there is no need to represent that flow explicitly in a control flow graph. We also adopt a much simpler register allocation algorithm, which can operate directly on the syntax tree prior to code generation, eliminating the need for virtual registers and the subsequent mapping onto architectural registers. Finally, we drop instruction scheduling. The resulting compiler structure appears in Figure 14.4. Its code generation phase closely resembles the intermediate code generation of Figure 14.1. ■

### 14.3.1 An Attribute Grammar Example

Like semantic analysis, intermediate code generation can be formalized in terms of an attribute grammar, though it is most commonly implemented via hand-

```

reg_names : array [0..k-1] of register_name := ["r1", "r2", ..., "rk"]
 -- ordered set of temporaries

program → id stmt
 ▷ stmt.next_free_reg := 0
 ▷ program.code := ["main:"]
 ▷ program.name := id.stp→name

while : stmt1 → expr stmt2 stmt3
 ▷ expr.next_free_reg := stmt2.next_free_reg := stmt3.next_free_reg := stmt1.next_free_reg
 ▷ L1 := new_label(); L2 := new_label()
 ▷ stmt1.code := ["goto" L1] + [L2 ":"]
 ▷ stmt2.code + [L1 ":"]
 ▷ expr.code + ["if" expr.reg "goto" L2] + stmt3.code

if : stmt1 → expr stmt2 stmt3 stmt4
 ▷ expr.next_free_reg := stmt2.next_free_reg := stmt3.next_free_reg := stmt4.next_free_reg :=
 stmt1.next_free_reg
 ▷ L1 := new_label(); L2 := new_label()
 ▷ stmt1.code := expr.code + ["if" expr.reg "goto" L1] + stmt3.code + ["goto" L2]
 ▷ [L1 ":"]
 ▷ stmt2.code + [L2 ":"]
 ▷ stmt4.code

assign : stmt1 → id expr stmt2
 ▷ expr.next_free_reg := stmt2.next_free_reg := stmt1.next_free_reg
 ▷ stmt1.code := expr.code + [id.stp→name "=" expr.reg] + stmt2.code

read : stmt1 → id1 id2 stmt2
 ▷ stmt1.code := ["a1 := &" id1.stp→name] -- file
 ▷ ["call" if id2.stp→type = int then "readint" else ...]
 ▷ [id2.stp→name ":= rv"] + stmt2.code

write : stmt1 → id expr stmt2
 ▷ expr.next_free_reg := stmt2.next_free_reg := stmt1.next_free_reg
 ▷ stmt1.code := ["a1 := &" id.stp→name] -- file
 ▷ ["a2 := " expr.reg] -- value
 ▷ ["call" if id.stp→type = int then "writeint" else ...] + stmt2.code

writeln : stmt1 → id stmt2
 ▷ stmt1.code := ["a1 := &" id.stp→name] + ["call writeln"] + stmt2.code

null : stmt → ε
 ▷ stmt.code := nil

'<>' : expr1 → expr2 expr3
 ▷ handle_op(expr1, expr2, expr3, "≠")

```

**Figure 14.5 Attribute grammar to generate code from a syntax tree.** Square brackets delimit individual target instructions. Juxtaposition indicates concatenation within instructions; the “+” operator indicates concatenation of instruction lists. The handle\_op macro is used in three of the attribute rules. (continued)

written ad hoc traversal of a syntax tree. We present an attribute grammar here for the sake of clarity.

In Figure 1.5 (page 29) we presented naive MIPS assembly language for the GCD program. We will use our attribute grammar example to generate a similar version here, in pseudo-assembly notation. Because this notation is now meant

```

'>' : expr1 → expr2 expr3
 ▷ handle_op(expr1, expr2, expr3, ">")

'-' : expr1 → expr2 expr3
 ▷ handle_op(expr1, expr2, expr3, "-")

id : expr → ε
 ▷ expr.reg := reg_names[expr.next_free_reg mod k]
 ▷ expr.code := [expr.reg ":"= expr.stp → name]

macro handle_op(ref result, L_operand, R_operand, op : syntax_tree_node)
 result.reg := L_operand.reg
 L_operand.next_free_reg := result.next_free_reg
 R_operand.next_free_reg := result.next_free_reg + 1
 if R_operand.next_free_reg < k
 spill_code := restore_code := nil
 else
 spill_code := ["*sp := reg_name[R_operand.next_free_reg mod k]
 + ["sp := sp - 4"]
 restore_code := ["sp := sp + 4"
 + [reg_names[R_operand.next_free_reg mod k] ":"= *sp"]]
 result.code := L_operand.code + spill_code + R_operand.code
 + [result.reg ":"= L_operand.reg op R_operand.reg] + restore_code

```

**Figure 14.5 (continued)**

to represent target code, rather than medium- or low-level intermediate code, we will assume a fixed, limited register set reminiscent of real machines. We will reserve several registers ( $a_1, a_2, sp, rv$ ) for special purposes; others ( $r_1 \dots r_k$ ) will be available for temporary values and expression evaluation.

#### EXAMPLE 14.4

An attribute grammar for code generation

Figure 14.5 contains a fragment of our attribute grammar. To save space, we have shown only those productions that actually appear in Figure 14.2. As in Chapter 4, notation like *while* : *stmt* on the left-hand side of a production indicates that a *while* node in the syntax tree is one of several kinds of *stmt* node; it may serve as the *stmt* in the right-hand side of its parent production. In our attribute grammar fragment, *program*, *expr*, and *stmt* all have a synthesized attribute code that contains a sequence of instructions. *Program* has a synthesized attribute name of type string. *Id* has a synthesized attribute *stp* that points to the symbol table entry for the identifier. *Expr* has a synthesized attribute *reg* that indicates the register that will hold the value of the computed expression at run time. *Expr* and *stmt* have an inherited attribute *next\_free\_reg* that indicates the next register (in an ordered set of temporaries) that is available for use (i.e., that will hold no useful value at run time) immediately before evaluation of a given expression or statement. ■

Because we use a symbol table in our example, and because symbol tables lie outside the formal attribute grammar framework, we must augment our attribute grammar with some extra code for storage management. Specifically,

prior to evaluating the attribute rules of Figure 14.5, we must traverse the symbol table in order to calculate stack frame offsets for local variables and parameters (none of which occur in the GCD program) and in order to generate assembler directives to allocate space for global variables (of which our program has two). Storage allocation and other assembler directives will be discussed in more detail in Section 14.5.

### 14.3.2 Register Allocation

**EXAMPLE 14.5**

Stack-based register allocation

Evaluation of the rules of the attribute grammar itself consists of two main tasks. In each subtree we first determine the registers that will be used to hold various quantities at run time; then we generate code. Our naive register allocation strategy uses the `next_free_reg` inherited attribute to manage registers  $r1 \dots rk$  as an expression evaluation stack. To calculate the value of  $(a + b) \times (c - (d / e))$  for example, we would generate the following.

```

r1 := a -- push a
r2 := b -- push b
r1 := r1 + r2 -- add
r2 := c -- push c
r3 := d -- push d
r4 := e -- push e
r3 := r3 / r4 -- divide
r2 := r2 - r3 -- subtract
r1 := r1 × r2 -- multiply

```

Allocation of the next register on the “stack” occurs in the production  $id : expr \longrightarrow \epsilon$ , where we use `expr.next_free_reg` to index into `reg_names`, the array of temporary register names, and in macro `handle_op`, where we increment `next_free_reg` to make this register unavailable during evaluation of the right-hand operand. There is no need to “pop” the “register stack” explicitly; this happens automatically when the attribute evaluator returns to a parent node and uses the parent’s (unmodified) `next_free_reg` attribute. In our example grammar, left-hand operands are the only constructs that tie up a register during the evaluation of anything else. In a more complete grammar, other long-term uses of registers would probably occur in constructs like `for` loops (for the step size, index, and bound).

In a particularly complicated fragment of code it is possible to run out of architectural registers. In this case we must *spill* one or more registers to memory. Our naive register allocator pushes a register onto the program’s subroutine call stack, reuses the register for another purpose, and then pops the saved value back into the register before it is needed again. In effect, architectural registers hold the top  $k$  elements of an expression evaluation stack of effectively unlimited size. ■

It should be emphasized that our register allocation algorithm, while correct, makes very poor use of machine resources. We have made no attempt to reor-

ganize expressions to minimize the number of registers used, or to keep commonly used variables in registers over extended periods of time (avoiding loads and stores). If we were generating medium-level intermediate code, instead of target code, we would employ virtual registers, rather than architectural ones, and would allocate a new one every time we needed it, never reusing one to hold a different value. Mapping of virtual registers to architectural registers would occur much later in the compilation process.

**EXAMPLE 14.6****GCD program target code**

Target code for the GCD program appears in Figure 14.6. The first few lines are generated during symbol table traversal, prior to attribute evaluation. Attribute `program.name` might be passed to the assembler, to tell it the name of the file into which to place the runnable program. A production-quality compiler would probably also generate assembler directives to embed symbol-table information in the target program. As in Figure 1.5, the quality of our code is very poor. We will investigate techniques to improve it in Chapter 15. In the remaining sections of the current chapter we will consider assembly and linking.

 **CHECK YOUR UNDERSTANDING**

1. What is a *code generator generator*? Why might it be useful?
2. What is a *basic block*? A *control flow graph*?
3. What are *virtual registers*? What purpose do they serve?
4. What is the difference between *local* and *global* code improvement?
5. What is *register spilling*?
6. Explain what is meant by the “level” of an intermediate form (IF). What are the comparative advantages and disadvantages of high-, medium-, and low-level IFs?
7. What is the IF most commonly used in Ada compilers?
8. Name two advantages of a stack-based IF. Name one disadvantage.
9. Explain the rationale for basing a family of compilers (several languages, several target machines) on a single IF.
10. Outline some of the major design alternatives for back-end compiler organization and structure.
11. Why might a compiler employ more than one IF?

```

-- first few lines generated during symbol table traversal
.data -- begin static data
i: .word 0 -- reserve one word to hold i
j: .word 0 -- reserve one word to hold j
.text -- begin text (code)
 -- remaining lines accumulated into program.code
main:
 a1 := &input -- "input" and "output" are file control blocks
 -- located in a library, to be found by the linker
 call readint -- "readint", "writeint", and "writeln" are library subroutines
 i := rv
 a1 := &input
 call readint
 j := rv
 goto L1
L2: r1 := i -- body of while loop
 r2 := j
 r1 := r1 > r2
 if r1 goto L3
 r1 := j -- "else" part
 r2 := i
 r1 := r1 - r2
 j := r1
 goto L4
L3: r1 := i -- "then" part
 r2 := j
 r1 := r1 - r2
 i := r1
L4:
L1: r1 := i -- test terminating condition
 r2 := j
 r1 := r1 ≠ r2
 if r1 goto L2
 a1 := &output
 r1 := i
 a2 := r1
 call writeint
 a1 := &output
 call writeln
 goto exit -- return to operating system

```

**Figure 14.6** Target code for the GCD program, generated from the syntax tree of Figure 14.2, using the attribute grammar of Figure 14.5.

## 14.4 Address Space Organization

Assemblers, linkers, and loaders typically operate on a pair of related file formats: *relocatable* object code and *executable* object code. Relocatable object code is acceptable as input to a linker; multiple files in this format can be combined to create an executable program. Executable object code is acceptable as input to a loader: it can be brought into memory and run. A relocatable object file includes the following descriptive information.

*import table:* Identifies instructions that refer to named locations whose addresses are unknown, but are presumed to lie in other files yet to be linked to this one.

*relocation table:* Identifies instructions that refer to locations within the current file, but that must be modified at link time to reflect the offset of the current file within the final, executable program.

*export table:* Lists the names and addresses of locations in the current file that may be referred to in other files.

Imported and exported names are known as *external symbols*.

An executable object file is distinguished by the fact that it contains no references to external symbols. It also defines a starting address for execution. An executable file may or may not be relocatable, depending on whether it contains the tables above.

Internally, an object file is typically divided into several sections, each of which is handled differently by the linker, loader, or operating system. The first section includes the import, export, and relocation tables, together with an indication of how much space will be required by the program for noninitialized static data. Other sections commonly include code (instructions), read-only data (constants, jump tables for case statements, etc.), initialized but writable static data, and high-level symbol table information saved by the compiler. The initial descriptive section is used by the linker and loader. The high-level symbol table section is used by debuggers and performance profilers. Neither of these tables is usually brought into memory at run time; neither is needed by most running programs (an exception occurs in the case of programs that employ *reflection* mechanisms to examine their own type structure).

In its runnable (loaded) form, a program is typically organized into several *segments*. On some machines (e.g., the x86 or PA-RISC), segments are visible to the assembly language programmer, and may be named explicitly in instructions. More commonly on modern machines, segments are simply subsets of the address space that the operating system manages in different ways. Two or three of them—code, constants, and initialized data—correspond to sections of the object file. Code and constants are usually read-only, and are often combined in a single segment; the operating system arranges to receive an interrupt if the program attempts to modify them. (In response to such an interrupt, it will

most likely print an error message and terminate the program.) Initialized data is writable. At load time, the operating system either reads code, constants, and initialized data from disk, or arranges to read them in at run time in response to “invalid access” (page fault) interrupts or dynamic linking requests.

In addition to code, constants, and initialized data, the typical running program has several additional segments:

*uninitialized data:* May be allocated at load time or on demand in response to page faults. Usually zero-filled, both to provide repeatable symptoms for programs that erroneously read data they have not yet written, and to enhance security on multiuser systems by preventing a program from reading the contents of pages written by previous users.

*stack:* May be allocated in some fixed amount at load time. More commonly, is given a small initial size and is then extended automatically by the operating system in response to (faulting) accesses beyond the current segment end.

*heap:* Like stack, may be allocated in some fixed amount at load time. More commonly, is given a small initial size and is then extended in response to explicit requests (via system call) from heap-management library routines.

*files:* In many systems, library routines allow a program to *map* a file into memory. The *map* routine interacts with the operating system to create a new segment for the file, and returns the address of the beginning of the segment. The contents of the segment are usually fetched from disk on demand, in response to page faults.

*dynamic libraries:* Modern operating systems typically arrange for most programs to share a single copy of the code for popular libraries (Section 14.7). From the point of view of an individual process, each such library tends to occupy a pair of segments: one for the shared code and one for a private copy of any writable data it may use.

## 14.5 Assembly

Some compilers translate source files directly into object files acceptable to the linker. More commonly, they generate assembly language that must subsequently be processed by an assembler to create an object file.

In our examples we have consistently employed a symbolic (textual) notation for code. Within a compiler, the representation would not be ASCII text, but it would still be symbolic, most likely consisting of records and linked lists. To translate this symbolic representation into executable code, we must

1. replace opcodes and operands with their machine language encodings, and
2. replace uses of symbolic names with actual addresses.

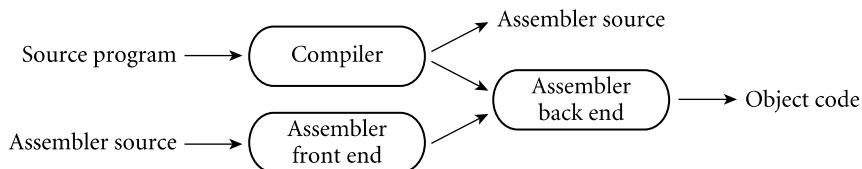
These are the principal tasks of an assembler.

In the early days of computing, most programmers wrote in assembly language. To simplify the more tedious and repetitive aspects of assembly programming, assemblers often provided extensive macro expansion facilities. With the move to high-level languages, such programmer-centric features have largely disappeared. Most assembly language programs now are written by compilers. At the same time, the evolution of compiler technology and the development of RISC machines have pushed new features into the assembler. In particular, some assemblers now perform some of the machine-specific parts of code improvement, such as instruction scheduling, register allocation, and peephole optimization (to be described in Section 15.2).

**EXAMPLE 14.7**

Assembly as a final compiler pass

When passing assembly language from the compiler to the assembler, it makes sense to use some internal (records and linked lists) representation. At the same time, we must provide a textual front end to accommodate the occasional need for human input:

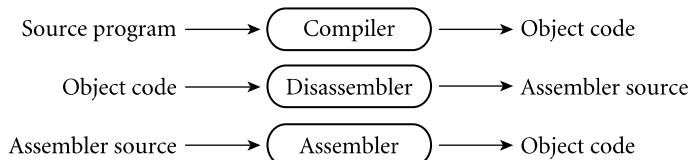


The text-based assembler front end simply translates ASCII source into internal symbolic form. By sharing the assembler back end, the compiler and assembler front end avoid duplication of effort. For debugging purposes, the compiler will generally have an option to dump a textual representation of the code it passes to the assembler. ■

**EXAMPLE 14.8**

Direct generation of object code

An alternative organization has the compiler generate object code directly:



This organization gives the compiler a bit more flexibility: operations normally performed by an assembler (e.g., assignment of addresses to variables) can be performed earlier if desired. Because there is no separate assembly pass, the overall translation to object code may be slightly faster. The stand-alone assembler can be relatively simple. If it is used only for small, special purpose code fragments, it probably doesn't need to perform instruction scheduling or other machine-specific code improvement. Using a disassembler instead of an assembly language dump from the compiler ensures that what the programmer sees corresponds precisely to what is in the object file. If the compiler uses a fancier assembler as a back end, then program modifications effected by the assembler will not be visible in the assembly language dumped by the compiler. ■

### 14.5.1 Emitting Instructions

The most basic task of the assembler is to translate symbolic representations of instructions into binary form. In some assemblers this is an entirely straightforward task, because there is a one-one correspondence between mnemonic operations and instruction opcodes. Many assemblers, however, extend the instruction set in minor ways (sometimes a large number of minor ways) to make the assembly language easier for human beings to read. Most MIPS assemblers, for example, provide a large number of pseudoinstructions that translate into different real instructions depending on their arguments, or that correspond to multi-instruction sequences. Here are a few examples.

#### EXAMPLE 14.9

##### Instruction variants

- Many of the arithmetic/logic instructions come in several variants, depending on whether they take one of their operands from an “immediate” constant field within the instruction, as opposed to taking both from registers. Strictly speaking, these are different instructions; the assembler picks the right one based on the syntax of the operands:

```
add $10, $8, $9 -- r10 := r8 + r9
```

is translated as is, while

add \$10, \$8, 0x12	becomes	addi \$10, \$8, 0x12
---------------------	---------	----------------------

#### EXAMPLE 14.10

##### Pseudoinstruction expansion

- Some pseudoinstructions actually generate multi-instruction sequences. For example, the pseudoinstruction

```
div $10, $8, $9
```

is meant to divide register 8 by register 9 and put the result in register 10. In actuality, the assembler translates it into the following 11-instruction sequence:

```
div $8, $9 -- LO := quotient; HI := remainder
bne $9, $0, L1 -- branch if divisor not zero
nop -- branch delay
break 0x7 -- trap to operating system
L1: li $1, -1
bne $9, $1, L2 -- branch if divisor not -1
lui $1, 0x8000 -- $1 := 0x80000000
bne $8, $1, L2 -- branch if dividend not minint
nop -- branch delay
break 0x6 -- overflow; trap to OS
L2: mflo $10 -- $10 := quotient
```

In most cases, the hardware integer divide instruction generates an exact quotient and remainder into the special registers LO and HI. The exceptions are division by zero and division of the largest-magnitude negative number (for which two’s complement arithmetic has no positive counterpart) by  $-1$ . Software must test for these cases; the hardware simply produces invalid results,

silently. Assuming the tests succeed, the instruction sequence ends by moving the quotient to the desired target register.

**EXAMPLE 14.11**

## Two-instruction loads

- Because instructions are 32 bits in length, a 32-bit constant cannot be loaded into a register with a single instruction:

li \$14, 0x12345abc	becomes	lui \$14, 0x1234 ori \$14, 0x5abc
---------------------	---------	--------------------------------------

The mnemonics `li`, `lui`, and `ori` stand for “load immediate,” “load upper immediate,” and “or immediate.” The `lui` instruction loads a 16-bit operand into the high-order 16 bits of the target register, and sets the low-order 16 bits to zero.

**EXAMPLE 14.12**

## Nontrivial conditional branches

- The assembler supports a large suite of conditional branch pseudoinstructions, many of which compare a register to a constant or to another register, and branch on the result. The hardware, on the other hand, can only compare a register to zero, or test two registers for equality. Thus

bge \$9, 0x10, foo	becomes	slti \$1, \$9, 0x10 beq \$1, \$0, foo
--------------------	---------	------------------------------------------

The `slti` instruction sets its destination register to one if the source register is less than the immediate operand, or to zero otherwise.

**EXAMPLE 14.13**

## Assembler directives

In addition to translating from symbolic to binary instruction representations, most assemblers respond to a variety of *directives*. Here are some examples from the MIPS assembler.

*segment switching:* The `.text` directive indicates that subsequent instructions and data should be placed in the code (text) segment. The `.data` directive indicates that subsequent instructions and data should be placed in the initialized data segment. (It is possible, though uncommon, to put instructions in the data segment, or data in the code segment.) The `.space n` directive indicates that *n* bytes of space should be reserved in the uninitialized data segment. (This latter directive is usually preceded by a label.)

*data generation:* The `.byte`, `.half`, `.word`, `.float`, and `.double` directives each take a sequence of arguments, which they place in successive locations in the current segment of the output program. They differ in the types of operands. The related `.ascii` directive takes a single character string as argument, which it places in consecutive bytes.

*symbol identification:* The `.globl name` directive indicates that *name* should be entered into the table of exported symbols.

*alignment:* The `.align n` directive causes the subsequent output to be aligned at an address evenly divisible by  $2^n$ .

In effect, most RISC assemblers implement a virtual machine whose instruction set is “nicer” than that of the real hardware. In addition to pseudoinstructions, the virtual machine may have nondelayed branches. If desired, the com-

piler or assembly language programmer can ignore the existence of branch delays. The assembler will move nearby instructions to fill delay slots if possible, or generate nops if necessary. (To minimize the number of nops, it may still be desirable for the compiler to place independent instructions near the branch, where the assembler will be able to find them. To support systems programmers, the assembler must also make it possible to specify [e.g., with a .set noreorder directive] that delay slots have already been filled.) Note that the task of filling branch delays is substantially easier in the presence of nullifying branches (Section 5.5.1). Assuming that the target of the branch lies within the current file, and is not itself a branch, we can always fill the delay slot with a duplicate of the target instruction, and increment the target address.

Some assemblers go beyond the simple filling of branch delays to provide the final pass of general purpose instruction scheduling. Though this job can be handled by the compiler, the existence of pseudoinstructions such as the division example above argues strongly for doing it in the assembler. In addition to having two branch delays that might be filled by neighboring instructions, the expanded division sequence can be used as a source of instructions to fill nearby branch, load, or functional unit delays.

#### 14.5.2 Assigning Addresses to Names

Like compilers, assemblers commonly work in several phases. If the input is textual, an initial phase scans and parses the input, and builds an internal representation. In the most common organization there are two additional phases. The first identifies all internal and external (imported) symbols, assigning locations to the internal ones. This phase is complicated by the fact that the length of some instructions (on a CISC machine) or the number of real instructions produced by a pseudoinstruction (on a RISC machine) may depend on the number of significant bits in an address. Given values for symbols, the final phase produces object code.

Within the object file, any symbol mentioned in a .globl directive must appear in the table of exported symbols, with an entry that indicates the symbol's address. Any symbol referred to in a directive or an instruction, but not defined in the input program, must appear in the table of imported symbols, with an entry that identifies all places in the code at which such references occur. Finally, any instruction or datum whose value depends on the placement of the current file within a final executable program must be listed in the relocation table.

##### EXAMPLE 14.14

Encoding of addresses in object files

Traditionally, assemblers for CISC machines distinguished between *absolute* and *relocatable* words in an object file. Absolute words are known at assembly time; they need not be changed by the linker. Examples include constants and register-register instructions. A relocatable word, on the other hand, must be modified by adding to it the address within the final program of the code or data segment of the current object file. A CISC jump instruction, for example, might consist of a one-byte jmp opcode followed by a four-byte target address.

For a local target, the address bytes in the object file would contain the symbol's offset within the file. The linker would finalize the address by adding the offset of the file's code segment within the final program.

On RISC machines, this single form of relocation no longer suffices. Addresses are encoded into instructions in many different ways, and these encodings must be reflected in the relocation table and the import table. On a MIPS processor, for example, a *j* (jump) instruction has a 26-bit target field. The processor left-shifts this field by 2 bits and tacks on the high-order 4 bits of the address of the instruction in the delay slot. To relocate such an instruction, the linker must right-shift and left-truncate the address of the file's code segment, add it into the low-order 26 bits of the instruction, and verify that the target and delay slot instructions share the same top 4 address bits. In a similar vein, a two-instruction load of a 32-bit quantity (as described in Example 14.11) requires the linker to recalculate the 16-bit operands of both instructions. ■

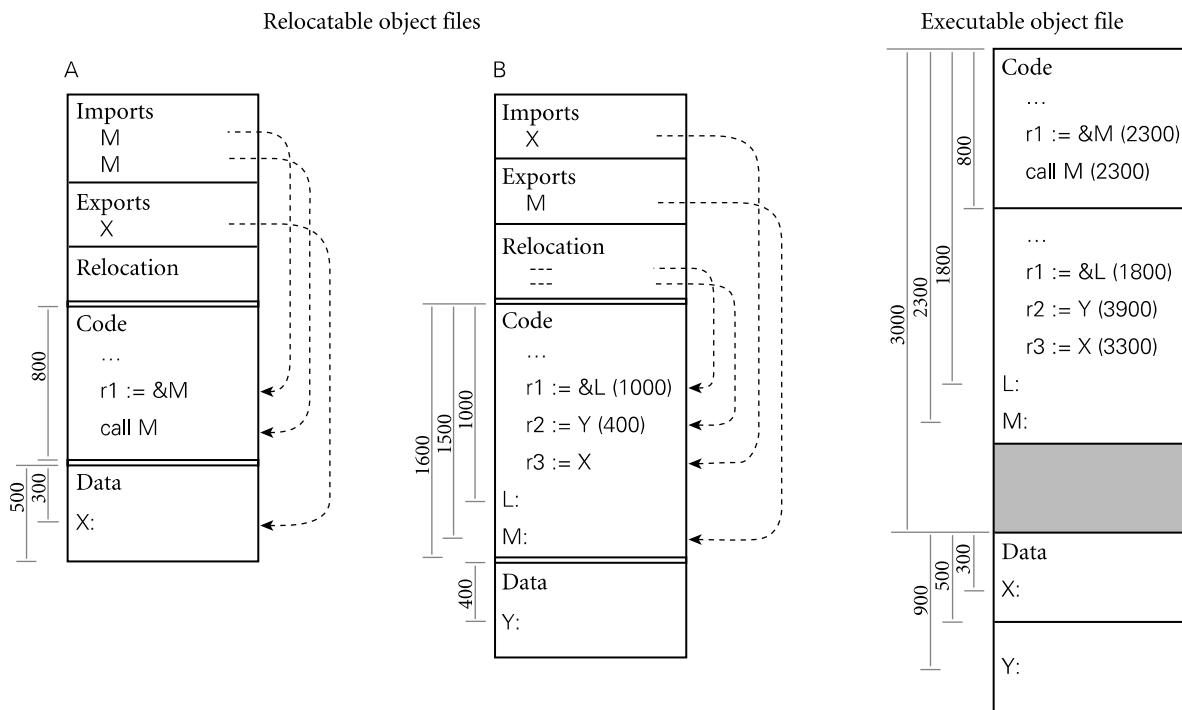
## 14.6 Linking

Most language implementations—certainly all that are intended for the construction of large programs—support separate compilation: fragments of the program can be compiled and assembled more or less independently. After compilation, these fragments (known as *compilation units*) are “glued together” by a *linker*. In many languages and environments, the programmer explicitly divides the program into modules or files, each of which is separately compiled. More integrated environments may abandon the notion of a file in favor of a database of subroutines, each of which is separately compiled.

The task of a linker is to join together compilation units. A *static linker* does its work prior to program execution, producing an executable object file. A *dynamic linker* (described in Section 14.7) does its work after the program has been brought into memory for execution.

Each of the compilation units of a program to be linked must be a relocatable object file. Typically, some of these files will have been produced by compiling fragments of the application being constructed, while others will be general purpose library packages needed by the application. Since most programs make use of libraries, even a “one-file” application typically needs to be linked.

Linking involves two subtasks: relocation and the resolution of external references. Some authors refer to relocation as *loading*, and call the entire “joining together” process “link-loading.” Other authors (including the current one) use “loading” to refer to the process of bringing an executable object file into memory for execution. On very simple machines, or on machines with very simple operating systems, loading entails relocation. More commonly, the operating system uses virtual memory to give every program the impression that it starts at some standard address (e.g., zero). In many systems loading also entails a certain amount of linking (Section 14.7).



**Figure 14.7** Linking relocatable object files A and B to make an executable object file. A's code section has been placed at offset 0, with B's code section immediately after, at offset 800. To allow the operating system to establish different protections for the code and data segments, A's data section has been placed at the next page boundary (offset 3000), with B's data section immediately after (offset 3500). External references to M and X have been set to use the appropriate addresses. Internal references to L and Y have been updated by adding in the starting addresses of B's code and data sections, respectively.

### 14.6.1 Relocation and Name Resolution

Each relocatable object file contains the information required for linking: the import, export, and relocation tables. A static linker uses this information in a two-phase process analogous to that described for assemblers in Section 14.5. In the first phase, the linker gathers all of the compilation units together, chooses an order for them in memory, and notes the address at which each will consequently lie. In the second phase, the linker processes each unit, replacing unresolved external references with appropriate addresses, and modifying instructions that need to be relocated to reflect the addresses of their units. These phases are illustrated pictorially in Figure 14.7. Addresses and offsets are assumed to be written in hexadecimal notation, with a page size of 4K ( $1000_{16}$ ) bytes. ■

#### EXAMPLE 14.15

Static linking

Libraries present a bit of a challenge. Many consist of hundreds of separately compiled program fragments, most of which will not be needed by any particular application. Rather than link the entire library into every application, the linker needs to search the library to identify the fragments that are referenced from the

main program. If these refer to additional fragments, then those must be included also, recursively. Many systems support a special library format for relocatable object files. A library in this format may contain an arbitrary number of code and data sections, together with an index that maps symbol names to the sections in which they appear.

### 14.6.2 Type Checking

Within a compilation unit, the compiler enforces static semantic rules. Across the boundaries between units, it uses module headers to enforce the rules pertaining to external references. In effect, the header for module  $M$  makes a set of promises regarding  $M$ 's interface to its users. When compiling the body of  $M$ , the compiler ensures that those promises are kept. Imagine what could happen, however, if we compiled the body of  $M$  and then changed the numbers and types of parameters for some of the subroutines in its header file before compiling some user module  $U$ . If both compilations succeed, then  $M$  and  $U$  will have very different notions of how to interpret the parameters passed between them; while they may still link together, chaos is likely to ensue at run time. To prevent this sort of problem, we must ensure whenever  $M$  and  $U$  are linked together that both were compiled using the same version of  $M$ 's header.

In most module-based languages, the following technique suffices. When compiling the body of module  $M$  we create a dummy symbol whose name uniquely characterizes the contents of  $M$ 's header. When compiling the body of  $U$  we create a reference to the dummy symbol. An attempt to link  $M$  and  $U$  together will succeed only if they agree on the name of the symbol.

One way to create the symbol name that characterizes  $M$  is to use an ASCII representation of the time of the most recent modification of  $M$ 's header. Because files may be moved across machines, however (e.g., to deliver source files to geographically distributed customers), modification times are problematic: clocks on different machines are often poorly synchronized, and file copy operations often change the modification time. A better candidate is a *checksum* of the header file: essentially the output of a hash function that uses the entire text of the file

#### EXAMPLE 14.16

Checksumming headers  
for consistency

#### DESIGN & IMPLEMENTATION

##### Type checking for separate compilation

The encoding of type information in symbol names works well in C++ but is too strict for use in C: it would outlaw programming tricks which, while questionable, are permitted by the language definition. Symbol-name encoding is facilitated in C++ by the use of structural equivalence for types. In principle, one could use it in a language with name equivalence, but given that such languages generally have well-structured modules, it is simpler just to use a checksum of the header.

as key. It is possible in theory for two different but valid files to have the same checksum, but with a good choice of hash function the odds of this error are exceedingly small.

The checksum strategy does require that we know when we're using a module header. Unfortunately, as described in Section 3.7, we don't know this in C and C++: headers in these languages are simply a programming convention, supported by the textual inclusion mechanism of the language's preprocessor. Most implementations of C do not enforce consistency of interfaces at link time; instead, programmers rely on configuration management tools (e.g., Unix's `make`) to recompile files when necessary. Such tools are typically driven by file modification times.

Most implementations of C++ adopt a different approach, sometimes called *name mangling*. The name of each imported or exported symbol in an object file is created by concatenating the corresponding name from the program source with a representation of its type. For an object, the type consists of the class name and a terse encoding of its structure. For a function, it consists of an encoding of the types of the arguments and the return value. For complicated objects or functions of many arguments, the resulting names can be very long. If the linker limits symbols to some too-small maximum length, the type information can be compressed by hashing, at some small loss in security [SF88].

One problem with any technique based on file modification times or checksums is that a trivial change to a header file (modification of a comment, for example, or definition of a new constant not needed by existing users of the interface) can prevent files from linking correctly. A similar problem occurs with configuration management tools: a trivial change may cause the tool to recompile files unnecessarily. A few programming environments address this issue by tracking changes at a granularity smaller than compilation units [Tic86]. Most just live with the need to recompile.

