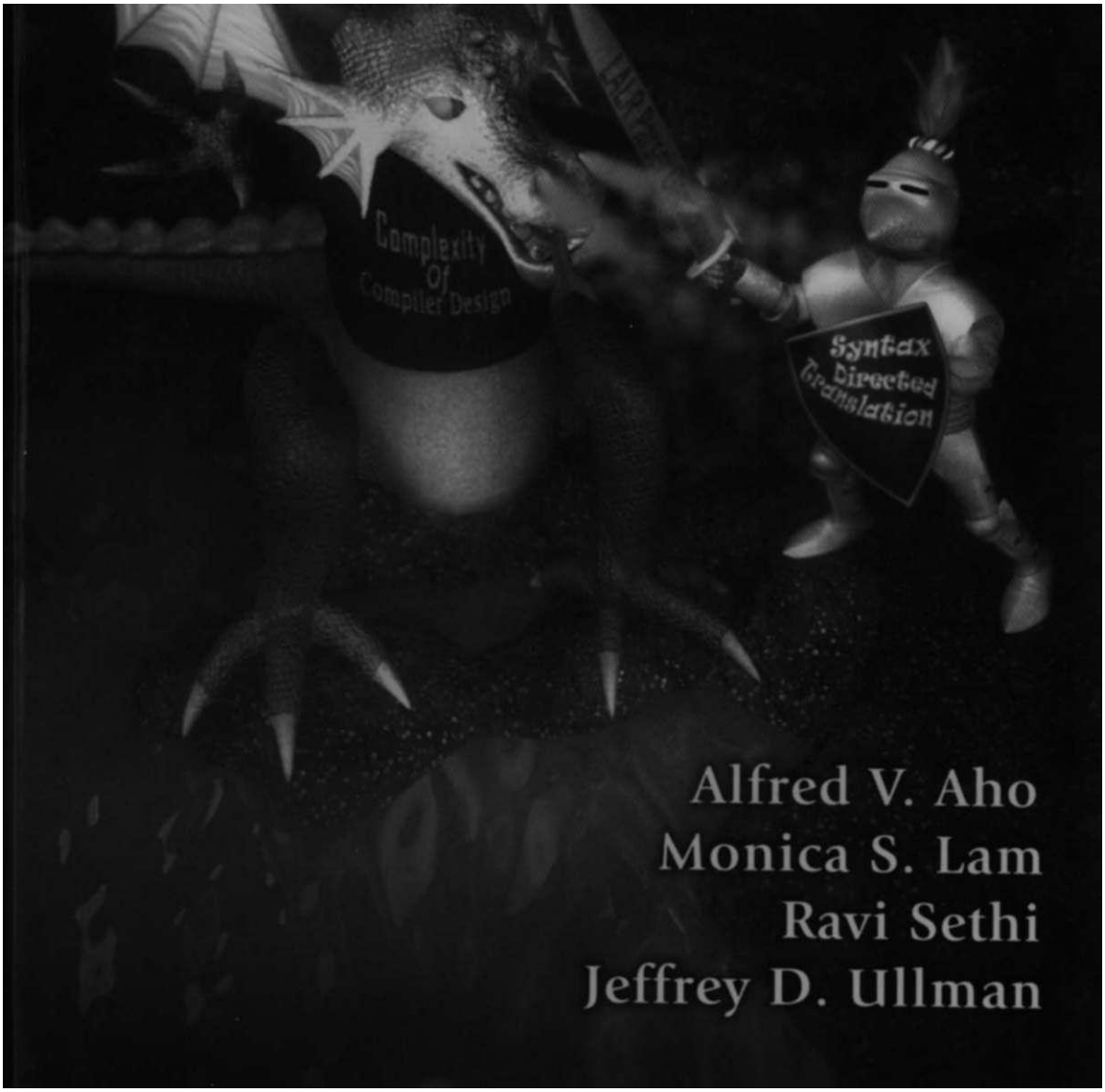
p

Compilers

Principles, Techniques, <§ Tools

Second Edition



Second
Edition

Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman

Compilers

Principles, Techniques, & Tools

Second Edition

Alfred V. Aho
Columbia University

Monica S. Lam
Stanford University

Ravi Sethi
Avaya

Jeffrey D. Ullman
Stanford University

PEARSON
Addison Wesley

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Publisher	Greg Tobin
Executive Editor	Michael Hirsch
Acquisitions Editor	Matt Goldstein
Project Editor	Katherine Harutunian
Associate Managing Editor	Jeffrey Holcomb
Cover Designer	Joyce Cosentino Wells
Digital Assets Manager	Marianne Groth
Media Producer	Bethany Tidd
Senior Marketing Manager	Michelle Brown
Marketing Assistant	Sarah Milmore
Senior Author Support/ Technology Specialist	Joe Vetere
Senior Manufacturing Buyer	Carol *Melville*
Cover Image	Scott Ullman of Strange Tonic Productions (www.strangetonic.com)

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

This interior of this book was composed in LaTeX.

Library of Congress Cataloging-in-Publication Data

Compilers : principles, techniques, and tools / Alfred V. Aho ... [et al.]. ~ 2nd ed.
p. cm.
Rev. ed. of: Compilers, principles, techniques, and tools / Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. 1986.
ISBN 0-321-48681-1 (alk. paper)
1. Compilers (Computer programs) I. Aho, Alfred V. II. Aho, Alfred V. Compilers, principles, techniques, and tools.
QA76.76.C65A37 2007
005.4'53~dc22

2006024333

Copyright © 2007 Pearson Education, Inc. All rights reserved. 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, recording, or otherwise, without the prior written permission of the publisher. Printed in the *United States of* America. For information on obtaining permission for use of material in this work, please submit a written request to Pearson Education, Inc., Rights and Contracts Department, 75 Arlington Street, Suite 300, Boston, MA 02116, fax your request to 617-848-7047, or e-mail at http://www.pearsoned.com/legal/permissions.htm.

2 3 4 5 6 7 8 9 10—CW—10 09 08 07 06

Preface

In the time since the 1986 edition of this book, the world of compiler design has changed significantly. Programming languages have evolved to present new compilation problems. Computer architectures offer a variety of resources of which the compiler designer must take advantage. Perhaps most interestingly, the venerable technology of code optimization has found use outside compilers. It is now used in tools that find bugs in software, and most importantly, find security holes in existing code. And much of the "front-end" technology — grammars, regular expressions, parsers, and syntax-directed translators — are still in wide use.

Thus, our philosophy from previous versions of the book has not changed. We recognize that few readers will build, or even maintain, a compiler for a major programming language. Yet the models, theory, and algorithms associated with a compiler can be applied to a wide range of problems in software design and software development. We therefore emphasize problems that are most commonly encountered in designing a language processor, regardless of the source language or target machine.

Use of the Book

It takes at least two quarters or even two semesters to cover all or most of the material in this book. It is common to cover the first half in an undergraduate course and the second half of the book — stressing code optimization — in a second course at the graduate or mezzanine level. Here is an outline of the chapters:

Chapter 1 contains motivational material and also presents some background issues in computer architecture and programming-language principles.

Chapter 2 develops a miniature compiler and introduces many of the important concepts, which are then developed in later chapters. The compiler itself appears in the appendix.

Chapter 3 covers lexical analysis, regular expressions, finite-state machines, and scanner-generator tools. This material is fundamental to text-processing of all sorts.

Chapter 4 covers the major parsing methods, top-down (recursive-descent, LL) and bottom-up (LR and its variants).

Chapter 5 introduces the principal ideas in syntax-directed definitions and syntax-directed translations.

Chapter 6 takes the theory of Chapter 5 and shows how to use it to generate intermediate code for a typical programming language.

Chapter 7 covers run-time environments, especially management of the run-time stack and garbage collection.

Chapter 8 is on object-code generation. It covers construction of basic blocks, generation of code from expressions and basic blocks, and register-allocation techniques.

Chapter 9 introduces the technology of code optimization, including flow graphs, data-flow frameworks, and iterative algorithms for solving these frameworks.

Chapter 10 covers instruction-level optimization. The emphasis is on the extraction of parallelism from small sequences of instructions and scheduling them on single processors that can do more than one thing at once.

Chapter 11 talks about larger-scale parallelism detection and exploitation. Here, the emphasis is on numeric codes that have many tight loops that range over multidimensional arrays.

Chapter 12 is on interprocedural analysis. It covers pointer analysis, aliasing, and data-flow analysis that takes into account the sequence of procedure calls that reach a given point in the code.

Courses from material in this book have been taught at Columbia, Harvard, and Stanford. At Columbia, a senior/first-year graduate course on programming languages and translators has been regularly offered using material from the first eight chapters. A highlight of this course is a semester-long project in which students work in small teams to create and implement a little language of their own design. The student-created languages have covered diverse application domains including quantum computation, music synthesis, computer graphics, gaming, matrix operations and many other areas. Students use compiler-component generators such as ANTLR, Lex, and Yacc and the syntax-directed translation techniques discussed in chapters two and five to build their compilers. A follow-on graduate course has focused on material in Chapters 9 through 12, emphasizing code generation and optimization for contemporary machines including network processors and multiprocessor architectures.

At Stanford, a one-quarter introductory course covers roughly the material in Chapters 1 through 8, although there is an introduction to global code optimization from Chapter 9. The second compiler course covers Chapters 9 through 12, plus the more advanced material on garbage collection from Chapter 7. Students use a locally developed, Java-based system called Joeq for implementing data-flow analysis algorithms.

Prerequisites

The reader should possess some "computer-science sophistication," including at least a second course on programming, and courses in data structures and discrete mathematics. Knowledge of several different programming languages is useful.

Exercises

The book contains extensive exercises, with some for almost every section. We indicate harder exercises or parts of exercises with an exclamation point. The hardest exercises have a double exclamation point.

Gradiance On-Line Homeworks

A feature of the new edition is that there is an accompanying set of on-line homeworks using a technology developed by Gradiance Corp. Instructors may assign these homeworks to their class, or students not enrolled in a class may enroll in an "omnibus class" that allows them to do the homeworks as a tutorial (without an instructor-created class). Gradiance questions look like ordinary questions, but your solutions are sampled. If you make an incorrect choice you are given specific advice or feedback to help you correct your solution. If your instructor permits, you are allowed to try again, until you get a perfect score.

A subscription to the Gradiance service is offered with all new copies of this text sold in North America. For more information, visit the Addison-Wesley web site www.aw.**com/gradiance** or send email to **computing**@aw.com.

Support on the World Wide Web

The book's home page is

dragonbook.Stanford.edu

Here, you will find errata as we learn of them, and backup materials. We hope to make available the notes for each offering of compiler-related courses as we teach them, including homeworks, solutions, and exams. We also plan to post descriptions of important compilers written by their implementers.

Acknowledgements

Cover art is by S. D. Ullman of Strange Tonic Productions.

Jon Bentley gave us extensive comments on a number of chapters of an earlier draft of this book. Helpful comments and errata were received from:

Domenico Bianculli, Peter Bosch, Marcio Buss, Marc Eaddy, Stephen Edwards, Vibhav Garg, Kim Hazelwood, Gaurav Kc, Wei Li, Mike Smith, Art Stamness, Krysta Svore, Olivier Tardieu, and Jia Zeng. The help of all these people is gratefully acknowledged. Remaining errors are ours, of course.

In addition, Monica would like to thank her colleagues on the SUIF compiler team for an 18-year lesson on compiling: Gerald Aigner, Dzintars Avots, Saman Amarasinghe, Jennifer Anderson, Michael Carbin, Gerald Cheong, Amer Diwan, Robert French, Anwar Ghuloum, Mary Hall, John Hennessy, David Heine, Shih-Wei Liao, Amy Lim, Benjamin Livshits, Michael Martin, Dror Maydan, Todd Mowry, Brian Murphy, Jeffrey Oplinger, Karen Pieper, Martin Rinard, Olatunji Ruwase, Constantine Sapuntzakis, Patrick Sathyanathan, Michael Smith, Steven Tjiang, Chau-Wen Tseng, Christopher Unkel, John Whaley, Robert Wilson, Christopher Wilson, and Michael Wolf.

A. V. A., Chatham NJ
M. S. L., Menlo Park CA
R. S., Far Hills NJ
J. D. U., Stanford CA
June, 2006

Table of Contents

1 Introduction **1**

1.1 Language Processors . . . 1

1.1.1 Exercises for Section 1.1 . . . 3

1.2 The Structure of a Compiler . . . 4

1.2.1 Lexical Analysis . . . 5

1.2.2 Syntax Analysis . . . 8

1.2.3 Semantic Analysis . . . 8

1.2.4 Intermediate Code Generation . . . 9

1.2.5 Code Optimization . . . 10

1.2.6 Code Generation . . . 10

1.2.7 Symbol-Table Management . . . 11

1.2.8 The Grouping of Phases into Passes . . . 11

1.2.9 Compiler-Construction Tools . . . 12

1.3 The Evolution of Programming Languages . . . 12

1.3.1 The Move to Higher-level Languages . . . 13

1.3.2 Impacts on Compilers . . . 14

1.3.3 Exercises for Section 1.3 . . . 14

1.4 The Science of Building a Compiler . . . 15

1.4.1 Modeling in Compiler Design and Implementation . . . 15

1.4.2 The Science of Code Optimization . . . 15

1.5 Applications of Compiler Technology . . . 17

1.5.1 Implementation of High-Level Programming Languages . 17

1.5.2 Optimizations for Computer Architectures . . . 19

1.5.3 Design of New Computer Architectures . . . 21

1.5.4 Program Translations . . . 22

1.5.5 Software Productivity Tools . . . 23

1.6 Programming Language Basics . . . 25

1.6.1 The Static/Dynamic Distinction . . . 25

1.6.2 Environments and States . . . 26

1.6.3 Static Scope and Block Structure . . . 28

1.6.4 Explicit Access Control . . . 31

1.6.5 Dynamic Scope . . . 31

1.6.6 Parameter Passing Mechanisms . . . 33

1.6.7 Aliasing . . . 35
1.6.8 Exercises for Section 1.6 . . . 35
1.7 Summary of Chapter 1 . . . 36
1.8 References for Chapter 1 . . . 38

A Simple Syntax-Directed Translator 39
2.1 Introduction . . . 40
2.2 Syntax Definition . . . 42
2.2.1 Definition of Grammars . . . 42
2.2.2 Derivations . . . 44
2.2.3 Parse Trees . . . 45
2.2.4 Ambiguity . . . 47
2.2.5 Associativity of Operators . . . 48
2.2.6 Precedence of Operators . . . 48
2.2.7 Exercises for Section 2.2 . . . 51
2.3 Syntax-Directed Translation . . . 52
2.3.1 Postfix Notation . . . 53
2.3.2 Synthesized Attributes . . . 54
2.3.3 Simple Syntax-Directed Definitions . . . 56
2.3.4 Tree Traversals . . . 56
2.3.5 Translation Schemes . . . 57
2.3.6 Exercises for Section 2.3 . . . 60
2.4 Parsing . . . 60
2.4.1 Top-Down Parsing . . . 61
2.4.2 Predictive Parsing . . . 64
2.4.3 When to Use e-Productions . . . 65
2.4.4 Designing a Predictive Parser . . . 66
2.4.5 Left Recursion . . . 67
2.4.6 Exercises for Section 2.4 . . . 68
2.5 A Translator for Simple Expressions . . . 68
2.5.1 Abstract and Concrete Syntax . . . 69
2.5.2 Adapting the Translation Scheme . . . 70
2.5.3 Procedures for the Nonterminals . . . 72
2.5.4 Simplifying the Translator . . . 73
2.5.5 The Complete Program . . . 74
2.6 Lexical Analysis . . . 76
2.6.1 Removal of White Space and Comments . . . 77
2.6.2 Reading Ahead . . . 78
2.6.3 Constants . . . 78
2.6.4 Recognizing Keywords and Identifiers . . . 79
2.6.5 A Lexical Analyzer . . . 81
2.6.6 Exercises for Section 2.6 . . . 84
2.7 Symbol Tables . . . 85
2.7.1 Symbol Table Per Scope . . . 86
2.7.2 The Use of Symbol Tables . . . 89

2.8 Intermediate Code Generation **91**
2.8.1 Two Kinds of Intermediate Representations **91**
2.8.2 Construction of Syntax Trees **92**
2.8.3 Static Checking **97**
2.8.4 Three-Address Code **99**
2.8.5 Exercises for Section **2.8** **105**
2.9 Summary of Chapter **2** **105**

3 Lexical Analysis
3.1 The Role of the Lexical Analyzer **109**
3.1.1 Lexical Analysis Versus Parsing **110**
3.1.2 Tokens, Patterns, and Lexemes Ill
3.1.3 Attributes for Tokens **112**
3.1.4 Lexical Errors **113**
3.1.5 Exercises for Section **3.1** **114**
3.2 Input Buffering **115**
3.2.1 Buffer Pairs **115**
3.2.2 Sentinels **116**
3.3 Specification of Tokens **116**
3.3.1 Strings and Languages **117**
3.3.2 Operations on Languages **119**
3.3.3 Regular Expressions **120**
3.3.4 Regular Definitions **123**
3.3.5 Extensions of Regular Expressions **124**
3.3.6 Exercises for Section **3.3** **125**
3.4 Recognition of Tokens **128**
3.4.1 Transition Diagrams **130**
3.4.2 Recognition of Reserved Words and Identifiers **132**
3.4.3 Completion of the Running Example **133**
3.4.4 Architecture of a Transition-Diagram-Based Lexical Analyzer **134**
3.4.5 Exercises for Section **3.4** **136**
3.5 The Lexical-Analyzer Generator **Lex** **140**
3.5.1 Use of **Lex** **140**
3.5.2 Structure of **Lex** Programs **141**
3.5.3 Conflict Resolution in **Lex** **144**
3.5.4 The Lookahead Operator **144**
3.5.5 Exercises for Section **3.5** **146**
3.6 Finite Automata **147**
3.6.1 Nondeterministic Finite Automata **147**
3.6.2 Transition Tables **148**
3.6.3 Acceptance of Input Strings by Automata **149**
3.6.4 Deterministic Finite Automata **149**
3.6.5 Exercises for Section **3.6** **151**
3.7 From Regular Expressions to Automata **152**