## 14.7 Dynamic Linking

On a multiuser system, it is common for several instances of a program (an editor or web browser, for example) to be executing simultaneously. It would be highly wasteful to allocate space in memory for a separate, identical copy of the code of such a program for every running instance. Many operating systems therefore keep track of the programs that are running, and set up memory mapping tables so that all instances of the same program share the same read-only copy of the program's code segment. Each instance receives its own writable copy of the data segment. Code segment sharing can save enormous amounts of space. It does not work, however, for instances of programs that are similar but not identical.

Many sets of programs, while not identical, have large amounts of library code in common, for example, to manage a graphical user interface. If every application has its own copy of the library, then large amounts of memory may be

wasted. Moreover, if programs are statically linked, then much larger amounts of disk space may be wasted on nearly identical copies of the library in separate executable object files.

#### IN MORE DEPTH

In the early 1990s, most operating system vendors adopted *dynamic linking* in order to save space in memory and on disk. Each dynamically linked library resides in its own code and data segments. Every program instance that uses a given library has a private copy of the library's data segment, but shares a single system-wide read-only copy of the library's code segment. These segments may be linked to the remainder of the code when the program is loaded into memory, or they may be linked incrementally on demand, during execution. In addition to saving space, dynamic linking allows a programmer or system administrator to install backward-compatible updates to a library without rebuilding all existing executable object files: the next time it runs, each program will obtain the new version of the library automatically.

#### CHECK YOUR UNDERSTANDING

12. What are the distinguishing characteristics of a *relocatable* object file? An *executable* object file?
13. Why do operating systems typically *zero-fill* pages used for uninitialized data?
14. List four tasks commonly performed by an *assembler*.
15. Summarize the comparative advantages of assembly language and object code as the output of a compiler.
16. Give three examples of *pseudoinstructions* and three examples of *directives* that an assembler might be likely to provide.
17. Why might a RISC assembler perform its own final pass of instruction scheduling?
18. Explain the distinction between *absolute* and *relocatable* words in an object file. Why is the notion of “relocatability” more complicated than it used to be?
19. What is the difference between *linking* and *loading*?
20. What are the principal tasks of a *linker*?
21. How can a linker enforce type checking across compilation units?
22. What is the motivation for *dynamic* linking?

## 14.8 Summary and Concluding Remarks

In this chapter we focused our attention on the back end of the compiler, and on *code generation, assembly, and linking*, in particular.

Compiler back ends vary greatly in internal structure. We discussed one plausible structure, in which semantic analysis is followed by, in order, intermediate code generation, machine-independent code improvement, target code generation, and machine-specific code improvement (including register allocation and instruction scheduling). The semantic analyzer passes a syntax tree to the intermediate code generator, which in turn passes a *control flow graph* to the machine-independent code improver. Within the nodes of the control flow graph, we suggested that code be represented by instructions in a pseudo-assembly language with an unlimited number of *virtual registers*. In order to delay discussion of code improvement to Chapter 15, we also presented a simpler back-end structure in which code improvement is dropped, naive register allocation happens early, and intermediate and target code generation are merged into a single phase. This simpler structure provided the context for our discussion of code generation.

We also discussed intermediate forms (IFs). These can be categorized in terms of their *level*, or degree of machine independence. On the PLP CD we considered two examples: the high-level, tree-based Diana language used by most Ada compilers, and the medium-level Register Transfer Language of the Free Software Foundation GNU compilers. A well-defined IF facilitates the construction of *compiler families*, in which front ends for one or more languages can be paired with back ends for many machines.

Intermediate code generation is typically performed via ad hoc traversal of a syntax tree. Like semantic analysis, the process can be formalized in terms of attribute grammars. We presented part of a small example grammar and used it to generate code for the GCD program introduced in Chapter 1. We noted in passing that target code generation is often automated, in whole or in part, using a *code generator generator* that takes as input a formal description of the target machine and produces code that performs pattern matching on instruction sequences or trees.

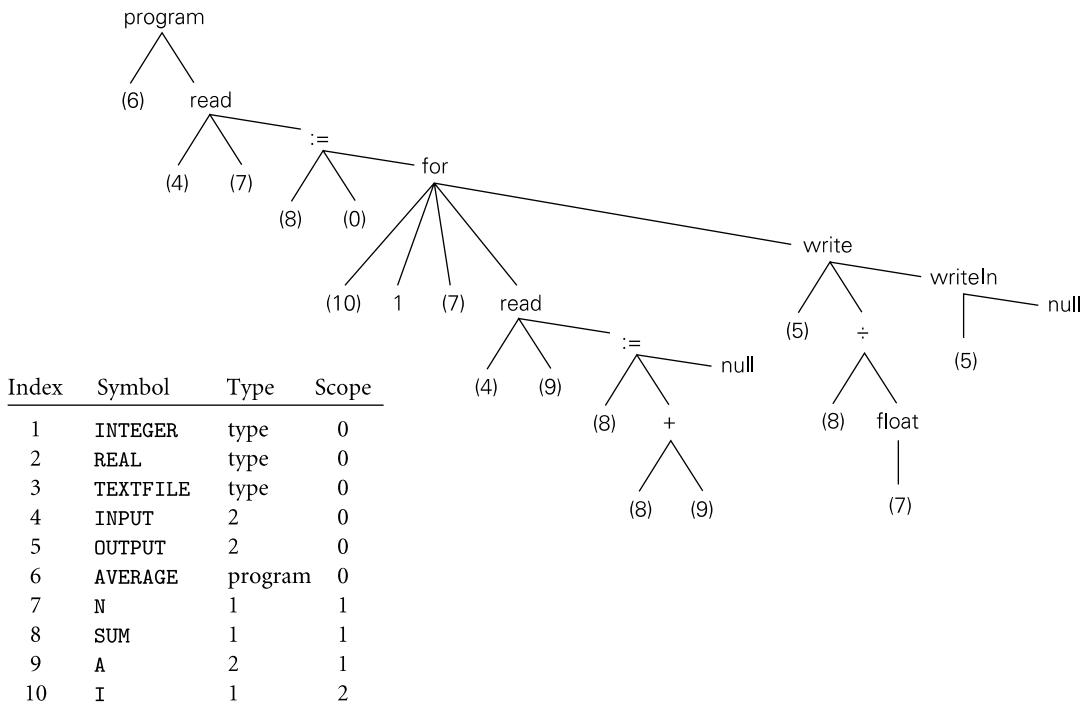
In our discussion of assembly and linking we described the format of *relocatable* and *executable* object files, and discussed the notions of *name resolution* and *relocation*. We noted that while not all compilers include an explicit assembly phase, all compilation systems must make it possible to generate assembly code for debugging purposes, and must allow the programmer to write special purpose routines in assembler. In compilers that use an assembler, the assembly phase is sometimes responsible for instruction scheduling and other low-level code improvement. The linker, for its part, supports separate compilation, by “gluing” together object files produced by multiple compilations. In many modern systems, significant portions of the linking task are delayed until load time or even run time, to allow programs to share the code segments of large, popular libraries. For many languages the linker must perform a certain amount of

semantic checking, to guarantee type consistency. In more aggressive optimizing compilation systems (not discussed in this text), the linker may also perform interprocedural code improvement.

As noted in Section 1.5, the typical programming environment includes a host of additional tools, including debuggers, performance profilers, configuration and version managers, style checkers, preprocessors, pretty-printers, and perusal and cross-referencing utilities. Many of these tools, particularly in well-integrated environments, are directly supported by the compiler. Many make use, for example, of symbol-table information embedded in object files. Performance profilers often rely on special instrumentation code inserted by the compiler at subroutine calls, loop boundaries, and other key points in the code. Perusal, style-checking, and pretty-printing programs may share the compiler’s scanner and parser. Configuration tools often rely on lists of interfile dependences, again generated by the compiler, to tell when a change to one part of a large system may require that other parts be recompiled.

## | 4.9 Exercises

- 14.1 If you were writing a two-pass compiler, why might you choose a high-level IF as the link between the front end and the back end? Why might you choose a medium-level IF?
- 14.2 Consider a language like Ada or Modula-2, in which a module  $M$  can be divided into a specification (header) file and an implementation (body) file for the purpose of separate compilation (Section 9.2.1). Should  $M$ ’s specification itself be separately compiled, or should the compiler simply read it in the process of compiling  $M$ ’s body and the bodies of other modules that use abstractions defined in  $M$ ? If the specification is compiled, what should the output consist of?
- 14.3 Many research compilers (e.g., for SR [AO93], Cedar [SZBH86], Lynx [Sco91], and Modula-3 [Har92]) use C as their IF. C is well documented and mostly machine-independent, and C compilers are much more widely available than alternative back ends. What are the disadvantages of generating C, and how might they be overcome?
- 14.4 List as many ways as you can think of in which the back end of a just-in-time compiler might differ from that of a more conventional compiler. What design goals dictate the differences?
- 14.5 Suppose that  $k$  (the number of temporary registers) in Figure 14.5 is 4 (this is an artificially small number for modern machines). Give an example of an expression that will lead to register spilling under our naive register allocation algorithm.



**Figure 14.8** Syntax tree and symbol table for a program that computes the average of  $N$  real numbers. The children of the for node are the index variable, the lower bound, the upper bound, and the body.

- 14.6 Modify the attribute grammar of Figure 14.5 in such a way that it will generate the control flow graph of Figure 14.3 instead of the linear assembly code of Figure 14.6.
- 14.7 Add productions and attribute rules to the grammar of Figure 14.5 to handle Ada-style for loops (described in Section 6.5.1). Using your modified grammar, hand-translate the syntax tree of Figure 14.8 into pseudo-assembly notation. Keep the index variable and the upper loop bound in registers.
- 14.8 One problem (of many) with the code we generated in Section 14.3 is that it computes at run time the value of expressions that could have been computed at compile time. Modify the grammar of Figure 14.5 to perform a simple form of *constant folding*: whenever both operands of an operator are compile-time constants, we should compute the value at compile time and then generate code that uses the value directly. Be sure to consider how to handle overflow.
- 14.9 Modify the grammar of Figure 14.5 to generate jump code for Boolean expressions, as described in Section 6.4.1. You should assume short-circuit evaluation (Section 6.1.5).

- 14.10 Our GCD program did not employ subroutines. Extend the grammar of Figure 14.5 to handle procedures without parameters (feel free to adopt any reasonable conventions on the structure of the syntax tree). Be sure to generate appropriate prologue and epilogue code for each subroutine, and to save and restore any needed temporary registers.
- 14.11 The grammar of Figure 14.5 assumes that all variables are global. In the presence of subroutines, we would need to generate different code (with fp-relative displacement mode addressing) to access local variables and parameters. In a language with nested scopes we would need to dereference the static chain (or index into the display) to access objects that are neither local nor global. Suppose that we are compiling a language with nested subroutines, and are using a static chain. Modify the grammar of Figure 14.5 to generate code to access objects correctly, regardless of scope.

You may find it useful to define a `to_register` subroutine that generates the code to load a given object. Be sure to consider both l-values and r-values, and parameters passed by both value and result.

© 14.12–14.14 In More Depth.

## 14.10 Explorations

- 14.15 Investigate and describe the IF of the compiler you use most often. Can you instruct the compiler to dump it to a file which you can then inspect? Are there tools other than the back end of the compiler that operate on the IF (e.g., debuggers, code improvers, configuration managers, etc.)? Is the same IF used by compilers for other languages or machines?
- 14.16 Implement Figure 14.5 in your favorite programming language. Define appropriate data structures to represent a syntax tree; then generate code for some sample trees via ad hoc tree traversal.
- 14.17 Augment your solution to the previous exercise to handle various other language features. Several interesting options have been mentioned in earlier exercises. Others include functions, first-class subroutines, case statements, records and `with` statements, arrays (particularly those of dynamic size), and iterators.
- 14.18 Find out what tools are available on your favorite system to inspect the content of object files (on a Unix system, use `nm` or `objdump`). Consider some program consisting of a modest number (three to six, say) of compilation units. Using the appropriate tool, list the imported and exported symbols in each compilation unit. Then link the files together. Draw an address map showing the locations at which the various code and data segments have been placed. Which instructions within the code segments have been changed by relocation?

- 14.19 If you have access to `g++` (the Gnu C++ compiler), or to a C++ compiler based on AT&T’s translator, investigate the encoding of type information in the names of external symbols. See if you can “reverse engineer” the algorithm used to generate the funny characters at the end of every name.
- © 14.20–14.23 In More Depth.

## 14.11 Bibliographic Notes

Standard compiler textbooks (e.g., those by Cooper and Torczon [CT04], Grune et al. [GBJL01], Appel [App97], Aho, Sethi, and Ullman [ASU86], or Fischer and LeBlanc [FL88]) are an accessible source of information on back-end compiler technology, though the last two have grown a bit dated. More detailed information can be found in the text of Muchnick [Muc97]. Fraser and Hanson provide a wealth of detail on code generation and (simple) code improvement in their `lcc` compiler [FH95].

The Diana intermediate form is documented by Goos, Wulf, Evans, and Butler [GWEB83]. A simpler tree-based IF is described by Fraser and Hanson [FH95, Chap. 5]. RTL is documented in a set of `texinfo` files distributed with `gcc` (available from [www.gnu.org/software](http://www.gnu.org/software)). Java byte code is documented by Lindholm and Yellin [LY97].

Ganapathi, Fischer, and Hennessy provide an early survey of automatic code generator generators [GFH82]. A later and more comprehensive survey is that of Henry and Damron [HD89]. The most widely used automatic code generation technique is based on LR parsing, and is due to Glanville and Graham [GG78].

Sources of information on assemblers, linkers, and software development tools include the texts of Beck [Bec97] and of Kernighan and Plauger [KP76]. Gingell et al. describe the implementation of shared libraries for the Sparc architecture and the SunOS variant of Unix [GLDW87]. Ho and Olsson describe a particularly ambitious dynamic linker for Unix [HO91]. Tichy presents a compilation system that avoids unnecessary recompilations by tracking dependences at a granularity finer than the source file [Tic86].

# 15

## Code Improvement

In Chapter 14 we discussed the **generation**, assembly, and linking of target code in the back end of a compiler. The techniques we presented led to correct but highly suboptimal code: there were many redundant computations, and inefficient use of the registers, multiple functional units, and cache of a modern microprocessor. This chapter takes a look at *code improvement*: the phases of compilation devoted to generating *good* (fast) code. As noted in Section 1.6.4, code improvement is often referred to as *optimization*, though it seldom makes anything optimal in any absolute sense.

Our study will consider simple *peephole* optimization, which “cleans up” generated target code within a very small instruction window; *local* optimization, which generates near-optimal code for individual basic blocks; and *global* optimization, which performs more aggressive code improvement at the level of entire subroutines. We will not cover interprocedural improvement; interested readers are referred to other texts (see the Bibliographic Notes at the end of the chapter). Moreover, even for the subjects we cover, our intent will be more to “demystify” code improvement than to describe the process in detail. Much of the discussion will revolve around the successive refinement of code for a single subroutine. This extended example will allow us to illustrate the effect of several key forms of code improvement without dwelling on the details of how they are achieved.

### IN MORE DEPTH

Chapter 15 can be found in its entirety on the PLP CD.



# Programming Languages Mentioned

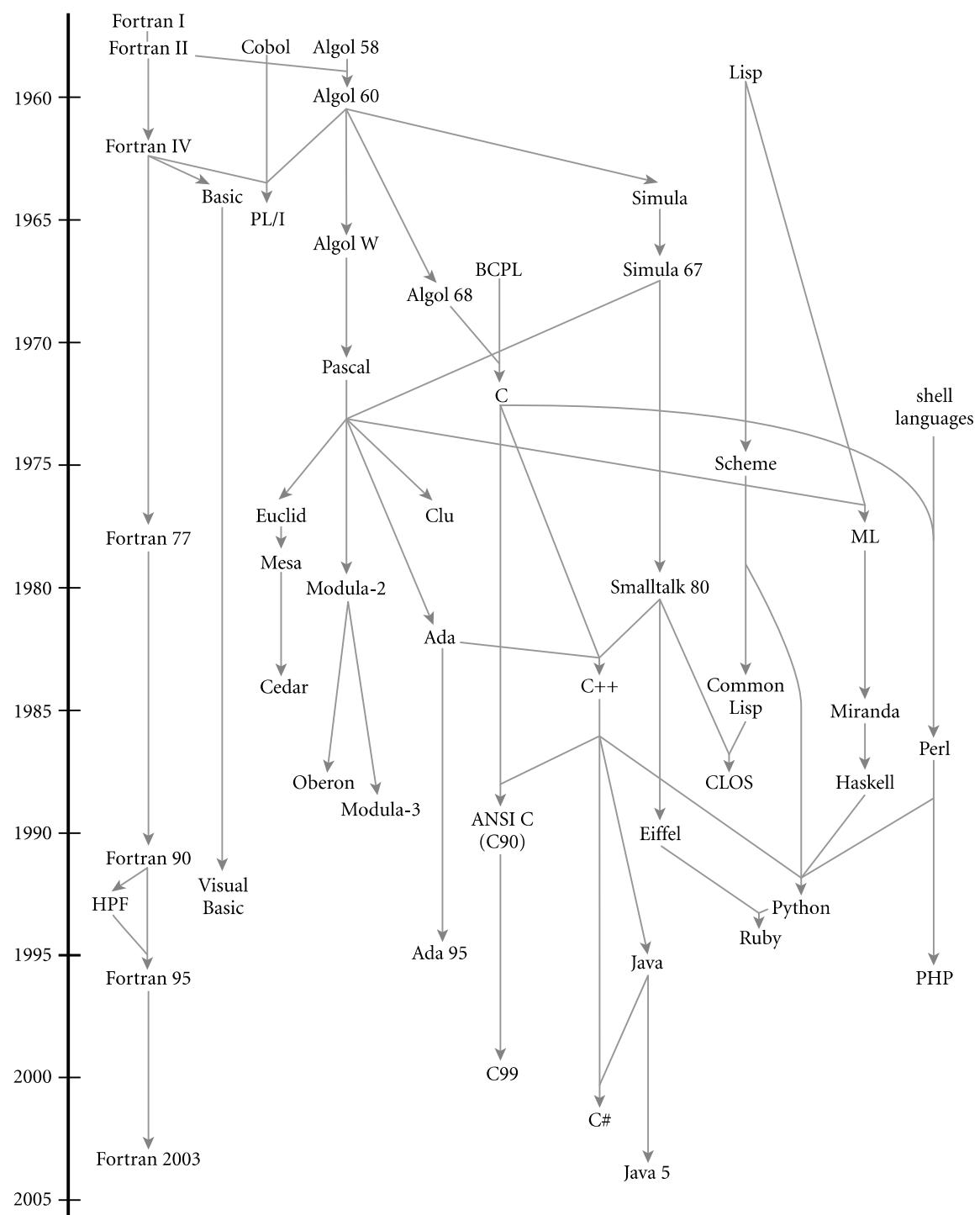


**This appendix provides brief descriptions,** bibliographic references, and (in many cases) URLs for online information concerning each of the principal programming languages mentioned in this book. The URLs are accurate as of May 2005, though they are subject to change as people move files around. Some additional URLs can be found in the bibliographic references.

For many languages XXX, there exists an Internet newsgroup *comp.lang.XXX*. Many of these newsgroups host frequently-asked-question (FAQ) lists. Bill Kinnersley maintains an extremely useful index of online materials for approximately 2500 programming languages at [people.ku.edu/~nkinners/LangList/Extras/langlist.htm](http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm). Other resources include the Google and Yahoo language indices ([directory.google.com/Top/Computers/Programming/Languages/](http://directory.google.com/Top/Computers/Programming/Languages/) and [dir.yahoo.com/Computers\\_and\\_Internet/Programming\\_and\\_Development/Languages/](http://dir.yahoo.com/Computers_and_Internet/Programming_and_Development/Languages/)), and the Open Directory and HyperNews languages lists ([www.dmoz.org/Computers/Programming/Languages/](http://www.dmoz.org/Computers/Programming/Languages/) and [www.hypernews.org/HyperNews/get/computing/lang-list.html](http://www.hypernews.org/HyperNews/get/computing/lang-list.html)).

Figure A.1 shows the genealogy of some of the more influential or widely used programming languages. The date for each language indicates the approximate time at which its features became widely known. Arrows indicate principal influences on design. Many influences, of course, cannot be shown in a single figure.

**Ada:** Originally intended to be the standard language for all software commissioned by the U.S. Department of Defense [Ame83]. Prototypes designed by teams at several sites; final '83 language developed by a team at Honeywell's Systems and Research Center in Minneapolis and Alsys Corp. in France, led by Jean Ichbiah. A very large language, descended largely from Pascal. Design rationale articulated in a remarkably clear companion document [IBFW91]. Ada 95 [Int95b] is a revision developed under government contract by a team at Intermetrics, Inc. It fixes several subtle problems in the earlier language, and adds objects, shared-memory synchronization, and several other features. Freely available implementation distributed by Ada Core Technolo-



**Figure A.1** Genealogy of selected programming languages. Dates are approximate.

gies ([www.gnat.com/](http://www.gnat.com/)) under terms of the Free Software Foundation’s GNU public license.

**Algol 60:** The original block-structured language. The original definition by Naur et al. [NBB<sup>+</sup>63] is considered a landmark of clarity and conciseness. It includes the original use of Backus-Naur Form (BNF).

**Algol 68:** A large and relatively complex successor to Algol 60, designed by a committee led by A. van Wijngaarden. Includes (among other things) structures and unions, expression-based syntax, reference parameters, a reference model of variables, and concurrency. The official definition [vMP<sup>+</sup>75] uses unconventional terminology and is very difficult to read; other sources [Pag76, Lv77] are more accessible.

**Algol W:** A smaller, simpler alternative to Algol 68, proposed by Niklaus Wirth and C. A. R. Hoare [WH66, Sit72]. The precursor to Pascal. Introduced the case statement.

**APL:** Designed by Kenneth Iverson in the late 1950s and early 1960s, primarily for the manipulation of numeric arrays. Functional. Extremely concise. Powerful set of operators. Employs an extended character set. Intended for interactive use. Original syntax [Ive62] was nonlinear; implementations generally use a revised syntax due to a team at IBM [IBM87]. Online resources at [www.acm.org/sigs/sigapl/](http://www.acm.org/sigs/sigapl/).

**Basic:** Simple imperative language, originally intended for interactive use. Original version developed by John Kemeny and Thomas Kurtz of Dartmouth College in the early 1960s. Dozens of dialects exist. Microsoft’s Visual Basic [Mic91], which bears little resemblance to the original, is the most widely used today. Minimal subset defined by ANSI standard [Ame78b].

**C:** One of the most successful imperative languages. Originally defined by Brian Kernighan and Dennis Ritchie of Bell Labs as part of the development of Unix [KR88]. Concise syntax. Unusual declaration syntax. Intended for systems programming. Weak type checking. No dynamic semantic checks. Standardized by ANSI/ISO in 1990 [Ame90]. Extensions for international character sets adopted in 1994. More extensive changes adopted in 1999 (the C99 standard) [Int99]. Freely available implementation (gcc) distributed for many platforms by the Free Software Foundation ([www.gnu.org/software/gcc/](http://www.gnu.org/software/gcc/)).

**C#:** Object-oriented language based heavily on C++ and Java. Designed by Anders Hejlsberg, Scott Wiltamuth, and associates at Microsoft Corporation in the late 1990s and early 2000s [HWG04, ECM02]. Intended as the principal language for the .NET platform, a run-time and middleware system for multilanguage distributed computing. Regarded by many as Microsoft’s alternative to Java. Includes most of Java’s features, plus many from C++ and Visual Basic, including both reference and value types, both contiguous and row-pointer arrays, both virtual and nonvirtual methods, operator overloading, delegates, and an “unsafe” superset with pointers. Standardized by ECMA/ISO in 2002 [ECM02]. Commercial resources at [msdn.microsoft.com/vcsharp/](http://msdn.microsoft.com/vcsharp/).

Open source implementations available from [www.gnu.org/projects/dotgnu/pnet.html](http://www.gnu.org/projects/dotgnu/pnet.html) and [www.go-mono.com/c-sharp.html](http://www.go-mono.com/c-sharp.html).

**C++:** The first object-oriented successor to C to gain widespread adoption. Still widely considered the one most suited to “industrial strength” computing. Designed by Bjarne Stroustrup of Bell Labs. Includes (among other things) generalized reference types, both static and dynamic method binding, extensive facilities for overloading and coercion, and multiple inheritance. No automatic garbage collection. Useful references include Stroustrup’s text [Str97] and the reference manual of Ellis and Stroustrup [ES90]. Standardized by the ISO [Int98]. Freely available implementation included in the `gcc` distribution (see C).

**Cedar:** See Mesa and Cedar.

**CLOS:** The Common Lisp Object System [Kee89; Ste90, Chap. 28]. A set of object-oriented extensions to Common Lisp, now incorporated into the ANSI standard language (see Common Lisp). The leading notation for object-oriented functional programming.

**Clu:** Developed by Barbara Liskov and associates at MIT in the late 1970s [LG86]. Designed to provide an unusually powerful set of features for data abstraction [LSAS77]. Also includes iterators and exception handling. Freely available implementations for most Unix platforms at <ftp://ftp.lcs.mit.edu/pub/pclu>.

**Cobol:** Originally developed by the U.S. Department of Defense in the late 1950s and early 1960s by a team led by Grace Murray Hopper [Uni60]. Long the most widely used programming language in the world. Standardized by ANSI in 1968; revised in 1974 and 1985 [Ame85]. Intended principally for business data processing. Introduced the concept of structures. Elaborate I/O facilities.

**Common Lisp:** The standard modern Lisp (see also Lisp). A large language. Includes (among other things) static scoping, an extensive type system, exception handling, and object-oriented features (see CLOS). For years the standard reference was the book by Guy Steele, Jr. [Ste90]. Subsequently standardized by ANSI [Ame96b]. An abridged hypertext version of the standard is available online at [www.lispworks.com/documentation/HyperSpec/Front/index.htm](http://www.lispworks.com/documentation/HyperSpec/Front/index.htm).

**CSP:** See Occam.

**Eiffel:** An object-oriented language developed by Bertrand Meyer and associates at the Société des Outils du Logiciel à Paris [Mey92]. Includes (among other things) multiple inheritance, automatic garbage collection, and powerful mechanisms for renaming of data members and methods in derived classes. Online resources at [www.eiffel.com/](http://www.eiffel.com/).

**Euclid:** Imperative language developed by Butler Lampson and associates at the Xerox Palo Alto Research Center in the mid-1970s [LHL<sup>+</sup>77]. Designed to eliminate many of the sources of common programming errors in Pascal, and

to facilitate formal verification of programs. Has closed scopes and module types.

**Forth:** A small and rather ingenious stack-based language designed for interpretation on machines with limited resources [Bro87, Int97]. Originally developed by Charles H. Moore in the late 1960s. Has a loyal following in the instrumentation and process-control communities.

**Fortran:** The original high-level imperative language. Developed in the mid-1950s by John Backus and associates at IBM. The most important versions are Fortran I, Fortran II, Fortran IV, Fortran 77, and Fortran 90. The latter two are documented in a pair of ANSI standards [Ame78a, Ame92]. Fortran 90 [MR96] (updated in 1995) is a major revision to the language, adding (among other things) recursion, pointers, new control constructs, and a wealth of array operations. Fortran 2003 [Int03b] adds object orientation. Fortran 77, however, is still very widely used. Freely available implementation distributed as part of the `gcc` compiler suite ([www.gnu.org/software/gcc/fortran/](http://www.gnu.org/software/gcc/fortran/)). Support for the older `g77` front end was discontinued as of `gcc` version 3.4.

**Haskell:** The leading purely functional language. Descended from Miranda. Designed by a committee of researchers beginning in 1987. Includes curried functions, higher-order functions, nonstrict semantics, static polymorphic typing, pattern matching, list comprehensions, modules, monadic I/O, and layout (indentation)-based syntactic grouping. Haskell 98 [Pey03] is the most recent as of this writing; design of Haskell 2 is under way. Online resources at [haskell.org/](http://haskell.org/).

**Icon:** The successor to Snobol. Developed by Ralph Griswold (Snobol's principal designer) at the University of Arizona [GG96]. Adopts more conventional control-flow constructs, but with powerful iteration and search facilities based on pattern-matching and backtracking. Online resources at [www.cs.arizona.edu/icon/](http://www.cs.arizona.edu/icon/).

**Java:** Object-oriented language based largely on a subset of C++. Developed by James Gosling and associates at Sun Microsystems in the early 1990s [AG98, GJS96]. Intended for the construction of highly portable, architecture-neutral programs. Defined in conjunction with an intermediate *byte code* format intended for execution on a Java *virtual machine* [LY97]. Includes (among other things) a reference model of (class-typed) variables, mix-in inheritance, threads, and extensive predefined libraries for graphics, communication, and other activities. Heavily used for transmission of program fragments (*applets*) over the Internet. Online resources at [www.sun.com/java/](http://www.sun.com/java/).

**JavaScript:** Simple scripting language developed by Brendan Eich at Netscape Corp. in the mid-1990s for the purpose of client-side web scripting. Has no connection to Java beyond superficial syntactic similarity. Embedded in most commercial web browsers. Microsoft's JScript is very similar. The two were merged into a single ECMA standard [ECM99] in 1997 (since revised).

**Linda:** A set of language extensions intended to add concurrency to conventional programming languages [ACG86]. Developed by David Gelernter for his doctoral research at SUNY Stony Brook in the early 1980s and later refined by Gelernter and his student, Nicholas Carriero, at Yale University. Based on the notion of a distributed, associative *tuple space*. Has inspired numerous implementations, including Sun’s JavaSpaces [FHA99] and IBM’s TSpaces ([www.almaden.ibm.com/cs/TSpaces/](http://www.almaden.ibm.com/cs/TSpaces/)).

**Lisp:** The original functional language [McC60]. Developed by John McCarthy in the late 1950s as a realization of Church’s lambda calculus. Many dialects exist. The two most common today are Common Lisp and Scheme (see separate entries). Historically important dialects include Lisp 1.5 [MAE<sup>+</sup>65], MacLisp [Moo78], and Interlisp [TM81].

**Mesa and Cedar:** Mesa [LR80] is a successor to Euclid developed in the 1970s at Xerox’s Palo Alto Research Center by a team led by Butler Lampson. Includes monitor-based concurrency. Along with Interlisp and Smalltalk, one of three companion projects that pioneered the use of personal workstations, with bitmapped displays, mice, and a graphical user interface. Cedar [Szbh86] is a successor to Mesa with (among other things) complete type safety, exceptions, and automatic garbage collection.

**Miranda:** Purely functional language designed by David Turner in the mid-1980s [Tur86]. Resembles ML in several respects; has type inference and automatic currying. Unlike ML, provides list comprehensions (Section 7.8), and uses lazy evaluation for all arguments. Uses indentation and line breaks for syntactic grouping. Commercial implementations available from Research Software Ltd. of Canterbury, England.

**ML:** Functional language with “Pascal-like” syntax. Originally designed in the mid- to late 1970s by Robin Milner and associates at the University of Edinburgh as the meta-language (hence the name) for a program verification system. Pioneered aggressive compile-time type inference and polymorphism. Has a few imperative features. Several dialects exist; the most widely used is Standard ML [MTHM97]. Stansifer’s book [Sta92] is an accessible introduction. Standard ML of New Jersey, a project of Princeton University and Bell Labs, has produced freely available implementations for many platforms; see [www.smlnj.org/](http://www.smlnj.org/).

**Modula and Modula-2:** The immediate successors to Pascal, developed by Niklaus Wirth. The original Modula [Wir77b] was an explicitly concurrent monitor-based language. It is sometimes called Modula (1) to distinguish it from its successors. The more commercially important Modula-2 [Wir85b] was originally designed with coroutines (Section 8.6), but no real concurrency. Both languages provide mechanisms for module-as-manager style data abstractions. Modula-2 was standardized by the ISO in 1996 [Int96]. A freely available implementation for x86 Linux (and moderately priced implementations for several other Unix variants) is available from the University of Karlsruhe, Germany at [www.info.uni-karlsruhe.de/~modula/](http://www.info.uni-karlsruhe.de/~modula/).

**Modula-3:** A major extension to Modula-2 developed by Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson at the Digital Systems Research Center and the Olivetti Research Center in the late 1980s [Har92]. Intended to provide a level of support for large, reliable, and maintainable systems comparable to that of Ada, but in a simpler and more elegant form. Online resources at [research.compaq.com/SRC/modula-3/html/](http://research.compaq.com/SRC/modula-3/html/).

**Oberon:** A deliberately minimal language designed by Niklaus Wirth [Wir88b, RW92]. Essentially a subset of Modula-2 [Wir88a], augmented with a mechanism for type extension (Section 9.2.3) [Wir88c]. Online resources at [www.oberon.ethz.ch/](http://www.oberon.ethz.ch/).

**Objective-C:** An object-oriented extension to C based on Smalltalk-style “messaging.” Designed by Brad Cox and StepStone corporation in the early 1980s. Adopted by NeXT Software, Inc. in the late 1980s for their NEXTStep operating system and programming environment. Adopted by Apple as the principal development language for MacOS X after Apple acquired NeXT in 1997. Substantially simpler than other object-oriented descendants of C. Distinguished by fully dynamic method dispatch and unusual messaging syntax. Freely available implementation included in the gcc distribution (see C). Online documentation at [developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/](http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/).

**Occam:** A concurrent language [JG89] based on CSP [Hoa78], Hoare’s notation for message-based communication using guarded commands and synchronization send. The language of choice for systems built from INMOS Corporation’s *transputer* processors, once widely used in Europe. Uses indentation and line breaks for syntactic grouping. Online resources at [wotug.kent.ac.uk/parallel/occam/](http://wotug.kent.ac.uk/parallel/occam/).

**Pascal:** Designed by Niklaus Wirth in the late 1960s [Wir71], largely in reaction to Algol 68, which was widely perceived as bloated. Heavily used in the 1970s and 1980s, particularly for teaching. Introduced subrange and enumeration types. Unified structures and unions. For many years the standard reference was Wirth’s book with Kathleen Jensen [JW91]; more recently, the language has been standardized by the ISO and ANSI [Int90]. Freely available implementation distributed by the Free Software Foundation ([directory.fsf.org-devel/prog/other/pascal/](http://directory.fsf.org-devel/prog/other/pascal/)).

**Perl:** A general purpose scripting language designed by Larry Wall in the late 1980s [WCO00]. Includes unusually extensive mechanisms for character string manipulation and pattern matching based on (extended) regular expressions. Borrows features from C, sed and awk [AKW88] (two earlier scripting languages), and various Unix *shell* (command interpreter) languages. Is famous/infamous for having multiple ways of doing almost anything. Enjoyed an upsurge in popularity in the late 1990s as a server-side web scripting language. Version 5 released in 1995; version 6 currently under development. Online resources at [www.perl.org/](http://www.perl.org/). Larry Wall’s own Perl page is at [www.wall.org/~larry/perl.html](http://www.wall.org/~larry/perl.html).

**PHP:** A descendant of Perl designed for server-side web scripting. Scripts are typically embedded in web pages. Originally created by Rasmus Lerdorf in 1995 to help manage his personal home page. The name is now officially a recursive acronym (PHP: Hypertext Preprocessor). More recent versions due to Andi Gutmans and Zeev Suraski, in cooperation with Lerdorf. Includes built-in support for a wide range of Internet protocols and for access to dozens of different commercial database systems. Version 5 (released in September 2004) includes extensive object-oriented features, mix-in inheritance, iterator objects, autoloading, structured exception handling, reflection, overloading, and optional type declarations for parameters. Online resources at [www.php.net/](http://www.php.net/).

**PL/I:** A large, general purpose language designed in the mid-1960s as a successor to Fortran, Cobol, and Algol [Bee70]. Never managed to displace its predecessors; kept alive largely through IBM corporate influence.

**Postscript:** A stack-based language for the description of graphics and print operations [Ado86, Ado90]. Developed and marketed by Adobe Systems, Inc. Based in part on the Forth programming language [Bro87]. Generated by many word processors and drawing programs. Most professional-quality printers contain a Postscript interpreter.

**Prolog:** The most widely used logic programming language. Developed in the early 1970s by Alain Colmerauer and Philippe Roussel of the University of Aix-Marseille in France and Robert Kowalski and associates at the University of Edinburgh in Scotland. Many dialects exist. Partially standardized in 1995 [Int95c]. Numerous implementations, both free and commercial, are available. The AI group at CMU maintains a large Prolog repository at [www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/lang/prolog/0.html](http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/lang/prolog/0.html); additional information can be found at [vl.fmfnet.info/logic-prog/#Prolog](http://vl.fmfnet.info/logic-prog/#Prolog).

**Python:** A general purpose, object-oriented scripting language designed by Guido van Rossum in the early 1990s. Uses indentation for syntactic grouping. Includes dynamic typing, nested functions with lexical scoping, lambda expressions and higher-order functions, true iterators, list comprehensions, array slices, reflection, structured exception handling, multiple inheritance, and modules and dynamic loading. Online resources at [www.python.org/](http://www.python.org/).

**R:** Open source scripting language intended primarily for statistical analysis. Based on the proprietary S statistical programming language, originally developed by John Chambers and others at Bell Labs. Supports first class and higher-order functions, unlimited extent, call-by-need, multidimensional arrays and slices, and an extensive library of statistical functions. Online resources at [www.r-project.org/](http://www.r-project.org/).

**Ruby:** An elegant, general purpose, object-oriented scripting language designed by Yukihiko “Matz” Matsumoto, beginning in 1993. First released in 1995. Inspired by Ada, Eiffel, and Perl, with traces of Python, Lisp, Clu, and Smalltalk. Includes dynamic typing, arbitrary precision arithmetic, true iterators, user-level threads, first-class and higher-order functions, continuations,

reflection, Smalltalk-style messaging, mix-in inheritance, autoloading, structured exception handling, and support for the Tk windowing toolkit. The text by Thomas and Hunt is a standard reference [TH04]. Online resources at [ruby-lang.org/en/](http://ruby-lang.org/en/).

**Scheme:** A small, elegant dialect of Lisp (see also Lisp) developed in the mid-1970s by Guy Steele and Gerald Sussman. Has static scoping and true first-class functions. Widely used for teaching. Current standard is “R5RS” [ADH<sup>+</sup>98]; R6RS is currently under development. Earlier version standardized by the IEEE and ANSI [Ins91]. The book by Abelson and Sussman [AS96], used for introductory programming classes at MIT and elsewhere, is a classic guide to fundamental programming concepts, and to functional programming in particular. Online resources at [www.schemers.org/](http://www.schemers.org/).

**Simula:** Designed at the Norwegian Computing Centre, Oslo, in the mid-1960s by Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard [BDMN73, ND78]. Extends Algol 60 with *classes* and *coroutines*. The name of the language reflects its suitability for discrete-event simulation (Section 8.6.4). Free Simula-to-C translator available at [www.ifi.uio.no/~cim/cim.html](http://www.ifi.uio.no/~cim/cim.html).

**Sisal:** A functional language with “imperative-style” syntax. Developed by James McGraw and associates at Lawrence Livermore National Laboratory in the early to mid-1980s [MSA<sup>+</sup>85, FCO90, Can92]. Intended primarily for high-performance scientific computing, with automatic parallelization. A descendant of the dataflow language Val [McG82]. No longer under development at LLNL; available open source from [sisal.sourceforge.net/](http://sisal.sourceforge.net/).

**Smalltalk:** The quintessential object-oriented language. Developed by Alan Kay, Adele Goldberg, Dan Ingalls, and associates at the Xerox Palo Alto Research Center throughout the 1970s, culminating in the Smalltalk-80 language [GR89]. Anthropomorphic programming model based on “messages” between active objects. The Smalltalk group at the University of Illinois maintains a variety of resources at [st-www.cs.uiuc.edu/](http://st-www.cs.uiuc.edu/). Online resources at [www.smalltalk.org](http://www.smalltalk.org).

**Snobol:** Developed by Ralph Griswold and associates at Bell Labs in the 1960s [GPP71]. The principal version is SNOBOL4. Intended primarily for processing character strings. Includes an extremely rich set of string-manipulating primitives and a novel control-flow mechanism based on the notions of *success* and *failure*. Online resources at [ftp://ftp.cs.arizona.edu/snobol/](http://ftp.cs.arizona.edu/snobol/) and [www.snobol4.org](http://www.snobol4.org).

**SR:** Concurrent programming language developed by Greg Andrews and colleagues at the University of Arizona in the 1980s [AO93]. Integrates not only sequential and concurrent programming but also shared memory, semaphores, message passing, remote procedures, and rendezvous into a single conceptual framework and simple syntax. Online resources at [ftp://ftp.cs.arizona.edu/sr/](http://ftp.cs.arizona.edu/sr/).

**Tcl/Tk:** Tcl (Tool command language, pronounced “tickle”) is a scripting language designed by John Ousterhout in the late 1980s [Ous94, WJH03]. Keyword-based syntax resembles Unix command-line invocations and switches; punctuation is relatively spare. Uses dynamic scoping. Supports reflection, recursive invocation of interpreter. Tk (pronounced “tee-kay”) is a set of Tcl commands for graphical user interface (GUI) programming. Designed by Ousterhout as an extension to Tcl, Tk has also been embedded in Ruby, Perl, and several other languages. Online resources at [www.tcl.tk/](http://www.tcl.tk/).

**Turing:** Derived from Euclid by Richard Holt and associates at the University of Toronto in the early 1980s [HMRC88]. Originally intended as a pedagogical language, but can be used for a wide range of applications. Turing Plus and Object-Oriented Turing are more recent descendants, also developed by Holt’s group. Online resources at [www.holtsoft.com/turing](http://www.holtsoft.com/turing).

**XSL:** The Extensible Stylesheet Language, standardized by the World Wide Web Consortium. Serves as the standard stylesheet language for XML (Extensible Markup Language), the increasingly ubiquitous standard for self-descriptive tree-structured data, of which XHTML, the successor to HTML, is a dialect. XSL includes three substandards: XSLT (XSL Transformations) [Wor04b], which specifies how to translate from one dialect of XML to another; XPath [Wor04a], used to name elements of an XML document; and XSL-FO (XSL Formatting Objects) [Wor01], which specifies how to format documents. XSLT, though highly specialized to the transformation of XML, is a Turing complete programming language [Kep04]. Standards and additional resources at [www.w3.org/](http://www.w3.org/).



# Language Design and Language Implementation

Throughout this text we have had occasion to remark on the many connections between language design and language implementation. Some of the more direct connections have been highlighted in separate sidebars. We list those sidebars here.

## Chapter I: Introduction

1	Introduction	7
2	Compiled and interpreted languages	15
3	The early success of Pascal	19
4	Powerful development environments	21

## Chapter 2: Programming Language Syntax

5	Formatting restrictions	40
6	Nested comments	47
7	Longest possible tokens	57
8	The dangling <code>else</code>	79

## Chapter 3: Names, Scopes, and Bindings

9	Binding time	105
10	Recursion in Fortran	109
11	Mutual recursion	120
12	Redeclarations	123
13	Dynamic scoping	133
14	Binding rules and extent	141
15	Pointers in C and Fortran	142
16	Coercion and overloading	146
17	Generics as macros	147
18	Separate compilation	CD 34

**Chapter 4: Semantic Analysis**

19	Dynamic semantic checks	164
20	Forward references	174
21	Attribute evaluators	179

**Chapter 5: Target Machine Architecture**

22	The processor/memory gap	198
23	How much is a megabyte?	CD 54
24	In-line subroutines	220

**Chapter 6: Control Flow**

25	Implementing the reference model	240
26	Safety v. performance	248
27	Evaluation order	251
28	Cleaning up continuations	259
29	Short-circuit evaluation	264
30	<b>Case</b> statements	268
31	Numerical imprecision	272
32	<b>For</b> loops	276
33	“True” iterators and iterator objects	280
34	<b>Inline</b> as a hint	292
35	Normal-order evaluation	294
36	Nondeterminacy and fairness	CD 76

**Chapter 7: Data Types**

37	Dynamic typing	310
38	Multilingual character sets	313
39	Decimal types	314
40	Multiple sizes of integers	317
41	Nonconverting casts	328
42	Unification	CD 83
43	The order of record fields	340
44	<b>With</b> statements	CD 91
45	The placement of variant fields	347
46	Is [] an operator?	351
47	Array layout	360
48	Lower bounds on array indices	363
49	Representing sets	368
50	Implementation of pointers	369
51	Pointers and arrays	377
52	Garbage collection	383

53 Reference counts v. tracing	388
54 <code>Car</code> and <code>cdr</code>	391
55 I/O	392

**Chapter 8: Subroutines and Control Abstraction**

56 Lexical nesting and displays	CD 107
57 Executing code in the stack	CD 117
58 Hints and directives	415
59 <code>Inline</code> and modularity	416
60 Parameter modes	419
61 Anonymous delegates in C# 2.0	425
62 Call by name	CD 123
63 Call by need	CD 124
64 Why erasure?	CD 131
65 Structured exceptions	447
66 <code>Setjmp</code>	451
67 Threads and coroutines	454
68 Coroutine stacks	457

**Chapter 9: Data Abstraction and Object Orientation**

69 What goes in a class declaration?	475
70 Opaque exports in Modula-2	482
71 The value/reference tradeoff	492
72 Initialization and assignment	493
73 Initialization of "expanded" objects	494
74 Reverse assignment	504
75 The fragile base class problem	504
76 Generics and dynamic method dispatch	507
77 The cost of multiple inheritance	CD 149

**Chapter 10: Functional Languages**

78 Iteration in functional programs	534
79 Lazy evaluation	541
80 Monads	544
81 Higher-order functions	547
82 Side effects and compilation	550

**Chapter 11: Logic Languages**

83 Homoiiconic languages	575
84 Reflection	578
85 Implementing logic	580
86 Alternative search strategies	580

**Chapter 12: Concurrency**

87	Hardware and software communication	602
88	Stack frames for nested threads	606
89	Counterintuitive implementation	612
90	Monitor signal semantics	632
91	The nested monitor problem	633
92	Conditional critical regions	635
93	Condition variables in Java	637
94	Side-effect freedom and implicit synchronization	640
95	The semantic impact of implementation issues	647
96	Emulation and efficiency	650
97	Peeking inside messages	655
98	Parameters to remote procedures	658

**Chapter 13: Scripting Languages**

99	Scripting on Microsoft platforms	673
100	Compiling interpreted languages	675
101	Canonical implementations	675
102	Built-in commands in the shell	680
103	Magic numbers	683
104	JavaScript and Java	710
105	Sandboxing	711
106	Thinking about dynamic scope	727
107	Automata for regular expressions	729
108	The <code>grep</code> command and the birth of Unix tools	729
109	Compiling regular expressions	735
110	Typeglobs in Perl	739
111	Executable class declarations	747
112	Worse Is Better	749

**Chapter 14: Building a Runnable Program**

113	Stack-based IFs	767
114	Postscript	767
115	Type checking for separate compilation	783

**Chapter 15: Code Improvement**

116	Peephole optimization	CD 208
117	Basic blocks	CD 209
118	Common subexpressions	CD 215
119	Pointer analysis	CD 216
120	Loop invariants	CD 228
121	Control flow analysis	CD 228

# Numbered Examples

<b>Chapter 1: Introduction</b>		
I.1 GCD program in MIPS machine language	3	
I.2 GCD program in MIPS assembler	3	
		I.20 GCD program parse tree
		I.21 GCD program abstract syntax tree
		I.22 GCD program assembly code
		I.23 GCD program optimization
<b>The Art of Language Design</b>		25
<b>The Programming Language Spectrum</b>		27
I.3 Classification of programming languages	8	28
		30
<b>Why Study Programming Languages?</b>		
<b>Compilation and Interpretation</b>		
I.4 Pure compilation	13	
I.5 Pure interpretation	14	
I.6 Mixing compilation and interpretation	14	
I.7 Preprocessing	15	
I.8 Library routines and linking	16	
I.9 Post-compilation assembly	16	
I.10 The C preprocessor	17	
I.11 Source-to-source translation (C++)	17	
I.12 Bootstrapping	18	
I.13 Compiling interpreted languages	20	
I.14 Dynamic and just-in-time compilation	20	
I.15 Microcode (firmware)	20	
<b>Programming Environments</b>		
<b>An Overview of Compilation</b>		
I.16 Phases of compilation	22	
I.17 GCD program in Pascal	23	
I.18 GCD program tokens	24	
I.19 Context-free grammar and parsing	24	
		2.1 Syntax of Arabic numerals
		2.2 Syntax of numbers in Pascal
		2.3 Syntactic nesting in expressions
		2.4 Extended BNF (EBNF)
		2.5 Derivation of <code>slope * x + intercept</code>
		2.6 Parse trees for <code>slope * x + intercept</code>
		2.7 Expression grammar with precedence and associativity
		2.8 Outline of a scanner for Pascal
		2.9 Finite automaton for part of a Pascal scanner
		2.10 Constructing an NFA for a given regular expression
		2.11 NFA for $(1^*01^*0)^*1^*$
		2.12 DFA for $(1^*01^*0)^*1^*$
		2.13 Minimal DFA for $(1^*01^*0)^*1^*$
		2.14 Nested <code>case</code> statement automaton
		2.15 The “dot-dot problem” in Pascal
		2.16 Look-ahead in Fortran scanning
		2.17 Table-driven scanning

**Parsing**

2.18	Top-down and bottom-up parsing	62
2.19	Bounding space with a bottom-up grammar	64
2.20	Top-down grammar for a calculator language	64
2.21	Recursive descent parser for the calculator language	66
2.22	Recursive descent parse of a "sum and average" program	66
2.23	Driver and table for top-down parsing	70
2.24	Table-driven parse of the "sum and average" program	71
2.25	Predict sets for the calculator language	72
2.26	Left recursion	77
2.27	Common prefixes	77
2.28	Eliminating left recursion	77
2.29	Left factoring	77
2.30	Parsing a "dangling else"	78
2.31	"Dangling else" program bug	78
2.32	End markers for structured statements	79
2.33	The need for <code>elsif</code>	79
2.34	Derivation of an <code>id</code> list	81
2.35	Bottom-up grammar for the calculator language	81
2.36	Bottom-up parse of the "sum and average" program	82
2.37	CFSM for the bottom-up calculator grammar	86
2.38	Epsilon productions in the bottom-up calculator grammar	86
2.39	CFSM with epsilon productions	90
2.40	A syntax error in C	93
2.41	Syntax error in C (reprise)	CD 1
2.42	The problem with panic mode	CD 2
2.43	Phrase-level recovery in recursive descent	CD 2
2.44	Cascading syntax errors	CD 3
2.45	Reducing cascading errors with context-specific look-ahead	CD 4
2.46	Recursive descent with full phrase-level recovery	CD 4
2.47	Error production for " <code>;</code> <code>else</code> "	CD 5
2.48	Insertion-only repair in FMQ	CD 8
2.49	FMQ with deletions	CD 8
2.50	Panic mode in <code>yacc/bison</code>	CD 10
2.51	Panic mode with statement terminators	CD 11
2.52	Phrase-level recovery in <code>yacc/bison</code>	CD 11

**Theoretical Foundations**

2.53	Formal DFA for $(1 * 01 * 0)^* 1^*$	CD 14
------	-------------------------------------	-------

2.54	Reconstructing the regular expression for a 2-state DFA	CD 15
2.55	$0^n 1^n$ is not a regular language	CD 17
2.56	Separation of grammar classes	CD 17
2.57	Separation of language classes	CD 18

**Chapter 3: Names, Scopes, and Bindings****The Notion of Binding Time****Object Lifetime and Storage Management**

3.1	Static allocation of local variables	108
3.2	Layout of the run-time stack	109
3.3	External fragmentation in the heap	111

**Scope Rules**

3.4	Nested scopes	117
3.5	Static chains	119
3.6	A "gotcha" in declare-before-use	121
3.7	Whole-block scope in C#	121
3.8	"Local if written" in Python	122
3.9	Declaration order in Scheme	122
3.10	Declarations v. definitions in C	122
3.11	Inner declarations in C	124
3.12	Static variables in C	125
3.13	Stack module in Modula-2	126
3.14	Module as "manager" for a type	128
3.15	Module types in Euclid	129
3.16	N-ary methods in C++	131
3.17	Static v. dynamic scope	132
3.18	Customization via dynamic scope	134
3.19	Multiple interface alternative	134
3.20	Static variable alternative	134

**Implementing Scope**

3.34	The LeBlanc-Cook symbol table	CD 24
3.35	Symbol table for a sample program	CD 25
3.36	A-list lookup in Lisp	CD 27
3.37	Central reference table	CD 27

**The Binding of Referencing Environments**

3.21	Deep and shallow binding	136
3.22	Binding rules with static scoping	139
3.23	Returning a first-class subroutine in Scheme	140

**Binding Within a Scope**

3.24	Aliasing with parameters	142
3.25	Aliases and code improvement	142
3.26	Overloaded enumeration constants in Ada	143
3.27	Resolving ambiguous overloads	143

3.28	Overloading in Ada and C++	144	4.26	Processing lists with a semantic stack	CD 49
3.29	Overloading built-in operators	144			
3.30	Overloading v. coercion	145	<b>Decorating a Syntax Tree</b>		
3.31	Generic <code>min</code> function in Ada	147	4.12	Bottom-up CFG for calculator language with types	182
3.32	Implicit polymorphism in Scheme	148	4.13	Syntax tree to average an integer and a real	182
3.33	Implicit polymorphism in Haskell	148	4.14	Tree grammar for the calculator language with types	182
	<b>Separate Compilation</b>		4.15	Tree AG for the calculator language with types	184
3.38	Namespaces in C++	CD 32			
3.39	Using names from another namespace	CD 33			
3.40	Packages in Java	CD 33			
3.41	Using names from another package	CD 33			
3.42	Multipart package names	CD 35			
<b>Chapter 4: Semantic Analysis</b>					
<b>The Role of the Semantic Analyzer</b>					
4.1	Assertions in Euclid	164	5.1	Memory hierarchy stats	196
4.2	Assertions in C	164			
<b>Attribute Grammars</b>					
4.3	Bottom-up CFG for constant expressions	166	<b>Data Representation</b>		
4.4	Bottom-up AG for constant expressions	166	5.2	Big- and little-endian	199
<b>Evaluating Attributes</b>					
4.5	Decoration of a parse tree	168	5.15	Hexadecimal numbers	CD 54
4.6	Top-down CFG and parse tree for subtraction	169	5.16	Two's complement	CD 55
4.7	Decoration with left-to-right attribute flow	170	5.17	Overflow in two's complement addition	CD 56
4.8	Top-down AG for subtraction	171	5.18	Biased exponents	CD 57
4.9	Top-down AG for constant expressions	171	5.19	IEEE floating point	CD 57
4.10	Bottom-up and top-down AGs to build a syntax tree	175			
<b>Action Routines</b>					
4.11	Top-down action routines to build a syntax tree	180	<b>Instruction Set Architecture</b>		
<b>Space Management for Attributes</b>					
4.16	Stack trace for bottom-up parse, with action routines	CD 39	5.3	An <code>if</code> statement in x86 assembler	202
4.17	Finding inherited attributes in "buried" records	CD 40	5.4	Compare and test instructions	202
4.18	Grammar fragment requiring context	CD 41	5.5	Conditional branches on the MIPS	203
4.19	Semantic hooks for context	CD 42			
4.20	Semantic hooks that break an LR CFG	CD 42	<b>Architecture and Implementation</b>		
4.21	Action routines in the trailing part	CD 43	5.6	The x86 ISA	208
4.22	Left factoring in lieu of semantic hooks	CD 43	5.7	The MIPS ISA	208
4.23	Operation of an LL attribute stack	CD 44	5.20	x86 and MIPS register sets	CD 59
4.24	Ad hoc management of a semantic stack	CD 47	5.8	Pseudo-assembler	209
4.25	Processing lists with an attribute stack	CD 48			
<b>Compiling for Modern Processors</b>					
5.9	Performance $\neq$ clock rate	211			
5.10	Filling a load delay slot	213			
5.11	Renaming registers for scheduling	213			
5.12	Filling a branch delay slot	215			
5.13	Register allocation for a simple loop	216			
5.14	Register allocation and instruction scheduling	218			
<b>Chapter 6: Control Flow</b>					
<b>Expression Evaluation</b>					
6.1	A typical function call	234			
6.2	Typical operators	235			
6.3	Cambridge Polish (prefix) notation	235			
6.4	Mixfix notation in Smalltalk	235			

6.5	Conditional expressions	235	6.46	<b>Elsif/elif</b>	262
6.6	A complicated Fortran expression	236	6.47	<b>Cond in Lisp</b>	262
6.7	Precedence in four influential languages	236	6.48	Code generation for a Boolean condition	263
6.8	A "gotcha" in Pascal precedence	236	6.49	Code generation for short-circuiting	263
6.9	Common rules for associativity	238	6.50	Short-circuit creation of a Boolean value	264
6.10	L-values and r-values	239	6.51	<b>Case statements and nested ifs</b>	265
6.11	L-values in C	239	6.52	Translation of nested <b>ifs</b>	265
6.12	L-values in C++	239	6.53	Jump tables	266
6.13	Variables as values and references	240	6.54	Fall-through in C <b>switch</b> statements	269
6.14	Wrapper objects in Java 2	241	6.55	Fortran computed <b>goto</b>	269
6.15	Boxing in Java 5	241	6.56	Algol 60 <b>switch</b>	269
6.16	Boxing in C#	241			
6.17	Expression orientation in Algol 68	242		<b>Iteration</b>	
6.18	A "gotcha" in C conditions	243	6.57	Early Fortran <b>do</b> loop	271
6.19	Updating assignments	243	6.58	Meaning of a <b>do</b> loop	271
6.20	Side effects and updates	243	6.59	Modula-2 <b>for</b> loop	273
6.21	Assignment operators	244	6.60	Obvious translation of a <b>for</b> loop	274
6.22	Prefix and postfix inc/dec	244	6.61	<b>For</b> loop translation with test at the bottom	274
6.23	Advantages of postfix inc/dec	244	6.62	Reverse direction <b>for</b> loop	274
6.24	Simple multiway assignment	245	6.63	<b>For</b> loop translation with iteration count	275
6.25	Advantages of multiway assignment	245	6.64	Index value after loop	275
6.26	Programs outlawed by definite assignment	248	6.65	Preserving the final index value	276
6.27	Indeterminate ordering	249	6.66	Algol 60 <b>for</b> loop	277
6.28	A value that depends on ordering	250	6.67	Combination ( <b>for</b> ) loop in C	277
6.29	An optimization that depends on ordering	250	6.68	Simple iterator in Clu	279
6.30	Optimization and mathematical "laws"	250	6.69	Clu iterator for tree enumeration	279
6.31	Reordering and numerical stability	252	6.70	Java iterator for tree enumeration	279
6.32	Short-circuited expressions	252	6.71	Iterator objects in C++	282
6.33	Saving time with short-circuiting	252	6.72	Passing the "loop body" to an iterator in Scheme	282
6.34	Short-circuit pointer chasing	252	6.73	Iteration with blocks in Smalltalk	283
6.35	Short-circuiting and other errors	253	6.74	Imitating iterators in C	283
6.36	When not to use short-circuiting	253	6.95	Simple generator in Icon	CD 69
6.37	Optional short-circuiting	254	6.96	A generator inside an expression	CD 69
			6.97	Generating in search of success	CD 70
			6.98	Backtracking with multiple generators	CD 70
			6.75	<b>While</b> loop in Pascal	284
6.38	Control flow with <b>gosub</b> s in Fortran	254	6.76	Imitating <b>while</b> loops in Fortran 77	284
6.39	Leaving the middle of a loop	256	6.77	Post-test loop in Pascal and Modula	284
6.40	Returning from the middle of a subroutine	256	6.78	Post-test loop in C	285
6.41	Escaping a nested subroutine	256	6.79	Midtest loop in Modula	285
6.42	Structured nonlocal transfers	257	6.80	<b>Exit</b> as a separate statement	286
6.43	Error-checking with status codes	258	6.81	<b>Break</b> statement in C	286
			6.82	Exiting a nested loop	286
	<b>Sequencing</b>				
6.44	Side effects in a random number generator	261		<b>Recursion</b>	
			6.83	A "naturally iterative" problem	288
			6.84	A "naturally recursive" problem	288
			6.85	Implementing problems "the other way"	288
			6.86	Implementation of tail recursion	289

6.87	By-hand creation of tail-recursive code	289	7.22	Type conversions in Ada	326
6.88	Naive recursive Fibonacci function	290	7.23	Unchecked conversions in Ada	327
6.89	Efficient iterative Fibonacci function	290	7.24	Type conversions in C	327
6.90	Efficient tail-recursive Fibonacci function	291	7.25	Coercion in Ada	329
6.91	Tail-recursive Fibonacci function in Sisal	291	7.26	Coercion in C	329
6.92	Divisibility macro in C	292	7.27	Java container of <code>Object</code>	332
6.93	"Gotchas" in C macros	293	7.28	Inference of subrange types	333
6.94	Lazy evaluation of an infinite data structure	294	7.29	Using inference to avoid run-time checks	333
<b>Nondeterminacy</b>					
6.99	Avoiding asymmetry with nondeterminism	CD 72	7.30	Heuristic nature of subrange inference	334
6.100	Selection with guarded commands	CD 73	7.31	Type inference on string operations	334
6.101	Looping with guarded commands	CD 73	7.32	Type inference for sets	334
6.102	Nondeterministic message receipt	CD 74	7.104	Fibonacci function in ML	CD 81
6.103	Nondeterministic server in SR	CD 74	7.105	Expression types	CD 81
6.104	Naive (unfair) implementation of nondeterminism	CD 75	7.106	Type inconsistency	CD 82
6.105	"Gotcha" in round-robin implementation of nondeterminism	CD 75	7.107	Polymorphic functions	CD 82
<b>Chapter 7: Data Types</b>					
7.1	Operations that leverage type information	307	7.108	Polymorphic list operators	CD 84
7.2	Errors captured by type information	307	7.109	List notation	CD 84
<b>Type Systems</b>					
7.3	Enumerations in Pascal	315	7.110	Resolving ambiguity with explicit types	CD 85
7.4	Enumerations as constants	315	7.111	Pattern matching of argument tuples	CD 85
7.5	Converting to and from enumeration type	315	7.112	<code>Swap</code> in ML	CD 85
7.6	Distinguished values for enums	316	7.113	Run-time pattern matching	CD 86
7.7	Emulating distinguished enum values in Java 5	316	7.114	ML <code>case</code> expression	CD 86
7.8	Subranges in Pascal	316	7.115	Coverage of <code>case</code> labels	CD 86
7.9	Subranges in Ada	316	7.116	Function as a series of alternatives	CD 87
7.10	Space requirements of subrange type	317	7.117	Pattern matching of return tuple	CD 87
7.11	<code>Void</code> (empty) type	319	7.118	ML records	CD 87
7.12	Making do without <code>void</code>	319	7.119	ML <code>datatype</code> s	CD 87
7.13	Aggregates in Ada	320	7.120	Recursive <code>datatype</code> s	CD 88
7.121	Type equivalence in ML	CD 88			
<b>Records (Structures) and Variants (Unions)</b>					
7.33	A Pascal record	337			
7.34	A C struct	337			
7.35	Accessing record fields	337			
7.36	Nested records	337			
7.37	ML records and tuples	338			
7.38	Memory layout for a record type	338			
7.39	Layout of <code>packed</code> types	339			
7.40	Assignment and comparison of records	339			
7.41	Minimizing holes by sorting fields	340			
7.42	Pascal <code>with</code> statement	341			
7.122	Elliptical references in Cobol and PL/I	CD 90			
7.123	Pascal <code>with</code> statement (reprise)	CD 90			
7.124	Modula-3 <code>with</code> statement	CD 91			
7.125	Multiple-object <code>with</code> statements	CD 91			
7.126	Nonrecord <code>with</code> statements	CD 91			
7.127	Emulating <code>with</code> in Scheme	CD 92			
7.128	Emulating <code>with</code> in C	CD 92			
7.43	Variant record in Pascal	341			
7.44	Fortran <code>equivalence</code> statement	342			
7.45	Mixing <code>structs</code> and <code>unions</code> in C	342			

7.46	Breaking type safety with <code>equivalence</code>	343	7.86	Assignment in Lisp	375
7.47	Union conformity in Algol 68	344	7.87	Array names and pointers in C	376
7.48	Tagged variant record in Pascal	344	7.88	Pointer comparison and subtraction in C	376
7.49	Breaking type safety with variant records	345	7.89	Pointer and array declarations in C	376
7.50	Untagged variants in Pascal	345	7.90	Arrays as parameters in C	377
7.51	Ada variants and tags (discriminants)	346	7.91	<code>Sizeof</code> in C	377
7.52	A discriminated subtype in Ada	347	7.92	Multidimensional array parameters in C	378
7.53	Discriminated array in Ada	347	7.93	Explicit storage reclamation	379
<b>Arrays</b>					
7.54	Array declarations	350	7.94	Dangling reference detection with tombstones	380
7.55	Multidimensional arrays	350	7.95	Dangling reference detection with locks and keys	381
7.56	Multidimensional v. built-up arrays	351	7.96	Reference counts and circular structures	384
7.57	Arrays of arrays in C	352	7.97	Heap tracing with pointer reversal	386
7.58	Array slice operations	352	<b>Lists</b>		
7.59	Stack allocation of elaborated arrays	354	7.98	Lists in ML and Lisp	390
7.60	Stack allocation of new arrays	355	7.99	List notation	390
7.61	Conformant array parameters	355	7.100	Basic list operations in Lisp	391
7.62	Local arrays of dynamic shape	356	7.101	Basic list operations in ML	391
7.63	Dynamic strings in Java and C#	356	7.102	List comprehensions	392
7.64	Elaborated arrays in Fortran 90	356	<b>Files and Input/Output</b>		
7.65	Row-major v. column-major array layout	358	7.129	Files as a built-in type	CD 95
7.66	Array layout and cache performance	358	7.130	The <code>open</code> operation	CD 95
7.67	Contiguous v. row-pointer array layout	360	7.131	The <code>close</code> operation	CD 95
7.68	Indexing a contiguous array	361	7.132	Formatted output in Fortran	CD 97
7.69	Pseudo-assembler for contiguous array indexing	362	7.133	Labeled formats	CD 97
7.70	Static and dynamic portions of an array index	362	7.134	Printing to standard output	CD 98
7.71	Indexing complex structures	363	7.135	Formatted output in Ada	CD 99
7.72	Pseudo-assembler for row-pointer array indexing	364	7.136	Overloaded <code>put</code> routines	CD 99
<b>Strings</b>					
7.73	Character escapes in C and C++	366	7.137	Formatted output in C	CD 100
7.74	<code>Char*</code> assignment in C	367	7.138	Text in format strings	CD 100
<b>Sets</b>					
7.75	Set types	367	7.139	Formatted input in C	CD 100
<b>Pointers and Recursive Types</b>					
7.76	Tree type in ML	371	7.140	Formatted output in C++	CD 101
7.77	Tree type in Lisp	371	7.141	Stream manipulators	CD 102
7.78	Mutually recursive types in ML	372	7.142	Array output in C++	CD 103
7.79	Tree types in Pascal, Ada, and C	373	7.143	Changing default format	CD 103
7.80	Allocating heap nodes	374	<b>Equality Testing and Assignment</b>		
7.81	Object-oriented allocation of heap nodes	374	7.103	Equality testing in Scheme	394
7.82	Pointer-based tree	374	<b>Chapter 8: Subroutines and Control Abstraction</b>		
7.83	Pointer dereferencing	374	<b>Review of Stack Layout</b>		
7.84	Implicit dereferencing in Ada	375	8.1	Layout of run-time stack (reprise)	408
7.85	Pointer dereferencing in ML	375	8.2	Offsets from frame pointer	408
			8.3	Static and dynamic links	409
			8.4	Visibility of nested routines	409