3.7.1 Conversion of an NFA to a DFA . . . 152
3.7.2 Simulation of an NFA . . . 156
3.7.3 Efficiency of NFA Simulation . . . 157
3.7.4 Construction of an NFA from a Regular Expression . . . 159
3.7.5 Efficiency of String-Processing Algorithms . . . 163
3.7.6 Exercises for Section 3.7 . . . 166
3.8 Design of a Lexical-Analyzer Generator . . . 166
3.8.1 The Structure of the Generated Analyzer . . . 167
3.8.2 Pattern Matching Based on NFA's . . . 168
3.8.3 DFA's for Lexical Analyzers . . . 170
3.8.4 Implementing the Lookahead Operator . . . 171
3.8.5 Exercises for Section 3.8 . . . 172
3.9 Optimization of DFA-Based Pattern Matchers . . . 173
3.9.1 Important States of an NFA . . . 173
3.9.2 Functions Computed From the Syntax Tree . . . 175
3.9.3 Computing *unliable*, *firstpos*, and *lastpos* . . . 176
3.9.4 Computing *followpos* . . . 177
3.9.5 Converting a Regular Expression Directly to a DFA . . . 179
3.9.6 Minimizing the Number of States of a DFA . . . 180
3.9.7 State Minimization in Lexical Analyzers . . . 184
3.9.8 Trading Time for Space in DFA Simulation . . . 185
3.9.9 Exercises for Section 3.9 . . . 186
3.10 Summary of Chapter 3 . . . 187
3.11 References for Chapter 3 . . . 189

4 Syntax Analysis 191
4.1 Introduction . . . 192
4.1.1 The Role of the Parser . . . 192
4.1.2 Representative Grammars . . . 193
4.1.3 Syntax Error Handling . . . 194
4.1.4 Error-Recovery Strategies . . . 195
4.2 Context-Free Grammars . . . 197
4.2.1 The Formal Definition of a Context-Free Grammar . . . 197
4.2.2 Notational Conventions . . . 198
4.2.3 Derivations . . . 199
4.2.4 Parse Trees and Derivations . . . 201
4.2.5 Ambiguity . . . 203
4.2.6 Verifying the Language Generated by a Grammar . . . 204
4.2.7 Context-Free Grammars Versus Regular Expressions . . . 205
4.2.8 Exercises for Section 4.2 . . . 206
4.3 Writing a Grammar . . . 209
4.3.1 Lexical Versus Syntactic Analysis . . . 209
4.3.2 Eliminating Ambiguity . . . 210
4.3.3 Elimination of Left Recursion . . . 212
4.3.4 Left Factoring . . . 214

4.3.5 Non-Context-Free Language Constructs . . . 215
4.3.6 Exercises for Section 4.3 . . . 216
4.4 Top-Down Parsing . . . 217
4.4.1 Recursive-Descent Parsing . . . 219
4.4.2 FIRST and FOLLOW . . . 220
4.4.3 LL(1) Grammars . . . 222
4.4.4 Nonrecursive Predictive Parsing . . . 226
4.4.5 Error Recovery in Predictive Parsing . . . 228
4.4.6 Exercises for Section 4.4 . . . 231
4.5 Bottom-Up Parsing . . . 233
4.5.1 Reductions . . . 234
4.5.2 Handle Pruning . . . 235
4.5.3 Shift-Reduce Parsing . . . 236
4.5.4 Conflicts During Shift-Reduce Parsing . . . 238
4.5.5 Exercises for Section 4.5 . . . 240
4.6 Introduction to LR Parsing: Simple LR . . . 241
4.6.1 Why LR Parsers? . . . 241
4.6.2 Items and the LR(0) Automaton . . . 242
4.6.3 The LR-Parsing Algorithm . . . 248
4.6.4 Constructing SLR-Parsing Tables . . . 252
4.6.5 Viable Prefixes . . . 256
4.6.6 Exercises for Section 4.6 . . . 257
4.7 More Powerful LR Parsers . . . 259
4.7.1 Canonical LR(1) Items . . . 260
4.7.2 Constructing LR(1) Sets of Items . . . 261
4.7.3 Canonical LR(1) Parsing Tables . . . 265
4.7.4 Constructing LALR Parsing Tables . . . 266
4.7.5 Efficient Construction of LALR Parsing Tables . . . 270
4.7.6 Compaction of LR Parsing Tables . . . 275
4.7.7 Exercises for Section 4.7 . . . 277
4.8 Using Ambiguous Grammars . . . 278
4.8.1 Precedence and Associativity to Resolve Conflicts . . . 279
4.8.2 The "Dangling-Else" Ambiguity . . . 281
4.8.3 Error Recovery in LR Parsing . . . 283
4.8.4 Exercises for Section 4.8 . . . 285
4.9 Parser Generators . . . 287
4.9.1 The Parser Generator **Yacc** . . . 287
4.9.2 Using **Yacc** with Ambiguous Grammars . . . 291
4.9.3 Creating **Yacc** Lexical Analyzers with Lex . . . 294
4.9.4 Error Recovery in **Yacc** . . . 295
4.9.5 Exercises for Section 4.9 . . . 297
4.10 Summary of Chapter 4 . . . 297
4.11 References for Chapter 4 . . . 300

5 Syntax-Directed Translation 303
5.1 Syntax-Directed Definitions . . . 304
5.1.1 Inherited and Synthesized Attributes . . . 304
5.1.2 Evaluating an SDD at the Nodes of a Parse Tree . . . 306
5.1.3 Exercises for Section 5.1 . . . 309
5.2 Evaluation Orders for SDD's . . . 310
5.2.1 Dependency Graphs . . . 310
5.2.2 Ordering the Evaluation of Attributes . . . 312
5.2.3 S-Attributed Definitions . . . 312
5.2.4 L-Attributed Definitions . . . 313
5.2.5 Semantic Rules with Controlled Side Effects . . . 314
5.2.6 Exercises for Section 5.2 . . . 317
5.3 Applications of Syntax-Directed Translation . . . 318
5.3.1 Construction of Syntax Trees . . . 318
5.3.2 The Structure of a Type . . . 321
5.3.3 Exercises for Section 5.3 . . . 323
5.4 Syntax-Directed Translation Schemes . . . 324
5.4.1 Postfix Translation Schemes . . . 324
5.4.2 Parser-Stack Implementation of Postfix SDT's . . . 325
5.4.3 SDT's With Actions Inside Productions . . . 327
5.4.4 Eliminating Left Recursion From SDT's . . . 328
5.4.5 SDT's for L-Attributed Definitions . . . 331
5.4.6 Exercises for Section 5.4 . . . 336
5.5 Implementing L-Attributed SDD's . . . 337
5.5.1 Translation During Recursive-Descent Parsing . . . 338
5.5.2 On-The-Fly Code Generation . . . 340
5.5.3 L-Attributed SDD's and LL Parsing . . . 343
5.5.4 Bottom-Up Parsing of L-Attributed SDD's . . . 348
5.5.5 Exercises for Section 5.5 . . . 352
5.6 Summary of Chapter 5 . . . 353
5.7 References for Chapter 5 . . . 354

6 Intermediate-Code Generation 357
6.1 Variants of Syntax Trees . . . 358
6.1.1 Directed Acyclic Graphs for Expressions . . . 359
6.1.2 The Value-Number Method for Constructing DAG's . . . 360
6.1.3 Exercises for Section 6.1 . . . 362
6.2 Three-Address Code . . . 363
6.2.1 Addresses and Instructions . . . 364
6.2.2 Quadruples . . . 366
6.2.3 Triples . . . 367
6.2.4 Static Single-Assignment Form . . . 369
6.2.5 Exercises for Section 6.2 . . . 370
6.3 Types and Declarations . . . 370
6.3.1 Type Expressions . . . 371

6.3.2 Type Equivalence . . . 372
6.3.3 Declarations . . . 373
6.3.4 Storage Layout for Local Names . . . 373
6.3.5 Sequences of Declarations . . . 376
6.3.6 Fields in Records and Classes . . . 376
6.3.7 Exercises for Section 6.3 . . . 378
6.4 Translation of Expressions . . . 378
6.4.1 Operations Within Expressions . . . 378
6.4.2 Incremental Translation . . . 380
6.4.3 Addressing Array Elements . . . 381
6.4.4 Translation of Array References . . . 383
6.4.5 Exercises for Section 6.4 . . . 384
6.5 Type Checking . . . 386
6.5.1 Rules for Type Checking . . . 387
6.5.2 Type Conversions . . . 388
6.5.3 Overloading of Functions and Operators . . . 390
6.5.4 Type Inference and Polymorphic Functions . . . 391
6.5.5 An Algorithm for Unification . . . 395
6.5.6 Exercises for Section 6.5 . . . 398
6.6 Control Flow . . . 399
6.6.1 Boolean Expressions . . . 399
6.6.2 Short-Circuit Code . . . 400
6.6.3 Flow-of-Control Statements . . . 401
6.6.4 Control-Flow Translation of Boolean Expressions . . . 403
6.6.5 Avoiding Redundant Gotos . . . 405
6.6.6 Boolean Values and Jumping Code . . . 408
6.6.7 Exercises for Section 6.6 . . . 408
6.7 Backpatching . . . 410
6.7.1 One-Pass Code Generation Using Backpatching . . . 410
6.7.2 Backpatching for Boolean Expressions . . . 411
6.7.3 Flow-of-Control Statements . . . 413
6.7.4 Break-, Continue-, and Goto-Statements . . . 416
6.7.5 Exercises for Section 6.7 . . . 417
6.8 Switch-Statements . . . 418
6.8.1 Translation of Switch-Statements . . . 419
6.8.2 Syntax-Directed Translation of Switch-Statements . . . 420
6.8.3 Exercises for Section 6.8 . . . 421
6.9 Intermediate Code for Procedures . . . 422
6.10 Summary of Chapter 6 . . . 424
6.11 References for Chapter 6 . . . 425

7 Run-Time Environments 427
7.1 Storage Organization . . . 427
7.1.1 Static Versus Dynamic Storage Allocation . . . 429
7.2 Stack Allocation of Space . . . 430
7.2.1 Activation Trees . . . 430
7.2.2 Activation Records . . . 433
7.2.3 Calling Sequences . . . 436
7.2.4 Variable-Length Data on the Stack . . . 438
7.2.5 Exercises for Section 7.2 . . . 440
7.3 Access to Nonlocal Data on the Stack . . . 441
7.3.1 Data Access Without Nested Procedures . . . 442
7.3.2 Issues With Nested Procedures . . . 442
7.3.3 A Language With Nested Procedure Declarations . . . 443
7.3.4 Nesting Depth . . . 443
7.3.5 Access Links . . . 445
7.3.6 Manipulating Access Links . . . 447
7.3.7 Access Links for Procedure Parameters . . . 448
7.3.8 Displays . . . 449
7.3.9 Exercises for Section 7.3 . . . 451
7.4 Heap Management . . . 452
7.4.1 The Memory Manager . . . 453
7.4.2 The Memory Hierarchy of a Computer . . . 454
7.4.3 Locality in Programs . . . 455
7.4.4 Reducing Fragmentation . . . 457
7.4.5 Manual Deallocation Requests . . . 460
7.4.6 Exercises for Section 7.4 . . . 463
7.5 Introduction to Garbage Collection . . . 463
7.5.1 Design Goals for Garbage Collectors . . . 464
7.5.2 Reachability . . . 466
7.5.3 Reference Counting Garbage Collectors . . . 468
7.5.4 Exercises for Section 7.5 . . . 470
7.6 Introduction to Trace-Based Collection . . . 470
7.6.1 A Basic Mark-and-Sweep Collector . . . 471
7.6.2 Basic Abstraction . . . 473
7.6.3 Optimizing Mark-and-Sweep . . . 475
7.6.4 Mark-and-Compact Garbage Collectors . . . 476
7.6.5 Copying collectors . . . 478
7.6.6 Comparing Costs . . . 482
7.6.7 Exercises for Section 7.6 . . . 482
7.7 Short-Pause Garbage Collection . . . 483
7.7.1 Incremental Garbage Collection . . . 483
7.7.2 Incremental Reachability Analysis . . . 485
7.7.3 Partial-Collection Basics . . . 487
7.7.4 Generational Garbage Collection . . . 488
7.7.5 The Train Algorithm . . . 490

7.7.6 Exercises for Section 7.7 . . . 493
7.8 Advanced Topics in Garbage Collection . . . 494
7.8.1 Parallel and Concurrent Garbage Collection . . . 495
7.8.2 Partial Object Relocation . . . 497
7.8.3 Conservative Collection for Unsafe Languages . . . 498
7.8.4 Weak References . . . 498
7.8.5 Exercises for Section 7.8 . . . 499
7.9 Summary of Chapter 7 . . . 500
7.10 References for Chapter 7 . . . 502

8 Code Generation 505
8.1 Issues in the Design of a Code Generator . . . 506
8.1.1 Input to the Code Generator . . . 507
8.1.2 The Target Program . . . 507
8.1.3 Instruction Selection . . . 508
8.1.4 Register Allocation . . . 510
8.1.5 Evaluation Order . . . 511
8.2 The Target Language . . . 512
8.2.1 A Simple Target Machine Model . . . 512
8.2.2 Program and Instruction Costs . . . 515
8.2.3 Exercises for Section 8.2 . . . 516
8.3 Addresses in the Target Code . . . 518
8.3.1 Static Allocation . . . 518
8.3.2 Stack Allocation . . . 520
8.3.3 Run-Time Addresses for Names . . . 522
8.3.4 Exercises for Section 8.3 . . . 524
8.4 Basic Blocks and Flow Graphs . . . 525
8.4.1 Basic Blocks . . . 526
8.4.2 Next-Use Information . . . 528
8.4.3 Flow Graphs . . . 529
8.4.4 Representation of Flow Graphs . . . 530
8.4.5 Loops . . . 531
8.4.6 Exercises for Section 8.4 . . . 531
8.5 Optimization of Basic Blocks . . . 533
8.5.1 The DAG Representation of Basic Blocks . . . 533
8.5.2 Finding Local Common Subexpressions . . . 534
8.5.3 Dead Code Elimination . . . 535
8.5.4 The Use of Algebraic Identities . . . 536
8.5.5 Representation of Array References . . . 537
8.5.6 Pointer Assignments and Procedure Calls . . . 539
8.5.7 Reassembling Basic Blocks From DAG's . . . 539
8.5.8 Exercises for Section 8.5 . . . 541
8.6 A Simple Code Generator . . . 542
8.6.1 Register and Address Descriptors . . . 543
8.6.2 The Code-Generation Algorithm . . . 544

8.6.3 Design of the Function *getReg* . . . 547
8.6.4 Exercises for Section 8.6 . . . 548
8.7 Peephole Optimization . . . 549
8.7.1 Eliminating Redundant Loads and Stores . . . 550
8.7.2 Eliminating Unreachable Code . . . 550
8.7.3 Flow-of-Control Optimizations . . . 551
8.7.4 Algebraic Simplification and Reduction in Strength . . . 552
8.7.5 Use of Machine Idioms . . . 552
8.7.6 Exercises for Section 8.7 . . . 553
8.8 Register Allocation and Assignment . . . 553
8.8.1 Global Register Allocation . . . 553
8.8.2 Usage Counts . . . 554
8.8.3 Register Assignment for Outer Loops . . . 556
8.8.4 Register Allocation by Graph Coloring . . . 556
8.8.5 Exercises for Section 8.8 . . . 557
8.9 Instruction Selection by Tree Rewriting . . . 558
8.9.1 Tree-Translation Schemes . . . 558
8.9.2 Code Generation by Tiling an Input Tree . . . 560
8.9.3 Pattern Matching by Parsing . . . 563
8.9.4 Routines for Semantic Checking . . . 565
8.9.5 General Tree Matching . . . 565
8.9.6 Exercises for Section 8.9 . . . 567
8.10 Optimal Code Generation for Expressions . . . 567
8.10.1 Ershov Numbers . . . 567
8.10.2 Generating Code From Labeled Expression Trees . . . 568
8.10.3 Evaluating Expressions with an Insufficient Supply of Registers . . . 570
8.10.4 Exercises for Section 8.10 . . . 572
8.11 Dynamic Programming Code-Generation . . . 573
8.11.1 Contiguous Evaluation . . . 574
8.11.2 The Dynamic Programming Algorithm . . . 575
8.11.3 Exercises for Section 8.11 . . . 577
8.12 Summary of Chapter 8 . . . 578
8.13 References for Chapter 8 . . . 579

9 Machine-Independent Optimizations 583
9.1 The Principal Sources of Optimization . . . 584
9.1.1 Causes of Redundancy . . . 584
9.1.2 A Running Example: Quicksort . . . 585
9.1.3 Semantics-Preserving Transformations . . . 586
9.1.4 Global Common Subexpressions . . . 588
9.1.5 Copy Propagation . . . 590
9.1.6 Dead-Code Elimination . . . 591
9.1.7 Code Motion . . . 592
9.1.8 Induction Variables and Reduction in Strength . . . 592

9.1.9 Exercises for Section 9.1 . . . 596
9.2 Introduction to Data-Flow Analysis . . . 597
9.2.1 The Data-Flow Abstraction . . . 597
9.2.2 The Data-Flow Analysis Schema . . . 599
9.2.3 Data-Flow Schemas on Basic Blocks . . . 600
9.2.4 Reaching Definitions . . . 601
9.2.5 Live-Variable Analysis . . . 608
9.2.6 Available Expressions . . . 610
9.2.7 Summary . . . 614
9.2.8 Exercises for Section 9.2 . . . 615
9.3 Foundations of Data-Flow Analysis . . . 618
9.3.1 Semilattices . . . 618
9.3.2 Transfer Functions . . . 623
9.3.3 The Iterative Algorithm for General Frameworks . . . 626
9.3.4 Meaning of a Data-Flow Solution . . . 628
9.3.5 Exercises for Section 9.3 . . . 631
9.4 Constant Propagation . . . 632
9.4.1 Data-Flow Values for the Constant-Propagation Framework . . . 633
9.4.2 The Meet for the Constant-Propagation Framework . . . 633
9.4.3 Transfer Functions for the Constant-Propagation Framework . . . 634
9.4.4 Monotonicity of the Constant-Propagation Framework . . . 635
9.4.5 Nondistributivity of the Constant-Propagation Framework 635
9.4.6 Interpretation of the Results . . . 637
9.4.7 Exercises for Section 9.4 . . . 637
9.5 Partial-Redundancy Elimination . . . 639
9.5.1 The Sources of Redundancy . . . 639
9.5.2 Can All Redundancy Be Eliminated? . . . 642
9.5.3 The Lazy-Code-Motion Problem . . . 644
9.5.4 Anticipation of Expressions . . . 645
9.5.5 The Lazy-Code-Motion Algorithm . . . 646
9.5.6 Exercises for Section 9.5 . . . 655
9.6 Loops in Flow Graphs . . . 655
9.6.1 Dominators . . . 656
9.6.2 Depth-First Ordering . . . 660
9.6.3 Edges in a Depth-First Spanning Tree . . . 661
9.6.4 Back Edges and Reducibility . . . 662
9.6.5 Depth of a Flow Graph . . . 665
9.6.6 Natural Loops . . . 665
9.6.7 Speed of Convergence of Iterative Data-Flow Algorithms . 667
9.6.8 Exercises for Section 9.6 . . . 669
9.7 Region-Based Analysis . . . 672
9.7.1 Regions . . . 672
9.7.2 Region Hierarchies for Reducible Flow Graphs . . . 673