**Calling Sequences**

8.5	A typical calling sequence	411
8.59	Nonlocal access using a display	CD 107
8.60	SGI MIPSpro C calling sequence	CD 111
8.61	Gnu Pascal x86 calling sequence	CD 115
8.62	Subroutine closure trampoline	CD 117
8.63	Register windows on the Sparc	CD 119
8.6	Requesting an <i>inline</i> subroutine	415
8.7	In-lining and recursion	416

**Parameter Passing**

8.8	Infix operators	418
8.9	Control abstraction in Lisp and Smalltalk	418
8.10	Passing a subroutine argument	418
8.11	Value and reference parameters	419
8.12	Emulating call-by-reference in C	420
8.13	<b>Const</b> parameters in C	421
8.14	Reference and value/result parameters	422
8.15	Reference parameters in C++	423
8.16	References as aliases in C++	423
8.17	Returning a reference from a function	424
8.18	Subroutines as parameters in Pascal	424
8.19	Subroutine types in Modula-2	424
8.20	Subroutine pointers in C and C++	425
8.21	First-class subroutines in Scheme	425
8.22	First-class subroutines in ML	426
8.64	Jensen's device	CD 122
8.23	Default parameters in Ada	428
8.24	Named parameters in Ada	429
8.25	Self-documentation with named parameters	429
8.26	Variable number of arguments in C	430
8.27	Variable number of arguments in Java	431
8.28	Variable number of arguments in C#	431
8.29	Returning a value from a function	432
8.30	Incremental computation of a return value	432
8.31	Explicitly named return values in SR	433

**Generic Subroutines and Modules**

8.32	Generic queues in Ada and C++	435
8.33	Generic <b>min</b> function in Ada (reprise)	435
8.34	Generic parameters	435
8.35	Simple constraints in Ada	437
8.36	<b>With</b> constraints in Ada	438
8.37	Generic sorting routine in Java	438
8.38	Generic sorting routine in C#	439
8.39	Generic sorting routine in C++	439
8.40	Generic class instance in C++	440
8.41	Generic subroutine instance in Ada	440
8.42	Implicit instantiation in C++	440

8.65	Generic <b>arbiter</b> class in C++	CD 125
8.66	Instantiation-time errors in C++ templates	CD 127
8.67	Generic <b>arbiter</b> class in Java	CD 128
8.68	Wildcards and bounds on Java generic parameters	CD 129
8.69	Type erasure and implicit casts	CD 130
8.70	<b>Unchecked</b> warnings in Java 5	CD 131
8.71	Java 5 generics and built-in types	CD 131
8.72	Sharing generic implementations in C#	CD 132
8.73	C# generics and built-in types	CD 132
8.74	Generic <b>arbiter</b> class in C#	CD 132

**Exception Handling**

8.43	<b>ON</b> conditions in PL/I	442
8.44	What is an exception?	444
8.45	Parameterized exceptions	444
8.46	Exception handler in Ada	445
8.47	Exception handler in C++	446
8.48	Exception handler in ML	447
8.49	<b>Finally</b> clause in Modula-3	447
8.50	<b>Catch</b> and <b>finally</b> in Java	448
8.51	Exceptions in a recursive descent parser	449
8.52	Stacked exception handlers	449
8.53	Multiple exceptions per handler	450
8.54	<b>Setjmp</b> and <b>longjmp</b> in C	451

**Coroutines**

8.55	Explicit interleaving of concurrent computations	453
8.56	Interleaving coroutines	454
8.57	Cactus stacks	456
8.58	Switching coroutines	457
8.75	Coroutine-based iterator invocation	CD 135
8.76	Coroutine-based iterator implementation	CD 135
8.77	Iterator usage in C#	CD 136
8.78	Implementation of C# iterators	CD 137
8.79	Sequential simulation of a complex physical system	CD 139
8.80	Initialization of a coroutine-based traffic simulation	CD 139
8.81	Traversing a street segment in the traffic simulation	CD 140
8.82	Scheduling a coroutine for future execution	CD 140
8.83	Queueing cars at a traffic light	CD 140
8.84	Waiting at a light	CD 141
8.85	Sleeping in anticipation of future execution	CD 141

**Chapter 9: Data Abstraction and Object Orientation****Object-Oriented Programming**

9.1	List_node class in C++	471
9.2	List class that uses list_node	473
9.3	Declaration of in-line (expanded) objects	473
9.4	Method declaration without definition	474
9.5	Separate method definition	474
9.6	Property and indexer methods in C#	475
9.7	Queue class derived from list	476
9.8	The Smalltalk class hierarchy	476
9.9	Base class for general purpose lists	478
9.10	Overloaded int_list_node constructor	478
9.11	Redefining a method in a derived class	479
9.12	Redefinition that builds on the base class method	479
9.13	Accessing base class members	479
9.14	Renaming methods in Eiffel	480

**Encapsulation and Inheritance**

9.15	Data hiding in Euclid	481
9.16	Opaque types in Modula-2	482
9.17	Data hiding in Ada	482
9.18	The hidden this parameter	484
9.19	Private base class in C++	484
9.20	Protected base class in C++	485
9.21	List and queue abstractions in Ada 95	486

**Initialization and Finalization**

9.22	Naming constructors in Eiffel	490
9.23	Metaclasses in Smalltalk	491
9.24	Declarations and constructors in C++	492
9.25	Copy constructors	492
9.26	Eiffel constructors and expanded objects	494
9.27	Specification of base class constructor arguments	495
9.28	Specification of member constructor arguments	495
9.29	Invocation of base class constructor in Java	496
9.30	Reclaiming space with destructors	496

**Dynamic Method Binding**

9.31	Derived class objects in a base class context	498
9.32	Static and dynamic method binding	498
9.33	The need for dynamic binding	499
9.34	Virtual methods in C++ and C#	500
9.35	Virtual methods in Simula	500
9.36	Class-wide types in Ada 95	500

9.37	Abstract methods in Java and C#	501
9.38	Abstract methods in C++	501
9.39	Vtables	502
9.40	Implementation of a virtual method call	502
9.41	Implementation of single inheritance	502
9.42	Casts in C++	503
9.43	Reverse assignment in Eiffel and C#	504
9.44	Inheritance and method signatures	506
9.45	Generics and inheritance	506
9.46	Like in Eiffel	507
9.47	Objects as closures	508
9.48	Encapsulating arguments	509

**Multiple Inheritance**

9.49	Deriving from two base classes	511
9.50	Deriving from two base classes (reprise)	CD 146
9.51	(Nonrepeated) multiple inheritance	CD 146
9.52	Method invocation with multiple inheritance	CD 147
9.53	This correction	CD 148
9.54	Methods found in more than one base class	CD 148
9.55	Overriding an ambiguous method	CD 149
9.56	Repeated multiple inheritance	CD 150
9.57	Shared inheritance in C++	CD 151
9.58	Replicated inheritance in Eiffel	CD 151
9.59	Using replicated inheritance	CD 151
9.60	Overriding methods with shared inheritance	CD 153
9.61	Implementation of shared inheritance	CD 153
9.62	Mixing interfaces into a derived class	CD 155
9.63	Compile-time implementation of mix-in inheritance	CD 155

**Object-Oriented Programming Revisited**

9.64	Operations as messages in Smalltalk	CD 158
9.65	Mixfix messages	CD 158
9.66	Selection as an ifTrue: ifFalse: message	CD 159
9.67	Iterating with messages	CD 159
9.68	Blocks as closures	CD 160
9.69	Logical looping with messages	CD 160
9.70	Defining control abstractions	CD 160
9.71	Recursion in Smalltalk	CD 161

**Chapter 10: Functional Languages****Historical Origins**

10.1	Comparing programming models	525
------	------------------------------	-----

### Functional Programming Concepts

#### A Review/Overview of Scheme

10.2	The read-eval-print loop	528
10.3	Significance of parentheses	528
10.4	Quoting	528
10.5	Dynamic typing	529
10.6	Type predicates	529
10.7	Liberal syntax for symbols	529
10.8	<b>Lambda</b> expressions	529
10.9	Function evaluation	530
10.10	<b>If</b> expressions	530
10.11	Nested scopes with <b>let</b>	530
10.12	Global bindings with <b>define</b>	531
10.13	Basic list operations	531
10.14	List search functions	532
10.15	Searching association lists	532
10.16	Multiway conditional expressions	533
10.17	Assignment	533
10.18	Sequencing	533
10.19	Iteration	533
10.20	Evaluating data as code	535
10.21	<b>Eval-apply</b> trace of a simple expression	536
10.22	Denotational semantics of Scheme	537
10.23	Simulating a DFA in Scheme	537

#### Evaluation Order Revisited

10.24	Applicative and normal-order evaluation	539
10.25	Normal-order avoidance of unnecessary work	540
10.26	Stream-based program execution	542
10.27	Interactive I/O with streams	543
10.28	The Haskell I/O monad	543
10.29	Invocation of actions with <b>do</b>	544
10.30	Functional composition of actions	544
10.31	Streams and the I/O monad	545

#### Higher-Order Functions

10.32	<b>Map</b> function in Scheme	545
10.33	Folding (reduction) in Scheme	545
10.34	Combining higher-order functions	546
10.35	Partial application with currying	546
10.36	General purpose <b>curry</b> function	546
10.37	Tuples as ML function arguments	546
10.38	Optional parentheses on singleton arguments	547
10.39	Simple curried function in ML	547
10.40	Shorthand notation for currying	548
10.41	Folding (reduction) in ML	548
10.42	Curried <b>fold</b> in ML	548
10.43	Currying in ML v. Scheme	548

### Theoretical Foundations

10.44	Declarative (nonconstructive) function definition	549
10.45	Functions as mappings	CD 166
10.46	Functions as sets	CD 166
10.47	Functions as powerset elements	CD 167
10.48	Function spaces	CD 167
10.49	Higher-order functions as sets	CD 167
10.50	Curried functions as sets	CD 168
10.51	Juxtaposition as function application	CD 168
10.52	Lambda calculus syntax	CD 168
10.53	Binding parameters with $\lambda$	CD 169
10.54	Free variables	CD 169
10.55	Naming functions for future reference	CD 169
10.56	Evaluation rules	CD 169
10.57	Delta reduction for arithmetic	CD 170
10.58	Eta reduction	CD 170
10.59	Reduction to simplest form	CD 170
10.60	Nonterminating applicative-order reduction	CD 171
10.61	Booleans and conditionals	CD 172
10.62	Beta abstraction for recursion	CD 172
10.63	The fixed-point combinator <b>Y</b>	CD 173
10.64	Lambda calculus list operators	CD 173
10.65	List operator identities	CD 175
10.66	Nesting of lambda expressions	CD 175
10.67	Paired arguments and currying	CD 176

### Functional Programming in Perspective

## Chapter 11: Logic Languages

#### Logic Programming Concepts

11.1	Horn clauses	560
11.2	Resolution	560
11.3	Unification	560

#### Prolog

11.4	Atoms, variables, scope, and type	561
11.5	Structures and predicates	561
11.6	Facts and rules	562
11.7	Queries	562
11.8	Resolution in Prolog	563
11.9	Unification in Prolog and ML	563
11.10	Equality and unification	564
11.11	Unification without instantiation	564
11.12	List notation in Prolog	564
11.13	Functions, predicates, and two-way rules	565
11.14	Arithmetic and the <b>is</b> predicate	565
11.15	Search tree exploration	566
11.16	Backtracking and instantiation	567

I1.17 Order of rule evaluation	568	I2.9 <b>Forall</b> in Fortran 90	607
I1.18 Infinite regression	568	I2.10 Elaborated tasks in Ada	607
I1.19 Tic-tac-toe in Prolog	569	I2.11 Co-begin v. fork/join	608
I1.20 The cut	572	I2.12 Task types in Ada	608
I1.21 <b>Not</b> and its implementation	572	I2.13 <b>Fork/Join</b> in Modula-3	609
I1.22 Pruning unwanted answers with the cut	572	I2.14 Forking a <b>proc</b> in SR	609
I1.23 Using the cut for selection	573	I2.15 Thread creation in Java 2	610
I1.24 Looping with <b>fail</b>	573	I2.16 Thread pools in Java 5	610
I1.25 Looping with an unbounded generator	573	I2.17 Modeling subroutines with fork/join	611
I1.26 Character input with <b>get</b>	574	I2.18 Returning without terminating	611
I1.27 Prolog programs as data	574	I2.19 Early reply in SR	613
I1.28 Modifying the Prolog database	575	I2.20 Early reply for initialization	613
I1.29 Tic-tac-toe (full game)	575	I2.21 Multiplexing threads on processes	614
I1.30 The <b>functor</b> predicate	577	I2.22 Cooperative multithreading on a uniprocessor	615
I1.31 Creating terms at run time	577	I2.23 A race condition in preemptive multithreading	617
I1.32 Pursuing a dynamic goal	577	I2.24 Disabling signals during context switch	617
I1.33 Custom database perusal	578		

#### Theoretical Foundations

I1.34 Predicates as mathematical objects	579
I1.39 Propositions	CD 180
I1.40 Different ways to say things	CD 181
I1.41 Conversion to clausal form	CD 181
I1.42 Conversion to Prolog	CD 182
I1.43 Disjunctive left-hand side	CD 182
I1.44 Empty left-hand side	CD 183
I1.45 Theorem proving as a search for contradiction	CD 183
I1.46 Skolem constants	CD 183
I1.47 Skolem functions	CD 184
I1.48 Limitations of Skolemization	CD 184

#### Logic Programming in Perspective

I1.35 Sorting incredibly slowly	581
I1.36 Quicksort in Prolog	581
I1.37 Negation as failure	581
I1.38 Negation and instantiation	582

## Chapter 12: Concurrency

#### Background and Motivation

I2.1 A race condition in the operating system	591
I2.2 Multithreaded web browser	593
I2.3 Dispatch loop web browser	595
I2.4 Direct and indirect networks	598
I2.5 The cache coherence problem	599

#### Concurrent Programming Fundamentals

I2.6 <b>Par begin</b> in Algol 68	604
I2.7 <b>Par</b> in Occam	605
I2.8 Parallel loops in SR	606

#### Shared Memory

I2.25 The basic <b>test_and_set</b> lock	620
I2.26 Test-and-test_and_set	621
I2.27 Atomic update with <b>LL/SC</b>	621
I2.28 The "sense-reversing" barrier	623
I2.29 Scheduling threads on processes	623
I2.30 A race condition in thread scheduling	625
I2.31 A "spin-then-yield" lock	625
I2.32 Semaphore implementation	627
I2.33 Bounded buffer with semaphores	628
I2.34 Bounded buffer monitor	630
I2.35 How to wait for a signal (hint or absolute)	631
I2.36 Original CCR syntax	634
I2.37 <b>Synchronized</b> statement in Java	635
I2.38 <b>Notify</b> as hint in Java	636
I2.39 <b>Lock</b> variables in Java 5	637
I2.40 Multiple <b>Conditions</b> in Java 5	638
I2.41 <b>Future</b> construct in Multilisp	639

#### Message Passing

I2.42 Naming processes, ports, and entries	642
I2.43 <b>Entry</b> calls in Ada	643
I2.44 Channels in Occam	643
I2.45 Datagram messages in Java	644
I2.46 Connection-based messages in Java	645
I2.47 Three main options for <b>send</b> semantics	646
I2.48 Buffering-dependent deadlock	648
I2.49 Acknowledgments	648
I2.50 Bounded buffer in Ada 83	651
I2.51 Timeout and distributed termination	652
I2.52 Bounded buffer in Occam	653

12.53 Asymmetry of synchronization <b>send</b>	653	13.38 Content versus appearance in HTML	712
12.54 Timeout in Occam receipt	654	13.39 Well-formed XHTML	713
12.55 Bounded buffer in SR	656	13.40 XHTML to display a favorite quote	714
12.56 Peeking at messages in SR	656	13.41 XPath names for XHTML elements	715
12.57 An RPC server system	658	13.42 Creating a reference list with XSLT	717
<b>Chapter 13: Scripting Languages</b>			
What Is a Scripting Language?			
13.1 Trivial programs in conventional and scripting languages	674	13.43 Scoping rules in Python	724
13.2 Coercion in Perl	676	13.44 Super-assignment in R	725
<b>Problem Domains</b>			
13.3 "Wildcards" and "globbing"	678	13.45 Scoping rules in Tcl	725
13.4 <b>For</b> loops in the shell	679	13.46 Static and dynamic scope in Perl	726
13.5 A whole loop on one line	679	13.47 Accessing globals in Perl	727
13.6 Conditional tests in the shell	679	13.48 Basic operations in POSIX REs	730
13.7 Pipes	680	13.49 Extra quantifiers in POSIX REs	730
13.8 Output redirection	680	13.50 Zero-length assertions	730
13.9 Redirection of <b>stderr</b> and <b>stdout</b>	681	13.51 Character classes	730
13.10 Heredocs (in-line input)	681	13.52 The dot (.) character	730
13.11 Single and double quotes	681	13.53 Negation and quoting in character classes	730
13.12 Subshells	682	13.54 Predefined POSIX character classes	731
13.13 Brace-quoted blocks in the shell	682	13.55 RE matching in Perl	731
13.14 Pattern-based list generation	682	13.56 Negating a match in Perl	731
13.15 User-defined shell functions	683	13.57 RE substitution in Perl	731
13.16 The <b>#!</b> convention in script files	683	13.58 Trailing modifiers on RE matches	731
13.17 Extracting HTML headers with <b>sed</b>	685	13.59 Greedy and minimal matching	733
13.18 One-line scripts in <b>sed</b>	686	13.60 Minimal matching of HTML headers	733
13.19 Extracting HTML headers with <b>awk</b>	686	13.61 Variable interpolation in extended REs	733
13.20 Fields in <b>awk</b>	687	13.62 Variable capture in extended REs	734
13.21 Capitalizing a title in <b>awk</b>	687	13.63 Backreferences in extended REs	734
13.22 Extracting HTML headers with Perl	688	13.64 Dissecting a floating-point literal	734
13.23 "Force quit" script in Perl	690	13.65 Implicit capture of prefix, match, and suffix	734
13.24 "Force quit" script in Tcl	692	13.66 Coercion in Ruby and Perl	736
13.25 "Force quit" script in Python	694	13.67 Coercion and context in Perl	736
13.26 Method call syntax in Ruby	696	13.68 Explicit conversion in Ruby	736
13.27 "Force quit" script in Ruby	697	13.69 Perl arrays	738
13.28 Numbering lines with Emacs Lisp	699	13.70 Perl hashes	738
<b>Scripting the World Wide Web</b>			
13.29 Remote monitoring with a CGI script	702	13.71 Arrays and hashes in Python and Ruby	738
13.30 Adder web form with a CGI script	703	13.72 Array access methods in Ruby	739
13.31 Remote monitoring with a PHP script	705	13.73 Tuples in Python	739
13.32 A fragmented PHP script	706	13.74 Sets in Python	740
13.33 Adder web form with a PHP script	706	13.75 Conflated types in PHP, Tcl, and JavaScript	740
13.34 Self-posting Adder web form	707	13.76 Multidimensional arrays in Python and other languages	740
13.35 Adder web form in JavaScript	708	13.77 Scalar and list context in Perl	741
13.36 Embedding an applet in a web page	709	13.78 Using <b>wantarray</b> to determine calling context	741
13.37 Embedding an object in a web page	710	13.79 A simple class in Perl	742
		13.80 Invoking methods in Perl	742
		13.81 Inheritance in Perl	743
		13.82 Inheritance via <b>use base</b>	744
		13.83 Prototypes in JavaScript	745
<b>Innovative Features</b>			

I3.84	Overriding instance methods in JavaScript	745	I5.2	Elimination of redundant loads and stores	CD 206
I3.85	Inheritance in JavaScript	746	I5.3	Constant folding	CD 206
I3.86	Constructors in Python and Ruby	746	I5.4	Constant propagation	CD 206
I3.87	Naming class members in Python and Ruby	746	I5.5	Common subexpression elimination	CD 207
			I5.6	Copy propagation	CD 207
			I5.7	Strength reduction	CD 207
			I5.8	Elimination of useless instructions	CD 208
			I5.9	Exploitation of the instruction set	CD 208
			I5.10	The <code>combinations</code> subroutine	CD 210
			I5.11	Syntax tree and naive control flow graph	CD 210
			I5.12	Result of local redundancy elimination	CD 215
			I5.13	Conversion to SSA form	CD 218
			I5.14	Global value numbering	CD 220
			I5.15	Data flow equations for available expressions	CD 222
			I5.16	Fixed point for available expressions	CD 222
			I5.17	Result of global common subexpression elimination	CD 224
			I5.18	Edge splitting transformations	CD 225
			I5.19	Data flow equations for live variables	CD 226
			I5.20	Fixed point for live variables	CD 226
			I5.21	Data flow equations for reaching definitions	CD 228
			I5.22	Result of hoisting loop invariants	CD 229
			I5.23	Induction variable strength reduction	CD 230
			I5.24	Induction variable elimination	CD 230
			I5.25	Result of induction variable optimization	CD 231
			I5.26	Remaining pipeline delays	CD 233
			I5.27	Value dependence DAG	CD 233
			I5.28	Result of instruction scheduling	CD 235
			I5.29	Result of loop unrolling	CD 237
			I5.30	Result of software pipelining	CD 237
			I5.31	Loop interchange	CD 241
			I5.32	Loop tiling (blocking)	CD 241
			I5.33	Loop distribution	CD 242
			I5.34	Loop fusion	CD 243
			I5.35	Obtaining a perfect loop nest	CD 243
			I5.36	Loop-carried dependences	CD 244
			I5.37	Loop reversal and interchange	CD 244
			I5.38	Loop skewing	CD 245
			I5.39	Coarse-grain parallelization	CD 246
			I5.40	Strip mining	CD 247
			I5.41	Live ranges of virtual registers	CD 248
			I5.42	Register coloring	CD 248
			I5.43	Optimized <code>combinations</code> subroutine	CD 249
I4.1	Phases of compilation	762			
I4.2	GCD program abstract syntax tree (reprise)	762			
<b>Chapter 14: Building a Runnable Program</b>					
<b>Back-End Compiler Structure</b>					
I4.17	<code>ExpressionTree</code> abstraction in Diana	CD 190			
I4.18	An RTL <code>insn</code> sequence	CD 193			
<b>Intermediate Forms</b>					
I4.17	<code>ExpressionTree</code> abstraction in Diana	CD 190			
I4.18	An RTL <code>insn</code> sequence	CD 193			
<b>Code Generation</b>					
I4.3	Simpler compiler structure	769			
I4.4	An attribute grammar for code generation	771			
I4.5	Stack-based register allocation	772			
I4.6	GCD program target code	773			
<b>Address Space Organization</b>					
<b>Assembly</b>					
I4.7	Assembly as a final compiler pass	777			
I4.8	Direct generation of object code	777			
I4.9	Instruction variants	778			
I4.10	Pseudoinstruction expansion	778			
I4.11	Two-instruction loads	779			
I4.12	Nontrivial conditional branches	779			
I4.13	Assembler directives	779			
I4.14	Encoding of addresses in object files	780			
<b>Linking</b>					
I4.15	Static linking	782			
I4.16	Checksumming headers for consistency	783			
<b>Dynamic Linking</b>					
I4.19	PIC under MIPS/IRIX	CD 196			
I4.20	Dynamic linking under MIPS/IRIX	CD 197			
<b>Chapter 15: Code Improvement</b>					
I5.1	Code Improvement Phases	CD 204			

# Bibliography

- [ACC<sup>+</sup>90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, Amsterdam, the Netherlands, June 1990. In *ACM Computer Architecture News*, 18(3), September 1990.
- [ACD<sup>+</sup>96] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. ThreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.
- [ACG86] Shakil Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.
- [AD00] Mohit Aron and Peter Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, August 2000.
- [ADH<sup>+</sup>98] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele, Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised<sup>5</sup> report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Edited by Richard Kelsey, William Clinger, and Jonathan Rees. Available at [www.schemers.org/Documents/Standards/R5RS/](http://www.schemers.org/Documents/Standards/R5RS/).
- [Ado86] Adobe Systems, Inc. *PostScript Language Tutorial and Cookbook*. Addison-Wesley, Reading, MA, 1986.
- [Ado90] Adobe Systems, Inc. *PostScript Language Reference Manual*. Addison-Wesley, Reading, MA, second edition, 1990.
- [AF84] Krzysztof R. Apt and Nissim Francez. Modeling the distributed termination convention of CSP. *ACM Transactions on Programming Languages and Systems*, 6(3):370–379, July 1984.
- [AG90] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, Reading, MA, 1990.
- [AG94] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, Redwood City, CA, second edition, 1994.
- [AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.
- [AG98] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.
- [AH95] Ole Agesen and Urs Hözle. Type feedback v. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of the Tenth ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–107, Austin, TX, October 1995. In *ACM SIGPLAN Notices*, 30(10), October 1995.
- [AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, San Francisco, CA, 2002.

- [AKW88] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, Reading, MA, 1988.
- [ALL89] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.
- [Ame78a] American National Standards Institute, New York, NY. *Programming Language FORTRAN*, 1978. ANSI X3.9–1978.
- [Ame78b] American National Standards Institute, New York, NY. *Programming Language Minimal BASIC*, 1978. ANSI X3.60–1978.
- [Ame83] American National Standards Institute, New York, NY. *Reference Manual for the Ada Programming Language*, January 1983. ANSI/MIL 1815 A–1983.
- [Ame85] American National Standards Institute, New York, NY. *Programming Language COBOL*, 1985. ANSI X3.23–1985. Supercedes earlier ANSI standards from 1968 and 1974.
- [Ame90] American National Standards Institute, New York, NY. *Programming Language C*, 1990. ANSI/ISO 9899–1990 (revision and redesignation of ANSI X3.159–1989).
- [Ame92] American National Standards Institute, New York, NY. *Programming Language, FORTRAN—Extended*, 1992. ANSI X3.198–1992. Also ISO 1539–1991 (E).
- [Ame96a] American National Standards Institute, New York, NY. *Information Technology—Programming Language REXX*, 1996. ANSI INCITS 274-1996/AMD1-2000 (R2001).
- [Ame96b] American National Standards Institute, New York, NY. *Programming Language—Common Lisp*, 1996. ANSI X3.226:1994.
- [And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, CA, 1991.
- [AO93] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings, Redwood City, CA, 1993.
- [App91] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1991.
- [App97] Andrew W. Appel. *Modern Compiler Implementation*. Cambridge University Press, Cambridge, England, 1997. Text available in ML, Java, and C versions. C version specialized by Maia Ginsburg.
- [App04] Apple Computer, Inc., Cupertino, CA. *The Objective-C Programming Language*, Mac OS X version 10.3 edition, February 2004.
- [AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.
- [AS96] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, second edition, 1996. With Julie Sussman. Full text and supplementary resources available at [mitpress.mit.edu/sicp/](http://mitpress.mit.edu/sicp/).
- [Ass93] Association for Computing Machinery, New York, NY. *Proceedings of the Second ACM SIGPLAN History of Programming Languages Conference*, Cambridge, MA, April 1993. In *ACM SIGPLAN Notices*, 28(3), March 1993.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
- [Atk73] M. Stella Atkins. Mutual recursion in Algol 60 using restricted compilers. *Communications of the ACM*, 16(1):47–48, 1973.
- [AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1972. Two-volume set.
- [AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, CA, January 1988.
- [Bac78] John W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. The 1977 Turing Award lecture.
- [Bag86] Rajive L. Bagrodia. A distributed algorithm to implement the generalized alternative command of CSP. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 422–427, Cambridge, MA, May 1986. IEEE Computer Society Press, Washington, DC.
- [BALL90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.
- [Ban97] Utpal Banerjee. *Dependence Analysis*, volume 3 of *Loop Transformations for Restructuring Compilers*. Kluwer Academic Publishers, Boston, MA, 1997.

- [Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, the Netherlands, revised edition, 1984.
- [BC93] Peter Bumbulis and Donald D. Cowan. RE2C: A more versatile scanner generator. *ACM Letters on Programming Languages and Systems*, 2(1–4):70–84, March–December 1993.
- [BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, BC, Canada, June 2000.
- [BDH<sup>+</sup>95] Jehoshua Bruck, Danny Dolev, Ching-Tien Ho, Marcel-Catalin Rosu, and Ray Strong. Efficient message passing interface (MPI) for parallel computing on clusters of workstations. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 64–73, Santa Barbara, CA, July 1995.
- [BDMN73] Graham M. Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA Begin*. Auerback Publishers, Inc., Philadelphia, PA, 1973.
- [Bec97] Leland L. Beck. *System Software: An Introduction to Systems Programming*. Addison-Wesley, Reading, MA, third edition, 1997.
- [Bee70] David Beech. A structural view of PL/I. *ACM Computing Surveys*, 2(1):33–64, March 1970.
- [Ben86] John L. Bentley. *Programming Pearls*. Addison-Wesley, Reading, MA, 1986.
- [Ber80] Arthur J. Bernstein. Output guards and nondeterminism in ‘Communicating Sequential Processes’. *ACM Transactions on Programming Languages and Systems*, 2(2):234–238, April 1980.
- [Ber85] Robert L. Bernstein. Producing good code for the case statement. *Software—Practice and Experience*, 15(10):1021–1024, October 1985. A correction, by Sampath Kannan and Todd A. Proebsting, appears in Volume 24, Number 2.
- [BFKM86] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley Series in Artificial Intelligence. Addison-Wesley, Reading, MA, 1986.
- [BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.
- [BHJ<sup>+</sup>87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Lawrence Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.
- [BHPS61] Yehoshua Bar-Hillel, Micha A. Perles, and Elihu Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14:143–172, 1961.
- [BI82] Alan H. Borning and Daniel H. H. Ingalls. Multiple inheritance in Smalltalk-80. In *AAAI-82: The National Conference on Artificial Intelligence*, pages 234–237, Pittsburgh, PA, August 1982. American Association for Artificial Intelligence.
- [BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.
- [BN84] Andrew D. Birrell and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [BO03] Randal E. Bryant and David O’Hallaron. *Computer Systems: A Programmer’s Perspective*. Prentice-Hall, Upper Saddle River, NJ, 2003.
- [BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the Thirteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 183–200, Vancouver, BC, Canada, October 1998.
- [Bou78] Stephen R. Bourne. An introduction to the UNIX shell. *Bell System Technical Journal*, 57(6, Part 2):2797–2822, July–August 1978.
- [Bri73] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, NJ, 1973.
- [Bri75] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, June 1975.
- [Bri78] Per Brinch Hansen. Distributed Processes: A concurrent programming concept. *Communications of the ACM*, 21(11):934–941, November 1978.
- [Bri81] Per Brinch Hansen. The design of Edison. *Software—Practice and Experience*, 11(4):363–396, April 1981.
- [Bro87] Leo Brodie. *Starting FORTH: An Introduction to the FORTH Language and Operating System for Beginners and Professionals*. Prentice-Hall Software Series. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1987.

- [Bro96] Kraig Brockschmidt. How OLE and COM solve the problems of component software design. *Microsoft Systems Journal*, 11(5):63–82, May 1996.
- [BS83a] Daniel G. Bobrow and Mark J. Stefik. The LOOPS manual. Technical report, Xerox Palo Alto Research Center, Palo Alto, CA, 1983.
- [BS83b] G. N. Buckley and A. Silbershatz. An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, April 1983.
- [BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Eleventh ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–341, San Jose, CA, October 1996.
- [BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.
- [CAC<sup>+</sup>81] Gregory Chaitin, Marc Auslander, Ashok Chandra, John Cocke, Martin Hopkins, and Peter Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.
- [Cai82] R. Cailliau. How to avoid getting SCHLONKED by Pascal. *ACM SIGPLAN Notices*, 17(12):31–40, December 1982.
- [Can92] David Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.
- [CDS03] Bruce R. Childers, Jack W. Davidson, and Mary Lou Soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *Proceedings of the NSF Workshop on Next Generation Software*, Nice, France, April 2003. Held in conjunction with the *International Parallel and Distributed Processing Symposium*. Available at [www.cs.pitt.edu/coco/papers/ipdps-nf-workshop.pdf](http://www.cs.pitt.edu/coco/papers/ipdps-nf-workshop.pdf).
- [CDW04] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the Network and Distributed System Security Symposium*, pages 171–185, San Diego, CA, February 2004.
- [Cer89] Paul Ceruzzi. *Beyond the Limits—Flight Enters the Computer Age*. MIT Press, Cambridge, MA, 1989.
- [CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, the Netherlands, 1958. With two sections by William Craig.
- [CFR<sup>+</sup>91] Ronald Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, IT-2(3):113–124, September 1956.
- [Cho62] Noam Chomsky. Context-free grammars and pushdown storage. In *Quarterly Progress Report No. 65*, pages 187–194. MIT Research Laboratory for Electronics, Cambridge, MA, 1962.
- [CHP71] Pierre-Jacques Courtois, F. Heymans, and David L. Parnas. Concurrent control with ‘readers’ and ‘writers’. *Communications of the ACM*, 14(10):667–668, October 1971.
- [Chu41] Alonzo Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies #6. Princeton University Press, Princeton, NJ, 1941.
- [Cia92] Paolo Ciancarini. Parallel programming with logic languages: A survey. *Computer Languages*, 17(4):213–239, October 1992.
- [CL83] Robert P. Cook and Thomas J. LeBlanc. A symbol table abstraction to implement languages with explicit scope control. *IEEE Transactions on Software Engineering*, SE-9(1):8–12, January 1983.
- [Cle86] J. Craig Cleaveland. *An Introduction to Data Types*. Addison-Wesley, Reading, MA, 1986.
- [CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.
- [CM84] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.
- [CM94] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, West Germany, fourth edition, 1994.
- [CMD<sup>+</sup>01] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, San Francisco, CA, 2001.
- [Coh81] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.
- [Con63] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, July 1963.
- [Cou84] Bruno Courcelle. Attribute grammars: Definitions, analysis of dependencies, proof methods. In

- Bernard Lorho, editor, *Methods and Tools for Compiler Construction: An Advanced Course*, pages 81–102. Cambridge University Press, Cambridge, England, 1984.
- [CR01] James H. Cross II and Eric Roberts, editors. *Computing Curricula 2001: Computer Science*. Joint Task Force on Computing Curricula, IEEE Computer Society, Association for Computing Machinery, 2001. Available at [www.computer.org/education/cc2001/](http://www.computer.org/education/cc2001/).
- [CS69] John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, New York, NY, 1969.
- [CS98] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA, 1998. With Anoop Gupta.
- [CT04] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, San Francisco, CA, 2004.
- [CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
- [CW05] Norman Chonacky and David Winch. Maple, Mathematica, and Matlab: The 3M’s without the tape. *Computing in Science and Engineering*, 7(1):8–16, 2005. Available as [csdl.computer.org/comp/mags/cs/2005/01/c1008.pdf](http://csdl.computer.org/comp/mags/cs/2005/01/c1008.pdf).
- [Dar90] Jared L. Darlington. Search direction by goal failure in goal-oriented programming. *ACM Transactions on Programming Languages and Systems*, 12(2):224–252, April 1990.
- [Dav63] Martin Davis. Eliminating the irrelevant from mechanical proofs. In *Proceedings of a Symposium in Applied Mathematics*, volume 15, pages 15–30. American Mathematical Society, Providence, RI, 1963.
- [DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.
- [DDH72] Ole-Johan Dahl, Edsger W. Dijkstra, and Charles Antony Richard Hoare. *Structured Programming*. A.P.I.C. Studies in Data Processing #8. Academic Press, New York, NY, 1972.
- [DeR69] Franklin L. DeRemer. *Practical Translators for LR(k) Languages*. Ph.D. dissertation, Massachusetts Institute of Technology, 1969.
- [DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, July 1971.
- [DGAFS<sup>+</sup>80] Robert B. K. Dewar, Jr., Gerald A. Fisher, Edmond Schonberg, Robert Froehlich, Stephen Bryant, Clinton F. Goss, and Michael Burke. The NYU Ada trans-
- lator and interpreter. In *Proceedings of the ACM SIGPLAN Symposium on the Ada Programming Language*, pages 194–201, Boston, MA, December 1980. In *ACM SIGPLAN Notices*, 15(11), November 1980.
- [Dij60] Edsger W. Dijkstra. Recursive programming. *Numerische Mathematik*, 2:312–318, 1960. Reprinted as pages 221–228 of *Programming Systems and Languages*, Saul Rosen, editor. MacGraw-Hill, New York, NY, 1967.
- [Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.
- [Dij68a] Edsger W. Dijkstra. Co-operating sequential processes. In F. Genys, editor, *Programming Languages*, pages 43–112. Academic Press, London, England, 1968.
- [Dij68b] Edsger W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.
- [Dij72] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. In Charles Antony Richard Hoare and Ronald H. Perrott, editors, *Operating Systems Techniques*, A.P.I.C. Studies in Data Processing #9, pages 72–93. Academic Press, London, England, 1972. Also *Acta Informatica*, 1(8):115–138, 1971.
- [Dij75] Edsger W. Dijkstra. Guarded commands, noneterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.
- [Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1976.
- [Dij82] Edsger W. Dijkstra. How do we tell truths that might hurt? *ACM SIGPLAN Notices*, 17(5):13–15, May 1982.
- [Dio78] Bernard A. Dion. *Locally Least-Cost Error Correctors for Context-Free and Context-Sensitive Parsers*. Ph. D. dissertation, University of Wisconsin–Madison, 1978. Computer Sciences Technical Report #344.
- [DMM96] Amer Diwan, J. Eliot, B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically typed object-oriented programs. In *Proceedings of the Eleventh ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 292–305, San Jose, CA, October 1996.
- [Dol97] Julian Dolby. Automatic inline allocation of objects. In *Proceedings of the SIGPLAN ’97 Conference on Programming Language Design and Implementation*, pages 7–17, Las Vegas, NV, June 1997. In *ACM SIGPLAN Notices*, 32(5), May 1997.
- [Dri93] Karel Driesen. Selector table indexing and sparse arrays. In *Proceedings of the Eighth ACM SIGPLAN Con-*

- ference on Object-Oriented Programming Systems, Languages, and Applications*, pages 259–270, Washington, DC, September 1993. In *ACM SIGPLAN Notices*, 28(10), October 1993.
- [DRSS96] Steven Dawson, C. R. Ramakrishnan, Steven Skiena, and Terrence Swift. Principles and practice of unification factoring. *ACM Transactions on Programming Languages and Systems*, 18(5):528–563, September 1996.
- [DS84] L. Peter Deutsch and Alan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, UT, January 1984.
- [Dya95] Lev J. Dyadkin. Multibox parsers: No more handwritten lexical analyzers. *IEEE Software*, 12(5):61–67, September 1995.
- [Ear70] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.
- [ECM99] ECMA International, Geneva, Switzerland. *ECMAScript Language Specification*, third edition, December 1999. ECMA-262, ISO/IEC 16262. Available as [www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf](http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf).
- [ECM02] ECMA International, Geneva, Switzerland. *C# Language Specification*, second edition, December 2002. ECMA-334, ISO/IEC 23270. Available as [www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf](http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf).
- [Eng84] Joost Engelfriet. Attribute grammars: Attribute evaluation methods. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction: An Advanced Course*, pages 103–138. Cambridge University Press, Cambridge, England, 1984.
- [ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.
- [Eve63] R. James Evey. Application of pushdown store machines. In *Proceedings of the 1963 Fall Joint Computer Conference*, pages 215–227, Las Vegas, NV, November 1963. AFIPS Press, Montvale, NJ.
- [FCO90] John T. Feo, David Cann, and Rod R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–365, December 1990.
- [FG84] Alan R. Feuer and Narain Gehani, editors. *Comparing and Assessing Programming Languages: Ada, C, Pascal*. Prentice-Hall Software Series. Prentice-Hall, Englewood Cliffs, NJ, 1984.
- [FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, Redwood City, CA, 1995.
- [FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Pearson Education, Upper Saddle River, NJ, 1999.
- [Fin96] Raphael A. Finkel. *Advanced Programming Language Design*. Addison-Wesley, Menlo Park, CA, 1996.
- [FL80] Charles N. Fischer and Richard J. LeBlanc, Jr. Implementation of runtime diagnostics in Pascal. *IEEE Transactions on Software Engineering*, SE-6(4):313–319, July 1980.
- [FL88] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a Compiler*. Benjamin/Cummings, Menlo Park, CA, 1988.
- [Fle76] Arthur C. Fleck. On the impossibility of content exchange through the by-name parameter transmission technique. *ACM SIGPLAN Notices*, 11(11):38–41, November 1976.
- [FMQ80] Charles N. Fischer, Donn R. Milton, and Sam B. Quiring. Efficient LL(1) error correction and recovery using only insertions. *Acta Informatica*, 13(2):141–154, February 1980.
- [Fos95] Ian Foster. Compositional C++. In *Debugging and Building Parallel Programs*, chapter 5, pages 167–204. Addison-Wesley, Reading, MA, 1995. Available in hypertext at [www.mcs.anl.gov/dbpp/text/node51.html](http://www.mcs.anl.gov/dbpp/text/node51.html).
- [Fra80] Nissim Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, January 1980.
- [FSS83] Stefan M. Freudenberger, Jacob T. Schwartz, and Micha Sharir. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems*, 5(1):26–45, January 1983.
- [FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, second edition, 2001.
- [FY69] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.
- [Gar70] Martin Gardner. The fantastic combinations of John Conway’s new solitaire game “Life.” *Scientific American*, 223(4):120–123, October 1970.
- [GBD<sup>+</sup>94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine: A Users’ Guide and Tutorial for Networked Parallel Computing*.

- Scientific and Engineering Computation series. MIT Press, Cambridge, MA, 1994. Available in hypertext at [www.netlib.org/pvm3/book/pvm-book.html](http://www.netlib.org/pvm3/book/pvm-book.html).
- [GBJL01] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design*. John Wiley & Sons, Chichester, England, 2001.
- [GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the Twelfth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 108–124, Atlanta, GA, October 1997. In *ACM SIGPLAN Notices*, 32(10), October 1997.
- [GFH82] Mahadevan Ganapathi, Charles N. Fischer, and John L. Hennessy. Retargetable compiler code generation. *ACM Computing Surveys*, 14(4):573–592, December 1982.
- [GG78] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 231–240, Tucson, AZ, January 1978.
- [GG89] Ron Goldman and Richard P. Gabriel. Qlisp: Parallel processing in Lisp. *IEEE Software*, 6(4):51–59, July 1989.
- [GG96] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, San Jose, CA, third edition, 1996. Out of print; available online at [www.cs.arizona.edu/icon/lb3.htm](http://www.cs.arizona.edu/icon/lb3.htm). Previous editions published by Prentice-Hall.
- [GJL<sup>+</sup>03] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *Proceedings of the Eighteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 115–134, Anaheim, CA, October 2003.
- [GJS96] James Gosling, Bill Joy, and Guy L. Steele, Jr. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, 1.0 edition, 1996. Available in hypertext at [java.sun.com/docs/books/jls/html/index.html](http://java.sun.com/docs/books/jls/html/index.html).
- [GL03] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. *Science of Computer Programming*, 58(1–2):83–114, October 2005.
- [GLDW87] Robert A. Gingell, Meng Lee, Xuong T. Dang, and Mary S. Weeks. Shared libraries in SunOS. In *Proceedings of the 1987 Summer USENIX Conference*, pages 131–145, Phoenix, AZ, June 1987.
- [GM86] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, Palo Alto, CA, July 1986. In *ACM SIGPLAN Notices*, 21(7), July 1986.
- [Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1984.
- [Gol03] David Goldberg. Computer arithmetic. Appendix H of Hennessy and Patterson [HP03]. Available at [books.elsevier.com/companions/1558605967/appendices/1558605967-appendix-h.pdf](http://books.elsevier.com/companions/1558605967/appendices/1558605967-appendix-h.pdf).
- [Goo75] John B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.
- [Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Springer-Verlag, New York, NY, 1979.
- [GPP71] Ralph E. Griswold, J. F. Poage, and I. P. Polonsky. *The Snobol4 Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1971.
- [GR62] Seymour Ginsburg and H. Gordon Rice. Two families of languages related to ALGOL. *Journal of the ACM*, 9(3):350–371, 1962.
- [GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1989.
- [Gri81] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1981.
- [GSB<sup>+</sup>93] William E. Garrett, Michael L. Scott, Ricardo Bianchini, Leonidas I. Kontothanassis, R. Andrew McCallum, Jeffrey A. Thomas, Robert Wisniewski, and Steve Luk. Linking shared segments. In *Proceedings of the USENIX Winter '93 Technical Conference*, pages 13–27, San Diego, CA, January 1993.
- [GTW78] Joseph A. Goguen, James W. Thatcher, and Eric G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice-Hall, Englewood Cliffs, NJ, 1978.
- [Gut77] John Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, June 1977.
- [GWEB83] Gerhard Goos, William A. Wulf, Arthur Evans, Jr., and Kenneth J. Butler, editors. *DIANA: An Intermediate Language for Ada*, volume 161 of *Lecture Notes*

- in Computer Science*. Springer-Verlag, Berlin, West Germany, 1983.
- [Hal85] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
- [Han81] David R. Hanson. Is block structure necessary? *Software—Practice and Experience*, 11(8):853–866, August 1981.
- [Han93] David R. Hanson. A brief introduction to Icon. In *Proceedings of the Second ACM SIGPLAN History of Programming Languages Conference*, pages 359–360, Cambridge, MA, April 1993. In *ACM SIGPLAN Notices*, 28(3), March 1993.
- [Har92] Samuel P. Harbison. *Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [HD68] E. A. Hauck and B. A. Dent. Burroughs' B6500/B7500 stack mechanism. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 32, pages 245–251, 1968. Reprinted as pages 244–250 of Siewiorek, Bell, and Newell [SBN82].
- [HD89] Robert R. Henry and Peter C. Damron. Algorithms for table-driven code generators using tree-pattern matching. Technical Report 89-02-03, Computer Science Department, University of Washington, Seattle, WA, February 1989.
- [Her91] Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the Eighteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Anaheim, CA, October 2003.
- [HGLS78] Richard C. Holt, G. Scott Graham, Edward D. Lazowska, and Mark A. Scott. *Structured Concurrent Programming with Operating Systems Applications*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1978.
- [HH97a] Michael Hauben and Ronda Hauben. *Netizens: On the History and Impact of Usenet and the Internet*. Wiley/IEEE Computer Society Press, New York, NY, 1997. Available at [www.columbia.edu/~hauben/netbook/](http://www.columbia.edu/~hauben/netbook/).
- [HH97b] Raymond J. Hookway and Mark A. Herdeg. DIGITAL FX!32: Combining emulation and binary translation. *DIGITAL Technical Journal*, 9(1):3–12, 1997.
- [Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, Snowbird, UT, June 2001. Available as [www.research.ibm.com/people/h/hind/paste01.ps](http://www.research.ibm.com/people/h/hind/paste01.ps).
- [HJBG81] John L. Hennessy, Norman Jouppi, Forest Bassett, and John Gill. MIPS: A VLSI processor architecture. In *Proceedings of the CMU Conference on VLSI Systems and Computations*, pages 337–346. Computer Science Press, Rockville, MD, October 1981.
- [HL94] Patricia M. Hill and John W. Lloyd. *The Gödel Programming Language*. Logic Programming Series. MIT Press, Cambridge, MA, 1994.
- [HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, Boston, MA, July 2003.
- [HMRC88] Richard C. Holt, Philip A. Matthews, J. Alan Rosselet, and James R. Cordy. *The Turing Programming Language: Design and Definition*. Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Boston, MA, second edition, 2001.
- [HO91] W. Wilson Ho and Ronald A. Olsson. An approach to genuine dynamic linking. *Software—Practice and Experience*, 21(4):375–390, April 1991.
- [Hoa69] Charles Antony Richard Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580+, October 1969.
- [Hoa74] Charles Antony Richard Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.
- [Hoa75] Charles Antony Richard Hoare. Recursive data structures. *International Journal of Computer and Information Sciences*, 4(2):105–132, June 1975.
- [Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
- [Hoa81] Charles Antony Richard Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, February 1981. The 1980 Turing Award lecture.
- [Hoa89] Charles Antony Richard Hoare. Hints on programming language design. In Cliff B. Jones, editor, *Essays in Computing Science*, pages 193–216. Prentice-Hall, New York, NY, 1989. Based on a keynote address presented at the *First ACM Symposium on Principles of Programming Languages*, Boston, MA, October 1973.

- [Hor51] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, March 1951.
- [Hor87] Ellis Horowitz. *Programming Languages: A Grand Tour*. Computer Software Engineering Series. Computer Science Press, Rockville, MD, third edition, 1987.
- [HP03] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, third edition, 2003.
- [Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.
- [Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.
- [HWG04] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Microsoft .NET Development Series. Addison-Wesley, Boston, MA, 2004.
- [IBFW91] Jean Ichbiah, John G. P. Barnes, Robert J. Firth, and Mike Woodger. *Rationale for the Design of the Ada Programming Language*. Ada Companion Series. Cambridge University Press, Cambridge, England, 1991.
- [IBM87] IBM Corporation. *APL2 Programming: Language Reference*, 1987. SH20-9227.
- [IEE87] IEEE Standards Committee. IEEE standard for binary floating-point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, February 1987. Standard adopted by IEEE, March 1985; by ANSI, July 1985.
- [Ing61] Peter Z. Ingerman. Thunks: A way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, January 1961.
- [Ins91] Institute of Electrical and Electronics Engineers, New York, NY. *IEEE/ANSI Standard for the Scheme Programming Language*, 1991. IEEE 1178-1990. Available at [standards.ieee.org/reading/ieee/std\\_public/description/busarch/1178-1990\\_desc.html](http://standards.ieee.org/reading/ieee/std_public/description/busarch/1178-1990_desc.html).
- [Int90] International Organization for Standardization, Geneva, Switzerland. *Information Technology—Programming Languages—Pascal*, 1990. ISO/IEC 7185:1990 (revision and redesignation of ANSI/IEEE 770X).
- [Int95a] Intermetrics, Inc., Cambridge, MA. *Ada 95 Rationale*, 1995.
- [Int95b] International Organization for Standardization, Geneva, Switzerland. *Information Technology—Programming Languages—Ada*, 1995. ISO/IEC 8652:1995 (E). Available in hypertext at [www.adahome.com/rm95/](http://www.adahome.com/rm95/).
- [Int95c] International Organization for Standardization, Geneva, Switzerland. *Information Technology—Programming Languages—Prolog—Part 1: General Core*, 1995. ISO/IEC 13211-1:1995.
- [Int96] International Organization for Standardization, Geneva, Switzerland. *Information Technology—Programming Languages—Part 1: Modula-2, Base Language*, 1996. ISO/IEC 10514-1:1996.
- [Int97] International Organization for Standardization, Geneva, Switzerland. *Programming Language Forth*, 1997. ISO/IEC 15145:1997 (revision and redesignation of ANSI X3.215-1994).
- [Int98] International Organization for Standardization, Geneva, Switzerland. *Programming Languages—C++*, 1998. ISO/IEC 14882:1998.
- [Int99] International Organization for Standardization, Geneva, Switzerland. *Programming Language—C*, December 1999. ISO/IEC 9899:1999(E).
- [Int03a] International Organization for Standardization, Geneva, Switzerland. *The C Rationale*, April 2003. ISO/IEC JTC 1/SC 22/WG 14, revision 5.10.
- [Int03b] International Organization for Standardization, Geneva, Switzerland. *Information Technology—Portable Operating System Interface (POSIX)*, fourth edition, August 2003. ISO/IEC 9945-1:2003. Also IEEE standard 1003.1, 2004 Edition, and The Open Group Technical Standard Base Specifications, issue 6. Available as [www.opengroup.org/onlinelibrary/009695399/](http://www.opengroup.org/onlinelibrary/009695399/).
- [Ive62] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, New York, NY, 1962.
- [JG89] Geraint Jones and Michael Goldsmith. *Programming in occam2*. Prentice-Hall International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1989.
- [JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, New York, NY, 1996.
- [JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
- [JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 20:503–581, May–July 1994.
- [Joh75] Stephen C. Johnson. Yacc—Yet another compiler compiler. Technical Report 32, Computing Science, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- [JOR75] Mehdi Jazayeri, William F. Ogden, and William C. Rounds. The intrinsically exponential complexity of the

- circularity problem for attribute grammars. *Communications of the ACM*, 18(12):697–706, December 1975.
- [Jor85] Harry F. Jordan. HEP architecture, programming and performance. In Janusz S. Kowalik, editor, *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*, pages 1–40. MIT Press, Cambridge, MA, 1985.
- [JPAR68] Walter L. Johnson, James H. Porter, Stephanie I. Ackley, and Douglas T. Ross. Automatic generation of efficient lexical processors using finite state techniques. *Communications of the ACM*, 11(12):805–813, December 1968.
- [JW91] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report: ISO Pascal Standard*. Springer-Verlag, New York, NY, fourth edition, 1991. Revised by Andrew B. Mickel and James F. Miner. ISBN 0-387-97649-3.
- [Kas65] T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.
- [Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer’s Guide to CLOS*. Addison-Wesley, Reading, MA, 1989. Contributions by Dan Gerzon.
- [Kep04] Stephan Kepser. A simple proof for the Turing-completeness of XSLT and XQuery. In *Proceedings, Extreme Markup Languages 2004*, Montréal, Canada, August 2004. Available as [www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01/EML2004Kepser01.html](http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01/EML2004Kepser01.html).
- [Ker81] Brian W. Kernighan. Why Pascal is not my favorite programming language. Technical Report 100, Computing Science, AT&T Bell Laboratories, Murray Hill, NJ, 1981. Reprinted as pages 170–186 of Feuer and Gehani [FG84].
- [Kes77] J. L. W. Kessels. An alternative to event queues for synchronization in monitors. *Communications of the ACM*, 20(7):500–503, July 1977.
- [KKR<sup>+</sup>86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN ’86 Symposium on Compiler Construction*, pages 219–233, Palo Alto, CA, June 1986. In *ACM SIGPLAN Notices*, 21(7), July 1986.
- [Kle56] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In Claude E. Shannon and John McCarthy, editors, *Automata Studies*, number 34 in Annals of Mathematical Studies, pages 3–41. Princeton University Press, Princeton, NJ, 1956.
- [KLS<sup>+</sup>94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1994.
- [KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, June 1977.
- [Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, December 1965.
- [Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction appears in Volume 5, pages 95–96.
- [Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.
- [Kor94] David G. Korn. ksh: An extensible high level language. In *Proceedings of the USENIX Very High Level Languages Symposium*, pages 129–146, Santa Fe, NM, October 1994.
- [KP76] Brian W. Kernighan and Phillip J. Plauger. *Software Tools*. Addison-Wesley, Reading, MA, 1976.
- [KP78] Brian W. Kernighan and Phillip J. Plauger. *The Elements of Programming Style*. McGraw-Hill, New York, NY, second edition, 1978.
- [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1988.
- [Krá73] Jaroslav Král. The equivalence of modes and the equivalence of finite automata. *ALGOL Bulletin*, 35:34–35, March 1973.
- [KS01] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of the SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 1–12, Snowbird, UT, June 2001. In *ACM SIGPLAN Notices*, 36(5), May 2001.
- [KWS97] Leonidas I. Kontothanassis, Robert Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.
- [Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

- [Lam94] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley Professional, Reading, MA, second edition, 1994.
- [Les75] Michael E. Lesk. Lex—A lexical analyzer generator. Technical Report 39, Computing Science, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- [LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1986.
- [LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [LHL<sup>+</sup>77] Butler W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. *ACM SIGPLAN Notices*, 12(2):1–79, February 1977.
- [Llo87] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, West Germany, second edition, 1987.
- [Lom75] David B. Lomet. Scheme for invalidating references to freed storage. *IBM Journal of Research and Development*, 19(1):26–35, January 1975.
- [Lom85] David B. Lomet. Making pointers safe in system programming languages. *IEEE Transactions on Software Engineering*, SE-11(1):87–96, January 1985.
- [Lou03] Kenneth C. Louden. *Programming Languages: Principles and Practice*. Brooks/Cole–Thomson Learning, Pacific Grove, CA, second edition, 2003.
- [LP80] David C. Luckham and W. Polak. Ada exception handling: An axiomatic approach. *ACM Transactions on Programming Languages and Systems*, 2(2):225–233, April 1980.
- [LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.
- [LRS74] Philip M. Lewis II, Daniel J. Rosenkrantz, and Richard E. Stearns. Attributed translations. *Journal of Computer and System Sciences*, 9(3):279–307, December 1974.
- [LS68] Philip M. Lewis II and Richard E. Stearns. Syntax-directed transduction. *Journal of the ACM*, 15(3):465–488, July 1968.
- [LS79] Barbara Liskov and Alan Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6):546–558, November 1979.
- [LS83] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.
- [LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN ’95 Conference on Programming Language Design and Implementation*, pages 291–300, La Jolla, CA, June 1995. In *ACM SIGPLAN Notices*, 30(6), June 1995.
- [LSAS77] Barbara Liskov, Alan Snyder, Russel Atkinson, and J. Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.
- [LT02] Rasmus Lerdorf and Kevin Tatroe. *Programming PHP*. O’Reilly and Associates, Cambridge, MA, 2002.
- [Lv77] C. H. Lindsey and S. G. van der Meulen. *Informal Introduction to ALGOL 68*. North-Holland, Amsterdam, the Netherlands, revised edition, 1977.
- [LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, 1997.
- [Mac77] M. Donald MacLaren. Exception handling in PL/I. In David B. Wortman, editor, *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 101–104, Raleigh, NC, 1977. In *ACM SIGPLAN Notices*, 12(3), March 1977.
- [MAE<sup>+</sup>65] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer’s Manual*. MIT Press, Cambridge, MA, second edition, 1965.
- [Mai90] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 382–401, San Francisco, CA, January 1990.
- [Mas76] John R. Mashey. Using a command language as a high-level programming language. In *Proceedings of the Second International IEEE Conference on Software Engineering*, pages 169–176, San Francisco, CA, October 1976.
- [Mas87] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–126, Palo Alto, CA, October 1987. In *ACM SIGPLAN Notices*, 22(10), October 1987.
- [McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.

- [McG82] James R. McGraw. The VAL language: Description and analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, January 1982.
- [McK04] Kathryn S. McKinley, editor. *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1979–1999*. ACM Press, New York, NY, 2004. Also *ACM SIGPLAN Notices*, 39(4), April 2004.
- [MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.
- [Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall Object-Oriented Series. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [Mic89] Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. International Computer Science Series. Addison-Wesley, Wokingham, England, 1989.
- [Mic91] Microsoft Corporation, Redmond, WA. *Microsoft Visual Basic Language Reference*, 1991. Document DB20664-0491.
- [Mic04] Maged Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 35–46, Washington, DC, June 2004.
- [Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.
- [MKH91] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.
- [MLB76] Michael Marcotty, Henry F. Ledgard, and Gre-gor V. Bochmann. A sampler of formal definitions. *ACM Computing Surveys*, 8(2):191–276, June 1976.
- [Moo78] David A. Moon. *MacLisp Reference Manual*. MIT Artificial Intelligence Laboratory, 1978.
- [Moo86] David A. Moon. Object-oriented programming with Flavors. In *OOPSLA '86 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 1–8, Portland, OR, September 1986. In *ACM SIGPLAN Notices*, 21(11), November 1986.
- [Mor70] Howard L. Morgan. Spelling correction in systems programs. *Communications of the ACM*, 13(2):90–94, 1970.
- [MOSS96] Stephan Murer, Stephen Omohundro, David Stoutamire, and Clemens Szyperski. Iteration abstraction in Sather. *ACM Transactions on Programming Languages and Systems*, 18(1):1–15, January 1996.
- [MR96] Michael Metcalf and John Reid. *Fortran 90/95 Explained*. Oxford University Press, London, England, 1996.
- [MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, Philadelphia, PA, May 1996.
- [MS98] Maged M. Michael and Michael L. Scott. Non-blocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51:1–26, 1998.
- [MSA<sup>+</sup>85] James R. McGraw, S. K. Skedzielewski, S. J. Allan, Rod R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. *SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2*. Lawrence Livermore National Laboratory, Livermore, CA, March 1985. Manual M-146, Revision 1.
- [MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML—Revised*. MIT Press, Cambridge, MA, 1997.
- [Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.
- [MYD95] Bruce J. McKenzie, Corey Yeatman, and Lorraine De Vere. Error repair in shift-reduce parsers. *ACM Transactions on Programming Languages and Systems*, 17(4):672–689, July 1995.
- [NBB<sup>+</sup>63] Peter Naur (ed.), J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–23, January 1963. Original version appeared in the May 1960 issue.
- [ND78] Kristen Nygaard and Ole-Johan Dahl. The development of the Simula languages. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN History of Programming Languages Conference*, ACM Monograph Series, 1981, pages 439–493, Los Angeles, CA, June 1978. Academic Press, New York, NY. In *ACM SIGPLAN Notices*, 13(8), August 1978.
- [Nel65] Theodor Holm Nelson. Complex information processing: A file structure for the complex, the changing, and the indeterminate. In *Proceedings of the Twentieth Annual Meeting of the Association for Mathematical Logic*, 1965.

- eth ACM National Conference*, pages 84–100, Cleveland, OH, August 1965.
- [Nel81] Bruce J. Nelson. *Remote Procedure Call*. Ph. D. dissertation, Carnegie-Mellon University, 1981. School of Computer Science Technical Report CMU-CS-81-119.
- [NL91] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.
- [Ope96] Open Software Foundation. *OSF DCE Application Development Reference, Release 1.1*. OSF DCE Series. Prentice-Hall, Upper Saddle River, NJ, 1996.
- [Ous82] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 22–30, Miami/Ft. Lauderdale, FL, October 1982. IEEE Computer Society Press, Silver Spring, MD.
- [Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Professional, Reading, MA, 1994.
- [Ous98] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.
- [Pag76] Frank G. Pagan. *A Practical Guide to Algol 68*. Wiley Series in Computing. John Wiley and Sons, London, England, 1976.
- [Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [Pat85] David A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, January 1985.
- [PD80] David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *Computer Architecture News*, 8(6):25–33, October 1980.
- [PD03] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, San Francisco, CA, third edition, 2003.
- [Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.
- [Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ, 1987.
- [Pey92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spinless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
- [Pey03] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, England, 2003. Available at [haskell.org/definition/](http://haskell.org/definition/).
- [PH05] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware-Software Interface*. Morgan Kaufmann, San Francisco, CA, third edition, 2005.
- [Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.
- [PQ95] Terrence J. Parr and R. W. Quong. ANTLR: A predicated- $LL(k)$  parser generator. *Software—Practice and Experience*, 25(7):789–810, July 1995.
- [Pug00] William Pugh. The Java memory model is fatally flawed. *Concurrency—Practice and Experience*, 12(6):445–455, May 2000.
- [Rad82] George Radin. The 801 minicomputer. In *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, Palo Alto, CA, March 1982. In *ACM SIGPLAN Notices*, 17(4), April 1982.
- [Ram87] S. Ramesh. A new efficient implementation of CSP with output guards. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 266–273, Berlin, West Germany, September 1987. IEEE Computer Society Press, Washington, DC.
- [Rep84] Thomas Reps. *Generating Language-Based Environments*. MIT Press, Cambridge, MA, 1984. Winner of the 1983 ACM Doctoral Dissertation Award.
- [RF93] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *Journal of Supercomputing*, 7(1/2):9–50, May 1993.
- [Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.
- [Rob83] John Alan Robinson. Logic programming—past, present, and future. *New Generation Computing*, 1(2):107–124, 1983.
- [RR64] Brian Randell and Lawford J. Russell, editors. *ALGOL 60 Implementation: The Translation and Use of ALGOL 60 Programs on a Computer*. A.P.I.C. Studies in Data Processing #5. Academic Press, New York, NY, 1964. SBN 12-578150-4.
- [RS59] Michael O. Rabin and Dana S. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.
- [RS70] Daniel J. Rosenkrantz and Richard E. Stearns. Properties of deterministic top-down grammars. *Information and Control*, 17(3):226–256, October 1970.

- [RT88] Thomas Reps and Timothy Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, NY, 1988.
- [Rub87] Frank Rubin. ‘GOTO considered harmful’ considered harmful. *Communications of the ACM*, 30(3):195–196, March 1987. Further correspondence appears in Volume 30, Numbers 6, 7, 8, 11, and 12.
- [Rut67] Heinz Rutishauser. *Description of ALGOL 60*. Springer-Verlag, New York, NY, 1967.
- [RVL<sup>+</sup>97] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, and Brian Bershad. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–8, Seattle, WA, August 1997.
- [RW92] Martin Reiser and Niklaus Wirth. *Programming in Oberon—Steps Beyond Pascal and Modula*. Addison-Wesley, Reading, MA, 1992.
- [SaMC91] Joel H. Saltz, Avi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.
- [SB90] Michael Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.
- [SBG<sup>+</sup>91] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing*. Prentice-Hall Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [SBN82] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. *Computer Structures: Principles and Examples*. McGraw-Hill Computer Science Series. McGraw-Hill, New York, NY, 1982.
- [SCK<sup>+</sup>93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.
- [Sco91] Michael L. Scott. The Lynx distributed programming language: Motivation, design, and experience. *Computer Languages*, 16(3/4):209–233, 1991.
- [SDB84] Mayer D. Schwartz, Norman M. Delisle, and Vimal S. Begwani. Incremental compilation in Magpie. In *Proceedings of the SIGPLAN ’84 Symposium on Compiler Construction*, pages 122–131, Montreal, Quebec, Canada, June 1984. In *ACM SIGPLAN Notices*, 19(6), June 1984.
- [SDDS86] Jacob T. Schwartz, Robert B. K. Dewar, Ed Dubinsky, and Edmond Schonberg. *Programming with Sets: An Introduction to SETL*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1986.
- [SE94] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN ’94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, June 1994. In *ACM SIGPLAN Notices*, 29(6), June 1994.
- [Seb04] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson/Addison-Wesley, Boston, MA, sixth edition, 2004.
- [Set96] Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, Reading, MA, second edition, 1996.
- [SF80] Marvin H. Solomon and Raphael A. Finkel. A note on enumerating binary trees. *Journal of the ACM*, 27(1):3–5, January 1980.
- [SF88] Michael L. Scott and Raphael A. Finkel. A simple mechanism for type security across compilation units. *IEEE Transactions on Software Engineering*, SE-14(8):1238–1239, August 1988.
- [SFL<sup>+</sup>94] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, San Jose, CA, October 1994. In *ACM SIGPLAN Notices*, 29(11), November 1994.
- [SG96] Daniel J. Scales and Kourosh Gharachorloo. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, MA, October 1996. In *ACM SIGPLAN Notices*, 31(9), September 1996.
- [SH92] A. S. M. Sajeev and A. John Hurst. Programming persistence in  $\chi$ . *IEEE Computer*, 25(9):57–66, September 1992.
- [Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989. Correction appears in Volume 21, Number 4.
- [Sie96] Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley and Sons, New York, NY, 1996.
- [Sip97] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, Boston, MA, 1997.