9.7.3 Overview of a Region-Based Analysis 676
9.7.4 Necessary Assumptions About Transfer Functions 678
9.7.5 An Algorithm for Region-Based Analysis 680
9.7.6 Handling Nonreducible Flow Graphs 684
9.7.7 Exercises for Section 9.7 686
9.8 Symbolic Analysis 686
9.8.1 Affine Expressions of Reference Variables 687
9.8.2 Data-Flow Problem Formulation 689
9.8.3 Region-Based Symbolic Analysis 694
9.8.4 Exercises for Section 9.8 699
9.9 Summary of Chapter 9 700
9.10 References for Chapter 9 703

10 Instruction-Level Parallelism 707
10.1 Processor Architectures 708
10.1.1 Instruction Pipelines and Branch Delays 708
10.1.2 Pipelined Execution 709
10.1.3 Multiple Instruction Issue 710
10.2 Code-Scheduling Constraints 710
10.2.1 Data Dependence 711
10.2.2 Finding Dependences Among Memory Accesses 712
10.2.3 Tradeoff Between Register Usage and Parallelism 713
10.2.4 Phase Ordering Between Register Allocation and Code Scheduling 716
10.2.5 Control Dependence 716
10.2.6 Speculative Execution Support 717
10.2.7 A Basic Machine Model 719
10.2.8 Exercises for Section 10.2 720
10.3 Basic-Block Scheduling 721
10.3.1 Data-Dependence Graphs 722
10.3.2 List Scheduling of Basic Blocks 723
10.3.3 Prioritized Topological Orders 725
10.3.4 Exercises for Section 10.3 726
10.4 Global Code Scheduling 727
10.4.1 Primitive Code Motion 728
10.4.2 Upward Code Motion 730
10.4.3 Downward Code Motion 731
10.4.4 Updating Data Dependences 732
10.4.5 Global Scheduling Algorithms 732
10.4.6 Advanced Code Motion Techniques 736
10.4.7 Interaction with Dynamic Schedulers 737
10.4.8 Exercises for Section 10.4 737
10.5 Software Pipelining 738
10.5.1 Introduction 738
10.5.2 Software Pipelining of Loops 740

10.5.3 Register Allocation and Code Generation 743
10.5.4 Do-Across Loops 743
10.5.5 Goals and Constraints of Software Pipelining 745
10.5.6 A Software-Pipelining Algorithm 749
10.5.7 Scheduling Acyclic Data-Dependence Graphs 749
10.5.8 Scheduling Cyclic Dependence Graphs 751
10.5.9 Improvements to the Pipelining Algorithms 758
10.5.10 Modular Variable Expansion 758
10.5.11 Conditional Statements 761
10.5.12 Hardware Support for Software Pipelining 762
10.5.13 Exercises for Section 10.5 763
10.6 Summary of Chapter 10 765
10.7 References for Chapter 10 766

11 Optimizing for Parallelism and Locality 769
11.1 Basic Concepts 771
11.1.1 Multiprocessors 772
11.1.2 Parallelism in Applications 773
11.1.3 Loop-Level Parallelism 775
11.1.4 Data Locality 777
11.1.5 Introduction to Affine Transform Theory 778
11.2 Matrix Multiply: An In-Depth Example 782
11.2.1 The Matrix-Multiplication Algorithm 782
11.2.2 Optimizations 785
11.2.3 Cache Interference 788
11.2.4 Exercises for Section 11.2 788
11.3 Iteration Spaces 788
11.3.1 Constructing Iteration Spaces from Loop Nests 788
11.3.2 Execution Order for Loop Nests 791
11.3.3 Matrix Formulation of Inequalities 791
11.3.4 Incorporating Symbolic Constants 793
11.3.5 Controlling the Order of Execution 793
11.3.6 Changing Axes 798
11.3.7 Exercises for Section 11.3 799
11.4 Affine Array Indexes 801
11.4.1 Affine Accesses 802
11.4.2 Affine and Nonaffine Accesses in Practice 803
11.4.3 Exercises for Section 11.4 804
11.5 Data Reuse 804
11.5.1 Types of Reuse 805
11.5.2 Self Reuse 806
11.5.3 Self-Spatial Reuse 809
11.5.4 Group Reuse 811
11.5.5 Exercises for Section 11.5 814
11.6 Array Data-Dependence Analysis 815

11.6.1 Definition of Data Dependence of Array Accesses 816
11.6.2 Integer Linear Programming 817
11.6.3 The GCD Test 818
11.6.4 Heuristics for Solving Integer Linear Programs 820
11.6.5 Solving General Integer Linear Programs 823
11.6.6 Summary 825
11.6.7 Exercises for Section 11.6 826
11.7 Finding Synchronization-Free Parallelism 828
11.7.1 An Introductory Example 828
11.7.2 Affine Space Partitions 830
11.7.3 Space-Partition Constraints 831
11.7.4 Solving Space-Partition Constraints 835
11.7.5 A Simple Code-Generation Algorithm 838
11.7.6 Eliminating Empty Iterations 841
11.7.7 Eliminating Tests from Innermost Loops 844
11.7.8 Source-Code Transforms 846
11.7.9 Exercises for Section 11.7 851
11.8 Synchronization Between Parallel Loops 853
11.8.1 A Constant Number of Synchronizations 853
11.8.2 Program-Dependence Graphs 854
11.8.3 Hierarchical Time 857
11.8.4 The Parallelization Algorithm 859
11.8.5 Exercises for Section 11.8 860
11.9 Pipelining 861
11.9.1 What is Pipelining? 861
11.9.2 Successive Over-Relaxation (SOR): An Example 863
11.9.3 Fully Permutable Loops 864
11.9.4 Pipelining Fully Permutable Loops 864
11.9.5 General Theory 867
11.9.6 Time-Partition Constraints 868
11.9.7 Solving Time-Partition Constraints by Farkas' Lemma . . 872
11.9.8 Code Transformations 875
11.9.9 Parallelism With Minimum Synchronization 880
11.9.10 Exercises for Section 11.9 882
11.10 Locality Optimizations 884
11.10.1 Temporal Locality of Computed Data 885
11.10.2 Array Contraction 885
11.10.3 Partition Interleaving 887
11.10.4 Putting it All Together 890
11.10.5 Exercises for Section 11.10 892
11.11 Other Uses of Affine Transforms 893
11.11.1 Distributed memory machines 894
11.11.2 Multi-Instruction-Issue Processors 895
11.11.3 Vector and SIMD Instructions 895
11.11.4 Prefetching 896

11.12 Summary of Chapter 11 . . . 897
11.13 References for Chapter 11 . . . 899

12 Interprocedural Analysis 903
12.1 Basic Concepts . . . 904
12.1.1 Call Graphs . . . 904
12.1.2 Context Sensitivity . . . 906
12.1.3 Call Strings . . . 908
12.1.4 Cloning-Based Context-Sensitive Analysis . . . 910
12.1.5 Summary-Based Context-Sensitive Analysis . . . 911
12.1.6 Exercises for Section 12.1 . . . 914
12.2 Why Interprocedural Analysis? . . . 916
12.2.1 Virtual Method Invocation . . . 916
12.2.2 Pointer Alias Analysis . . . 917
12.2.3 Parallelization . . . 917
12.2.4 Detection of Software Errors and Vulnerabilities . . . 917
12.2.5 SQL Injection . . . 918
12.2.6 Buffer Overflow . . . 920
12.3 A Logical Representation of Data Flow . . . 921
12.3.1 Introduction to Datalog . . . 921
12.3.2 Datalog Rules . . . 922
12.3.3 Intensional and Extensional Predicates . . . 924
12.3.4 Execution of Datalog Programs . . . 927
12.3.5 Incremental Evaluation of Datalog Programs . . . 928
12.3.6 Problematic Datalog Rules . . . 930
12.3.7 Exercises for Section 12.3 . . . 932
12.4 A Simple Pointer-Analysis Algorithm . . . 933
12.4.1 Why is Pointer Analysis Difficult . . . 934
12.4.2 A Model for Pointers and References . . . 935
12.4.3 Flow Insensitivity . . . 936
12.4.4 The Formulation in Datalog . . . 937
12.4.5 Using Type Information . . . 938
12.4.6 Exercises for Section 12.4 . . . 939
12.5 Context-Insensitive Interprocedural Analysis . . . 941
12.5.1 Effects of a Method Invocation . . . 941
12.5.2 Call Graph Discovery in Datalog . . . 943
12.5.3 Dynamic Loading and Reflection . . . 944
12.5.4 Exercises for Section 12.5 . . . 945
12.6 Context-Sensitive Pointer Analysis . . . 945
12.6.1 Contexts and Call Strings . . . 946
12.6.2 Adding Context to Datalog Rules . . . 949
12.6.3 Additional Observations About Sensitivity . . . 949
12.6.4 Exercises for Section 12.6 . . . 950
12.7 Datalog Implementation by BDD's . . . 951
12.7.1 Binary Decision Diagrams . . . 951

12.7.2 Transformations on BDD's 953
12.7.3 Representing Relations by BDD's 954
12.7A Relational Operations as BDD Operations 954
12.7.5 Using BDD's for Points-to Analysis 957
12.7.6 Exercises for Section 12.7 958
12.8 Summary of Chapter 12 958
12.9 References for Chapter 12 961

A A Complete Front End 965
A.1 The Source Language 965
A.2 Main 966
A.3 Lexical Analyzer 967
A.4 Symbol Tables and Types 970
A.5 Intermediate Code for Expressions 971
A.6 Jumping Code for Boolean Expressions 974
A.7 Intermediate Code for Statements 978
A.8 Parser 981
A.9 Creating the Front End 986

B Finding Linearly Independent Solutions 989

Index 993

Chapter 1

Introduction

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.

The software systems that do this translation are called *compilers*.

This book is about how to design and implement compilers. We shall discover that a few basic ideas can be used to construct translators for a wide variety of languages and machines. Besides compilers, the principles and techniques for compiler design are applicable to so many other domains that they are likely to be reused many times in the career of a computer scientist. The study of compiler writing touches upon programming languages, machine architecture, language theory, algorithms, and software engineering.

In this preliminary chapter, we introduce the different forms of language translators, give a high level overview of the structure of a typical compiler, and discuss the trends in programming languages and machine architecture that are shaping compilers. We include some observations on the relationship between compiler design and computer-science theory and an outline of the applications of compiler technology that go beyond compilation. We end with a brief outline of key programming-language concepts that will be needed for our study of compilers.

1.1 Language Processors

Simply stated, a compiler is a program that can read a program in one language — the *source* language — and translate it into an equivalent program in another language — the *target* language; see Fig. 1.1. An important role of the compiler is to report any errors in the source program that it detects during the translation process.

source program

Compiler

target program

Figure 1.1: A compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs; see Fig. 1.2.

input Target Program output

Figure 1.2: Running the target program

An *interpreter* is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in Fig. 1.3.

source program
input Interpreter output

Figure 1.3: An interpreter

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Example 1.1: Java language processors combine compilation and interpretation, as shown in Fig. 1.4. A Java source program may first be compiled into an intermediate form called *bytecodes*. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

In order to achieve faster processing of inputs to outputs, some Java compilers, called *just-in-time* compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input. •

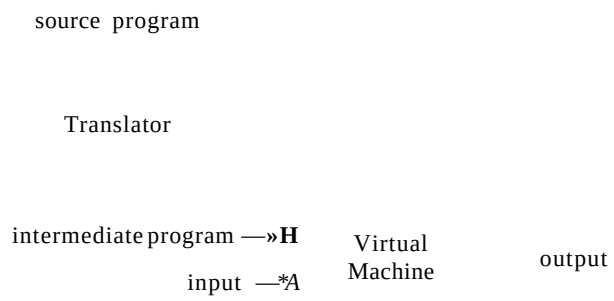


Figure 1.4: A hybrid compiler

In addition to a compiler, several other programs may be required to create an executable target program, as shown in Fig. 1.5. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a *preprocessor*. The preprocessor may also expand shorthands, called macros, into source language statements.

The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an *assembler* that produces relocatable machine code as its output.

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file. The *loader* then puts together all of the executable object files into memory for execution.

1.1.1 Exercises for Section 1.1

Exercise 1.1.1: What is the difference between a compiler and an interpreter?

Exercise 1.1.2: What are the advantages of (a) a compiler over an interpreter (b) an interpreter over a compiler?

Exercise 1.1.3 : What advantages are there to a language-processing system in which the compiler produces assembly language rather than machine language?

Exercise 1.1.4: A compiler that translates a high-level language into another high-level language is called a *source-to-source* translator. What advantages are there to using C as a target language for a compiler?

Exercise 1.1.5: Describe some of the tasks that an assembler needs to perform.

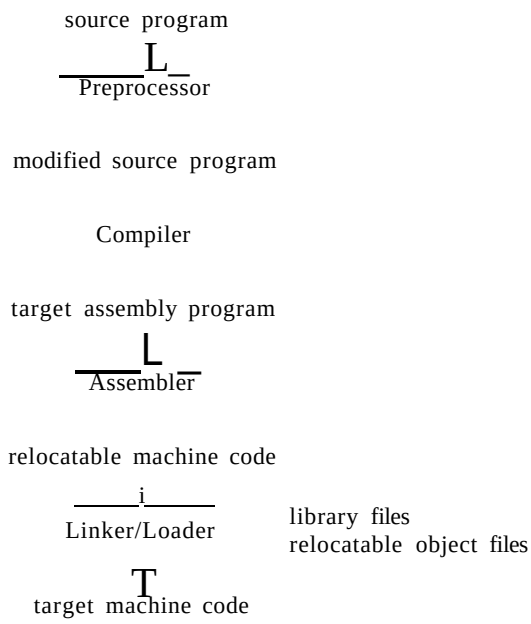


Figure 1.5: A language-processing system

1.2 The Structure of a Compiler

Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.

The *analysis* part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a *symbol table*, which is passed along with the intermediate representation to the synthesis part.

The *synthesis* part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the *front end* of the compiler; the synthesis part is the *back end*.

If we examine the compilation process in more detail, we see that it operates as a sequence of *phases*, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in Fig. 1.6. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the

character stream

Lexical Analyzer

token stream

Syntax Analyzer

syntax tree

Semantic Analyzer

syntax tree

Symbol Table

Intermediate Code Generator

intermediate representation

Machine-Independent Code Optimizer

intermediate representation

Code Generator

target-machine code

Machine-Dependent Code Optimizer

target-machine code

Figure 1.6: Phases of a compiler

entire source program, is used by all phases of the compiler.

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in Fig. 1.6 may be missing.

1.2.1 Lexical Analysis

The first phase of a compiler is called *lexical analysis* or *scanning*. The lexical analyzer reads the stream of characters making up the source program

and groups the characters into meaningful sequences called *lexemes.* For each lexeme, the lexical analyzer produces as output a *token* of the form

(token-name, attribute-value)

that it passes on to the subsequent phase, syntax analysis. In the token, the first component *token-name* is an abstract symbol that is used during syntax analysis, and the second component *attribute-value* points to an entry in the symbol table for this token. Information from the symbol-table entry Is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

position = initial + rate * 60 (1.1)

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. position is a lexeme that would be mapped into a token **(id,** 1), where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol-table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. The assignment symbol = is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

3. initial is a lexeme that is mapped into the token **(id,** 2), where 2 points to the symbol-table entry for initial.

4. + is a lexeme that is mapped into the token (+).

5. rate is a lexeme that is mapped into the token **(id,** 3), where 3 points to the symbol-table entry for rate.

6. * is a lexeme that is mapped into the token (*).

7. 60 is a lexeme that is mapped into the token (60).[1]

Blanks separating the lexemes would be discarded by the lexical analyzer.

Figure 1.7 shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens

(id,l) <=) (id, 2) (+) **(id,** 3) (*) (60) (1.2)

In this representation, the token names =, +, and * are abstract symbols for the assignment, addition, and multiplication operators, respectively.

[1] Technically speaking, for the lexeme 60 we should make up a token like **(number,** 4), where 4 points to the symbol table for the internal representation of integer 60 but we shall defer the discussion of tokens for numbers until Chapter 2. Chapter 3 discusses techniques for building lexical analyzers.

```
position = initial + rate * 60
```

Lexical Analyzer

```
(id,l> {=) <id,2> <+) <id,3> (.> (60)
```

Syntax Analyzer

```
<id,l)        +
      (id,2)-
            (id,3)        60
```

Semantic Analyzer

position	
initial	
rate	

SYMBOL TABLE

```
(id,l)
      (id, 2)
            (id, 3}      inttofloat
                              |
                              60
```

Intermediate Code Generator

```
tl = inttofloat(60)
t2 = id3 * tl
t3 = id2 + t2
idl = t3
```

Code Optimizer

```
tl = id3 * 60.0
idl = id2 + tl
```

Code Generator

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  Rl, id2
ADDF Rl, Rl, R2
STF  idl, Rl
```

Figure 1.7: Translation of an assignment statement

1.2.2 Syntax Analysis

The second phase of the compiler is *syntax analysis* or *parsing.* The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a *syntax tree* in which each interior node represents an operation and the children of the node represent the arguments of the operation. A syntax tree for the token stream (1.2) is shown as the output of the syntactic analyzer in Fig. 1.7.

This tree shows the order in which the operations in the assignment

position = initial + **rate * 60**

are to be performed. The tree has an interior node labeled * with **(id, 3)** as its left child and the integer **60** as its right child. The node **(id, 3)** represents the identifier **rate.** The node labeled * makes it explicit that we must first multiply the value of **rate** by **60.** The node labeled + indicates that we must add the result of this multiplication to the value of initial. The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier position. This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program. In Chapter 4 we shall use context-free grammars to specify the grammatical structure of programming languages and discuss algorithms for constructing efficient syntax analyzers automatically from certain classes of grammars. In Chapters 2 and 5 we shall see that syntax-directed definitions can help specify the translation of programming language constructs.

1.2.3 Semantic Analysis

The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is *type checking,* where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

The language specification may permit some type conversions called *coercions.* For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Such a coercion appears in Fig. 1.7. Suppose that position, initial, and rate have been declared to be floating-point numbers, and that the lexeme 60 by itself forms an integer. The type checker in the semantic analyzer in Fig. 1.7 discovers that the operator * is applied to a floating-point number rate and an integer 60. In this case, the integer may be converted into a floating-point number. In Fig. 1.7, notice that the output of the semantic analyzer has an extra node for the operator **inttofloat,** which explicitly converts its integer argument into a floating-point number. Type checking and semantic analysis are discussed in Chapter 6.

1.2.4 Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

In Chapter 6, we consider an intermediate form called *three-address code,* which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register. The output of the intermediate code generator in Fig. 1.7 consists of the three-address code sequence

```
tl • inttofloat(60)
t2•• id3 * tl
ts •• id2 + t2
idl  t3
```
(1.3)

There are several points worth noting about three-address instructions. First, each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done; the multiplication precedes the addition in the source program (1.1). Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction. Third, some "three-address instructions" like the first and last in the sequence (1.3), above, have fewer than three operands.

In Chapter 6, we cover the principal intermediate representations used in compilers. Chapters 5 introduces techniques for syntax-directed translation that are applied in Chapter 6 to type checking and intermediate-code generation for typical programming language constructs such as expressions, flow-of-control constructs, and procedure calls.

1.2.5 Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. For example, a straightforward algorithm generates the intermediate code (1.3), using an instruction for each operator in the tree representation that comes from the semantic analyzer.

A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the **inttofloat** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, **t3** is used only once to transmit its value to idl so the optimizer can transform (1.3) into the shorter sequence

```
tl = id3 * 60.0
idl = id2 + tl
```
(1.4)

There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much. The chapters from 8 on discuss machine-independent and machine-dependent optimizations in detail.

1.2.6 Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers Or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

For example, using registers Rl and **R2,** the intermediate code in (1.4) might get translated into the machine code

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  Rl,  id2
ADDF Rl,  Rl, R2
STF  idl, Rl
```
(1.5)

The first operand of each instruction specifies a destination. The **F** in each instruction tells us that it deals with floating-point numbers. The code in

(1.5) loads the contents of address id3 into register R2, then multiplies it with floating-point constant 60.0. The # signifies that 60.0 is to be treated as an immediate constant. The third instruction moves id2 into register Rl and the fourth adds to it the value previously computed in register R2. Finally, the value in register Rl is stored into the address of idl, so the code correctly implements the assignment statement (1.1). Chapter 8 covers code generation.