- [Sit72] Richard L. Sites. Algol W reference manual. Technical Report STAN-CS-71-230, Computer Science Department, Stanford University, Stanford, CA, February 1972.
- [SOHL<sup>+</sup>98] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, William Gropp, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, and William Saphir. *MPI: The Complete Reference*. Scientific and Engineering Computation series. MIT Press, Cambridge, MA, second edition, 1998. Two-volume set. First edition available in hypertext at [www.netlib.org/utk/papers/mpi-book/mpi-book.html](http://www.netlib.org/utk/papers/mpi-book/mpi-book.html).
- [Sri95] Raj Srinivasan. RPC: Remote procedure call protocol specification version 2. Internet Request for Comments #1831, August 1995. Available at [www.rfc-archive.org/getrfc.php?rfc=1831](http://www.rfc-archive.org/getrfc.php?rfc=1831).
- [SS71] Dana S. Scott and Christopher Strachey. Toward a mathematical semantics for computer language. In Jerome Fox, editor, *Proceedings, Symposium on Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn Press, New York, NY, 1971.
- [SS89] Richard Snodgrass and Karen P. Shannon. *The Interface Description Language: Definition and Use*. Computer Science Press, Rockville, MD, 1989.
- [Sta92] Ryan D. Stansifer. *ML Primer*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [Sta95] Ryan D. Stansifer. *The Study of Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ, 1995.
- [Ste90] Guy L. Steele, Jr. *Common Lisp—The Language*. Digital Press, Bedford, MA, second edition, 1990. Available in hypertext at [www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html](http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html).
- [Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, volume 1 of *MIT Press Series in Computer Science*. MIT Press, Cambridge, MA, 1977.
- [Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, second edition, 1991.
- [Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, third edition, 1997.
- [Sun90] Vaidyalingam S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency—Practice and Experience*, 2(4):315–339, December 1990.
- [Sun97] Sun Microsystems, Mountain View, CA. *Javabeans*, July 1997. Available at [java.sun.com/products/javabeans/docs/spec.html](http://java.sun.com/products/javabeans/docs/spec.html).
- [Sun04] Håkan Sundell. *Efficient and Practical Non-Blocking Data Structures*. Ph.D. dissertation, Department of Computing Science, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, 2004. Available as [www.cs.chalmers.se/~tsigas/papers/Haakan-Thesis.pdf](http://www.cs.chalmers.se/~tsigas/papers/Haakan-Thesis.pdf).
- [SW67] Herbert Schorr and William M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.
- [SW94] James E. Smith and Shlomo Weiss. PowerPC 601 and Alpha 21064: A tale of two RISCs. *IEEE Computer*, 27(6):46–58, June 1994.
- [SZ90] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):54–64, May 1990.
- [SZBH86] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, October 1986.
- [Tan78] Andrew S. Tanenbaum. A comparison of Pascal and ALGOL 68. *The Computer Journal*, 21(4):316–323, November 1978.
- [Tan02] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, Upper Saddle River, NJ, fourth edition, 2002.
- [TH04] David Thomas and Andrew Hunt. *Programming Ruby—The Pragmatic Programmer’s Guide*. The Pragmatic Programmers, LLC, Raleigh, NC, second edition, 2004. First edition available in hypertext at [www.rubycentral.com/book/](http://www.rubycentral.com/book/).
- [Tic86] Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.
- [Tic91] Evan Tick. *Parallel Logic Programming*. Logic Programming Series. MIT Press, Cambridge, MA, 1991.
- [TM81] Warren Teitelman and Larry Masinter. The Interlisp programming environment. *IEEE Computer*, 14(4):25–33, April 1981.
- [TR81] Timothy Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.
- [Tur86] David A. Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, December 1986.
- [UKGS94] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Experiences with locking in a NUMA multiprocessor operating system kernel. In *Proceedings of the First USENIX Symposium on Operat-*

- ing Systems Design and Implementation*, pages 139–152, Monterey, CA, November 1994.
- [Ull85] Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(3):289–321, September 1985.
- [Uni60] United States Department of Defense. *COBOL, Initial Specifications for a Common Business Oriented Language*, 1960. Revised in 1961 and again in 1962.
- [Uni03] University of Chicago Press Staff. *The Chicago Manual of Style*. University of Chicago Press, Chicago, IL, fifteenth edition, 2003.
- [UW97] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems*. Prentice-Hall, Upper Saddle River, NJ, 1997.
- [VF82] Thomas R. Virgilio and Raphael A. Finkel. Binding strategies and scope rules are independent. *Computer Languages*, 7(2):61–67, 1982.
- [VF94] Jack E. Veenstra and Robert J. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 201–207, Durham, NC, January 1994. IEEE Computer Society Press, Los Alamitos, CA.
- [vMP<sup>+</sup>75] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5(1–3):1–236, 1975. Also *ACM SIGPLAN Notices*, 12(5):1–70, May 1977.
- [vRD03] Guido van Rossum and Fred L. Drake, Jr. (Editor). *The Python Language Reference Manual*. Network Theory, Ltd., Bristol, UK, 2003.
- [Wad97] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997.
- [Wad98a] Philip Wadler. An angry half-dozen. *ACM SIGPLAN Notices*, 33(2):25–30, February 1998. (NB: Table of contents on cover of issue is incorrect.)
- [Wad98b] Philip Wadler. Why no one uses functional languages. *ACM SIGPLAN Notices*, 33(8):23–27, August 1998.
- [Wat77] David Anthony Watt. The parsing problem for affix grammars. *Acta Informatica*, 8(1):1–20, 1977.
- [WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly and Associates, Cambridge, MA, third edition, 2000.
- [Web89] Fred Webb. Fortran story—The real scoop. Submitted to `alt.folklore.computers`, 1989. Quoted by Mark Brader in the ACM *RISKS* online forum, volume 9, issue 54, December 12, 1989.
- [Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990. Expanded version of the keynote address from *OOPSLA '89*.
- [Wet78] Horst Wettstein. The problem of nested monitor calls revisited. *ACM Operating Systems Review*, 12(1):19–23, January 1978.
- [Wex78] Richard L. Wexelblat, editor. *Proceedings of the ACM SIGPLAN History of Programming Languages Conference*, ACM Monograph Series, 1981, Los Angeles, CA, June 1978. Academic Press, New York, NY. In *ACM SIGPLAN Notices*, 13(8), August 1978.
- [WH66] Niklaus Wirth and Charles Antony Richard Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9(6):413–431, June 1966.
- [WHA03] Damien Watkins, Mark Hammond, and Brad Abrams. *Programming in the .NET Environment*. Microsoft .NET Development Series. Addison-Wesley, Boston, MA, 2003.
- [Wil92a] Paul R. Wilson. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992.
- [Wil92b] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer Verlag, Berlin, West Germany, 1992. Workshop held at St. Malo, France, September 1992. Expanded version available as <ftp://ftp.cs.utexas.edu/pub/garbage/big surv.ps>.
- [Wir71] Niklaus Wirth. The programming language Pascal. *Acta Informatica*, 1(1):35–63, 1971.
- [Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1976.
- [Wir77a] Niklaus Wirth. Design and implementation of Modula. *Software—Practice and Experience*, 7(1):67–84, January–February 1977.
- [Wir77b] Niklaus Wirth. Modula: A language for modular multiprogramming. *Software—Practice and Experience*, 7(1):3–35, January–February 1977.
- [Wir80] Niklaus Wirth. The module: A system structuring facility in high-level programming languages. In Jeffrey M. Tobias, editor, *Language Design and Programming Methodology*, volume 79 of *Lecture Notes in Computer*

- Science*, pages 1–24. Springer Verlag, Berlin, West Germany, 1980. Proceedings of a symposium held at Sydney, Australia, September 1979.
- [Wir85a] Niklaus Wirth. From programming language design to computer construction. *Communications of the ACM*, 28(2):159–164, February 1985. The 1984 Turing Award lecture.
- [Wir85b] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, third corrected edition, 1985.
- [Wir88a] Niklaus Wirth. From Modula to Oberon. *Software—Practice and Experience*, 18(7):661–670, July 1988.
- [Wir88b] Niklaus Wirth. The programming language Oberon. *Software—Practice and Experience*, 18(7):671–690, July 1988.
- [Wir88c] Niklaus Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988. Relevant correspondence appears in Volume 13, Number 4.
- [WJ93] Paul R. Wilson and Mark S. Johnstone. Real-time non-copying garbage collection. In *OOPSLA ’93 Workshop on Memory Management and Garbage Collection*, Washington, DC, September 1993.
- [WJH03] Brent B. Welch, Ken Jones, and Jeffrey Hobbs. *Practical Programming in Tcl and Tk*. Prentice-Hall, Upper Saddle River, NJ, fourth edition, 2003. Sample chapters from previous editions available at [www.beedub.com/book/](http://www.beedub.com/book/).
- [WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, Wokingham, England, 1995. Translated from the German by Stephen S. Wilson.
- [WMWM87] Janet H. Walker, David A. Moon, Daniel L. Weinreb, and Mike McMahon. The Symbolics Genera programming environment. *IEEE Software*, 4(6):36–45, November 1987.
- [Wol96] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.
- [Wor01] World Wide Web Consortium. *Extensible Stylesheet Language XSL Version 1.0*, October 2001. Available as [www.w3.org/TR/2001/REC-xsl-200111015/](http://www.w3.org/TR/2001/REC-xsl-200111015/).
- [Wor04a] World Wide Web Consortium. *XML Path Language (XPath) 2.0*, October 2004. Working draft; available as [www.w3.org/TR/xpath20/](http://www.w3.org/TR/xpath20/).
- [Wor04b] World Wide Web Consortium. *XSL Transformations (XSLT) Version 2.0*, November 2004. Working draft; available as [www.w3.org/TR/2004/WD-xslt20-20041105/](http://www.w3.org/TR/2004/WD-xslt20-20041105/).
- [Wor05] World Wide Web Consortium. *Character Model for the World Wide Web 1.0: Fundamentals*, February 2005. Available as [www.w3.org/TR/2005/REC-charmod-20050215/](http://www.w3.org/TR/2005/REC-charmod-20050215/).
- [WRH71] William A. Wulf, Donald B. Russel, and A. Nico Habermann. Bliss: A language for systems programming. *Communications of the ACM*, 14(12):780–790, December 1971.
- [WSH77] Jim Welsh, W. J. Sneeringer, and Charles Antony Richard Hoare. Ambiguities and insecurities in Pascal. *Software—Practice and Experience*, 7(6):685–696, November–December 1977.
- [YA93] Jae-Heon Yang and James H. Anderson. Fast, scalable synchronization with minimal hardware support (extended abstract). In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, pages 171–182, Ithaca, NY, August 1993.
- [You67] Daniel H. Younger. Recognition and parsing of context-free languages in time  $n^3$ . *Information and Control*, 10(2):189–208, February 1967.
- [YTEM04] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth USENIX Symposium on Operating Systems Design and Implementation*, pages 273–288, San Francisco, CA, December 2004.
- [ZHL<sup>+</sup>89] Benjamin G. Zorn, Kimson Ho, James Larus, Luigi Semenzato, and Paul Hilfinger. Multiprocessing extensions in Spur Lisp. *IEEE Software*, 6(4):41–49, July 1989.
- [Zho96] Neng-Fa Zhou. Parameter passing and control stack management in Prolog implementation revisited. *ACM Transactions on Programming Languages and Systems*, 18(6):752–779, November 1996.



# Index

A-lists, 532  
Abelson, Harold, 159, 558, 757  
abstract, definition, 103  
abstraction, 231. *See also* control abstractions  
    conceptual load, 471  
    control abstraction, 104  
    data, 407  
    definition, 103  
    fault containment, 471  
    independence, 471  
    procedural, 233  
    stack abstractions, 126  
abstraction-based point of view, types, 311  
abstractions  
    bounded buffers, 626–627  
    data abstractions, classes, 104  
    extensions, 471  
    refinements, 471  
acknowledgement messages,  
    concurrency and, 648–649  
action routines, 179–181  
actual parameters, subroutines, 407, 417–418  
ad hoc polymorphism, 148  
ad hoc translation schemes, 179  
Ada  
    overview, 793, 795  
    parameter modes, 422–423  
    text I/O, 98–99–CD  
Ada 95, synchronization, 634–635  
addresses  
    array memory, 361–364  
    assignment to names, assemblers and, 780–781  
    data representation and, 199

dynamic libraries, 776  
files, 776  
heaps, 776  
space organization, 775–776  
stacks, 776  
addressing mode, 111  
    displacement, 201  
    indexed, 201  
instruction set architecture, 201–202  
    register indirect addressing, 201  
aggregates, functional programming, 526  
Aho, Alfred V., 35, 194, 686–687, 790  
Algol, limited extent, 141  
Algol 60, overview, 795  
Algol 68, overview, 795  
Algol W, overview, 795  
algorithms  
    CYK (Cocke–Younger–Kasami), 61  
    Earley, Jay, 61  
aliases  
    alias analysis, 165  
    binding with scopes, 142–143  
alignment, 779  
Allen, Frances, 259–CD, 358  
allocation, arrays, 353–357  
Almasi, George S., 670  
Alpern, Bowen, 259–CD  
alphabet, formal language and, 94  
ambiguity, 44–45  
    predict–predict conflict, 76–77  
Anderson, Thomas E., 669  
Andrews, Gregory R., 669, 801  
annotations, semantic analysis, 27  
anthropomorphism, 512  
antidependence, 213

APL  
    dynamic scoping, 132  
    overview, 795  
Appel, Andrew W., 35, 194, 258–CD, 558, 790  
applicative evaluation, 291–295  
applicative-order evaluation, 539–540  
apply function, Scheme, 535–536  
architecture  
    CISC, 196  
    examples, 208–209  
    implementation and, 204–210  
    microprocessors and, 206  
    microprogramming, 205–206  
    pseudo-assembly notation, 209–210  
    RISC, 207–208  
instruction set, 201–204  
    addressing modes, 201–202  
    branches, 202–204  
    conditions, 202–204  
MIPS, 208–209  
multiprocessors, 597–600  
    distributed memory, 598  
processor architecture, 195  
RISC, 196  
x86, 208–209  
arguments  
    subroutines and, 109  
    variable number of, 429–431  
arithmetic operations  
    Prolog, 565–566  
    semantic analysis and, 27  
array operations, 352–353  
arrays, 377  
    allocation, 353–357  
    associative, 349  
    bounds, 353–357

- C, 376
- component types, 349
- composite types, 318
- conformant, 355–356
  - parameter passing and, 427
- dimensions, 353–357
- dynamic, 356–357
- element types, 349
- index types, 349
- indices, 363
- memory layout, 358–365
  - addresses, 361–364
  - dope vectors, 364–365
  - row-pointer layout, 359–364
- operations, 349–353
- semantic analysis and, 27
- shapes, 353–355
- slices, 352
- syntax, 349–353
  - declarations and, 350–352
- assemblers, 12
  - address assignment, to names, 780–781
  - implementation and, 759
  - instructions, 778–780
  - introduction, 4
  - tasks, 776–777
- assembly languages
  - compilers and, 16
  - introduction, 3–4
- assertions, 164
- assignment, 493
  - addresses to names, 780–781
  - expressions, 238–246
    - boxing, 241–242
    - multiway, 245
    - operators, 243–245
    - orthogonality, 242–243
    - references, 239–241
    - values, 239–241
  - initialization, 494
  - reverse assignment, 504
  - Scheme, 533–534
  - types, 393–395
- Association for Computing Machinery, 35
- association list
  - dynamic scoping and, 135
  - scope implementation, 27–29-CD
- associative arrays, 349
  - awk, 686
- associativity
  - caches, 211
  - expressions and, 236–238
  - operators, 45
- AST (abstract syntax tree), 27
- attribute evaluators, 179
- attribute flow, 162
  - declarative notations, 172
  - L-attributed, 173
  - synthesized attributes and, 168
  - translation schemes and, 173
- attribute grammars, 2
  - attributes and, 166
  - code generation, 769–772
  - copy rules, 166
  - noncircular, 173
  - one-pass compilers, 174–175
  - S-attributed, 168, 173–174
  - semantic analysis and, 162
  - semantic functions, 166
  - syntax trees, 175–176
  - syntax trees and, 182
  - translation schemes, 173
  - well-defined, 173
- attributes
  - bottom-up parsers, 181
  - inherited, 40–41-CD, 169–172
  - space management, 181–182
    - bottom-up evaluation, 39–44-CD
    - synthesized, 168–169
  - top-down parsers, 181
- Auslander, Marc, 259-CD
- automata theory, 94
- automatic header inference, separate compilation, 33–34-CD
- awk, text processing and, 686–687
- axioms, logic programming, 560
- back end
  - compiler structure, phases, 762–766
  - introduction, 22
- backoff strategy, 621
- backtracking, Prolog, 566
- Backus, John, 42, 102, 558, 797
- backward chaining, Prolog, 566–569
- Bacon, David F., 259-CD, 518
- Banerjee, Utpal, 259-CD
- Bar-Hillel, Yehoshua, 102
- Barendregt, Hendrik Pieter, 558
- base classes, 478
  - fragile base class problem, 504
  - methods, 479–480
  - protected keyword, 485
- bash (Bourne again shell), 678
- Basic
  - comments, performance and, 16
    - overview, 795
  - batch processing, 591
  - batches, scripting and, 674
  - Bauer, F.L., 102
  - Beck, Leland L., 790
  - Bell, C. Gordon, 227
- Bernstein, Robert L., 305
- best fit algorithm, 111
- big-endian organization, 199
- binary integers, data representation and, 199
- binary operations, RISC machines, 201
- binary semaphores, 628
- binding, 2, 104–105
  - deep binding, 115, 138
  - dynamic method binding, 497–500
  - dynamic scope rules, 115
  - exercises, 36–37-CD
  - explorations, 38-CD
  - lifetime, 106–114
  - referencing environment, 115, 136–142
- rules, 115
- rules and extent, 141
- Scheme, 530–531
- scopes and, 142
  - aliases, 142–143
  - overloading, 143–145
  - polymorphism, 145–148
- shallow binding, 115, 137
- static method binding, 499
- static scope rules, 115, 139
- storage allocation, 107
- subroutines, 138
- time, 105
  - compile time and, 105
  - definition, 104
  - introduction, 104–105
  - language design and, 105
  - language implementation and, 105
  - link time and, 105
  - load time and, 105
  - program writing and, 105
  - run time and, 106
- Birtwistle, Graham M., 468
- blocking synchronization, 602
- blocks, nested blocks, 123–124
- BMP (Basic Multilingual Plane), 313
- BNF (Backus-Naur Form), 42
  - Algol 60, 795
- Bobrow, Daniel G., 406
- Bochmann, Gregor V., 194
- Boehm, Hans-Juergen, 406
- bookkeeping information, subroutines and, 109
- Boolean expressions, short-circuit evaluation, 252–254
- bootstrapping, 18
- Borning, Alan H., 518
- bottom-up parsers, 62
  - attributes and, 181
  - canonical derivations, 81–82
- CFSM, 84–85

- epsilon productions, 86–92  
 evaluation, 39–44-CD  
 introduction, 80–81  
 LR variants, 84–85  
 modeling, LR items, 82–84  
 syntax errors, 9–11-CD  
 table-driven parsing, 86  
 bounded buffer abstractions, 626–627, 652  
 bounds, arrays, 353–357  
 Bourne, Stephen, 678, 757  
 boxing, 241–242  
 Bracha, Gilad, 468  
 branches  
     delayed branches, 212  
     instruction set architecture, 202–204  
     pipeline and, 214–216  
 Brinch Hansen, Per, 629  
 Bryant, Randal E., 227  
 buddy system, dynamic pool  
     adjustment, 112  
 buffering, messaging passing,  
     concurrency, 647–648  
 built-in objects, 117  
 Burrows, Michael, 670  
 busy-wait, I/O, 591  
 busy-wait synchronization, 620  
     barriers, 622–623  
     spin locks, 620–622  
 Butler, Kenneth J., 790  
 byte code, Java, 20
- C**  
 arrays, 376  
 calling sequences, MIPS, 414  
 compilers, 17  
 overview, 795  
 pointers, 142, 376  
     to subroutines, 425  
 separate compilation, 30–33-CD  
 text I/O, 99–101-CD
- C#**  
 generics, 132–134-CD, 440–441  
 overview, 795–796
- C++**  
 compiler, as preprocessor, 18  
 generics, 125–127-CD, 440–441  
 implementations, early compilers, 17  
 overview, 796  
 parameters, references, 423–424  
 text I/O, 101–103-CD
- caches  
     associativity, 211  
     cache hits, 198  
     cache misses, 198  
     pipeline and, 211
- memory hierarchy and, 197  
     primary caches, 197  
     primary, loads, 213
- Cailliau, R., 159
- call by name mode (parameters), 122–124-CD, 426–427  
 call by reference mode (parameters), 419  
 ambiguity, 420–421  
 call by sharing mode (parameters), 420  
 call by value mode (parameters), 419  
 callers, subroutines, 407  
 calling sequences, 407  
     C on MIPS, 414  
     displays, 413  
     example, 411–412  
     format parameters, 417–418  
     in-line expansion, 415–416  
     Pascal on x86, 414  
     register windows, 414–415  
     registers, saving/restoring, 410–411  
     static chain, 411  
     subroutines, 110
- Cambridge Polish notation, 528–530
- Cann, David, 551
- canonical derivations, 44  
     parsers, bottom-up, 81–82  
 canonical implementations, 675
- Car function, 391
- Cardelli, Luca, 159, 406, 799
- Carrierro, Nicholas, 798
- cascading errors, 93
- case statements, 265–268  
     finite automaton and, 58  
     nested, 54–55
- CCRs (conditional critical regions), 634–638
- CD accompanying book. *See* PLP CD  
 pushdown automata and, 95
- cdr function, 391
- Cedar, 796  
     overview, 798
- central reference table  
     dynamic scoping and, 135  
     scope implementation, 27–29-CD
- CFGs (context-free grammars), 39
- CFL (context-free language), 39
- CFSM (characteristic finite-state machine), 84–85
- CGI (Common Gateway Interface)  
     scripts, 702–703
- chaining, Prolog, 566–569
- Chaitin, Gregory, 259-CD
- Chambers, John, 690
- Chandra, Ashok, 259-CD
- character sets  
     localization, 40  
     multilingual, 313
- characters, data representation and, 199
- child classes, 476
- Chomsky, Noam, 102
- Chomsky hierarchy, 94
- Chonacky, Norman, 757
- Church, Alonzo, 468, 525, 558
- Church's thesis, 524  
     lambda calculus, 525
- Ciancarini, Paolo, 670
- CIL (Common Intermediate Language), .NET, 20
- CISC (complex instruction set computer) architecture, 196
- condition codes, 202
- register-memory architecture, 201
- class constructs, object-oriented programming, 130
- classes  
     abstract, object-oriented  
     programming, 501–502  
     base classes, 478  
     methods, modifying, 479–480  
     protected keyword, 485
- child classes, 476
- data abstractions, 104
- declaration, 475  
     executable, 747
- derived, object-oriented  
     programming, 476–477
- grammars  
     LL, 61  
     LR, 61
- hierarchies, 476
- inheritance, 130, 484–486
- modules, 128–131  
     subclasses, 476
- classification of languages, 9
- clausal forms, logic programming and, 579
- Cleaveland, Craig, 405
- client-side scripts, 708
- CLOS (Common Lisp Object System)  
     notation, 470  
     overview, 796
- closed scopes, modules, 128
- closed world assumption, Prolog, 581–582
- closures, subroutines, 138–140  
     dynamic method binding, 508–509  
     as parameters, 424–426
- Clu, overview, 796
- clusters, 126  
     computer clusters, multiprocessors, 597
- Cobol, overview, 796
- Cocke, John, 227, 259-CD
- code generation

attribute grammars and, 769–772  
 register allocation, 772–773  
 code improvement, 30, 202–204-CD, 759  
 global, 763  
 global redundancy, 217–227-CD  
 instruction scheduling, 232–236-CD  
 interprocedural, 764  
 local, 763  
 loop improvement, 227–228-CD,  
   236–237-CD  
 induction variables, 229–232-CD  
 invariants, 228–229-CD  
 register allocation, 248–251-CD  
 reordering, 241–248-CD  
 software pipelining, 237–240-CD  
 unrolling, 237–240-CD  
 peephole optimization, 206–209-CD,  
   791  
 phases, 204–205-CD  
 redundancy elimination, basic blocks,  
   209–217-CD  
 coercion, 145–146  
   type compatibility, 328–331  
 Cohen, Jacques, 406  
 coherence, multiprocessors, 599–600  
 collective communication, 603  
 Colmerauer, Alain, 800  
 combination loops, 277–278  
 commands, guarded, 651  
 comments  
   nested, 47  
   performance and, Basic, 16  
   scanning and, 24  
 Common Lisp, 21, 796  
 communication, concurrency and,  
   601–602  
   collective, 603  
   message passing, 601  
   partners, names, 642–646  
   RMIs (remote method invocations),  
    603  
   RPCs (remote procedure calls), 603  
   shared memory, 601  
 compilation  
   back end, 22  
   back-end compiler structure, 761–762  
    phases, 762–766  
   binding time and, 105  
   conditional, 17  
   front end, 22  
   interpreted languages, 675  
   introduction, 13  
   lexical analysis, 23–25  
   modern processors, 210–221  
    register allocation, 216–221  
   overview, 22–31  
   passes, 23

phases, 22, 23  
 machine-independent code  
   improvement, 23  
 machine-specific code improvement,  
   23  
 parser, 23  
 scanner, 23  
 semantic analysis and intermediate  
   code generation, 23  
   target code generation, 23  
 regular expressions, 735  
 separate, 149  
   automatic header inference,  
    33–34-CD  
   C, 30–33-CD  
   packages, 33–34-CD  
   syntax analysis, 23–25  
 compile-time constants, 109  
 compiled languages  
   error checking, 15  
   sidebar, 15  
 compiler-based language  
   implementation, 106  
 compilers  
   assembly language and, 16  
   C, 17  
   C++  
    early implementations, 17  
    as preprocessor, 18  
   conservative, 165  
   IFs (intermediate forms), 766–769  
   introduction, 4, 13–14  
   Java, just-in-time, 20  
   language design and, 6–7  
   late binding and, 20  
   machine language and, 16  
   memory hierarchy and, 196  
   one-pass, 174–175  
   optimistic, 165  
   parallelizing, 604  
   Pascal, 18  
   query language processors, 20  
   semantic analysis and, 163  
   TeX as, 20  
   troff as, 20  
 component types, arrays, 349  
 composite types, 317–318  
   arrays, 318  
   files, 318  
   lists, 318  
   pointers, 318  
   records, 318  
   scripting languages, 738–740  
   sets, 318  
   variant records, 318  
 compound statements, 260  
 computer arithmetic, target machine  
   architecture, 54–58-CD  
*Computer Curricula 2001* report, xxvii  
 computers, history of, 3–4  
 conceptual load, abstraction and, 471  
 concrete syntax tree, parse tree, 27  
 concurrency, 233  
   communication, 601–602  
    collective, 603  
    message passing, 601  
   RMIs, 603  
   RPCs, 603  
    shared memory, 601  
   context switches and, 592  
   distribution and, 592–593  
   history of, 590–593  
   interrupt-driven I/O and, 591–592  
   introduction, 589  
   languages, 603–604  
   libraries, 603–604  
   message passing  
    communication partners, 642–646  
    peeking inside, 655  
    receiving, 651–656  
    RPCs, 656–659  
    sending, 646–650  
   multiprocessors, architecture, 597–600  
   multiprogramming and, 591–592  
   multithreaded programs, 593–597  
    dispatch loop alternative, 595–597  
   parallelism comparison, 601  
   personal computers and, 597  
   processes, 601  
   shared memory, 619–620  
    CCRs, 634–638  
    monitors, 629–634  
    semaphores, 627–628  
   synchronization, 601–602  
    Ada 95, 634–635  
    Java, 635–368  
   tasks, 601  
   threads, 601  
    creation syntax, 604–613  
    implementation, 613–618  
    timesharing and, 592–593  
 cond (Scheme), 533–534  
 condition codes, 197  
   CISC machines, 202  
   processor status register, 202  
 condition synchronization, 619  
 condition variables  
   Java, 637  
   shared memory, 630  
 conditional compilation, 17  
 conditions  
   instruction set architecture, 202–204  
   scripting languages, 679–680

- conformant arrays, 355–356  
 parameter passing and, 427
- conservative collection, garbage collection, 389
- conservative compilers, 165
- constants  
 compile-time, 109  
 elaboration-time, 109
- constraint-based languages, 9
- constructive notation, 549
- constructive point of view, types, 311
- constructive proofs, 525  
 logic programming and, 525
- constructors, 473  
 copy constructors, 493  
 expression evaluation, 247–248  
 functional programming, 526  
 initialization and, 489, 490–491  
 overloaded, 478
- containers, object-oriented  
 programming, 480
- context blocks, coroutines, 457
- context-free grammars, 2, 38–39  
 derivations, 43–46  
 non-terminals, 42  
 parse trees and, 43–46  
 parsing and, 24  
 productions, 42  
 syntax and, 25, 38  
 terminals, 42  
 tree grammars and, 182  
 variables, 42
- context-free languages, 94
- context-sensitive look-aheads, syntax errors, 3–4-CD, 93
- context switches, concurrency and, 592
- continuation-passing style, recursion, 289
- control abstractions, 104, 231, 407  
 C on MIPS, 111–114-CD  
 call by name, 122–124-CD  
 coroutines, 453–459  
 discrete event simulation, 139–142-CD  
 displays, 107–110-CD  
 exercises, 143–144-CD  
 explorations, 145-CD  
 iterator implementation, 135–137-CD  
 Pascal on the x86, 114–117-CD  
 register window, 119–121-CD
- control flow, 231. *See also* ordering  
 continuations, 259–260  
 exercises, 78–79-CD  
 explorations, 80-CD  
 expression evaluation, 234–236  
 assignments, 238–246  
 associativity, 236–238  
 precedence, 236–238
- finite automata and, 58  
 functional languages, 171–173-CD  
 graph, 762
- Icon generators, 69–71-CD
- iteration, 270–287
- nondeterminacy, 72–77-CD, 295
- Prolog, 571–574
- recursion  
 applicative evaluation, 291–295  
 iteration, 287–291  
 normal-order evaluation, 291–295
- Scheme, 533–534
- selection, 261–262  
 Case statements, 265–268  
 short-circuited conditions, 262–264  
 switch statement, 268–269
- sequencing, 260–261
- structured, 234, 254–255  
 goto alternatives, 255–259
- unstructured, 234, 254–255
- control hazards, pipeline and, 211
- conversion, type conversion, 325–327
- Conway, Melvin E., 468
- Cooper, Keith D., 35, 194, 228, 258-CD, 790
- cooperative multithreading, 616
- copy constructors, 493
- copy rules, attribute grammars, 166
- coroutines  
 context blocks, 457  
 discrete event simulation, 458–459  
 introduction, 453  
 iterator implementation, 458  
 stack allocation, 455–457  
 stacks, 457  
 threads and, 454  
 transfers, 457–458
- coscheduling, 615
- counterintuitive implementation, 611
- Courcelle, Bruno, 194
- Courtois, Pierre-Jacques, 669
- Cox, Brad, 799
- CS 340 (Compiler Construction), xxvii
- CS 344 (Functional Programming), xxvii
- CS 341 (Programming Language Design), xxvii
- CS 343 (Programming Paradigms), xxvii
- CS 346 (Scripting Languages), xxvii
- CTSS (Compatible Time Sharing System), 678
- cubic time, parsing and, 61
- Culler, David E., 670
- Curry, Haskell, 546, 558
- currying functions, 546
- CYK (Cocke-Younger-Kasami) algorithm, 61
- Cytron, Ronald, 259-CD
- DAGs (directed acyclic graphs), 766–767
- Dahl, Ole-Johan, 158, 305, 468, 470, 801
- Damron, Peter C., 790
- dangling else, 79
- dangling references  
 garbage collection and, 113  
 locks and keys, 381–382  
 pointers, 379–382  
 tombstones, 380–381
- Darlington, Jared L., 580
- data abstractions, 231, 407  
 classes, 104  
 dynamic method binding, 469  
 exercises, 162–164-CD  
 explorations, 165-CD  
 inheritance, 469  
 multiple, 146–155-CD
- data generation, 779
- data hazards, pipeline and, 211
- data members, fields, 471
- data-parallel dialect (Fortran), 604
- data representation  
 arithmetic, 199–200  
 binary integers, 199  
 data formats, 199  
 integers, 199  
 operations, 199
- data types. *See also* type checking; type systems  
 arrays, 349–365  
 assignment, 393–395  
 equality testing, 393–395  
 exercises, 105-CD  
 explorations, 106-CD  
 file-based I/O, 94–96-CD  
 files, 392–393  
 interactive I/O, 93–94-CD  
 lists, 389–392  
 ML type system, 81–89-CD  
 operations, 337–338  
 pointers, 369–389  
 records, 336  
 memory layout and, 338–341  
 with statements, 341  
 variant, 341–348  
 recursive types, 369–389  
 scripting languages, 676–677, 736–741  
 composite types, 738–740  
 context, 740–741  
 numeric, 737  
 object orientation, 741–747
- sets, 367–368
- with statements, 90–92-CD
- strings, 366–367

- structures, 336  
 syntax, 337–338  
 text I/O, 96–104-CD  
 unions, 336
- databases, Prolog, 574–579  
 dataflow languages, 9  
 Davie, Bruce S., 670  
 debuggers, 12  
 decimal types, 314  
 declaration order, 119–124  
     nested blocks, 123–124
- declarations  
     classes, 475  
     executable, 747  
     forward declarations, 122  
     scripting languages, 676
- declarative languages, 8  
 deep binding, 115, 138  
 delayed branches, 212  
 denotational point of view, types, 311  
     denotational semantics, 311  
 dependences, varieties, 213  
 DeRemer, Franklin L., 101  
 derivation  
     canonical, 44  
     parse trees, 44  
     right-most, 44
- derived classes, object-oriented  
     programming, 476–477
- design, languages, implementation and,  
     2
- design & implementation sidebars, 7,  
     xxvi
- array indices, lower bounds, 363  
 array layout, 360  
 arrays, 377  
 assignment, 493  
 attribute evaluators, 179  
 binding rules and extent, 141  
 binding time and, 105  
 built-in commands in shell, 680  
 canonical implementations, 675  
 Car function, 391  
 CCRs (conditional critical regions),  
     635  
 cdr function, 391  
 class declarations, 475  
     executable, 747  
 coercion, 146  
 compilation in functional  
     programming, 550  
 compiled languages, 15  
 compiling interpreted languages, 675  
 condition variables, Java, 637  
 continuations, 259  
 coroutine stacks, 457  
 counterintuitive implementation, 611
- dangling else, 79  
 decimal types, 314  
 development environments, 21  
 dynamic method dispatch, 507  
 dynamic scope, 727  
 dynamic scoping, 133  
 dynamic semantic checks, 164  
 dynamic typing, 310  
 emulation, efficiency and, 650  
 evaluation order, expressions, 251  
 formatting restrictions, 40  
 forward references, 174  
 fragile base class problem, 504  
 garbage collection, 383  
 generics as macros, 147  
 grep command (Unix), 729  
 hardware communication, 602  
 higher-order functions, 547  
 I/O, 392  
 IFs, stack-based, 767  
 implicit synchronization, side-effect  
     freedom and, 640
- in-line subroutines, 220  
 initialization, 493  
     expanded objects, 494
- inline keyword, 292  
     directives/hints, 415  
     modularity, 416
- integers, multiple sizes, 317  
 interpreted languages, 15  
 iteration in functional programs, 534  
 iterator objects, 280  
 Java, 710  
 JavaScript, 710  
 lazy evaluation, 541–542  
 logic implementation, 580  
 for loops, 276  
 Modula-2 opaque exports, 481  
 monads, 544  
 monitor signal semantics, 632  
 multilingual character sets, 313  
 mutual recursion, 120  
 nested comments, 47  
 nested threads, stack frames for, 606  
 nonconverting casts, 328  
 normal-order evaluation, 294  
 numerical imprecision, 272  
 overloading, 146  
 parameter modes, 419  
 Pascal's early success, 19  
 peeking inside messages, 655  
 pointer implementation, 369  
 pointers, 142, 377  
 Postscript, 767  
 processor/memory gap, 198  
 record field order, 340  
 recursion in Fortran, 109
- redeclarations, 123  
 reference counts, 388  
 reference model implementation, 240  
 reflection, 578  
 regular expressions  
     automata, 729  
     compiling, 735  
 remote procedures, parameters, 658  
 reverse assignment, 504  
 safety v. performance, 248  
 sandboxing, 711  
 scripting on Microsoft platforms, 673  
 search strategy alternatives, Prolog,  
     580
- semantic impact of implementation  
     issues, 647
- set representation, 368  
 short-circuit evaluation, 264  
 side effects of functional  
     programming, 550  
 software communication, 602  
 square brackets, 351  
 stack frames for nested threads, 606  
 structured exceptions, 447  
 threads and coroutines, 454  
 tokens, 57  
 tracing, 388  
 true iterators, 280  
 type checking for separate  
     compilation, 783  
 typeglob, 739  
 value/reference tradeoff, 492  
 variant field placement, 347  
 destructors, 473  
 Deutsch, L. Peter, 406, 518  
 development environments, 21  
 devices, memory hierarchy and, 196  
 DFA (deterministic finite automaton),  
     38  
     minimizing, 53–54  
     Scheme example, 537–538  
 Diana, 189–192-CD  
 Dijkstra, Edsger W., 35, 305, 467, 620,  
     627–628, 669
- dimensions, arrays, 353–357  
 Dion, Bernard A., 102  
 discrete event simulation  
     coroutines, 458–459  
     subroutines, 139–142-CD
- discrete types, 314  
 displacement addressing mode, 201  
     register indirect addressing, 201
- displays  
     static chains, 413  
     subroutines, 107–110-CD
- distributed memory, multiprocessors,  
     598

- distribution, concurrency and, 592–593  
 Diwan, Amer, 518  
 Donahue, Jim, 799  
 dope vectors, 364–365  
 dot-dot problem (Pascal), 57  
 double precision floating-point numbers, 199  
 DTD (document type definition), 714  
 Dyadkin, Lev J., 57  
 dynamic arrays, 356–357  
 dynamic linkers, 781, 784–785  
   fully dynamic (lazy) linking, 196–198-CD  
   position-independent code and, 195–196-CD  
 dynamic links, subroutines, 409  
 dynamic method binding, 469, 497–500  
   closures, 508–509  
   nonvirtual methods, 500–501  
   subtype polymorphism, 498  
   virtual methods, 500–501  
 dynamic method dispatch, 507  
 dynamic pool adjustment  
   buddy system, 112  
   Fibonacci heap, 112  
 dynamic scope, 115, 131–134, 727  
   APL, 132  
   association list, 135  
   central reference table, 135  
   Lisp, 132  
   Perl, 132  
   Snobol, 132  
 dynamic semantics, 27, 161  
   eliminating checks, 166  
   semantic checks, 164  
 dynamic translation schemes, 173  
 dynamic typing, 310  
   scripting languages and, 676
- Earley, Jay, 61  
 EBNF (extended BNF), 42  
 Eich, Brendan, 797  
 Eiffel, overview, 796  
 elaboration-time constants, 109  
 element types, arrays, 349  
 elliptical references, 341  
 Ellis, Margaret A., 518  
 else statement, dangling, 79  
 elseif keyword, 80  
 emulation of alternatives, message passing, 649–650  
 encapsulation  
   modules and, 481–484  
   type extensions, 486–488  
 Engelfreit, Joost, 194  
 Enigma code, 524
- enumeration-controlled loops, 270, 271–277  
 enumeration types, 315–316  
 environments, 21–22  
 epilogue, subroutines, 110  
 epsilon production, bottom-up parsing, 86–92  
 epsilon transitions, 50  
 equality testing, types, 393–395  
   Scheme, 532  
 equational reasoning, functional programming, 549  
 equivalence, type  
   casts, 325–327  
   conversion, 325–327  
   name, 321  
   variants, 323–325  
   structural, 321  
 error checking  
   compiled languages, 15  
   interpreted languages and, 15  
 error productions, 93  
 error reporting, message passing, concurrency, 648–649  
 errors  
   cascading errors, 93  
   goto alternatives, 258  
   lexical errors, 58–59  
   semantic, syntax trees and, 182  
   syntax, 93  
     context-sensitive look-ahead, 93  
     panic mode, 93  
     phrase-level recovery, 93  
 escape analysis, 165  
 Euclid, overview, 796–797  
 Euclid's algorithm, GCD and, 3  
 eval function, Scheme, 535–536  
 evaluation order  
   applicative-order evaluation, 539–540  
   functional programming, 539–541  
     lazy evaluation, 541–542  
   I/O, 542–545  
   lazy evaluation, 541–542  
   normal-order evaluation, 539–540  
 Evans, Arthur Jr., 790  
 Evey, R. James, 102  
 evolution of language design, 5  
 examples  
   numbered, xxvi  
   titled, xxvi  
 exception handling  
   cleanup, 447  
   defining exceptions, 443–445  
   exception propagation, 445–448  
   expressions, 447  
   goto alternatives, 258  
   implementing exceptions, 449–452
- introduction, 441  
 PL/I, 442  
 structure exceptions, 447  
 suppressing exceptions, 444  
 without exceptions, 450–451
- execution order  
   initialization and, 489, 495–496  
   logic languages and, 580  
   Prolog, 580
- exercises, xxvi–xxvii  
   binding, 36–37-CD  
   building runnable program, 201-CD  
   control abstractions, 143–144-CD  
   control flow, 78–79-CD  
   data abstraction, 162–164-CD  
   data types, 105-CD  
   functional languages, 177–178-CD  
   logic languages, 186–187-CD  
   names, 36–37-CD  
   object orientation, 162–164-CD  
   programming language syntax, 21-CD  
   scope, 36–37-CD  
   semantic analysis, 51–52-CD  
   subroutines, 143–144-CD  
   target machine architecture, 66–67-CD
- expanded objects, initialization, 494  
 expansion, scripting languages, 681–682  
 explorations  
   binding, 38-CD  
   building runnable program, 201-CD  
   control abstraction, 145-CD  
   control flow, 80-CD  
   data abstractions, 165-CD  
   data types, 106-CD  
   functional languages, 179-CD  
   logic languages, 188-CD  
   names, 38-CD  
   object orientation, 165-CD  
   programming language syntax, 22-CD  
   scope, 38-CD  
   semantic analysis, 53-CD  
   subroutines, 145-CD  
   target machine architecture, 68-CD
- exponents, floating-point numbers, 200  
 export tables, 775  
 exports  
   modules and, 126  
   opaque, 127  
   Modula-2, 481  
   variables, 127
- expression types (Scheme), 541
- expressions  
   assignments  
     boxing, 241–242  
     multiway assignment, 245  
     operators, 243–245  
     orthogonality, 242–243

evaluation, 231, 234–236  
 assignments, 238–246  
 associativity, 236–238  
 initialization, 246–249  
 ordering within, 249–252  
 precedence, 236–238  
 short-circuit, 252–254  
 exception handling, 447  
 referentially transparent, 238  
 semantic analysis and, 25  
 extension languages, 12, 698–701  
 scripting languages and, 674  
 extensions, abstractions, 471  
 external fragmentation, 112  
 external symbols, 775

fault containment, abstraction and, 471  
 Fenichel, Robert R., 406  
 Ferrante, Jeanne, 259-CD  
 Feuer, Alan R., 305  
 Feys, Robert, 558  
 Fibonacci heap, dynamic pool adjustment, 112  
 fields, 471  
 awk, 686  
 file-based I/O, 94–96-CD  
 filenames, scripting languages, 678–679  
 files, 392–393  
 composite types, 318  
 finalization, overview, 489  
 finite automaton, 13–15-CD, 48  
 control flow and, 58  
 generation, 49–54  
 scanners, 94  
 transition tables, 58  
 firmware, 20  
 first-class function values, functional programming, 526  
 first-class subroutines, 140–142  
 first fit algorithm, 111  
 first-order predicate calculus, 559  
 Fischer, Charles N., 35, 93, 102, 194, 406, 790  
 Fisher, Joseph A., 259-CD  
 fixed format, 40  
 fixed-point types, 314  
 Fleck, Arthur C., 468  
 floating-point numbers  
   data representation and, 199  
   double precision, 56-CD, 199  
   scientific notation and, 200  
   single precision, 56-CD, 199  
 flow dependence, 213  
 for-each, Scheme, 534  
 for loops, 276  
 fork operations, threads, 608–611

formal languages  
 automata theory and, 94  
 Chomsky hierarchy, 94  
 format parameters, subroutines, 407  
 formatting, 40  
 Fortran and, 16  
 Forth, overview, 797  
 Fortran  
   formatting and, 16  
   implementation, 16  
   overview, 797  
   pointers, 142  
   recursion, 109  
   scope rules, 116  
   subroutines, libraries, 16  
   text I/O, 97–98-CD  
 forward chaining, Prolog, 566–569  
 forward declarations, 122  
 forward references, 174  
 fragile base class problem, 504  
 fragmentation, 111  
   external, 112  
 frame pointer, 408  
 frames, frame pointer, 110  
 Fraser, Keir, 670, 790  
 free format languages, 40  
 free list, 111  
 Friedman, Daniel P., 468  
 front end, introduction, 22  
 funarg problem (Lisp), 138  
 function prototypes, 430  
 functional forms, 545–549  
 functional languages, 9, 166–168-CD, 523  
   control flow, 171–173-CD  
   exercises, 177–178-CD  
   explorations, 179-CD  
   lambda calculus, 168–171-CD  
   structures, 173–176-CD  
 functional programming. *See also*  
   Scheme  
 constructors, 526  
 equational reasoning, 549  
 evaluation order, 539–541  
   lazy evaluation, 541–542  
 functions, 526  
   higher-order functions, 526  
   structured function returns, 526  
   values, 526  
 garbage collection, 526  
 higher-order functions, 545–549  
 iteration, 534  
 Lisp, features, 527  
 list types, 526  
 operators, 526  
 polymorphism, 526  
 recursion, 526

structured function returns, 526  
 theoretical foundations, 549  
 trivial update problem, 551  
 functional units, processor implementation, 204  
 functions  
   currying, 546  
   higher-order functions, 526  
   functional programming and, 545–549  
   notation, 549  
   returns, 432–433  
   scripting languages, 683  
   semantic functions, 166  
   type predicate functions, 529  
 functor (Prolog), 561

Gabriel, Richard, 757  
 Ganapathi, Mahadevan, 790  
 gang scheduling, 615  
 garbage collection, 113–114, 383–389  
   conservative collection, 389  
   dangling references, 113  
   functional programming, 526  
   initialization and, 490, 496–497  
   mark-and-sweep, 385–386  
   pointer reversal, 386–387  
   reference counts, 384–385  
   stop-and-copy, 387–388  
   tracing collection, 385–389  
 Garcia, Ronald, 468  
 Gardner, Martin, 406  
 GCD (greatest common divisor), 3, 23–24  
   parse tree, 25  
 Gehani, Narain, 305  
 Gelernter, David, 798  
 generic subroutines, 434–441  
   C#, 132–134-CD  
   C++, 125–127-CD  
   implementation, options, 435–437  
   implicit instantiation, 440  
   Java, 128–130-CD  
   parameters, constraints, 437–439  
   type erasure, 130–131-CD  
 genericity, 146  
   generics as macros, 147  
   object-oriented programming, 507  
 Gibbons, Philip B., 259-CD  
 Ginsberg, Seymour, 102  
 Glanville, R. Steven, 790  
 global optimization, 791  
 global redundancy, 217–227-CD  
 global variables, scope, 116  
 glocal code improvement, 763  
 glue languages, 672

- scripting and, 690–698
- GNU RTL (Register Transfer Language), 192–194-CD
- goals (Prolog), 562
- Goguen, Joseph A., 405
- Goldberg, Adele, 801
- Goldberg, David, 228
- Goodenough, John B., 468
- Goos, Gerhard, 790
- Gordon, Michael J.C., 194
- Gosling, James, 797
- goto statements, 255
  - alternatives, 255–259
- Gottlieb, Allan, 670
- Graham, G. Scott, 669
- Graham, Susan L., 259-CD, 790
- grammars
  - attribute, 2
  - classes
    - LL, 61
    - LR, 61
  - context-free, 2
- Green, J., 102
- grep (Unix), 729
  - regular expressions and, 41
- Gries, David, 194
- Griswold, Ralph, 797, 801
- Grune, Dick, 35, 258-CD, 558, 790
- guarded commands, 651
- Gutmans, Andi, 800
- Guttag, John, 159, 405, 468
  
- Hanson, David R., 467, 790
- Harbison, Samuel P., 159, 467–468
- Harris, Tim, 670
- Haskell, overview, 797
- Haskell I/O, 543–544
- Hauben, Michael, 757
- Hauben, Ronda, 757
- Haynes, Christopher, 468
- headers, modules, 484
- heap-based allocation, 111–113
  - garbage collection, 113–114
- heap objects
  - free list, 111
  - storage allocation, 107, 111–113
- heavyweight processes, 601
- Hejlsberg, Anders, 795
- Hennessy, John L., 227, 790
- Henry, 790
- Henry, Robert R., 790
- Herlihy, Maurice P., 670
- Heymans, F., 669
- hiding information, 125
- higher-order functions, 526, 547
  - currying, 546
  
- functional programming and, 545–549
- history of computers, 3–4
- Hoare, Charles Anthony, 35, 194, 305, 405, 406, 629, 795
- Holt, Richard C., 669, 801
- homoiconic languages, 575
- Hopcraft, John E., 101
- Hopkins, Martin, 259-CD
- Hopper, Grace Murray, 796
- Horn, Alfred, 587
- Horn clause, logic programming, 560
  - Prolog, 561
- Horowitz, Ellis, 35
- Hudak, Paul, 558, 670
- Hughes, John, 558
  
- I/O (input/output), 392–393
  - busy wait, 591
  - Haskell I/O, 543–544
  - interrupt-driven, 591
  - monads, 542–545
  - overlap, 591
  - streams, 542–545
- Icon
  - generators, control flow and, 69–71-CD
  - iteration, 284
  - overview, 797
- identifiers
  - lowercase letters, 39
  - semantic analysis and, 25
  - uppercase letters, 39
- IFs (intermediate forms), 766–769
- ILP (instruction-level parallelism), 212
- immutable integers, 240
- imperative languages, 8, 10
- implementation
  - architecture and, 204–210
    - microprocessors, 206
    - microprogramming, 205–206
    - pseudo-assembly notation, 209–210
    - RISC, 207–208
  - assemblers, 759
  - canonical, 675
  - code improvement and, 759
  - exceptions, 449–452
  - generic subroutines, options, 435–437
  - iterators, 135–137-CD
    - coroutines, 458
  - language compiler-based, 106
  - language design and, 2, 6
  - linkers, 759
  - logic, 580
  - optimization and, 759
  - pointers, 369
  
- processors, 195
  - functional units, 204
- RPCs, 658–659
- schedulers, 623–627
- scope, 135–136
  - association links, 27–29-CD
  - central reference tables, 27–29-CD
  - symbol tables, 23–27-CD
- sidebars used in book (*See* design & implementation sidebars)
- threads, 613–618
  - virtual methods, 503
- implicit parametric polymorphism, 148
  - type systems, 309
- implicit receipt, thread creation, 611, 651
- implicit synchronization, concurrency
  - and, 638–641
- import tables, 775
- imports
  - modules and, 126
  - stack abstraction and, 126
- in-line expansion, calling sequences, 415–416
- in-line subroutines, 220
- in-place mutation, 550
- independence, abstraction and, 471
- index types, arrays, 349
- indexed addressing mode, 201
- indexer mechanisms, 476
- induction variables, code improvement
  - and, 229–232-CD
- inference, types, 332–335
- information hiding, 125
- Ingalls, Daniel H. H., 518, 801
- Ingerman, Peter Z., 468
- inheritance, 469
  - classes, 130, 484–486
  - modules and, 481–484
  - multiple inheritance, 146–147-CD, 470, 511–512
    - ambiguities, 148–151-CD
    - mix-in, 152–157-CD
    - replicated, 151–152-CD
    - repeated, 511
  - replicated, 151–152-CD, 512
  - shared, 512
  - single, implementation, 503
- inherited attributes, 40–41-CD, 169–172
- initialization, 493
  - assignment and, 494
  - complex structures, 550
  - constructors and, 490–491
  - execution order, 495–496
  - expanded objects, 494
  - expression evaluation, 246–249
    - constructors, 247–248

- definite assignment, 248
- dynamic checks, 249
- garbage collection and, 490, 496–497
- overview, 489
- references and, 491–494
- values and, 491–494
- inline keyword, 292
- instantiation
  - implicit, generic subroutines, 440
  - Prolog, 561
  - variables, unification and, 563
- instruction scheduling, 212
  - code improvement and, 232–236-CD
  - register allocation and, 216–221
- instruction set architecture
  - addressing modes, 201–202
  - branches, 202–204
  - conditions, 202–204
- instructions
  - assemblers and, 778–780
  - data representation and, 199
- integers
  - data representation and, 199
  - immutable, 240
  - multiple size, 317
  - n-bit unsigned integers, 199
- interaction, functional programming, 534
- interactive I/O, 93–94-CD
- interconnection networks, 598–599
- interlock hardware, 214
- intermediate forms
  - Diana, 189–192-CD
  - GNU RTL (Register Transfer Language), 192–194-CD
- interpretation, 14
  - introduction, 13
  - late binding, 14
  - metacircular interpreters, 535
  - P-code interpreter, 18–19
  - source code, 14
- interpreted languages
  - compiling, 675
  - error checking and, 15
  - preprocessors, 15
  - sidebar, 15
- interpreters, Postscript printing and, 20
- interprocedural code improvement, 764
- interrupt-driven I/O, 591
  - multiprogramming and, 591–592
- invariants, 164
- iteration, 231, 233, 270
  - enumeration-controlled loops, 271–277
  - functions, 282–283
  - Icon, 284
  - iterators, 278–284
  - none, 283–284
  - objects, 279–282
- loops
  - combination, 277–278
  - logically-controlled, 284–286
- macros, 293
- naive implementation, 289
- recursion and, 287–291
  - tail recursion, 289–290
- iterators, 12
  - implementation, 135–137-CD
  - coroutines, 458
- Iverson, Kenneth E., 757, 795
- Järvi, Jaakko, 468
- Jaswinder, Pal Singh, 670
- Java, 710
  - applets, scripting and, 708–711
  - byte code, 20
  - condition variables, 637
  - generics, 128–130-CD, 440–441
  - just-in-time compiler, 20
  - overview, 797
  - synchronization, 635–368
- JavaScript, 710
  - object orientation, scripting languages, 742–744
  - overview, 797
- Jazayeri, Mehdi, 194
- Jensen, Kathleen, 799
- jobs, 591
- Johnson, Stephen, 101
- Johnson, Walter L., 101
- Johnstone, Mark S., 406
- join operations, threads, 608–611
- Jones, Richard, 406
- Jones, Simon Peyton, 558
- Jordan, Mick, 799
- Joy, Bill, 678
- just-in-time compiler, Java, 20
- Kalsow, Bill, 799
- Kasami, T., 61
- Katz, C., 102
- Kay, Alan, 801
- Kemeny, John, 795
- Kennedy, Andrew, 259-CD, 468
- Kernighan, Brian W., 35, 686–687, 795
- keywords, 54. *See also* reserved words
  - elsif, 80
  - packed, 339
  - predefined identifiers and, 54
  - protected, 485
  - virtual, 500–501
- Kleene, Stephen, 37, 102, 525
  - Alonzo Church and, 525
- Kleene closure, 39
- Kleene star metasymbol, 37
  - regular expressions, 41
- Knuth, Donald E., 101, 194
- Korn, David, 678, 757
- Kowalski, Robert, 800
- Kurtz, Thomas, 795
- L-attributed attribute grammars, 173
- l-values, 239
- lambda calculus, 168–171-CD, 525
  - functional languages and, 9
- Lamport, Leslie, 620
- Lampson, Butler, 796–798
- Landin, Peter, 159
- languages. *See also* programming languages
  - design
    - binding time and, 105
    - compilers and, 6–7
    - ease of use, 6
    - economics, 7
    - evolution, 5
    - implementation and, 6
    - inertia, 7
    - open source and, 6
    - patronage, 7
    - preferences, 5
    - purposes, 5
    - glue languages, 672
  - implementation
    - binding time and, 105
    - compiler-based, 106
    - integration, message passing, 650
  - late binding, 14
    - compilers and, 20
  - Lazowska, Edward D., 669
  - lazy evaluation, 541–542
  - lazy linking, 196–198-CD
  - leaf routine, 413
  - LeBlanc, Richard J. Jr., 35, 194, 406, 790
  - Ledgard, Henry F., 194
  - left recursion, LL(1) grammars, 77
  - Lerdorf, Rasmus, 672, 800
  - Lesk, Michael E., 101
  - Lewis, Philip M. II, 101, 194
  - lexical analysis
    - compilation, 23–25
    - scanning, 24
  - lexical errors, 58–59
  - lexical scope, 114
  - Li, Kai, 670
  - libraries
    - concurrency and, 603–604
    - Fortran, 16
  - lightweight processes, 601

- limited extent, objects, 141  
 Linda, overview, 798  
 Lindholm, Tim, 790  
 line breaks, 40  
 linkers, 12
  - dynamic, 781, 784–785
  - implementation, 759
  - static, 781
 linking
  - binding time and, 105
  - dynamic linkers, 781, 784–785
  - name resolution, 782–783
  - relocation and, 782–783
  - static linkers, 781
  - type checking and, 783–784
 Lins, Rafael, 406  
 Liskov, Barbara, 159, 468, 796  
 Lisp
  - dynamic scoping, 132
  - funarg problem, 138
  - functional programming and, features, 527
  - overview, 798
 list types, functional programming, 526  
 lists
  - composite types, 318
  - notation and, 390
  - programs as (Scheme), 535–537
 Prolog, 564–565  
 Scheme, 531–532  
 literals, numeric, regular expressions, 41  
 little-endian organization, 199  
 LL class (grammars), 61
  - writing, 77–80
 LL parser generators, 179  
 Lloyd, John W., 587  
 Lo, Virginia, 670  
 load-store architecture (RISC machines), 201  
 loading, binding time and, 105  
 loads, primary cache, 213  
 local code improvement, 763  
 local optimization, 791  
 local variables, scope, 116  
 locks and keys, dangling references, 381–382  
 logic implementation, 580  
 logic languages, 9, 180–181-CD. *See also*
  - Prolog
  - arithmetic operators, 565–566
  - axioms, 560
  - clausal forms and, 181–182-CD, 579
  - concepts, 560–561
  - constructive proofs and, 525
  - exercises, 186–187-CD
  - explorations, 188-CD
  - first-order predicate calculus, 559
 Horn clause, 560  
 limitations, 182–183-CD  
 lists, 564–565  
 reflection, 578  
 skolemization, 183–185-CD  
 theoretical foundations, 579  
 logical functions, Scheme, 532  
 logically controlled loops, 270
  - midtest loops, 285–286
  - post-test, 284–285
 Lomet, David B., 406  
 loops
  - bounds, 273
  - code improvement and, 236–237-CD
    - induction variables, 229–232-CD
    - invariants, 228–229-CD
    - register allocation, 248–251-CD
    - reordering, 241–248-CD
    - software pipelining, 237–240-CD
    - unrolling loops, 237–240-CD
  - combination, 277–278
  - direction, 274–275
  - enumeration-controlled, 270, 271–277
  - finite automata and, 58
  - indices, 273
  - logically controlled, 270
    - midtest loops, 285–286
    - post-test, 284–285
  - for loops, 276
 Louden, Kenneth C., 35  
 LR class (grammars), 61  
 Luchangco, Victor, 670  
 Luckam, David C., 468  
 Lumsdaine, Andrew, 468  
 machine dependence/independence, 103  
 machine-independent code
  - improvement compilation phase, 23
 machine language
  - compilers and, 16
  - introduction, 3
 machine-specific code improvement
  - compilation phase, 23
 MacLaren, M. Donald, 468  
 macros
  - generics as, 147
  - iteration, 293
 Mairson, Harry G., 405  
 mantissa, floating-point numbers, 200  
 Marcotty, Michael, 194  
 mark-and-sweep, garbage collection
  - and, 385–386
 Markstein, Peter, 259-CD  
 Mashey, John, 678, 757  
 mathematical identities, expressions, 250–252  
 mathematics, scripting languages, 689–690  
 Matsumoto, Yukihiro “Matz,” 696–698, 800  
 Maurer, Dieter, 558  
 McCarthy, J., 102, 467–468, 798  
 McGraw, James, 801  
 Mellor-Crummey, John, 669  
 member lookup (object-oriented programming), 502–505  
 memory
  - array layout, 358–365
    - addresses, 361–364
    - dope vectors, 364–365
    - row-pointer layout, 359–364
  - coherence, multiprocessors and, 599–600
  - data alignment, 198
  - data types, records, 338–341
  - hierarchy, 196–199
    - caches, 197
    - condition codes, 197
    - processor status register, 197
  - Mesa, overview, 798
  - message passing, concurrency, 601
    - communication partners, naming, 642–646
    - message-passing models, 521
    - peeking inside, 655
    - receiving, 651–656
    - RPCs, 656
      - implementation, 658–659
      - semantics, 657
    - sending
      - buffering, 647–648
      - emulation of alternatives, 649–650
      - error reporting, 648–649
      - failure semantics, 646
      - resource management, 646
      - return parameters, 646
      - synchronization semantics, 646–647
      - syntax and language, 650
    - metacircular interpreters, 535
    - $\Rightarrow$  metasymbol, 44
    - metasymbols,  $\Rightarrow$ , 44
    - methods, 471
      - base classes, modifying, 479–480
      - dynamic method binding, 497–500
        - nonvirtual methods, 500–501
        - subtype polymorphism, 498
        - virtual methods, 500–501
      - nonvirtual, 500–501
      - object-oriented programming, 475
        - property mechanism, 475
      - static method binding, 499

- virtual, 500–501
- vtable (virtual method table), 502
- Meyer, Bertrand, 518, 796
- Michaelson, Greg, 558
- microcode, 20
- microprocessors, 206
- microprogramming
  - IBM, 205–206
  - implementation and, 205–206
- midtest loops, 285–286
- Milner, Robin, 335, 405
- Milton, Donn R., locally least-cost syntax repair algorithm, 93
- MIPS
  - C, calling sequences, 414
  - example, 208–209
  - target machine architecture, 59–64-CD
- Miranda, overview, 798
- mix-in inheritance, 152–157-CD
- ML, overview, 798
- ML type system, 81–89-CD, 335–336
- Modula, overview, 798
  - Modula-2, 798
  - Modula-3, 799
- module hierarchies, separate compilation, 35-CD
- modules, 124–128
  - classes, 128–131
  - closed scopes, 128
  - clusters, 126
  - encapsulation and, 481–484
  - exports and, 126
  - headers, 484
  - imports and, 126
  - information hiding and, 125
  - inheritance and, 481–484
  - inline and, 416
  - module constructs, 126
  - object-oriented programming, 471
  - open scopes, 128
  - packages, 126
  - this parameter, 483–484
  - types, 128–131
- Moir, Mark, 670
- monads, 544
- monitors
  - semantics, 631–634
  - shared memory, 629–634
    - condition variables, 630
    - nested monitor calls, 633
- Moore, Charles, 797
- Motwani, Rajeev, 101
- Muchnick, Steven S., 228, 259-CD, 790
- multidimensional arrays, memory layout, 358
- multilevel returns, goto and, 256–257
- multilingual character sets, 313
- multiple inheritance, 146–147-CD, 470
  - ambiguities, 148–151-CD
  - repeated inheritance, 511
  - replicated, 151–152-CD, 512
  - shared, 152–157-CD, 512
- multiprocessors
  - architecture, 597–600
  - interconnection networks, 598–599
  - memory, coherence, 599–600
  - scheduling, 618
  - symmetry, 598
  - vector processors, 598
- multiprogramming, 591
  - interrupt-driven I/O and, 591–592
- multithreaded programs
  - concurrency and, 593–597
  - dispatch loop alternative, 595–597
  - need for, 593
- multiway assignments, 245
- Murer, Stephan, 468
- mutation, in-place, 550
- mutual exclusion, synchronization, 619
- mutual recursion, 120
- Myhrhaug, Bjorn, 158, 468, 801
- n-bit unsigned integers, 199
- naive implementation, 289
- name equivalence, types, variants, 323–325
- name resolution, linking and, 782–783
- named parameters, 428–429
- names, 2
  - definition, 103
  - exercises, 36–37-CD
  - explorations, 38-CD
  - pervasive, 126
  - predefined, 126
  - qualifiers, 118
  - scope resolution operators, 118
  - scripting languages, 723–728
- Naur, Peter, 42, 102
- negation, Prolog, 581–582
- Nelson, Bruce J., 670
- Nelson, Greg, 799
- nesting
  - blocks, 123–124
  - constructs
    - cast statements, 54–55
    - comments, 47
    - regular expressions and, 42
  - monitors, shared memory, 633–634
  - scope, bindings, 530–531
  - subroutines, 117–119
    - local variables and, 118
    - nonlocal objects, 119
  - open scopes, 128
- Oberon, overview, 799
- object model, Smalltalk, 158–161-CD
- object orientation, 231
  - data types, scripting languages, 741–747
  - exercises, 162–164-CD
  - explorations, 165-CD
- object-oriented languages, 10
- object-oriented programming, 471
  - classes
    - abstract, 501–502
    - base classes, 478
    - constructs, 130
- parameters and, 118
- static links, 119
- .NET, CIL (Common Intermediate Language), 20
- networks, interconnection networks, 598–599
- Newell, Allen, 227
- NFAs (nondeterministic finite automata), 50
  - DFA transition, 51–53
  - regular expressions transition, 50–51
- Nitzberg, Bill, 670
- no-wait sends, 646
- nodes, syntax trees, 182
- non-terminals, context-free grammars, 42
- nonbinding prefetches, 165
- noncircular attribute grammars, 173
- nonconstructive proofs, 525
- nonconverting casts, 326–327, 328
- nondeterminacy, 234, 295
  - control flow and, 72–77-CD
- nonlocal objects, access, 119
- nonvirtual methods, dynamic method binding, 500–501
- normal-order evaluation, 291–295, 539–540
- notation
  - Cambridge Polish, 528–530
  - constructive, 549
  - lists, 390
  - pseudo-assembly notation, 209–210
  - semantic functions, 167–168
  - numbered examples, xxvi
  - numbers, Scheme, 531–532
  - numeric functions, Scheme, 532
  - numeric literals, regular expressions, 41
  - numeric types, 312–315
    - fixed-point, 314
    - scripting languages, 737
  - numerical imprecision sidebar, 272
  - Nygaard, Kristen, 158, 468, 470, 801