This discussion of code generation has ignored the important issue of storage allocation for the identifiers in the source program. As we shall see in Chapter 7, the organization of storage at run-time depends on the language being compiled. Storage-allocation decisions are made either during intermediate code generation or during code generation.

1.2.7 Symbol-Table Management

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly. Symbol tables are discussed in Chapter 2.

1.2.8 The Grouping of Phases into Passes

The discussion of phases deals with the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a *pass* that reads an input file and writes an output file. For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.

Some compiler collections have been created around carefully designed intermediate representations that allow the front end for a particular language to interface with the back end for a certain target machine. With these collections, we can produce compilers for different source languages for one target machine by combining different front ends with the back end for that target machine. Similarly, we can produce compilers for different target machines, by combining a front end with back ends for different target machines.

1.2.9 Compiler-Construction Tools

The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on. In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.

These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms. The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler. Some commonly used compiler-construction tools include

1. *Parser generators* that automatically produce syntax analyzers from a grammatical description of a programming language.

2. *Scanner generators* that produce lexical analyzers from a regular-expression description of the tokens of a language.

3. *Syntax-directed translation engines* that produce collections of routines for walking a parse tree and generating intermediate code.

4. *Code-generator generators* that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

5. *Data-flow analysis engines* that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

6. *Compiler-construction toolkits* that provide an integrated set of routines for constructing various phases of a compiler.

We shall describe many of these tools throughout this book.

1.3 The Evolution of Programming Languages

The first electronic computers appeared in the 1940's and were programmed in machine language by sequences of O's and l's that explicitly told the computer what operations to execute and in what order. The operations themselves were very low level: move data from one location to another, add the contents of two registers, compare two values, and so on. Needless to say, this kind of programming was slow, tedious, and error prone. And once written, the programs were hard to understand and modify.

1.3.1 The Move to Higher-level Languages

The first step towards more people-friendly programming languages was the development of mnemonic assembly languages in the early 1950's. Initially, the instructions in an assembly language were just mnemonic representations of machine instructions. Later, macro instructions were added to assembly languages so that a programmer could define parameterized shorthands for frequently used sequences of machine instructions.

A major step towards higher-level languages was made in the latter half of the 1950's with the development of Fortran for scientific computation, Cobol for business data processing, and Lisp for symbolic computation. The philosophy behind these languages was to create higher-level notations with which programmers could more easily write numerical computations, business applications, and symbolic programs. These languages were so successful that they are still in use today.

In the following decades, many more languages were created with innovative features to help make programming easier, more natural, and more robust. Later in this chapter, we shall discuss some key features that are common to many modern programming languages.

Today, there are thousands of programming languages. They can be classified in a variety of ways. One classification is by generation. *First-generation languages* are the machine languages, *second-generation* the assembly languages, and *third-generation* the higher-level languages like Fortran, Cobol, Lisp, C, C++, C#, and Java. *Fourth-generation languages* are languages designed for specific applications like NOMAD for report generation, SQL for database queries, and Postscript for text formatting. The term *fifth-generation language* has been applied to logic- and constraint-based languages like Prolog and OPS5.

Another classification of languages uses the term *imperative* for languages in which a program specifies *how* a computation is to be done and *declarative* for languages in which a program specifies *what* computation is to be done. Languages such as C, C++, C#, and Java are imperative languages. In imperative languages there is a notion of program state and statements that change the state. Functional languages such as ML and Haskell and constraint logic languages such as Prolog are often considered to be declarative languages.

The term *von Neumann language* is applied to programming languages whose computational model is based on the von Neumann computer architecture. Many of today's languages, such as Fortran and C are von Neumann languages.

An *object-oriented language* is one that supports object-oriented programming, a programming style in which a program consists of a collection of objects that interact with one another. Simula 67 and Smalltalk are the earliest major object-oriented languages. Languages such as C++, C#, Java, and Ruby are more recent object-oriented languages.

Scripting languages are interpreted languages with high-level operators designed for "gluing together" computations. These computations were originally

called "scripts." Awk, JavaScript, Perl, PHP, Python, Ruby, and Tel are popular examples of scripting languages. Programs written in scripting languages are often much shorter than equivalent programs written in languages like C.

1.3.2 Impacts on Compilers

Since the design of programming languages and compilers are intimately related, the advances in programming languages placed new demands on compiler writers. They had to devise algorithms and representations to translate and support the new language features. Since the 1940's, computer architecture has evolved as well. Not only did the compiler writers have to track new language features, they also had to devise translation algorithms that would take maximal advantage of the new hardware capabilities.

Compilers can help promote the use of high-level languages by minimizing the execution overhead of the programs written in these languages. Compilers are also critical in making high-performance computer architectures effective on users' applications. In fact, the performance of a computer system is so dependent on compiler technology that compilers are used as a tool in evaluating architectural concepts before a computer is built.

Compiler writing is challenging. A compiler by itself is a large program. Moreover, many modern language-processing systems handle several source languages and target machines within the same framework; that is, they serve as collections of compilers, possibly consisting of millions of lines of code. Consequently, good software-engineering techniques are essential for creating and evolving modern language processors.

A compiler must translate correctly the potentially infinite set of programs that could be written in the source language. The problem of generating the optimal target code from a source program is undecidable in general; thus, compiler writers must evaluate tradeoffs about what problems to tackle and what heuristics to use to approach the problem of generating efficient code.

A study of compilers is also a study of how theory meets practice, as we shall see in Section 1.4.

The purpose of this text is to teach the methodology and fundamental ideas used in compiler design. It is not the intention of this text to teach all the algorithms and techniques that could be used for building a state-of-the-art language-processing system. However, readers of this text will acquire the basic knowledge and understanding to learn how to build a compiler relatively easily.

1.3.3 Exercises for Section 1.3

Exercise 1.3.1: Indicate which of the following terms:

a) imperative	b) declarative	c) von Neumann
d) object-oriented	e) functional	f) third-generation
g) fourth-generation	h) scripting	

apply to which of the following languages:

1) C	2) C++	3) Cobol	4) Fortran	5) Java
6) Lisp	7) ML	8) Perl	9) Python	10) VB.

1.4 The Science of Building a Compiler

Compiler design is full of beautiful examples where complicated real-world problems are solved by abstracting the essence of the problem mathematically. These serve as excellent illustrations of how abstractions can be used to solve problems: take a problem, formulate a mathematical abstraction that captures the key characteristics, and solve it using mathematical techniques. The problem formulation must be grounded in a solid understanding of the characteristics of computer programs, and the solution must be validated and refined empirically.

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

1.4.1 Modeling in Compiler Design and Implementation

The study of compilers is mainly a study of how we design the right mathematical models and choose the right algorithms, while balancing the need for generality and power against simplicity and efficiency.

Some of most fundamental models are finite-state machines and regular expressions, which we shall meet in Chapter 3. These models are useful for describing the lexical units of programs (keywords, identifiers, and such) and for describing the algorithms used by the compiler to recognize those units. Also among the most fundamental models are context-free grammars, used to describe the syntactic structure of programming languages such as the nesting of parentheses or control constructs. We shall study grammars in Chapter 4. Similarly, trees are an important model for representing the structure of programs and their translation into object code, as we shall see in Chapter 5.

1.4.2 The Science of Code Optimization

The term "optimization" in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code. "Optimization" is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.

In modern times, the optimization of code that a compiler performs has become both more important and more complex. It is more complex because processor architectures have become more complex, yielding more opportunities to improve the way code executes. It is more important because massively parallel computers require substantial optimization, or their performance suffers by orders of magnitude. With the likely prevalence of multicore machines (computers with chips that have large numbers of processors on them), all compilers will have to face the problem of taking advantage of multiprocessor machines.

It is hard, if not impossible, to build a robust compiler out of "hacks." Thus, an extensive and useful theory has been built up around the problem of optimizing code. The use of a rigorous mathematical foundation allows us to show that an optimization is correct and that it produces the desirable effect for all possible inputs. We shall see, starting in Chapter **9,** how models such as graphs, matrices, and linear programs are necessary if the compiler is to produce well optimized code.

On the other hand, pure theory alone is insufficient. Like many real-world problems, there are no perfect answers. In fact, most of the questions that we ask in compiler optimization are undecidable. One of the most important skills in compiler design is the ability to formulate the right problem to solve. We need a good understanding of the behavior of programs to start with and thorough experimentation and evaluation to validate our intuitions.

Compiler optimizations must meet the following design objectives:

- The optimization must be correct, that is, preserve the meaning of the compiled program,
- The optimization must improve the performance of many programs,
- The compilation time must be kept reasonable, and
- The engineering effort required must be manageable.

It is impossible to overemphasize the importance of correctness. It is trivial to write a compiler that generates fast code if the generated code need not be correct! Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct.

The second goal is that the compiler must be effective in improving the performance of many input programs. Normally, performance means the speed of the program execution. Especially in embedded applications, we may also wish to minimize the size of the generated code. And in the case of mobile devices, it is also desirable that the code minimizes power consumption. Typically, the same optimizations that speed up execution time also conserve power. Besides performance, usability aspects such as error reporting and debugging are also important.

Third, we need to keep the compilation time short to support a rapid development and debugging cycle. This requirement has become easier to meet as

machines get faster. Often, a program is first developed and debugged without program optimizations. Not only is the compilation time reduced, but more importantly, unoptimized programs are easier to debug, because the optimizations introduced by a compiler often obscure the relationship between the source code and the object code. Turning on optimizations in the compiler sometimes exposes new problems in the source program; thus testing must again be performed on the optimized code. The need for additional testing sometimes deters the use of optimizations in applications, especially if their performance is not critical.

Finally, a compiler is a complex system; we must keep the system simple to assure that the engineering and maintenance costs of the compiler are manageable. There is an infinite number of program optimizations that we could implement, and it takes a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations, implementing only those that lead to the greatest benefits on source programs encountered in practice.

Thus, in studying compilers, we learn not only how to build a compiler, but also the general methodology of solving complex and open-ended problems. The approach used in compiler development involves both theory and experimentation. We normally start by formulating the problem based on our intuitions on what the important issues are.

1.5 Applications of Compiler Technology

Compiler design is not only about compilers, and many people use the technology learned by studying compilers in school, yet have never, strictly speaking, written (even part of) a compiler for a major programming language. Compiler technology has other important uses as well. Additionally, compiler design impacts several other areas of computer science. In this section, we review the most important interactions and applications of the technology.

1.5.1 Implementation of High-Level Programming Languages

A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language. Generally, higher-level programming languages are easier to program in, but are less efficient, that is, the target programs run more slowly. Programmers using a low-level language have more control over a computation and can, in principle, produce more efficient code. Unfortunately, lower-level programs are harder to write and — worse still — less portable, more prone to errors, and harder to maintain. Optimizing compilers include techniques to improve the performance of generated code, thus offsetting the inefficiency introduced by high-level abstractions.

Example 1.2: The **register** keyword in the C programming language is an early example of the interaction between compiler technology and language evolution. When the C language was created in the mid 1970s, it was considered necessary to let a programmer control which program variables reside in registers. This control became unnecessary as effective register-allocation techniques were developed, and most modern programs no longer use this language feature.

In fact, programs that use the **register** keyword may lose efficiency, because programmers often are not the best judge of very low-level matters like register allocation. The optimal choice of register allocation depends greatly on the specifics of a machine architecture. Hardwiring low-level resource-management decisions like register allocation may in fact hurt performance, especially if the program is run on machines other than the one for which it was written. •

The many shifts in the popular choice of programming languages have been in the direction of increased levels of abstraction. C was the predominant systems programming language of the 80's; many of the new projects started in the 90's chose C++; Java, introduced in 1995, gained popularity quickly in the late 90's. The new programming-language features introduced in each round spurred new research in compiler optimization. In the following, we give an overview on the main language features that have stimulated significant advances in compiler technology.

Practically all common programming languages, including C, Fortran and Cobol, support user-defined aggregate data types, such as arrays and structures, and high-level control flow, such as loops and procedure invocations. If we just take each high-level construct or data-access operation and translate it directly to machine code, the result would be very inefficient. A body of compiler optimizations, known as *data-flow optimizations,* has been developed to analyze the flow of data through the program and removes redundancies across these constructs. They are effective in generating code that resembles code written by a skilled programmer at a lower level.

Object orientation was first introduced in Simula in 1967, and has been incorporated in languages such as Smalltalk, C++, C#, and Java. The key ideas behind object orientation are

1. Data abstraction and
2. Inheritance of properties,

both of which have been found to make programs more modular and easier to maintain. Object-oriented programs are different from those written in many other languages, in that they consist of many more, but smaller, procedures (called *methods* in object-oriented terms). Thus, compiler optimizations must be able to perform well across the procedural boundaries of the source program. Procedure inlining, which is the replacement of a procedure call by the body of the procedure, is particularly useful here. Optimizations to speed up virtual method dispatches have also been developed.

Java has many features that make programming easier, many of which have been introduced previously in other languages. The Java language is type-safe; that is, an object cannot be used as an object of an unrelated type. All array accesses are checked to ensure that they lie within the bounds of the array. Java has no pointers and does not allow pointer arithmetic. It has a built-in garbage-collection facility that automatically frees the memory of variables that are no longer in use. While all these features make programming easier, they incur a run-time overhead. Compiler optimizations have been developed to reduce the overhead, for example, by eliminating unnecessary range checks and by allocating objects that are not accessible beyond a procedure on the stack instead of the heap. Effective algorithms also have been developed to minimize the overhead of garbage collection.

In addition, Java is designed to support portable and mobile code. Programs are distributed as Java bytecode, which must either be interpreted or compiled into native code dynamically, that is, at run time. Dynamic compilation has also been studied in other contexts, where information is extracted dynamically at run time and used to produce better-optimized code. In dynamic optimization, it is important to minimize the compilation time as it is part of the execution overhead. A common technique used is to only compile and optimize those parts of the program that will be frequently executed.

1.5.2 Optimizations for Computer Architectures

The rapid evolution of computer architectures has also led to an insatiable demand for new compiler technology. Almost all high-performance systems take advantage of the same two basic techniques: *parallelism* and *memory hierarchies*. Parallelism can be found at several levels: at the *instruction level*, where multiple operations are executed simultaneously and at the *processor level*, where different threads of the same application are run on different processors. Memory hierarchies are a response to the basic limitation that we can build very fast storage or very large storage, but not storage that is both fast and large.

Parallelism

All modern microprocessors exploit instruction-level parallelism. However, this parallelism can be hidden from the programmer. Programs are written as if all instructions were executed in sequence; the hardware dynamically checks for dependencies in the sequential instruction stream and issues them in parallel when possible. In some cases, the machine includes a hardware scheduler that can change the instruction ordering to increase the parallelism in the program. Whether the hardware reorders the instructions or not, compilers can rearrange the instructions to make instruction-level parallelism more effective.

Instruction-level parallelism can also appear explicitly in the instruction set. VLIW (Very Long Instruction Word) machines have instructions that can issue

multiple operations in parallel. The Intel IA64 is a well-known example of such an architecture. All high-performance, general-purpose microprocessors also include instructions that can operate on a vector of data at the same time. Compiler techniques have been developed to generate code automatically for such machines from sequential programs.

Multiprocessors have also become prevalent; even personal computers often have multiple processors. Programmers can write multithreaded code for multiprocessors, or parallel code can be automatically generated by a compiler from conventional sequential programs. Such a compiler hides from the programmers the details of finding parallelism in a program, distributing the computation across the machine, and minimizing synchronization and communication among the processors. Many scientific-computing and engineering applications are computation-intensive and can benefit greatly from parallel processing. Parallelization techniques have been developed to translate automatically sequential scientific programs into multiprocessor code.

Memory Hierarchies

A memory hierarchy consists of several levels of storage with different speeds and sizes, with the level closest to the processor being the fastest but smallest. The average memory-access time of a program is reduced if most of its accesses are satisfied by the faster levels of the hierarchy. Both parallelism and the existence of a memory hierarchy improve the potential performance of a machine, but they must be harnessed effectively by the compiler to deliver real performance on an application.

Memory hierarchies are found in all machines. A processor usually has a small number of registers consisting of hundreds of bytes, several levels of caches containing kilobytes to megabytes, physical memory containing megabytes to gigabytes, and finally secondary storage that contains gigabytes and beyond. Correspondingly, the speed of accesses between adjacent levels of the hierarchy can differ by two or three orders of magnitude. The performance of a system is often limited not by the speed of the processor but by the performance of the memory subsystem. While compilers traditionally focus on optimizing the processor execution, more emphasis is now placed on making the memory hierarchy more effective.

Using registers effectively is probably the single most important problem in optimizing a program. Unlike registers that have to be managed explicitly in software, caches and physical memories are hidden from the instruction set and are managed by hardware. It has been found that cache-management policies implemented by hardware are not effective in some cases, especially in scientific code that has large data structures (arrays, typically). It is possible to improve the effectiveness of the memory hierarchy by changing the layout of the data, or changing the order of instructions accessing the data. We can also change the layout of code to improve the effectiveness of instruction caches.

1.5.3 Design of New Computer Architectures

In the early days of computer architecture design, compilers were developed after the machines were built. That has changed. Since programming in high-level languages is the norm, the performance of a computer system is determined not by its raw speed but also by how well compilers can exploit its features. Thus, in modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectural features.

RISC

One of the best known examples of how compilers influenced the design of computer architecture was the invention of the RISC (Reduced Instruction-Set Computer) architecture. Prior to this invention, the trend was to develop progressively complex instruction sets intended to make assembly programming easier; these architectures were known as CISC (Complex Instruction-Set Computer). For example, CISC instruction sets include complex memory-addressing modes to support data-structure accesses and procedure-invocation instructions that save registers and pass parameters on the stack.

Compiler optimizations often can reduce these instructions to a small number of simpler operations by eliminating the redundancies across complex instructions. Thus, it is desirable to build simple instruction sets; compilers can use them effectively and the hardware is much easier to optimize.

Most general-purpose processor architectures, including PowerPC, SPARC, MIPS, Alpha, and PA-RISC, are based on the RISC concept. Although the x86 architecture—the most popular microprocessor—has a CISC instruction set, many of the ideas developed for RISC machines are used in the implementation of the processor itself. Moreover, the most effective way to use a high-performance x86 machine is to use just its simple instructions.

Specialized Architectures

Over the last three decades, many architectural concepts have been proposed. They include data flow machines, vector machines, VLIW (Very Long Instruction Word) machines, SIMD (Single Instruction, Multiple Data) arrays of processors, systolic arrays, multiprocessors with shared memory, and multiprocessors with distributed memory. The development of each of these architectural concepts was accompanied by the research and development of corresponding compiler technology.

Some of these ideas have made their way into the designs of embedded machines. Since entire systems can fit on a single chip, processors need no longer be prepackaged commodity units, but can be tailored to achieve better cost-effectiveness for a particular application. Thus, in contrast to general-purpose processors, where economies of scale have led computer architectures

to converge, application-specific processors exhibit a diversity of computer architectures. Compiler technology is needed not only to support programming for these architectures, but also to evaluate proposed architectural designs.

1.5.4 Program Translations

While we normally think of compiling as a translation from a high-level language to the machine level, the same technology can be applied to translate between different kinds of languages. The following are some of the important applications of program-translation techniques.

Binary Translation

Compiler technology can be used to translate the binary code for one machine to that of another, allowing a machine to run programs originally compiled for another instruction set. Binary translation technology has been used by various computer companies to increase the availability of software for their machines. In particular, because of the domination of the x86 personal-computer market, most software titles are available as x86 code. Binary translators have been developed to convert x86 code into both Alpha and Sparc code. Binary translation was also used by Transmeta Inc. in their implementation of the x86 instruction set. Instead of executing the complex x86 instruction set directly in hardware, the Transmeta Crusoe processor is a VLIW processor that relies on binary translation to convert x86 code into native VLIW code.

Binary translation can also be used to provide backward compatibility. When the processor in the Apple Macintosh was changed from the Motorola MC 68040 to the PowerPC in 1994, binary translation was used to allow PowerPC processors run legacy MC 68040 code.

Hardware Synthesis

Not only is most software written in high-level languages; even hardware designs are mostly described in high-level hardware description languages like Verilog and VHDL (Very high-speed integrated circuit Hardware Description Language). Hardware designs are typically described at the register transfer level (RTL), where variables represent registers and expressions represent combinational logic. Hardware-synthesis tools translate RTL descriptions automatically into gates, which are then mapped to transistors and eventually to a physical layout. Unlike compilers for programming languages, these tools often take hours optimizing the circuit. Techniques to translate designs at higher levels, such as the behavior or functional level, also exist.

Database Query Interpreters

Besides specifying software and hardware, languages are useful in many other applications. For example, query languages, especially SQL (Structured Query

Language), are used to search databases. Database queries consist of predicates containing relational and boolean operators. They can be interpreted or compiled into commands to search a database for records satisfying that predicate.

Compiled Simulation

Simulation is a general technique used in many scientific and engineering disciplines to understand a phenomenon or to validate a design. Inputs to a simulator usually include the description of the design and specific input parameters for that particular simulation run. Simulations can be very expensive. We typically need to simulate many possible design alternatives on many different input sets, and each experiment may take days to complete on a high-performance machine. Instead of writing a simulator that interprets the design, it is faster to compile the design to produce machine code that simulates that particular design natively. Compiled simulation can run orders of magnitude faster than an interpreter-based approach. Compiled simulation is used in many state-of-the-art tools that simulate designs written in Verilog or VHDL.

1.5.5 Software Productivity Tools

Programs are arguably the most complicated engineering artifacts ever produced; they consist of many many details, every one of which must be correct before the program will work completely. As a result, errors are rampant in programs; errors may crash a system, produce wrong results, render a system vulnerable to security attacks, or even lead to catastrophic failures in critical systems. Testing is the primary technique for locating errors in programs.

An interesting and promising complementary approach is to use data-flow analysis to locate errors statically (that is, before the program is run). Data-flow analysis can find errors along all the possible execution paths, and not just those exercised by the input data sets, as in the case of program testing. Many of the data-flow-analysis techniques, originally developed for compiler optimizations, can be used to create tools that assist programmers in their software engineering tasks.

The problem of finding all program errors is undecidable. A data-flow analysis may be designed to warn the programmers of all possible statements violating a particular category of errors. But if most of these warnings are false alarms, users will not use the tool. Thus, practical error detectors are often neither sound nor complete. That is, they may not find all the errors in the program, and not all errors reported are guaranteed to be real errors. Nonetheless, various static analyses have been developed and shown to be effective in finding errors, such as dereferencing null or freed pointers, in real programs. The fact that error detectors may be unsound makes them significantly different from compiler optimizations. Optimizers must be conservative and cannot alter the semantics of the program under any circumstances.

In the balance of this section, we shall mention several ways in which program analysis, building upon techniques originally developed to optimize code in compilers, have improved software productivity. Of special importance are techniques that detect statically when a program might have a security vulnerability.

Type Checking

Type checking is an effective and well-established technique to catch inconsistencies in programs. It can be used to catch errors, for example, where an operation is applied to the wrong type of object, or if parameters passed to a procedure do not match the signature of the procedure. Program analysis can go beyond finding type errors by analyzing the flow of data through a program. For example, if a pointer is assigned null and then immediately dereferenced, the program is clearly in error.

The same technology can be used to catch a variety of security holes, in which an attacker supplies a string or other data that is used carelessly by the program. A user-supplied string can be labeled with a type "dangerous." If this string is not checked for proper format, then it remains "dangerous," and if a string of this type is able to influence the control-flow of the code at some point in the program, then there is a potential security flaw.

Bounds Checking

It is easier to make mistakes when programming in a lower-level language than a higher-level one. For example, many security breaches in systems are caused by buffer overflows in programs written in C. Because C does not have array-bounds checks, it is up to the user to ensure that the arrays are not accessed out of bounds. Failing to check that the data supplied by the user can overflow a buffer, the program may be tricked into storing user data outside of the buffer. An attacker can manipulate the input data that causes the program to misbehave and compromise the security of the system. Techniques have been developed to find buffer overflows in programs, but with limited success.

Had the program been written in a safe language that includes automatic range checking, this problem would not have occurred. The same data-flow analysis that is used to eliminate redundant range checks can also be used to locate buffer overflows. The major difference, however, is that failing to eliminate a range check would only result in a small run-time cost, while failing to identify a potential buffer overflow may compromise the security of the system. Thus, while it is adequate to use simple techniques to optimize range checks, sophisticated analyses, such as tracking the values of pointers across procedures, are needed to get high-quality results in error detection tools.

Memory-Management Tools

Garbage collection is another excellent example of the tradeoff between efficiency and a combination of ease of programming and software reliability. Automatic memory management obliterates all memory-management errors (e.g., "memory leaks"), which are a major source of problems in C and C++ programs. Various tools have been developed to help programmers find memory management errors. For example, Purify is a widely used tool that dynamically catches memory management errors as they occur. Tools that help identify some of these problems statically have also been developed.

1.6 Programming Language Basics

In this section, we shall cover the most important terminology and distinctions that appear in the study of programming languages. It is not our purpose to cover all concepts or all the popular programming languages. We assume that the reader is familiar with at least one of C, C++, C#, or Java, and may have encountered other languages as well.

1.6.1 The Static/Dynamic Distinction

Among the most important issues that we face when designing a compiler for a language is what decisions can the compiler make about a program. If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a *static* policy or that the issue can be decided at *compile time*. On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a *dynamic policy* or to require a decision at *run time*.

One issue on which we shall concentrate is the scope of declarations. The *scope* of a declaration of x is the region of the program in which uses of x refer to this declaration. A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses *dynamic scope*. With dynamic scope, as the program runs, the same use of x could refer to any of several different declarations of x.

Most languages, such as C and Java, use static scope. We shall discuss static scoping in Section 1.6.3.

Example 1.3 : As another example of the static/dynamic distinction, consider the use of the term "static" as it applies to data in a Java class declaration. In Java, a variable is a name for a location in memory used to hold a data value. Here, "static" refers not to the scope of the variable, but rather to the ability of the compiler to determine the location in memory where the declared variable can be found. A declaration like

```
public static int x;
```

makes *x* a *class variable* and says that there is only one copy of *x, no* matter how many objects of this class are created. Moreover, the compiler can determine a location in memory where this integer *x* will be held. In contrast, had "static" been omitted from this declaration, then each object of the class would have its own location where *x* would be held, and the compiler could not determine all these places in advance of running the program. •

1.6.2 Environments and States

Another important distinction we must make when discussing programming languages is whether changes occurring as the program runs affect the values of data elements or affect the interpretation of names for that data. For example, the execution of an assignment such as **x = y +1** changes the value denoted by the name *x*. More specifically, the assignment changes the value in whatever location is denoted by *x*.

It may be less clear that the location denoted by *x* can change at run time. For instance, as we discussed in Example 1.3, if *x* is not a static (or "class") variable, then every object of the class has its own location for an instance of variable *x*. In that case, the assignment to *x* can change any of those "instance" variables, depending on the object to which a method containing that assignment is applied.

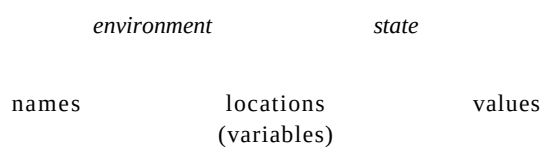


Figure 1.8: Two-stage mapping from names to values

The association of names with locations in memory (the *store)* and then with values can be described by two mappings that change as the program runs (see Fig. 1.8):

1. The *environment* is a mapping from names to locations in the store. Since variables refer to locations ("1-values" in the terminology of C), we could alternatively define an environment as a mapping from names to variables.

2. The *state* is a mapping from locations in store to their values. That is, the state maps 1-values to their corresponding r-values, in the terminology of C.

Environments change according to the scope rules of a language.

Example 1.4: Consider the C program fragment in Fig. 1**.9.** Integer *i* is declared a global variable, and also declared as a variable local to function /. When / is executing, the environment adjusts so that name *i* refers to the

```
int i;                  /* global i        */

void f(---) {
      int i;            /* local i         */

      i = 3;            /* use of local i  */

}

      x = i + 1;        /* use of global i */
```

Figure **1.9:** Two declarations of the name *i*

location reserved for the *i* that is local to /, and any use of *i*, such as the assignment **i = 3** shown explicitly, refers to that location. Typically, the local *i* is given a place on the run-time stack.

Whenever a function *g* other than / is executing, uses of *i* cannot refer to the *i* that is local to /. Uses of name *i* in *g* must be within the scope of some other declaration of *i*. An example is the explicitly shown statement x = **i+1,** which is inside some procedure whose definition is not shown. The *i* in *i* **+ 1** presumably refers to the global *i*. As in most languages, declarations in C must precede their use, so a function that comes before the global *i* cannot refer to it. •

The environment and state mappings in Fig. **1**.8 are dynamic, but there are a few exceptions:

1. *Static versus dynamic binding* of names to locations. Most binding of names to locations is dynamic, and we discuss several approaches to this binding throughout the section. Some declarations, such as the global *i* in Fig. **1.9,** can be given a location in the store once and for all, as the compiler generates object code.[2]

2. *Static versus dynamic binding* of locations to values. The binding of locations to values (the second stage in Fig. **1**.8), is generally dynamic as well, since we cannot tell the value in a location until we run the program. Declared constants are an exception. For instance, the C definition

```
#define ARRAYSIZE 1000
```

[2] Technically, the C compiler will assign a location in virtual memory for the global ***i,*** leaving it to the loader and the operating system to determine where in the physical memory of the machine ***i*** will be located. However, we shall not worry about "relocation" issues such as these, which have no impact on compiling. Instead, we treat the address space that the compiler uses for its output code as if it gave physical memory locations.

Names, Identifiers, and Variables

Although the terms "name" and "variable," often refer to the same thing, we use them carefully to distinguish between compile-time names and the run-time locations denoted by names.

An *identifier* is a string of characters, typically letters or digits, that refers to (identifies) an entity, such as a data object, a procedure, a class, or a type. All identifiers are names, but not all names are identifiers. Names can also be expressions. For example, the name *x.y* might denote the field *y* of a structure denoted by *x*. Here, *x* and *y* are identifiers, while *x.y* is a name, but not an identifier. Composite names like *x.y* are called *qualified* names.

A *variable* refers to a particular location of the store. It is common for the same identifier to be declared more than once; each such declaration introduces a new variable. Even if each identifier is declared just once, an identifier local to a recursive procedure will refer to different locations of the store at different times.

binds the name **ARRAYS** IZE to the value 1000 statically. We can determine this binding by looking at the statement, and we know that it is impossible for this binding to change when the program executes.

1.6.3 Static Scope and Block Structure

Most languages, including C and its family, use static scope. The scope rules for C are based on program structure; the scope of a declaration is determined implicitly by where the declaration appears in the program. Later languages, such as C++, Java, and C#, also provide explicit control over scopes through the use of keywords like **public, private,** and **protected.**

In this section we consider static-scope rules for a language with blocks, where a *block* is a grouping of declarations and statements. C uses braces { and } to delimit a block; the alternative use of **begin** and **end** for the same purpose dates back to Algol.

Example 1.5 : To a first approximation, the C static-scope policy is as follows:

1. A C program consists of a sequence of top-level declarations of variables and functions.

2. Functions may have variable declarations within them, where variables include local variables and parameters. The scope of each such declaration is restricted to the function in which it appears.

Procedures, Functions, and Methods

To avoid saying "procedures, functions, or methods," each time we want to talk about a subprogram that may be called, we shall usually refer to all of them as "procedures." The exception is that when talking explicitly of programs in languages like C that have only functions, we shall refer to them as "functions." Or, if we are discussing a language like Java that has only methods, we shall use that term instead.

A function generally returns a value of some type (the "return type"), while a procedure does not return any value. C and similar languages, which have only functions, treat procedures as functions that have a special return type "void," to signify no return value. Object-oriented languages like Java and C++ use the term "methods." These can behave like either functions or procedures, but are associated with a particular class.

3. The scope of a top-level declaration of a name *x* consists of the entire program that follows, with the exception of those statements that lie within a function that also has a declaration of *x*.

The additional detail regarding the C static-scope policy deals with variable declarations within statements. We examine such declarations next and in Example 1.6. •

In C, the syntax of blocks is given by

1. One type of statement is a block. Blocks can appear anywhere that other types of statements, such as assignment statements, can appear.

2. A block is a sequence of declarations followed by a sequence of statements, all surrounded by braces.

Note that this syntax allows blocks to be nested inside each other. This nesting property is referred to as *block structure.* The C family of languages has block structure, except that a function may not be defined inside another function.

We say that a declaration *D* "belongs" to a block *B* if *B* is the most closely nested block containing *D;* that is, *D* is located within *B,* but not within any block that is nested within *B.*

The static-scope rule for variable declarations in a block-structured languages is as follows. If declaration *D* of name *x* belongs to block *B,* then the scope of *D* is all of *B,* except for any blocks *B'* nested to any depth within J5, in which *x* is redeclared. Here, *x* is redeclared in *B'* if some other declaration *D'* of the same name *x* belongs to *B'.*

An equivalent way to express this rule is to focus on a use of a name *x*. Let *Bi,* i?2, • • • , *Bk* be all the blocks that surround this use of *x,* with *Bk* the smallest, nested within *Bk-i,* which is nested within *Bk-2,* and so on. Search for the largest *i* such that there is a declaration of *x* belonging to *B*^. This use of *x* refers to the declaration in *B{*. Alternatively, this use of *x* is within the scope of the declaration in *Bi.*

```
mainQ {
    int a = 1;
    int b = 1;
    {
        int b = 2;
        int a = 3;
        cout « a « b;
    }
    {
        int b - 4;
        cout << a « b;
    }
    cout << a « b;
    cout << a << b;
}
```

Figure 1.10: Blocks in a C++ program

Example 1.6: The C++ program in Fig. 1.10 has four blocks, with several definitions of variables *a* and *b.* As a memory aid, each declaration initializes its variable to the number of the block to which it belongs.

For instance, consider the declaration int a = 1 in block *Bi.* Its scope is all of *Bi,* except for those blocks nested (perhaps deeply) within *B*± that have their own declaration of a. Z?2, nested immediately within *B*±, does not have a declaration of *a,* but *B3* does. *B4* does not have a declaration of a, so block *B3* is the only place in the entire program that is outside the scope of the declaration of the name *a* that belongs to *B*\. That is, this scope includes *B4* and all of Z?, except for the part of *B2* that is within *B3.* The scopes of all five declarations are summarized in Fig. 1.11.

From another point of view, let us consider the output statement in block *B4* and bind the variables *a* and *b* used there to the proper declarations. The list of surrounding blocks, in order of increasing size, is *B,,B2,Bi.* Note that #3 does not surround the point in question. *B4* has a declaration of *b,* so it is to this declaration that this use of *b* refers, and the value of 6 printed is 4. However, *B4* does not have a declaration of o, so we next look at £?2- That block does not have a declaration of *a* either, so we proceed to *B*\. Fortunately,

DECLARATION		SCOPE
int **a** = 1		B_1 - B_3
int b = 1		B_2 - B_2
int b = 2		B_3 — B_3
int **a** = 3		B_3
int b = 4		B_3

Figure 1.11: Scopes of declarations in Example 1.6

there is a declaration int **a** = 1 belonging to that block, so the value of *a* printed is 1. Had there been no such declaration, the program would have been erroneous. •

1.6.4 Explicit Access Control

Classes and structures introduce a new scope for their members. If *p* is an object of a class with a field (member) *x*, then the use of *x* in *p.x* refers to field *x* in the class definition. In analogy with block structure, the scope of a member declaration *x* in a class *C* extends to any subclass *C*, except if *C* has a local declaration of the same name *x*.