- declaration, 475
    - derived, 476–477
  - constructors, 473
    - overloaded, 478
  - containers, 480
  - destructors, 473
  - dynamic method dispatch, 507
  - generics, 507
  - initialization
    - assignment and, 494
    - constructors and, 490–491
    - execution order, 495–496
    - expanded objects, 494
    - garbage collection and, 490
    - overview, 489
    - references and, 491–494
    - values and, 491–494
  - member lookup, 502–505
  - methods, property mechanism, 475
  - modules, 471
    - pointers, 480
  - polymorphism, 505–508
  - private labels, 473–474
  - public labels, 473–474
  - subroutines and, 474–475
  - uniform object model, 512
  - visibility, 485
- Objective-C, overview, 799
- objects
- expanded, initialization, 494
  - heap, storage allocation, 107
  - lifetime, 106–114
  - protected, 651
  - stack, storage allocation, 107
  - static, storage allocation, 107
  - unlimited extent, 141
- oblivious translation schemes, 173
- Occam, overview, 799
- Odersky, Martin, 468
- Ogden, William F., 194
- O'Hallaron, David, 227
- Omohundro, Stephen, 468
- one-pass compilers, 174–175
- opaque exports, 127
  - Modula-2, 481
- opaque types, 481
- open scopes, 128
  - nested subroutines, 128
- open source, language design and, 6
- operations
  - arrays, 349–353
  - data types, 337–338
  - pointers, 370–378
- operators
  - assignment, 243–245
  - associativity, 45
  - expressions, 234
- functional programming, 526
  - precedence, 45
  - scope resolution, 474
- optimistic compilers, 165
- optimization, 30, 759
  - peephole optimization, 206–209-CD, 791
  - speculative, 165
  - unsafe, 165
- ordering, 233
  - concurrency, 233
  - within expressions, 249–252
  - mathematical identities, 250–252
- iteration, 233
- nondeterminacy, 234
- procedural abstraction, 233
- Prolog, 566–571
  - recursion, 233
  - selection, 233
  - sequencing, 233
- orthogonality, 242–243
- Ousterhout, John, 673, 757, 801
- output dependence, 213
- overlap I/O, 591
- overloaded constructors, 478
- overloading, binding with scopes, 143–146
- P-code interpreter, 18–19
- packages, 126
  - separate compilation, 33–34-CD
- packed keyword, 339
- panic mode, syntax error recovery, 1–2-CD, 93
- parallelism
  - concurrency comparison, 601
  - data-parallel dialect (Fortran), 604
  - parallel loops, thread creation, 606
  - parallelizing computers, 604
  - task-parallel programs, 604
- parameters
  - Ada, 422–423
  - arguments, variable numbers of, 429–431
  - C++ references, 423–424
  - closures as, 424–426
  - conformat arrays, 427
  - default, 427–428
  - generic subroutines, constraints, 437–439
  - named, 428–429
  - passing, 417–418
    - call by name, 426–427
    - call by reference, 419
    - call by sharing, 420
    - call by value mode, 419
- modes, 418–426
- positional, 428–429
- read-only, 421
- remote procedures, 658
- this, 483–484
- parametric polymorphism, 146
  - implicit, 148
- parentheses, regular expressions, 41
- Parnas, David L., 159, 669
- Parr, Terrence, 64
- parse tree
  - concrete syntax tree, 27
  - context-free grammars, 43–46
  - derivations, 44
  - GCD and, 25
  - semantic analysis and, 162
- parser compilation phase, 23
  - context-free grammar, 24
  - cubic time and, 61
  - language recognizer, 61
  - parse tree and, 24
  - syntax analysis, 25
  - syntax and, 38
  - syntax-directed translation and, 61
- parsers
  - bottom-up, 62
  - canonical derivations, 81–82
  - CFSM, 84–85
  - epsilon productions, 86–92
  - introduction, 80–81
  - LR variants, 84–85
  - modeling, LR items and, 82–84
  - table-driven, 86
  - recursive descent, 64–70
  - top-down, 62
    - table-driven, 70–80
- Pascal
  - calling sequences, x86, 414
  - compilers, 18
  - dot-dot problem, 57
  - early success, sidebar, 19
  - order of declaration, 121
  - overview, 799
- passes, compilation, 23
- pattern matching
  - awk, 686–687
  - minimal, 733
  - scripting languages, 676
- patterns, scripting languages, 728–735
- Patterson, David A., 227
- PDA (push-down automation), 38
- peeking inside messages, 655
- peephole optimization, 206–209-CD, 791
- performance, comments and, Basic, 16
- Perl, 672
  - dynamic scoping, 132

- extensions, scripting languages and, 731
- overview, 799
- Perl 5, object orientation, scripting languages, 742–744
- scope, 726–728
- typeglobs, 739
- Wall, Larry, 687
- Perles, Micha A., 102
- Perlis, A.J., 102
- personal computers, concurrency and, 597
- pervasive names, 126
- Peterson, Gary L., 620
- Peterson, Larry L., 670
- phases of code improvement, 204–205-CD
- phases of compilation
  - introduction, 22
  - machine-independent code improvement, 23
  - machine-specific code improvement, 23
  - parser, 23
  - scanner, 23
  - semantic analysis and intermediate code generation, 23
  - target code generation, 23
- PHP
  - object orientation, scripting languages, 742–744
  - overview, 800
- phrase-level recovery, syntax errors, 2–3-CD, 93
  - recursive descent parser, 448–449
- Pierce, Benjamin C., 405
- pipelined organization, 205
  - branches, 214–216
  - code improvement, 237–240-CD
  - instruction scheduling, 212
  - stalled processor, 211–216
- pipes in scripting languages, 680–681
- PL/I, overview, 800
- PL/I exception handlers, 442
- PLP CD, introduction, xxvi
- pointer reversal, garbage collection, 386–387
- pointers, 369, 142
  - C, 376
  - composite types, 318
  - dangling references, 379–382
  - frame pointer, 408
  - implementation, 369
  - object-oriented programming, 480
  - operations, 370–378
  - semantic analysis and, 27
  - stack pointer, 408
- to subroutines, C, 425
  - syntax, 370–378
- Polak, W., 468
- polling, 651
- polymorphism, 106
  - ad hoc, 148
  - binding with in scopes and, 145–148
  - functional programming, 526
  - object-oriented programming, 505–508
  - parametric, 146
  - subtype, 146, 310
    - dynamic method binding and, 498
    - type systems and, 309–311
- position-independent code, dynamic linking and, 195–196-CD
- positional parameters, 428–429
- POSIX, regular expressions, 730–731
- post-test loops, 284–285
- postconditions, 164
- PostScript, 679–680, 767
  - overview, 800
- pragmas, introduction, 60
- precedence
  - expressions and, 236–238
  - operators, 45
- preconditions, 164
- predefined identifiers, keywords and, 54
- predefined names, 126
- predefined objects, 117
- predicate calculus, logic languages and, 579
- predict-predict conflict, 76–77
- predict sets, 72–77
- preemption, 592
  - threads, 616–617
- prefixes, LL(1) grammars, 77
- preprocessors
  - C++ compiler, 18
  - interpreted languages, 15
- primary caches, memory hierarchy and, 197
- primitive expression types (Scheme), 541
- private keyword, 474
- private labels, object-oriented programming, 473–474
- procedural abstraction, 233
- procedures, subroutines, 407
- processes
  - heavyweight, 601
  - lightweight, 601
  - threads and, 601
- processors
  - architecture, 195
  - compiling and
    - modern processors, 210–221
- register allocation, 216–221
- implementation, 195
  - functional units, 204
- instruction scheduling, 212
- microprocessors, 206
- partitioning, 615
- status register, 197
  - condition codes, 202
  - vector processors, 598
- productions, context-free grammars, 42
- program counters (PC), 197
- program writing, binding and, 105
- programming environments, 21–22
- Programming Language Pragmatics, second edition changes, xxiv
- Programming Language Pragmatics, First Edition, xxiv
- programming languages. *See also* specific languages
  - assembly, introduction, 3–4
  - classification, 9
  - compiled, sidebar, 15
  - concurrency and, 603–604
  - constraint-based, 9
  - context-free, 94
  - dataflow, 9
  - declarative, 8
  - design (*See also* language design)
    - implementation and, 2
  - distinctions, 10
  - extension languages, 12
  - formal, 94
  - free format, 40
  - functional, 9
  - homoiconic, 575
  - imperative, 8
  - interpreted, sidebar, 15
  - machine language, introduction, 3
  - object-oriented, 10
  - reasons to study, 11–13
  - regular, 94
  - scripting, 10
  - syntax
    - exercises, 21-CD
    - explorations, 22-CD
  - von Neumann, 9
- programs as lists (Scheme), 535–537
- Prolog
  - arithmetic in, 565–566
  - backtracking, 566
  - backward chaining, 566–569
  - closed world assumption, 581–582
  - control flow, 571–574
  - databases and, 574–579
  - execution order and, 580
  - forward chaining, 566–569
  - functor, 561

- goals, 562
- Horn clauses, 561
- instantiation, 561
- lists, 564–565
- negation, 581–582
- not predicate, 580
- overview, 800
- queries, 562
- reflection, 578
- resolution, 563–564
- search strategy alternatives, 580
- tic-tac-toe example, 569–571
- unification, 563–564
- prologue, subroutines, 110
- property mechanisms, 475
- propositions, logic languages and, 579
- protected keyword, 485
- protected objects, 651
- pseudo-assembly notation, 209–210
- pthreads standard (POSIX), 603
- public keyword, 474
- public labels, object-oriented programming, 473–474
- pure virtual methods, 501
- push-down automata
  - CD and, 95
  - parsers, 94
  - syntax, 16–17-CD
- Python, 694–696
  - object orientation, 746–747
  - overview, 800
- queries
  - Prolog, 562
  - scripting languages, 679–680
- query language processors, 20
- Quiring, Sam B., 93
- quoting, scripting languages, 681–682
- R, overview, 800
- Rabin, Michael O., 102
  - Alonzo Church and, 525
- race condition, 592
- Randell, Brian, 467
- Rau, B. Ramakrishna, 259-CD
- read-only parameters, 421
- reader-writer locks, 622
- records
  - composite types, 318
  - data types, 336
    - memory layout and, 338–341
    - with statements, 341
  - fields, memory, 338
  - variants, fields, 347
- recursion, 39, 231, 233
- algorithmically inferior programs, 290
  - applicative evaluation, 291–295
  - lazy, 293–295
- continuation-passing style, 289
- Fortran, 109
- functional programming, 526
- iteration and, 287–291
- mutual recursion, 120
- normal-order evaluation, 291–295
- Scheme, 533–534
  - tail recursion, 289–290
- recursive descent, 64–70
  - phrase-level recover, syntax errors, 448–449
- recursive types, 369
- redeclarations, 123
- redirection, scripting languages, 680–681
- redundancy elimination, basic blocks, 209–217-CD
- reference counts, garbage collection, 384–385
- reference model, implementation, 240
- references
  - elliptical, 341
  - expressions, 239–241
  - forward, 174
  - initialization and, 489, 491–494
  - subroutines, 136
- referencing environment, 115
  - binding and, 136–142
- refinement of abstractions, 471
- reflection, Prolog and, 578
- register allocation, 216–221
  - code generation and, 772–773
  - code improvement and, 248–251-CD
- register indirect addressing, 201
- register-memory architecture, CISC machines, 201
- register-register architecture (RISC machines), 201
- register renaming, 208
- register window, 408
  - calling sequences, 414–415
  - control abstraction, 119–121-CD
  - subroutines, 119–121-CD
- registers
  - memory hierarchy and, 196
  - PC (program counter), 197
  - processor status, 197
  - saving/restoring, 410–411
- regular expressions, 2, 38–39
  - automata, 729
  - compiling, 735
  - description, 40–41
  - grep and (Unix), 41
- Kleene stars, 41
- literals, numeric, 41
- nested constructs and, 42
- NFA transition, 50–51
- parentheses, 41
- POSIX, 730–731
- syntax and, 38
  - tokens and, 39–41
- regular language, 39
- regular sets, 39
- relocation, linking and, 782–783
- relocation tables, 775
- remote-invocation sends, 646
- reordering loops, 241–248-CD
- repeated inheritance, 511
- replicated inheritance, 151–152-CD, 512
- Reps, Thomas, 194
- reserved words, 54. *See also* keywords
- resident monitor programs, 591
- resolution, Prolog, 563–564
- resource hazards, pipeline and, 211
- return values, subroutines and, 109
- reverse assignment, 504
- Rice, H. Gordon, 102
- right-most derivations, 44
- RISC (reduced instruction set computer) architecture, 196
- binary operations, 201
- implementation and, 207–208
- load-store architecture, 201
- register-register architecture, 201
- Ritchie, Dennis, 795
- RMIs (remote method invocation), 603
- Robinson, John Alan, 587
- Rochester Plan, xxviii
- Rosen, Barry, 259-CD
- Rosenkrantz, Daniel J., 101, 194
- Rounds, William C., 194
- Roussel, Philippe, 800
- row-pointer memory layout, arrays, 359–364
- RPCs (remote procedure calls), 603
  - implementation, 658–659
  - message passing, concurrency, 656
  - parameters, 658
  - semantics, 657
- Rubin, Frank, 305
- Ruby, 696–698
  - object orientation, 746–747
  - overview, 800–801
- run time, binding time and, 106
- Russell, Lawford J., 467
- Rutishauser, H., 102
- S-attributed attribute grammars, 168, 173–174

S-DSM (software distributed shared memory), 602  
 S-expressions (Scheme), 535  
 Samelson, K., 102  
 sandboxing, 711  
 scanner compilation phase, 23  
 code, 54–58  
 dot-dot problem, 57  
 finite automatons, 48–54  
 lexical analysis and, 24  
 lexical errors, 58–59  
 syntax analysis, 46  
 syntax and, 38  
 table-driven scanning, 58  
 tokens, 24  
 scanner generator tool, finite automatons and, 49  
 scheduler-based synchronization, 602  
 schedule implementation and, 626–627  
 schedulers, implementation, concurrency and, 623–627  
 scheduling  
 coscheduling, 615  
 gang scheduling, 615  
 instruction scheduling, 212  
 instructions, 30  
 multiprocessors, 618  
 uniprocessor, 615–616  
 Scheme  
 apply function, 535–536  
 assignment, 533–534  
 bindings, 530–531  
 cond, 533–534  
 control flow, 533–534  
 DFA (deterministic finite automaton) simulation, 537–538  
 equality testing, 532  
 eval function, 535–536  
 expression types, 541  
 for-each, 534  
 lambda expressions, 529–530  
 lists, 531–532  
 logical functions, 532  
 numbers, 531–532  
 numeric functions, 532  
 overview, 528–530, 801  
 programs as lists, 535–537  
 recursion, 533–534  
 S-expressions, 535  
 searches, 532  
 self-definition, 536–537  
 special forms, 530  
 symbols, 529  
 type predicate functions, 529  
 Scherer, William N. III, 670  
 Schiffman, Alan M., 518

Schneider, Fred B., 669  
 Schorr, Herbert, 406  
 Schroeder, Michael, 670  
 Schwartz, Jacob T., 259–CD  
 scientific notation, floating-point numbers, 200  
 scope, 2, 104  
 binding within  
 aliases, 142–143  
 overloading, 143–145  
 polymorphism, 145–148  
 dynamic, 131–134  
 exercises, 36–37–CD  
 explorations, 38–CD  
 global variables, 116  
 implementation, 135–136  
 association lists, 27–29–CD  
 central reference tables, 27–29–CD  
 symbol tables, 23–27–CD  
 Perl, 726–728  
 resolution operator, 474  
 rules  
 dynamic scope, 115  
 Fortran, 116  
 introduction, 114–115  
 lexical scope, 114  
 static scope, 114–116  
 scripting languages, 676, 723–728  
 variables, undeclared, 724–725  
 scopes  
 closed, modules, 128  
 open, 128  
 nested subroutines, 128  
 Scott, Dana S., 102, 194  
 Alonzo Church and, 525  
 Scott, Mark, 669  
 Scott, Michael L., 669  
 scripting languages, 10  
 #! conventions, 683–684  
 characteristics, 674–677  
 conditions, 679–680  
 data types, 676–677, 736–741  
 composite, 738–740  
 context, 740–741  
 numeric types, 737  
 object orientation, 741–747  
 declarations, 676  
 dynamic typing, 676  
 economy of expression, 674  
 escape sequences, 731–732  
 expansion, 681–682  
 extension languages and, 674, 698–701  
 filename expansion, 678–679  
 functions, 683  
 general purpose, 690–698  
 glue languages, 690–698  
 introduction, 671–672  
 magic numbers, 683  
 mathematics and, 689–690  
 Microsoft platforms, 673  
 modifiers, 731–732  
 names, 723–728  
 overview, 672–674  
 pattern manipulation, 728–735  
 minimal matches, 733  
 pattern matching, 676  
 Perl, 687–689  
 Perl extensions, 731  
 pipes, 680–681  
 POSIX regular expressions, 730–731  
 Python, 694–696  
 queries, 679–680  
 quoting, 681–682  
 redirection, 680–681  
 report generation, 684–689  
 Ruby, 696–698  
 sandboxing, 711  
 scope, 723–728  
 shell languages, 677–684  
 statistics, 689–690  
 string manipulation, 676, 728–735  
 Tcl, 692–694  
 tests, 679–680  
 text processing, 684–689  
 awk, 686–687  
 sed, 685–686  
 variable expansion, 678–679  
 variable interpolation, 733–735  
 World Wide Web  
 client-side scripts, 708  
 Java applets, 708–711  
 XHTML, 713–714  
 XML, 713–714  
 XSLT, 712–721  
 World Wide Web and, 701–702  
 CGI scripts, 702–703  
 server-side scripts, embedded, 703–707  
 searches, Scheme, 532  
 Sebesta, Robert W., 35  
 second-class subroutines, 140–142  
 sed, text processing and, 685–686  
 segment switching, 779  
 selection, 231, 233  
 selection statements, 261–262  
 Case statement, 265–268  
 conditions, short-circuited, 262–264  
 switch statement, 268–269  
 self-definition, Scheme and, 536–537  
 self-study, xxvii–xxviii  
 semantic action routines, 27  
 semantic analysis, 23  
 analyzers, 162–166  
 annotation, 27

- assertions, 164
- attribute space management
  - bottom-up evaluation, 39–44-CD
  - top-down evaluation, 44–49-CD
- back end, 163
- compilers, 163
- dynamic semantics, 27
- exercises, 51–52-CD
- explorations, 53-CD
- expression types, 25
- front end, 163
- identifier types, 25
- invariants, 164
- left corners, 43–44-CD
- optimization, 165
- postconditions, 164
- preconditions, 164
- static analysis, 165–166
- static semantics, 27
- symbol table and, 25
- semantic errors, syntax trees and, 182
- semantic functions, 166
  - notation, 167–168
- semantic hooks, 42–43-CD
- semantics, 2
  - annotations, 162
  - attribute grammars, 162
  - attributes, 162
  - decoration, 162
  - dynamic, 161
  - monitors, 631–634
  - parse tree and, 162
  - RPCs (remote procedure calls), 657
  - static, 161
  - syntax comparison, 38
  - syntax tree and, 162
- semaphores, 627–628
- sending messages. *See message passing, concurrency*
- sentential forms, 44
- separate compilation, 149
  - automatic header inference, 33–34-CD
  - C, 30–33-CD
  - module hierarchies, 35-CD
  - packages, 33–34-CD
- sequencing, 231, 233, 260–261
- server-side scripts, embedded, 703–707
- Sethi, Ravi, 35, 194, 790
- setjmp, 451–452
- sets, 367–368
  - composite types, 318
- SGML (standard generalized markup language), 712
- shallow binding, 115, 137
- Shamir, Eliahu, 102
- Shapiro, Ehud, 670
- shared inheritance, 512
- shared memory, concurrency, 601, 619–620
- CCRs (conditional critical regions), 634–638
- monitors, 629–634
  - condition variables, 630
  - nested monitor calls, 633
- schedulers, implementation, 623–627
- semaphores, 627–628
- synchronization
  - busy-wait, 620–623
  - implicit, 638–641
- shared-memory models, 521
- Sharp, Oliver J., 259-CD
- shell languages, 677–678. *See also scripting languages*
- short-circuit evaluation, 252–254
- sidebars. *See design & implementation sidebar*
- Siek, Jeremy, 468
- Siewiorek, Daniel P., 227
- significand, floating-point numbers, 200
- significant comments, 60. *See also pragmas*
- Simula, 158
  - overview, 801
- single inheritance, implementation, 503
- single precision floating-point numbers, 199
- Sipser, Michael, 102
- Sisal, overview, 801
- skolemization, 183–185-CD
- slices, arrays and, 352
- Smalltalk
  - environment, 21
  - object model, 158–161-CD
  - object-oriented programming and, 513
  - overview, 801
- Smith, James E., 228
- Sneeringer, W.J., 405
- Snobol
  - dynamic scoping, 132
  - overview, 801
- Snyder, Alan, 468
- software pipelining, code improvement and, 237–240-CD
- space sharing, 615
- special forms, Scheme, 530
- speculative optimization, 165
  - nonbinding prefetches, 165
  - optimistic compilers, 165
  - trace scheduling, 165
- spin locks, busy-waiting
  - synchronization, 620–622
- spinning, synchronization and, 602
- SR, overview, 801
- stack abstractions, 126
- stack allocation, coroutines, 455–457
- stack-based allocation, 109–111
- stack-based languages, 767
- stack frame, 412
- stack layout, subroutines, 408–410
- stack objects, storage allocation, 107, 109–111
- stack pointer, 408
- stacks, coroutines, 457
  - changing, 457–458
- stalled processor, pipeline and, 211–216
- stand-alone mode, 590
- Stansifer, Ryan D., 558, 798
- statement labels, 255
- statements, expressions and, 239
- static analysis
  - alias analysis, 165
  - escape analysis, 165
  - optimization, 165
  - semantics, 165–166
- static chains, 119, 120
  - displays, 413
  - maintaining, 411
  - subroutines, 409
- static linkers, 781
- static links
  - objects, 119
  - subroutines and, 409
- static method binding, 499
- static objects, storage allocation, 107–109
- static scope, 114
  - binding and, 139
  - overview, 115–116
  - subroutines and, 409
- static semantics, 27, 161
- statistics, scripting languages, 689–690
- Stearns, Richard E., 101, 194
- Steele, Guy Jr., 796, 801
- stop-and-copy garbage collection, 387–388
- storage allocation
  - bindings, 107
  - dynamic pool adjustment, 112
- storage management, 106–114
- Stoutamire, David, 468
- Stoy, Joseph E., 194
- Strachey, Christopher, 194
- streams, I/O, 542–545
- strings, 366–367
  - derivations, 44
  - generating, regular expressions and, 41
  - regular sets, 39
- scripting languages, 676, 728–735
- shell languages, 684
- Stroustrup, Bjarne, 518, 796

structured control flow, 234, 254–255  
 goto alternatives, 255–259  
 structured exceptions, 447  
 structured function returns, functional programming, 526  
 structured programming, 255  
 structures  
   data types, 336  
   functional languages, 173–176-CD  
 Stumm, Michael, 670  
 stylesheet languages, 713  
 subclasses, 476  
 subrange types, 316–317, 333–334  
 subroutines  
   actual parameters, 417–418  
   arguments and, 109  
     actual parameters, 407  
     format parameters, 407  
   bindings, 138  
   bookkeeping information and, 109  
 C on MIPS, 111–114-CD  
 call by name, 122–124-CD  
 callers, 407  
 calling sequence, 110, 407  
   displays, 413  
   example, 411–412  
   in-line expansion, 415–416  
   register windows, 414–415  
   saving/restoring registers, 410–411  
     static chain, 411  
 closures, 138–140  
   as parameters, 424–426  
 control abstractions, 104  
 discrete event simulation, 139–142-CD  
 displays, 107–110-CD  
 dynamic links, 409  
 epilogue, 110  
 exercises, 143–144-CD  
 explorations, 145-CD  
 first-class, 140–142  
 format parameters, 417–418  
 frame pointer, 408  
 frames, frame pointer, 110  
 generic, 434–441  
   C#, 132–134-CD  
   Java, 128–130-CD  
 generics, C++, 125–127-CD  
 goto alternatives, 256  
 iterator implementation, 135–137-CD  
 leaf routines, 413  
 libraries, Fortran, 16  
 members, 471  
 nesting, 117–119  
 object-oriented programming,  
   474–475  
 parameter passing, 417–418  
   call by name mode, 426–427

call by reference mode, 419  
 call by sharing mode, 420  
 call by value mode, 419  
 modes, 418–426  
 read-only parameters, 421  
 Pascal on the x86, 114–117-CD  
 pointers to, C, 425  
 procedures, 407  
 prologue, 110  
 references, 136  
 register allocation and, 219–221  
 register windows, 119–121-CD  
 return values and, 109  
 second-class, 140–142  
 stack frames, 408  
 stack layout, 408–410  
 stack pointer, 408  
 static allocation, 108  
 static chain, 409  
 static link, 409  
 static scoping and, 409  
 temporaries and, 109  
 subtype polymorphism, 146, 310  
   dynamic method binding and, 498  
 summarization, 550  
 supplemental materials, xxvii  
 Suraski, Zeev, 800  
 Sussman, Gerald Jay, 159, 558, 801  
 Sweeney, Peter F., 518  
 switch statement, 268–269  
 symbol identification, 779  
 symbol table  
   inherited attributes, 169  
   scope implementation and, 23–27-CD,  
     135  
   semantic analyzer and, 25  
 symbols, useless, 45  
 Syme, Don, 468  
 synchronization, concurrency and,  
   601–602  
   Ada 95, 634–635  
   blocking, 602  
   busy-waiting and, 602, 620  
     barriers, 622–623  
     spin locks, 620–622  
   condition synchronization, 619  
   implicit synchronization, 638–641  
   Java, 635–368  
   mutual exclusion, 619  
   scheduler based, 602, 626–627  
   semantics, message passing and,  
     646–650  
   spinning and, 602  
 synchronization sends, 646  
 syntactic analysis  
   compilation, 23–25  
   parsing and, 25  
 scanning and, 46  
 syntactic sugar, 159  
 syntax, 2  
 arrays, 349–353  
   declarations and, 350–352  
 AST (abstract syntax tree), 27  
 context-free grammars and, 25, 38  
 data types, 337–338  
 errors, 93  
 finite automata, 13–15-CD  
 grammar and language classes,  
   17–18-CD  
 introduction, 37  
 message passing, concurrency, 650  
 pointers, 370–378  
 push-down automata, 16–17-CD  
 recursion and, 39  
 regular expressions and, 38  
 semantics comparison, 38  
 thread creation, 604–613  
 syntax error recovery, 93  
   bottom-up parsers, 9–11-CD  
   context-sensitive look-ahead, 93  
   context-specific look-ahead, 3–4-CD  
   error productions, 5–6-CD  
   exception-based recover, 5-CD  
   panic mode, 1–2-CD, 93  
   phrase-level recovery, 2–3-CD, 93,  
     448–449  
 table-driven LL parsers, 6–9-CD  
 syntax tree  
   attribute grammars and, 175–176, 182  
   nodes, 182  
   semantic analysis and, 162  
   semantic errors and, 182  
   synthesized attributes, 168–169  
 Szyperski, Clemens, 468  
 table-driven parsers, 86  
 table-driven scanning, 58  
 table-driven top-down parsing, 70–80  
   LL(1) grammars, 77–80  
   predict sets, 72–77  
 tables  
   export, 775  
   import, 775  
   relocation, 775  
   vtable (virtual method table), 502  
 tail recursion, 289–290  
 Tanenbaum, Andrew S., 670  
 target code generation compilation  
   phase, 23  
   overview, 28–30  
 target machine architecture  
   computer arithmetic, 54–58-CD  
   exercises, 66–67-CD

- explorations, 68-CD
- MIPS, 59–64-CD
- x86, 59–64-CD
- task-parallel programs, 604
- tasks, 601
- Tcl/Tk, overview, 802
- Tcl (tool command language), 692–694
- Teitelbaum, Timothy, 194
- temporaries, subroutines and, 109
- terminals, context-free grammars, 42
- tests, scripting languages, 679–680
- TeX, 20
- text I/O, 96–97-CD
  - Ada, 98–99-CD
  - C, 99–101-CD
  - C++, 101–103-CD
  - Fortran, 97–98-CD
- text processing, scripting languages
  - awk, 686–687
  - sed, 685–686
- Thatcher, James W., 405
- this parameter, 483–484
- Thompson shell, 678
- threads
  - concurrency, implementation, 613–618
  - concurrency and, 601
    - creation syntax, 604–613
  - cooperative multithreading, 616
  - coroutines and, 454
  - implicit receipt, 611, 651
  - preemption, 616–617
  - processes, 601
  - race condition, 592
- tic-tac-toe example, Prolog, 569–571
- Tichy, Walter F., 790
- Tick, Evan, 670
- timesharing, concurrency and, 592–593
- titled examples, xxvi
- T.J. Watson Research Center, 227
- Tokens, 57
  - misspellings, 43
  - prefixes, 56
  - regular expressions and, 39–41
  - scanning and, 24
- tombstones, dangling references, 380–381
- top-down parsers, 62
  - attributes and, 181
  - evaluation, 44–49-CD
  - table-driven, 70–80
    - LL(1) grammars, 77–80
    - predict sets, 72–77
- Torczon, Linda, 35, 194, 228, 258-CD, 790
- trace scheduling, 165
- transfers, coroutines, 457–458
- transition tables, 58
- translation schemes
  - ad hoc, 179
  - attribute flow and, 173
  - dynamic, 173
  - oblivious, 173
- tree grammars, 182
  - context-free grammars and, 182
- trivial update problem, 551
- troff, 20
- Turing, overview, 802
- Turing, Alan, 524
  - Alonzo Church and, 525
- Turing machine, 524–525
- Turner, David, 798
- type checking, 231, 309
  - compatibility, 327–332
    - coercion, 328–331
    - composite types, 334–335
    - equivalence, 321–323
      - casts, 325–327
      - conversion, 325–327
      - name, 323–325
    - generic reference types, 331–332
    - inference, 332–335
      - composite types, 334–335
        - subranges, 333–334
      - linking and, 783–784
      - ML system, 335–336
      - type predicate functions, 529
    - type equivalence, 307
      - type systems and, 308
    - type extensions, encapsulation, 486–488
    - type inference, type systems and, 308
    - type predicate functions, 529
    - type systems, 231, 308–309
      - dynamic typing, 310
      - orthogonality, 319–319
    - polymorphism and, 309–311
      - implicit parametric polymorphism, 309
    - type checking, 309
    - type compatibility and, 308
    - type definition, 311–312
    - type equivalence and, 308
    - type inference and, 308
    - typeglob, Perl, 739
    - types, 231
      - abstraction-based point of view, 311
      - classifications, 312–318
      - compatibility, 307
        - type systems and, 308
      - composite, 317–318
      - constructive point of view, 311
      - decimal, 314
      - denotational point of view, 311
      - discrete, 314
    - enumeration, 315–316
    - numeric, 312–315
      - fixed-point, 314
    - opaque, 481
    - purpose, 307
    - subrange, 316–317, 333–334
  - UCS (Universal Character Set), 313
  - Ullman, Jeffrey D., 35, 101, 194, 790
  - undeclared variables, scope, 724–725
  - Unicode, 313
  - unification (Prolog), 563–564
  - uniform object model, 512
  - unions, data types, 336
  - uniprocessor scheduling, 615–616
  - Unix
    - grep, regular expressions and, 41
    - magic numbers, 683
  - unlimited extent, objects, 141
  - unsafe optimization, 165
  - unstructured control flow, 234, 254–255
  - useless symbols, 45
  - values
    - expressions, 239–241
    - initialization and, 489, 491–494
    - l-values, 239
  - van Rossum, Guido, 800
  - variables
    - context-free grammars, 42
    - exporting, 127
    - induction, code improvement and, 229–232-CD
  - instantiation
    - Prolog, 561
    - unification and, 563
  - interpolation, 733–735
  - reference models, 239–240
  - scope, 116
  - scripting languages, 678–679
  - semantic analysis and, 27
  - undeclared, scope, 724–725
  - variant records
    - composite types, 318
    - fields, 347
  - Vauquois, B., 102
  - VBScript, 673
  - vector processors, 598
  - virtual keyword, 500–501
  - virtual methods
    - dynamic method binding, 500–501
    - implementation, 503
  - virtual registers, 762
  - visibility, object-oriented programming, 485

- von Neumann languages, 9  
scripting languages and, 10  
vtable (virtual method table), 502
- Wadler, Philip, 468, 551  
Waite, William M., 406  
Wall, Larry, 672, 687, 799  
Wand, Mitchell, 468  
Watt, David Anthony, 194  
Wegman, Mark N., 259-CD  
Wegner, Peter, 159, 406, 518  
Wegstein, J.H., 102  
Weinberger, Peter, 686–687  
Weiser, Mark, 406  
Weiss, Shlomo, 228  
well-defined attribute grammars, 173  
Welsh, Jim, 405  
white space, 40  
van Wijngaarden, A., 102  
Wilhelm, Reinhard, 558  
Willcock, Jeremiah, 468
- Wilson, Paul R., 406  
Wiltamuth, Scott, 795  
Winch, David, 757  
Windows Script, 673  
Wirth, Niklaus, 18, 93, 159, 305, 798, 799  
with statements, 90–92-CD  
  records, 341  
Wolfe, Michael, 259-CD  
Woodger, M., 102  
World Wide Web, scripting and, 701–702  
  CGI scripts, 702–703  
  client-side scripts, 708  
  Java applets, 708–711  
  server-side scripts, embedded, 703–707  
  XHTML, 713–714  
  XML, 713–714  
  XSLT, 712–721  
wormhole routing, 599  
Wulf, William A., 790
- x86  
  example, 208–209  
Pascal, calling sequences, 414  
target machine architecture, 59–64-CD  
XHTML, 713–714  
XML (extensible markup language), 712, 713–714  
XPath, 715–716  
XSL (extensible stylesheet language), 712, 802  
XSL-FO, 715–716  
XSLT, 712–721  
  bibliographic formatting, 717  
  XPath and, 715–716  
  XSL-FO and, 715–716
- Yellin, Frank, 790  
Yochelson, Jerome C., 406  
Younger, Daniel H., 61
- Zadeck, F. Kenneth, 259-CD  
Zhou, Songnian, 670