Through the use of keywords like **public, private,** and **protected,** object-oriented languages such as C++ or Java provide explicit control over access to member names in a superclass. These keywords support *encapsulation* by restricting access. Thus, private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C++ term). Protected names are accessible to subclasses. Public names are accessible from outside the class.

In C++, a class definition may be separated from the definitions of some or all of its methods. Therefore, a name *x* associated with the class *C* may have a region of the code that is outside its scope, followed by another region (a method definition) that is within its scope. In fact, regions inside and outside the scope may alternate, until all the methods have been defined.

1.6.5 Dynamic Scope

Technically, any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term *dynamic scope,* however, usually refers to the following policy: a use of a name *x* refers to the declaration of *x* in the most recently called procedure with such a declaration. Dynamic scoping of this type appears only in special situations. We shall consider two examples of dynamic policies: macro expansion in the C preprocessor and method resolution in object-oriented programming.

Declarations and Definitions

The apparently similar terms "declaration" and "definition" for programming-language concepts are actually quite different. Declarations tell us about the types of things, while definitions tell us about their values. Thus, `int i` is a declaration of *i*, while `i = 1` is a definition of *i*.

The difference is more significant when we deal with methods or other procedures. In C++, a method is declared in a class definition, by giving the types of the arguments and result of the method (often called the *signature* for the method. The method is then defined, i.e., the code for executing the method is given, in another place. Similarly, it is common to define a C function in one file and declare it in other files where the function is used.

Example 1.7: In the C program of Fig. 1.12, identifier a is a macro that stands for expression (a? + 1). But what is *xl* We cannot resolve *x* statically, that is, in terms of the program text.

```
#define a (x+1)
int x = 2;
void b() { int x = 1; printf("°/.d\n", a); }
void c() { printf("·/.d\n", a); }
void mainO { b(); c(); }
```

Figure 1.12: A macro whose names must be scoped dynamically

In fact, in order to interpret *x*, we must use the usual dynamic-scope rule. We examine all the function calls that are currently active, and we take the most recently called function that has a declaration of It is to this declaration that the use of *x* refers.

In the example of Fig. 1.12, the function *main* first calls function 6. As *b* executes, it prints the value of the macro *a*. Since *(x* + 1) must be substituted for a, we resolve this use of *x* to the declaration `int x=l` in function *b*. The reason is that *b* has a declaration of *x*, so the (x + 1) in the `printf` in *b* refers to this *x*. Thus, the value printed is 1.

After *b* finishes, and c is called, we again need to print the value of macro *a*. However, the only *x* accessible to c is the global *x*. The `printf` statement in c thus refers to this declaration of *x*, and value 2 is printed. •

Dynamic scope resolution is also essential for polymorphic procedures, those that have two or more definitions for the same name, depending only on the

Analogy Between Static and Dynamic Scoping

While there could be any number of static or dynamic policies for scoping, there is an interesting relationship between the normal (block-structured) static scoping rule and the normal dynamic policy. In a sense, the dynamic rule is to time as the static rule is to space. While the static rule asks us to find the declaration whose unit (block) most closely surrounds the physical location of the use, the dynamic rule asks us to find the declaration whose unit (procedure invocation) most closely surrounds the time of the use.

types of the arguments. In some languages, such as ML (see Section 7.3.3), it is possible to determine statically types for all uses of names, in which case the compiler can replace each use of a procedure name p by a reference to the code for the proper procedure. However, in other languages, such as Java and C++, there are times when the compiler cannot make that determination.

Example 1.8: A distinguishing feature of object-oriented programming is the ability of each object to invoke the appropriate method in response to a message. In other words, the procedure called when *x.mQ* is executed depends on the class of the object denoted by *x* at that time. A typical example is as follows:

1. There is a class *C* with a method named m().
2. *D* is a subclass of *C*, and *D* has its own method named ra().
3. There is a use of *m* of the form *x.mQ*, where *x* is an object of class *C*.

Normally, it is impossible to tell at compile time whether *x* will be of class *C* or of the subclass *D*. If the method application occurs several times, it is highly likely that some will be on objects denoted by *x* that are in class *C* but not *D*, while others will be in class *D*. It is not until run-time that it can be decided which definition of *m* is the right one. Thus, the code generated by the compiler must determine the class of the object *x*, and call one or the other method named m. •

1.6.6 Parameter Passing Mechanisms

All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments. In this section, we shall consider how the *actual parameters* (the parameters used in the call of a procedure) are associated with the *formal parameters* (those used in the procedure definition). Which mechanism is used determines how the calling-sequence code treats parameters. The great majority of languages use either "call-by-value," or "call-by-reference," or both. We shall explain these terms, and another method known as "call-by-name," that is primarily of historical interest.

Call-by-Value

In *call-by-value,* the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure. This method is used in C and Java, and is a common option in C++, as well as in most other languages. Call-by-value has the effect that all computation involving the formal parameters done by the called procedure is local to that procedure, and the actual parameters themselves cannot be changed.

Note, however, that in C we can pass a pointer to a variable to allow that variable to be changed by the callee. Likewise, array names passed as parameters in C, C++, or Java give the called procedure what is in effect a pointer or reference to the array itself. Thus, if *a* is the name of an array of the calling procedure, and it is passed by value to corresponding formal parameter *x,* then an assignment such as x[i] = 2 really changes the array element a[2]. The reason is that, although *x* gets a copy of the value of *a,* that value is really a pointer to the beginning of the area of the store where the array named *a* is located.

Similarly, in Java, many variables are really references, or pointers, to the things they stand for. This observation applies to arrays, strings, and objects of all classes. Even though Java uses call-by-value exclusively, whenever we pass the name of an object to a called procedure, the value received by that procedure is in effect a pointer to the object. Thus, the called procedure is able to affect the value of the object itself.

Call-by-Reference

In *call-by-reference,* the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller. Changes to the formal parameter thus appear as changes to the actual parameter.

If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own. Changes to the formal parameter change this location, but can have no effect on the data of the caller.

Call-by-reference is used for "ref" parameters in C++ and is an option in many other languages. It is almost essential when the formal parameter is a large object, array, or structure. The reason is that strict call-by-value requires that the caller copy the entire actual parameter into the space belonging to the corresponding formal parameter. This copying gets expensive when the parameter is large. As we noted when discussing call-by-value, languages such as Java solve the problem of passing arrays, strings, or other objects by copying only a reference to those objects. The effect is that Java behaves as if it used call-by-reference for anything other than a basic type such as an integer or real.

Call-by-Name

A third mechanism — call-by-name — was used in the early programming language Algol **60.** It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct). When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.

1.6.7 Aliasing

There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value. It is possible that two formal parameters can refer to the same location; such variables are said to be *aliases* of one another. As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other, as well.

Example 1.9: Suppose *a* is an array belonging to a procedure *p,* and *p* calls another procedure *q(x,y)* with a call *q(a,a).* Suppose also that parameters are passed by value, but that array names are really references to the location where the array is stored, as in C or similar languages. Now, *x* and *y* have become aliases of each other. The important point is that if within *q* there is an assignment **x[10] = 2,** then the value of *y[10]* also becomes **2.** •

It turns out that understanding aliasing and the mechanisms that create it is essential if a compiler is to optimize a program. As we shall see starting in Chapter **9,** there are many situations where we can only optimize code if we can be sure certain variables are not aliased. For instance, we might determine that **x = 2** is the only place that variable *x* is ever assigned. If so, then we can replace a use of *x* by a use of **2;** for example, replace **a = x+3** by the simpler **a = 5.** But suppose there were another variable *y* that was aliased to *x.* Then an assignment **y = 4** might have the unexpected effect of changing *x.* It might also mean that replacing **a = x+3** by **a = 5** was a mistake; the proper value of *a* could be **7** there.

1.6.8 Exercises for Section 1.6

Exercise 1.6.1: For the block-structured C code of Fig. **1.13**(a), indicate the values assigned to *w, x, y,* and *z.*

Exercise 1.6.2: Repeat Exercise **1.6.1** for the code of Fig. **1.13**(b).

Exercise 1.6.3: For the block-structured code of Fig. **1.14,** assuming the usual static scoping of declarations, give the scope for each of the twelve declarations.

```
int w, x, y, z;
int i = 4; int j = 5;
<   int j = 7;
    i = 6;
    w = i + j;
}
x = i + j;
{   int i = 8;
    y = i + j;
}
z = i + j;
```

(a) Code for Exercise **1.6.1**

```
int w, x, y, z;
int i = 3; int j = 4;
{   int i = 5;
    w = i + j;
}
x = i + j;
{   int j = 6;
    i = 7;
    y = i + j;
}
z = i + j;
```

(b) Code for Exercise **1.6.2**

Figure **1.13:** Block-structured code

```
int w, x, y, z;     /* Block Bl */
{   int x, z;       /* Block B2 */
    {   int w, x;   /* Block B3 */ }
}
{   int w, x;       /* Block B4 */
    {   int y, z;   /* Block B5 */ }
}
```

Figure **1.14:** Block structured code for Exercise **1.6.3**

Exercise 1.6.4: What is printed by the following C code?

```
#define a (x+1)
int x = 2;
void b() { x = a; printf("°/,d\n", x) ; }
void c() { int x = 1; printf("°/,d\n"), a; >
void main0 { b(); c(); }
```

1.7 Summary of Chapter 1

- *Language Processors.* An integrated software development environment includes many different kinds of language processors such as compilers, interpreters, assemblers, linkers, loaders, debuggers, profilers.

- *Compiler Phases.* A compiler operates as a sequence of phases, each of which transforms the source program from one intermediate representation to another.

4> *Machine and Assembly Languages.* Machine languages were the first-generation programming languages, followed by assembly languages. Programming in these languages was time consuming and error prone.

4 *Modeling in Compiler Design.* Compiler design is one of the places where theory has had the most impact on practice. Models that have been found useful include automata, grammars, regular expressions, trees, and many others.

4 *Code Optimization.* Although code cannot truly be "optimized," the science of improving the efficiency of code is both complex and very important. It is a major portion of the study of compilation.

4 *Higher-Level Languages.* As time goes on, programming languages take on progressively more of the tasks that formerly were left to the programmer, such as memory management, type-consistency checking, or parallel execution of code.

4 *Compilers and Computer Architecture.* Compiler technology influences computer architecture, as well as being influenced by the advances in architecture. Many modern innovations in architecture depend on compilers being able to extract from source programs the opportunities to use the hardware capabilities effectively.

4 *Software Productivity and Software Security.* The same technology that allows compilers to optimize code can be used for a variety of program-analysis tasks, ranging from detecting common program bugs to discovering that a program is vulnerable to one of the many kinds of intrusions that "hackers" have discovered.

4 *Scope Rules.* The *scope* of a declaration of x is the context in which uses of x refer to this declaration. A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses *dynamic scope.*

4 *Environments.* The association of names with locations in memory and then with values can be described in terms of *environments,* which map names to locations in store, and *states,* which map locations to their values.

4- *Block Structure.* Languages that allow blocks to be nested are said to have *block structure.* A name a: in a nested block B is in the scope of a declaration D of x in an enclosing block if there is no other declaration of x in an intervening block.

4 *Parameter Passing.* Parameters are passed from a calling procedure to the callee either by value or by reference. When large objects are passed by value, the values passed are really references to the objects themselves, resulting in an effective call-by-reference.

- *Aliasing.* When parameters are (effectively) passed by reference, two formal parameters can refer to the same object. This possibility allows a change in one variable to change another.

1.8 References for Chapter 1

For the development of programming languages that were created and in use by 1967, including Fortran, Algol, Lisp, and Simula, see [7]. For languages that were created by 1982, including C, C++, Pascal, and Smalltalk, see [1].

The GNU Compiler Collection, gcc, is a popular source of open-source compilers for C, C++, Fortran, Java, and other languages [2]. Phoenix is a compiler-construction toolkit that provides an integrated framework for building the program analysis, code generation, and code optimization phases of compilers discussed in this book [3].

For more information about programming language concepts, we recommend [5,6]. For more on computer architecture and how it impacts compiling, we suggest [4].

1. Bergin, T. J. and R. G. Gibson, *History of Programming Languages,* ACM Press, New York, 1996.

2. `http://gcc.gnu.org/`.

3. `http://research.microsoft.com/phoenix/default.aspx` .

4. Hennessy, J. L. and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface,* Morgan-Kaufmann, San Francisco, CA, 2004.

5. Scott, M. L., *Programming Language Pragmatics, second edition,* Morgan-Kaufmann, San Francisco, CA, 2006.

6. Sethi, R., *Programming Languages: Concepts and Constructs,* Addison-Wesley, 1996.

7. Wexelblat, R. L., *History of Programming Languages,* Academic Press, New York, 1981.

Chapter 2

A Simple Syntax-Directed Translator

This chapter is an introduction to the compiling techniques in Chapters 3 through 6 of this book. It illustrates the techniques by developing a working Java program that translates representative programming language statements into three-address code, an intermediate representation. In this chapter, the emphasis is on the front end of a compiler, in particular on lexical analysis, parsing, and intermediate code generation. Chapters 7 and 8 show how to generate machine instructions from three-address code.

We start small by creating a syntax-directed translator that maps infix arithmetic expressions into postfix expressions. We then extend this translator to map code fragments as shown in Fig. 2.1 into three-address code of the form in Fig. 2.2.

The working Java translator appears in Appendix A. The use of Java is convenient, but not essential. In fact, the ideas in this chapter predate the creation of both Java and C.

```
{
      int i; int j; float[100] a; float v; float x;

      while ( true ) {
            do i = i+1; while ( a[i] < v );
            do j = j-1; while ( a[j] > v );
            if ( i >= j ) break;
            x = a[i]; a[i] = a[j]; a[j] = x;
      }
}
```

Figure 2.1: A code fragment to be translated

```
 1   i = i + 1
 2   tl = a [ i ]
 3:  if tl < v goto 1
 4:  J = J - 1
 5:  t2 - a [ j ]
 6:  if t2 > v goto 4
 7:  ifFalse i >= j goto 9
 8:  goto 14
 9:  x = a [ i ]
10:  t3 = a [ j ]
11:  a [ i ] = t3
12:  a [ j ] = x
13:  goto 1
14:
```

Figure 2.2: Simplified intermediate code for the program fragment in Fig. 2.1

2.1 Introduction

The analysis phase of a compiler breaks up a source program into constituent pieces and produces an internal representation for it, called intermediate code. The synthesis phase translates the intermediate code into the target program.

Analysis is organized around the "syntax" of the language to be compiled. The *syntax* of a programming language describes the proper form of its programs, while the *semantics* of the language defines what its programs mean; that is, what each program does when it executes. For specifying syntax, we present a widely used notation, called context-free grammars or BNF (for Backus-Naur Form) in Section 2.2. With the notations currently available, the semantics of a language is much more difficult to describe than the syntax. For specifying semantics, we shall therefore use informal descriptions and suggestive examples.

Besides specifying the syntax of a language, a context-free grammar can be used to help guide the translation of programs. In Section 2.3, we introduce a grammar-oriented compiling technique known as *syntax-directed translation.* Parsing or syntax analysis is introduced in Section 2.4.

The rest of this chapter is a quick tour through the model of a compiler front end in Fig. 2.3. We begin with the parser. For simplicity, we consider the syntax-directed translation of infix expressions to postfix form, a notation in which operators appear after their operands. For example, the postfix form of the expression 9 - 5 + 2 is 95 - 2+. Translation into postfix form is rich enough to illustrate syntax analysis, yet simple enough that the translator is shown in full in Section 2.5. The simple translator handles expressions like 9 - 5 + 2, consisting of digits separated by plus and minus signs. One reason for starting with such simple expressions is that the syntax analyzer can work directly with the individual characters for operators and operands.

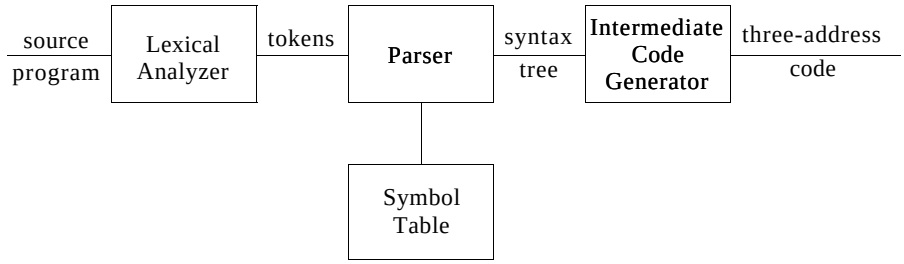


Figure 2.3: A model of a compiler front end

A lexical analyzer allows a translator to handle multicharacter constructs like identifiers, which are written as sequences of characters, but are treated as units called *tokens* during syntax analysis; for example, in the expression **count**+1, the identifier **count** is treated as a unit. The lexical analyzer in Section 2.6 allows numbers, identifiers, and "white space" (blanks, tabs, and newlines) to appear within expressions.

Next, we consider intermediate-code generation. Two forms of intermediate code are illustrated in Fig. 2.4. One form, called *abstract syntax trees* or simply *syntax trees*, represents the hierarchical syntactic structure of the source program. In the model in Fig. 2.3, the parser produces a syntax tree, that is further translated into three-address code. Some compilers combine parsing and intermediate-code generation into one component.

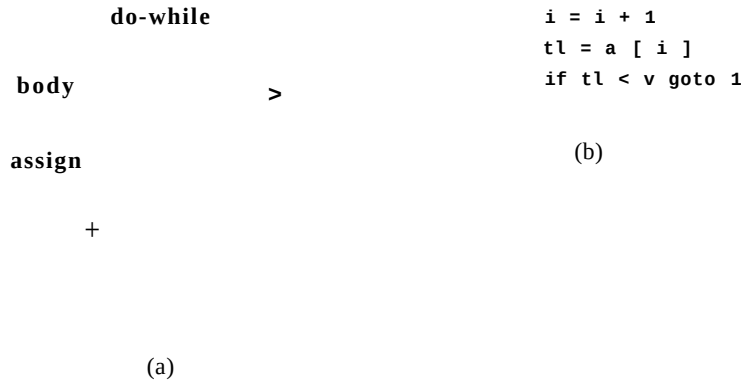


Figure 2.4: Intermediate code for **"do** i = i + 1; while (**a** [i] < **v);"**

The root of the abstract syntax tree in Fig. 2.4(a) represents an entire do-while loop. The left child of the root represents the body of the loop, which consists of only the assignment i = i + 1;. The right child of the root represents the condition **a**[i] **<v.** An implementation of syntax trees appears in Section 2.8(a).

The other common intermediate representation, shown in Fig. 2.4(b), is a

sequence of "three-address" instructions; a more complete example appears in Fig. 2.2. This form of intermediate code takes its name from instructions of the form $x = y$ **op** z, where **op** is a binary operator, y and z the are addresses for the operands, and x is the address for the result of the operation. A three-address instruction carries out at most one operation, typically a computation, a comparison, or a branch.

In Appendix A, we put the techniques in this chapter together to build a compiler front end in Java. The front end translates statements into assembly-level instructions.

2.2 Syntax Definition

In this section, we introduce a notation — the "context-free grammar," or "grammar" for short — that is used to specify the syntax of a language. Grammars will be used throughout this book to organize compiler front ends.

A grammar naturally describes the hierarchical structure of most programming language constructs. For example, an if-else statement in Java can have the form

if (expression) statement **else** statement

That is, an if-else statement is the concatenation of the keyword **if,** an opening parenthesis, an expression, a closing parenthesis, a statement, the keyword **else,** and another statement. Using the variable *expr* to denote an expression and the variable *stmt* to denote a statement, this structuring rule can be expressed as

stmt —> **if (** *expr)* *stmt* **else** *stmt*

in which the arrow may be read as "can have the form." Such a rule is called a *production.* In a production, lexical elements like the keyword **if** and the parentheses are called *terminals.* Variables like *expr* and *stmt* represent sequences of terminals and are called *nonterminals.*

2.2.1 Definition of Grammars

A *context-free grammar* has four components:

1. A set of *terminal* symbols, sometimes referred to as "tokens." The terminals are the elementary symbols of the language defined by the grammar.

2. A set of *nonterminals,* sometimes called "syntactic variables." Each nonterminal represents a set of strings of terminals, in a manner we shall describe.

3. A set of *productions,* where each production consists of a nonterminal, called the *head* or *left side* of the production, an arrow, and a sequence of

Tokens Versus Terminals

In a compiler, the lexical analyzer reads the characters of the source program, groups them into lexically meaningful units called lexemes, and produces as output tokens representing these lexemes. A token consists of two components, a token name and an attribute value. The token names are abstract symbols that are used by the parser for syntax analysis. Often, we shall call these token names *terminals*, since they appear as terminal symbols in the grammar for a programming language. The attribute value, if present, is a pointer to the symbol table that contains additional information about the token. This additional information is not part of the grammar, so in our discussion of syntax analysis, often we refer to tokens and terminals synonymously.

terminals and/or nonterminals, called the *body* or *right side* of the production. The intuitive intent of a production is to specify one of the written forms of a construct; if the head nonterminal represents a construct, then the body represents a written form of the construct.

4. A designation of one of the nonterminals as the *start* symbol.

We specify grammars by listing their productions, with the productions for the start symbol listed first. We assume that digits, signs such as < and <=, and boldface strings such as **while** are terminals. An italicized name is a nonterminal, and any nonitalicized name or symbol may be assumed to be a terminal.[1] For notational convenience, productions with the same nonterminal as the head can have their bodies grouped, with the alternative bodies separated by the symbol |, which we read as "or."

Example 2.1: Several examples in this chapter use expressions consisting of digits and plus and minus signs; e.g., strings such as 9-5+2, 3-1, or 7. Since a plus or minus sign must appear between two digits, we refer to such expressions as "lists of digits separated by plus or minus signs." The following grammar describes the syntax of these expressions. The productions are:

$$list \rightarrow list + digit \tag{2.1}$$

$$list \rightarrow list - digit \tag{2.2}$$

$$list \rightarrow digit \tag{2.3}$$

$$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \tag{2.4}$$

[1] Individual italic letters will be used for additional purposes, especially when grammars are studied in detail in Chapter 4. For example, we shall use ***X, Y,*** and ***Z*** to talk about a symbol that is either a terminal or a nonterminal. However, any italicized name containing two or more characters will continue to represent a nonterminal.

The bodies of the three productions with nonterminal *list* as head equivalently can be grouped:

list -+ *list* + *digit* | *list* - *digit* | *digit*

According to our conventions, the terminals of the grammar are the symbols

+ - 0 1 2 3 4 5 6 7 8 9

The nonterminals are the italicized names *list* and *digit,* with *list* being the start symbol because its productions are given first. •

We say a production is *for* a nonterminal if the nonterminal is the head of the production. A string of terminals is a sequence of zero or more terminals. The string of zero terminals, written as e, is called the *empty* string.[2]

2.2.2 Derivations

A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal. The terminal strings that can be derived from the start symbol form the *language* defined by the grammar.

Example 2.2: The language defined by the grammar of Example **2.1** consists of lists of digits separated by plus and minus signs. The ten productions for the nonterminal *digit* allow it to stand for any of the terminals **0,1,... ,9.** From production **(2.3),** a single digit by itself is a list. Productions **(2.1)** and **(2.2)** express the rule that any list followed by a plus or minus sign and then another digit makes up a new list.

Productions **(2.1)** to **(2.4)** are all we need to define the desired language. For example, we can deduce that **9-5+2** is a *list* as follows.

a) **9** is a *list* by production **(2.3),** since **9** is a *digit.*

b) **9-5** is a *list* by production **(2.2),** since **9** is a *list* and **5** is a *digit.*

c) **9-5+2** is a *list* by production **(2.1),** since **9-5** is a *list* and **2** is a *digit.*

•

Example 2.3: A somewhat different sort of list is the list of parameters in a function call. In Java, the parameters are enclosed within parentheses, as in the call **max(x ,y)** of function **max** with parameters x and **y.** One nuance of such lists is that an empty list of parameters may be found between the terminals (and). We may start to develop a grammar for such sequences with the productions:

[2]Technically, e can be a string of zero symbols from any alphabet (collection of symbols).

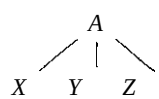
call -> **id** (*optparams*)
optparams -> *params* \ *e*
params —> *params* , *param* \ *param*

Note that the second possible body for *optparams* ("optional parameter list") is e, which stands for the empty string of symbols. That is, *optparams* can be replaced by the empty string, so a *call* can consist of a function name followed by the two-terminal string (). Notice that the productions for *params* are analogous to those for *list* in Example 2.1, with comma in place of the arithmetic operator + or -, and *param* in place of *digit.* We have not shown the productions for *param,* since parameters are really arbitrary expressions. Shortly, we shall discuss the appropriate productions for the various language constructs, such as expressions, statements, and so on. •

Parsing is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar, and if it cannot be derived from the start symbol of the grammar, then reporting syntax errors within the string. Parsing is one of the most fundamental problems in all of compiling; the main approaches to parsing are discussed in Chapter **4.** In this chapter, for simplicity, we begin with source programs like 9-5+2 in which each character is a terminal; in general, a source program has multicharacter lexemes that are grouped by the lexical analyzer into tokens, whose first components are the terminals processed by the parser.

2.2.3 Parse Trees

A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. If nonterminal *A* has a production *A* -> *XYZ,* then a parse tree may have an interior node labeled *A* with three children labeled *X, Y,* and *Z,* from left to right:



Formally, given a context-free grammar, a *parse tree* according to the grammar is a tree with the following properties:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a terminal or by e.
3. Each interior node is labeled by a nonterminal.
4. If *A* is the nonterminal labeling some interior node and $X_1, X_2, \cdots, X_n$ are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1 X_2 \cdots X_n$. Here, $X_1, X_2, \ldots, X_n$ each stand

Tree Terminology

Tree data structures figure prominently in compiling.

- A tree consists of one or more *nodes.* Nodes may have *labels,* which in this book typically will be grammar symbols. When we draw a tree, we often represent the nodes by these labels only.
- Exactly one node is the *root.* All nodes except the root have a unique *parent;* the root has no parent. When we draw trees, we place the parent of a node above that node and draw an edge between them. The root is then the highest (top) node.
- If node *N* is the parent of node *M,* then *M* is a *child* of *N.* The children of one node are called *siblings.* They have an order, *from the left,* and when we draw trees, we order the children of a given node in this manner.
- A node with no children is called a *leaf.* Other nodes — those with one or more children — are *interior nodes.*
- A *descendant* of a node *N* is either *N* itself, a child of *N,* a child of a child of *N,* and so on, for any number of levels. We say node *N* is an *ancestor* of node *M* if *M* is a descendant of *N.*

for a symbol that is either a terminal or a nonterminal. As a special case, if *A* ->• e is a production, then a node labeled *A* may have a single child labeled e.

Example 2.4: The derivation of 9-5+2 in Example 2.2 is illustrated by the tree in Fig. 2.5. Each node in the tree is labeled by a grammar symbol. An interior node and its children correspond to a production; the interior node corresponds to the head of the production, the children to the body.

In Fig. 2.5, the root is labeled *list,* the start symbol of the grammar in Example 2.1. The children of the root are labeled, from left to right, *list,* +, and *digit.* Note that

list —>• *list* + *digit*

is a production in the grammar of Example 2.1. The left child of the root is similar to the root, with a child labeled - instead of +. The three nodes labeled *digit* each have one child that is labeled by a digit. •

From left to right, the leaves of a parse tree form the *yield* of the tree, which is the string *generated* or *derived* from the nonterminal at the root of the parse

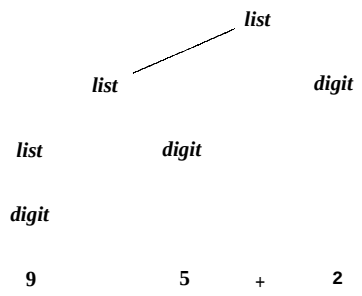


Figure 2.5: Parse tree for 9-5+2 according to the grammar in Example 2.1

tree. In Fig. 2.5, the yield is 9-5+2; for convenience, all the leaves are shown at the bottom level. Henceforth, we shall not necessarily line up the leaves in this way. Any tree imparts a natural left-to-right order to its leaves, based on the idea that if *X* and *Y* are two children with the same parent, and *X* is to the left of y, then all descendants of *X* are to the left of descendants of *Y*.

Another definition of the language generated by a grammar is as the set of strings that can be generated by some parse tree. The process of finding a parse tree for a given string of terminals is called *parsing* that string.

2.2.4 Ambiguity

We have to be careful in talking about *the* structure of a string according to a grammar. A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be *ambiguous.* To show that a grammar is ambiguous, all we need to do is find a terminal string that is the yield of more than one parse tree. Since a string with more than one parse tree usually has more than one meaning, we need to design unambiguous grammars for compiling applications, or to use ambiguous grammars with additional rules to resolve the ambiguities.

Example 2.5: Suppose we used a single nonterminal *string* and did not distinguish between digits and lists, as in Example 2.1. We could have written the grammar

string -» *string* + *string* \ *string* - *string* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 J 9

Merging the notion of *digit* and *list* into the nonterminal *string* makes superficial sense, because a single *digit* is a special case of a *list.*

However, Fig. 2.6 shows that an expression like 9-5+2 has more than one parse tree with this grammar. The two trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5)+2 and 9-(5+2). This second parenthesization gives the expression the unexpected value 2 rather than the customary value 6. The grammar of Example 2.1 does not permit this interpretation. •

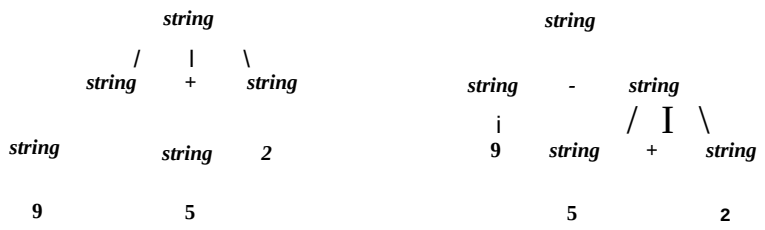


Figure **2.6:** Two parse trees for **9-5+2**

2.2.5 Associativity of Operators

By convention, **9+5+2** is equivalent to **(9+5)+2** and **9-5-2** is equivalent to **(9-5)-2.** When an operand like **5** has operators to its left and right, conventions are needed for deciding which operator applies to that operand. We say that the operator + *associates* to the left, because an operand with plus signs on both sides of it belongs to the operator to its left. In most programming languages the four arithmetic operators, addition, subtraction, multiplication, and division are left-associative.

Some common operators such as exponentiation are right-associative. As another example, the assignment operator = in C and its descendants is right-associative; that is, the expression **a**=b=c is treated in the same way as the expression **a**=(b=c).

Strings like **a**=b=c with a right-associative operator are generated by the following grammar:

right ->• *letter* = *right* | *letter*
letter -> **a** | **b** | • • • | **z**

The contrast between a parse tree for a left-associative operator like - and a parse tree for a right-associative operator like = is shown by Fig. **2.7.** Note that the parse tree for **9-5-2** grows down towards the left, whereas the parse tree for a-b=c grows down towards the right.

2.2.6 Precedence of Operators

Consider the expression **9+5*2.** There are two possible interpretations of this expression: **(9+5)*2** or **9+(5*2).** The associativity rules for + and * apply to occurrences of the same operator, so they do not resolve this ambiguity. Rules defining the relative precedence of operators are needed when more than one kind of operator is present.

We say that * has *higher precedence* than + if * takes its operands before + does. In ordinary arithmetic, multiplication and division have higher precedence than addition and subtraction. Therefore, **5** is taken by * in both **9+5*2** and **9*5+2;** i.e., the expressions are equivalent to **9+(5*2)** and **(9*5)+2,** respectively.

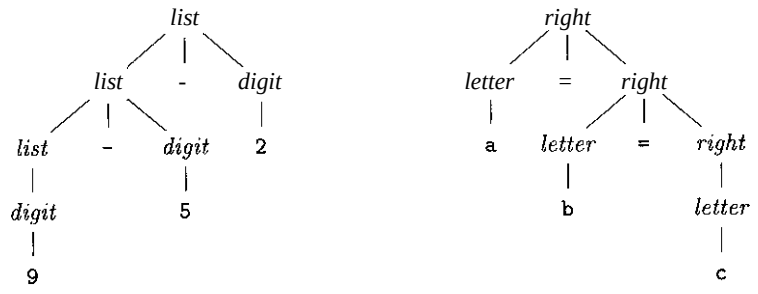


Figure 2.7: Parse trees for left- and right-associative grammars

Example 2.6 : A grammar for arithmetic expressions can be constructed from a table showing the associativity and precedence of operators. We start with the four common arithmetic operators and a precedence table, showing the operators in order of increasing precedence. Operators on the same line have the same associativity and precedence:

left-associative: + -
left-associative: * /

We create two nonterminals *expr* and *term* for the two levels of precedence, and an extra nonterminal *factor* for generating basic units in expressions. The basic units in expressions are presently digits and parenthesized expressions.

factor —> **digit** | **(** *expr)*

Now consider the binary operators, * and /, that have the highest precedence. Since these operators associate to the left, the productions are similar to those for lists that associate to the left.

term —> *term* * *factor*
| *term* / *factor*
| *factor*

Similarly, *expr* generates lists of terms separated by the additive operators.

expr —> *expr* + *term*
| *expr* - *term*
| *term*

The resulting grammar is therefore

expr -» *expr* + *term* \ *expr* - *term* \ *term*
term -> *term* * *factor* | *term* / *factor* | *factor*
factor -•• **digit** | **(** *expr)*

Generalizing the Expression Grammar of Example 2.6

We can think of a factor as an expression that cannot be "torn apart" by any operator. By "torn apart," we mean that placing an operator next to any factor, on either side, does not cause any piece of the factor, other than the whole, to become an operand of that operator. If the factor is a parenthesized expression, the parentheses protect against such "tearing," while if the factor is a single operand, it cannot be torn apart.

A term (that is not also a factor) is an expression that can be torn apart by operators of the highest precedence: * and /, but not by the lower-precedence operators. An expression (that is not a term or factor) can be torn apart by any operator.

We can generalize this idea to any number *n* of precedence levels. We need *n+1* nonterminals. The first, like *factor* in Example 2.6, can never be torn apart. Typically, the production bodies for this nonterminal are only single operands and parenthesized expressions. Then, for each precedence level, there is one nonterminal representing expressions that can be torn apart only by operators at that level or higher. Typically, the productions for this nonterminal have bodies representing uses of the operators at that level, plus one body that is just the nonterminal for the next higher level.

With this grammar, an expression is a list of terms separated by either + or - signs, and a term is a list of factors separated by * or / signs. Notice that any parenthesized expression is a factor, so with parentheses we can develop expressions that have arbitrarily deep nesting (and arbitrarily deep trees). •

Example 2.7: Keywords allow us to recognize statements, since most statement begin with a keyword or a special character. Exceptions to this rule include assignments and procedure calls. The statements defined by the (ambiguous) grammar in Fig. 2.8 are legal in Java.

In the first production for *stmt,* the terminal **id** represents any identifier. The productions for *expression* are not shown. The assignment statements specified by the first production are legal in Java, although Java treats = as an assignment operator that can appear within an expression. For example, Java allows **a** = b = **c,** which this grammar does not.

The nonterminal *stmts* generates a possibly empty list of statements. The second production for *stmts* generates the empty list e. The first production generates a possibly empty list of statements followed by a statement.

The placement of semicolons is subtle; they appear at the end of every body that does not end in *stmt.* This approach prevents the build-up of semicolons after statements such as if- and while-, which end with nested substatements. When the nested substatement is an assignment or a do-while, a semicolon will be generated as part of the substatement. •

stmt	—>	**id** = *expression* ;
	|	**if** (*expression*) **s£ra£**
	|	**if** (*expression*) **stmi else** *stmt*
	|	**while** (*expression*) **sirai**
	|	**do** *stmt* **while** (*expression*) ;
	|	{ *stmts* }
stmts	->•	s£m£s s£ra£

Figure 2.8: A grammar for a subset of Java statements

2.2.7 Exercises for Section 2.2

Exercise 2.2.1: Consider the context-free grammar

S -> 5 5 + | 5 5 * | a

a) Show how the string **aa+a*** can be generated by this grammar.

b) Construct a parse tree for this string.

c) What language does this grammar generate? Justify your answer.

Exercise 2.2.2: What language is generated by the following grammars? In each case justify your answer.

a) 5 -4 0 5 1 | 0 1

b) 5 -»• + 5 5 | - 5 5 | **a**

c) 5 -> 5 (5) 5 | e

d) 5 -» a 5 b 5 | **b** 5 **a** 5 | e

e) 5 -> **a** | 5 + 5 | 5 5 | 5 * | (5)

Exercise 2.2.3: Which of the grammars in Exercise 2.2.2 are ambiguous?

Exercise 2.2.4: Construct unambiguous context-free grammars for each of the following languages. In each case show that your grammar is correct.

a) Arithmetic expressions in postfix notation.

b) Left-associative lists of identifiers separated by commas.

c) Right-associative lists of identifiers separated by commas.

d) Arithmetic expressions of integers and identifiers with the four binary operators +, -, *, /.

! e) Add unary plus and minus to the arithmetic operators of (d).

Exercise 2.2.5:

a) Show that all binary strings generated by the following grammar have values divisible by 3. *Hint.* Use induction on the number of nodes in a parse tree.

$$num \rightarrow 11 \mid 1001 \mid numO \mid num\ num$$

b) Does the grammar generate all binary strings with values divisible by 3?

Exercise 2.2.6: Construct a context-free grammar for roman numerals.

2.3 Syntax-Directed Translation

Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar. For example, consider an expression *expr* generated by the production

$$expr \rightarrow expr_1 + term$$

Here, *expr* is the sum of the two subexpressions $expr_1$ and *term.* (The subscript in $expr_1$ is used only to distinguish the instance of *expr* in the production body from the head of the production). We can translate *expr* by exploiting its structure, as in the following pseudo-code:

```
translate expr1;
translate term;
handle +;
```

Using a variant of this pseudocode, we shall build a syntax tree for *expr* in Section 2.8 by building syntax trees for $expr_1$ and *term* and then handling + by constructing a node for it. For convenience, the example in this section is the translation of infix expressions into postfix notation.

This section introduces two concepts related to syntax-directed translation:

- *Attributes.* An *attribute* is any quantity associated with a programming construct. Examples of attributes are data types of expressions, the number of instructions in the generated code, or the location of the first instruction in the generated code for a construct, among many other possibilities. Since we use grammar symbols (nonterminals and terminals) to represent programming constructs, we extend the notion of attributes from constructs to the symbols that represent them.

- *(Syntax-directed) translation schemes.* A *translation scheme* is a notation for attaching program fragments to the productions of a grammar. The program fragments are executed when the production is used during syntax analysis. The combined result of all these fragment executions, in the order induced by the syntax analysis, produces the translation of the program to which this analysis/synthesis process is applied.

Syntax-directed translations will be used throughout this chapter to translate infix expressions into postfix notation, to evaluate expressions, and to build syntax trees for programming constructs. A more detailed discussion of syntax-directed formalisms appears in Chapter **5.**

2.3.1 Postfix Notation

The examples in this section deal with translation into postfix notation. The *postfix notation* for an expression E can be defined inductively as follows:

1. If E is a variable or constant, then the postfix notation for E is E itself.
2. If E is an expression of the form $E\backslash$ **op** E_2, where **op** is any binary operator, then the postfix notation for E is $E[\ E'_2$ **op,** where $E[$ and E'_2 are the postfix notations for $E\backslash$ and E_2, respectively.
3. If E is a parenthesized expression of the form *(Ei),* then the postfix notation for E is the same as the postfix notation for $E\backslash$.

Example 2.8 : The postfix notation for **(9-5)+2** is **95-2+.** That is, the translations of **9, 5,** and **2** are the constants themselves, by rule **(1).** Then, the translation of **9-5** is **95-** by rule **(2).** The translation of **(9-5)** is the same by rule **(3).** Having translated the parenthesized subexpression, we may apply rule **(2)** to the entire expression, with **(9-5)** in the role of $E\backslash$ and **2** in the role of E_2, to get the result **95-2+.**

As another example, the postfix notation for **9- (5+2)** is **952+-.** That is, **5+2** is first translated into **52+,** and this expression becomes the second argument of the minus sign. •

No parentheses are needed in postfix notation, because the position and *arity* (number of arguments) of the operators permits only one decoding of a postfix expression. The "trick" is to repeatedly scan the postfix string from the left, until you find an operator. Then, look to the left for the proper number of operands, and group this operator with its operands. Evaluate the operator on the operands, and replace them by the result. Then repeat the process, continuing to the right and searching for another operator.

Example 2.9: Consider the postfix expression **952+-3*.** Scanning from the left, we first encounter the plus sign. Looking to its left we find operands **5** and **2.** Their sum, **7,** replaces **52+,** and we have the string **97-3*.** Now, the leftmost

operator is the minus sign, and its operands are **9** and **7.** Replacing these by the result of the subtraction leaves **23*.** Last, the multiplication sign applies to **2** and **3,** giving the result **6.** •

2.3.2 Synthesized Attributes

The idea of associating quantities with programming constructs—for example, values and types with expressions—can be expressed in terms of grammars. We associate attributes with nonterminals and terminals. Then, we attach rules to the productions of the grammar; these rules describe how the attributes are computed at those nodes of the parse tree where the production in question is used to relate a node to its children.

A *syntax-directed definition* associates

1. With each grammar symbol, a set of attributes, and

2. With each production, a set of *semantic rules* for computing the values of the attributes associated with the symbols appearing in the production.

Attributes can be evaluated as follows. For a given input string *x,* construct a parse tree for *x.* Then, apply the semantic rules to evaluate attributes at each node in the parse tree, as follows.

Suppose a node A?' in a parse tree is labeled by the grammar symbol *X.* We write *X.a* to denote the value of attribute *a* of *X* at that node. A parse tree showing the attribute values at each node is called an *annotated* parse tree. For example, Fig. **2.9** shows an annotated parse tree for **9-5+2** with an attribute *t* associated with the nonterminals *expr* and *term.* The value **95-2+** of the attribute at the root is the postfix notation for **9-5+2.** We shall see shortly how these expressions are computed.

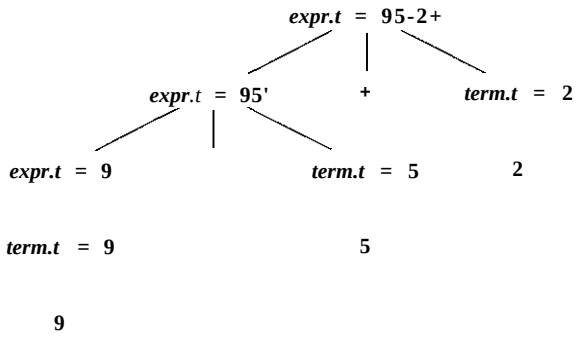


Figure **2.9:** Attribute values at nodes in a parse tree

An attribute is said to be *synthesized* if its value at a parse-tree node *N* is determined from attribute values at the children of *N* and at *N* itself. Synthesized

attributes have the desirable property that they can be evaluated during a single bottom-up traversal of a parse tree. In Section 5.1.1 we shall discuss another important kind of attribute: the "inherited" attribute. Informally, inherited attributes have their value at a parse-tree node determined from attribute values at the node itself, its parent, and its siblings in the parse tree.

Example 2.10: The annotated parse tree in Fig. 2.9 is based on the syntax-directed definition in Fig. 2.10 for translating expressions consisting of digits separated by plus or minus signs into postfix notation. Each nonterminal has a string-valued attribute *t* that represents the postfix notation for the expression generated by that nonterminal in a parse tree. The symbol || in the semantic rule is the operator for string concatenation.

PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
$term \rightarrow 9$	$term.t = '9'$

Figure 2.10: Syntax-directed definition for infix to postfix translation

The postfix form of a digit is the digit itself; e.g., the semantic rule associated with the production *term* -» 9 defines *term.t* to be 9 itself whenever this production is used at a node in a parse tree. The other digits are translated similarly. As another example, when the production *expr* *term* is applied, the value of *term.t* becomes the value of *expr.t.*

The production $expr \rightarrow expr_1 + term$ derives an expression containing a plus operator.[3] The left operand of the plus operator is given by $expr_1$ and the right operand by *term.* The semantic rule

$$expr.t = expr.t \mid\mid term.t \mid\mid '+'$$

associated with this production constructs the value of attribute *expr.t* by concatenating the postfix forms *expr.t* and *term.t* of the left and right operands, respectively, and then appending the plus sign. This rule is a formalization of the definition of "postfix expression." •

[3] In this and many other rules, the same nonterminal (***expr,*** here) appears several times. The purpose of the subscript 1 in $expr_1$ is to distinguish the two occurrences of ***expr*** in the production; the "1" is not part of the nonterminal. See the box on "Convention Distinguishing Uses of a Nonterminal" for more details.

Convention Distinguishing Uses of a Nonterminal

In rules, we often have a need to distinguish among several uses of the same nonterminal in the head and/or body of a production; e.g., see Example 2.10. The reason is that in the parse tree, different nodes labeled by the same nonterminal usually have different values for their translations. We shall adopt the following convention: the nonterminal appears unsubscripted in the head and with distinct subscripts in the body. These are all occurrences of the same nonterminal, and the subscript is not part of its name. However, the reader should be alert to the difference between examples of specific translations, where this convention is used, and generic productions like $A \rightarrow X_1 X_2, \cdots, X_n$, where the subscripted X's represent an arbitrary list of grammar symbols, and are *not* instances of one particular nonterminal called *X*.

2.3.3 Simple Syntax-Directed Definitions

The syntax-directed definition in Example 2.10 has the following important property: the string representing the translation of the nonterminal at the head of each production is the concatenation of the translations of the nonterminals in the production body, in the same order as in the production, with some optional additional strings interleaved. A syntax-directed definition with this property is termed *simple.*

Example 2.11: Consider the first production and semantic rule from Fig. 2.10:

PRODUCTION	SEMANTIC RULE
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$

(2.5)

Here the translation *expr.t* is the concatenation of the translations of $expr_1$ and *term,* followed by the symbol +. Notice that $expr_1$ and *term* appear in the same order in both the production body and the semantic rule. There are no additional symbols before or between their translations. In this example, the only extra symbol occurs at the end. •

When translation schemes are discussed, we shall see that a simple syntax-directed definition can be implemented by printing only the additional strings, in the order they appear in the definition.

2.3.4 Tree Traversals

Tree traversals will be used for describing attribute evaluation and for specifying the execution of code fragments in a translation scheme. A *traversal* of a tree starts at the root and visits each node of the tree in some order.

A *depth-first* traversal starts at the root and recursively visits the children of each node in any order, not necessarily from left to right. It is called "depth-first" because it visits an unvisited child of a node whenever it can, so it visits nodes as far away from the root (as "deep") as quickly as it can.

The procedure *visit(N)* in Fig. 2.11 is a depth first traversal that visits the children of a node in left-to-right order, as shown in Fig. 2.12. In this traversal, we have included the action of evaluating translations at each node, just before we finish with the node (that is, after translations at the children have surely been computed). In general, the actions associated with a traversal can be whatever we choose, or nothing at all.

```
procedure visit (node N) {
        for ( each child C of N, from left to right ) {
                visit (C);
        }
        evaluate semantic rules at node N;
}
```

Figure 2.11: A depth-first traversal of a tree

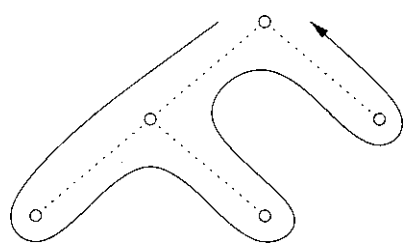


Figure 2.12: Example of a depth-first traversal of a tree

A syntax-directed definition does not impose any specific order for the evaluation of attributes on a parse tree; any evaluation order that computes an attribute a after all the other attributes that *a* depends on is acceptable. Synthesized attributes can be evaluated during any *bottom-up* traversal, that is, a traversal that evaluates attributes at a node after having evaluated attributes at its children. In general, with both synthesized and inherited attributes, the matter of evaluation order is quite complex; see Section 5.2.

2.3.5 Translation Schemes

The syntax-directed definition in Fig. 2.10 builds up a translation by attaching strings as attributes to the nodes in the parse tree. We now consider an alternative approach that does not need to manipulate strings; it produces the same translation incrementally, by executing program fragments.

Preorder and Postorder Traversals

Preorder and postorder traversals are two important special cases of depth-first traversals in which we visit the children of each node from left to right.

Often, we traverse a tree to perform some particular action at each node. If the action is done when we first visit a node, then we may refer to the traversal as a *preorder traversal.* Similarly, if the action is done just before we leave a node for the last time, then we say it is a *postorder traversal* of the tree. The procedure *visit(N)* in Fig. 2.11 is an example of a postorder traversal.

Preorder and postorder traversals define corresponding orderings on nodes, based on when the action at a node would be performed. The *preorder* of a (sub)tree rooted at node JV consists of *N,* followed by the preorders of the subtrees of each of its children, if any, from the left. The *postorder* of a (sub)tree rooted at TV' consists of the postorders of each of the subtrees for the children of *N,* if any, from the left, followed by *N* itself.

A syntax-directed translation scheme is a notation for specifying a translation by attaching program fragments to productions in a grammar. A translation scheme is like a syntax-directed definition, except that the order of evaluation of the semantic rules is explicitly specified.

Program fragments embedded within production bodies are called *semantic actions.* The position at which an action is to be executed is shown by enclosing it between curly braces and writing it within the production body, as in

rest —> + *term* {print('+')} *rest*\

We shall see such rules when we consider an alternative form of grammar for expressions, where the nonterminal *rest* represents "everything but the first term of an expression." This form of grammar is discussed in Section 2.4.5. Again, the subscript in *rest*\ distinguishes this instance of nonterminal *rest* in the production body from the instance of *rest* at the head of the production.

When drawing a parse tree for a translation scheme, we indicate an action by constructing an extra child for it, connected by a dashed line to the node that corresponds to the head of the production. For example, the portion of the parse tree for the above production and action is shown in Fig. 2.13. The node for a semantic action has no children, so the action is performed when that node is first seen.

Example 2.12: The parse tree in Fig. 2.14 has print statements at extra leaves, which are attached by dashed lines to interior nodes of the parse tree. The translation scheme appears in Fig. 2.15. The underlying grammar generates expressions consisting of digits separated by plus and minus signs. The

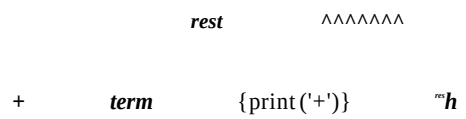


Figure 2.13: An extra leaf is constructed for a semantic action

actions embedded in the production bodies translate such expressions into postfix notation, provided we perform a left-to-right depth-first traversal of the tree and execute each print statement when we visit its leaf.

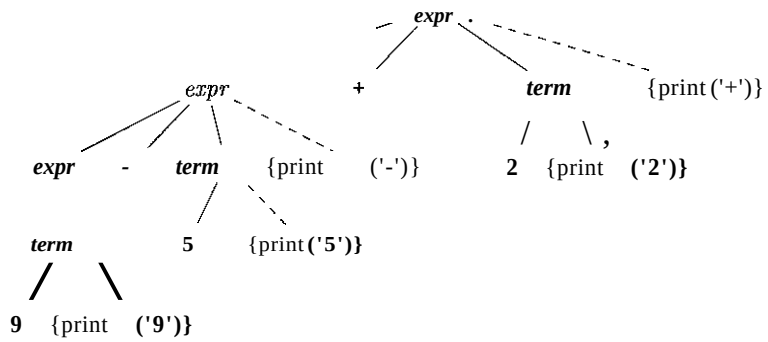


Figure 2.14: Actions translating 9-5+2 into 95-2+

expr	-¥	*expr*ₓ	+	*term*	{print('+')}
expr	->	*expr*ₓ	-	*term*	{print ('-')}
expr	-¥	*term*			
term	-*t*	0			{printCO')}
term	-»	1			{print(T)}
term	->	9			{print('9')}

Figure 2.15: Actions for translating into postfix notation

The root of Fig. 2.14 represents the first production in Fig. 2.15. In a postorder traversal, we first perform all the actions in the leftmost subtree of the root, for the left operand, also labeled *expr* like the root. We then visit the leaf + at which there is no action. We next perform the actions in the subtree for the right operand *term* and, finally, the semantic action { print('+') } at the extra node.

Since the productions for *term* have only a digit on the right side, that digit is printed by the actions for the productions. No output is necessary for the production *expr* -> *term,* and only the operator needs to be printed in the

action for each of the first two productions. When executed during a postorder traversal of the parse tree, the actions in Fig. 2.14 print 95-2+. •

Note that although the schemes in Fig. 2.10 and Fig. 2.15 produce the same translation, they construct it differently; Fig. 2.10 attaches strings as attributes to the nodes in the parse tree, while the scheme in Fig. 2.15 prints the translation incrementally, through semantic actions.

The semantic actions in the parse tree in Fig. 2.14 translate the infix expression 9-5+2 into 95-2+ by printing each character in 9-5+2 exactly once, without using any storage for the translation of subexpressions. When the output is created incrementally in this fashion, the order in which the characters are printed is significant.

The implementation of a translation scheme must ensure that semantic actions are performed in the order they would appear during a postorder traversal of a parse tree. The implementation need not actually construct a parse tree (often it does not), as long as it ensures that the semantic actions are performed as if we constructed a parse tree and then executed the actions during a postorder traversal.

2.3.6 Exercises for Section 2.3

Exercise 2.3.1: Construct a syntax-directed translation scheme that translates arithmetic expressions from infix notation into prefix notation in which an operator appears before its operands; e.g., $-xy$ is the prefix notation for $x - y$. Give annotated parse trees for the inputs 9-5+2 and 9-5*2.

Exercise 2.3.2: Construct a syntax-directed translation scheme that translates arithmetic expressions from postfix notation into infix notation. Give annotated parse trees for the inputs 95-2* and 952*-.

Exercise 2.3.3: Construct a syntax-directed translation scheme that translates integers into roman numerals.

Exercise 2.3.4: Construct a syntax-directed translation scheme that translates roman numerals into integers.

Exercise 2.3.5: Construct a syntax-directed translation scheme that translates postfix arithmetic expressions into equivalent infix arithmetic expressions.

2.4 Parsing

Parsing is the process of determining how a string of terminals can be generated by a grammar. In discussing this problem, it is helpful to think of a parse tree being constructed, even though a compiler may not construct one, in practice. However, a parser must be capable of constructing the tree in principle, or else the translation cannot be guaranteed correct.

This section introduces a parsing method called "recursive descent," which can be used both to parse and to implement syntax-directed translators. A complete Java program, implementing the translation scheme of Fig. 2.15, appears in the next section. A viable alternative is to use a software tool to generate a translator directly from a translation scheme. Section 4.9 describes such a tool — Yacc; it can implement the translation scheme of Fig. 2.15 without modification.

For any context-free grammar there is a parser that takes at most $0(n^3)$ time to parse a string of *n* terminals. But cubic time is generally too expensive. Fortunately, for real programming languages, we can generally design a grammar that can be parsed quickly. Linear-time algorithms suffice to parse essentially all languages that arise in practice. Programming-language parsers almost always make a single left-to-right scan over the input, looking ahead one terminal at a time, and constructing pieces of the parse tree as they go.

Most parsing methods fall into one of two classes, called the *top-down* and *bottom-up* methods. These terms refer to the order in which nodes in the parse tree are constructed. In top-down parsers, construction starts at the root and proceeds towards the leaves, while in bottom-up parsers, construction starts at the leaves and proceeds towards the root. The popularity of top-down parsers is due to the fact that efficient parsers can be constructed more easily by hand using top-down methods. Bottom-up parsing, however, can handle a larger class of grammars and translation schemes, so software tools for generating parsers directly from grammars often use bottom-up methods.

2.4.1 Top-Down Parsing

We introduce top-down parsing by considering a grammar that is well-suited for this class of methods. Later in this section, we consider the construction of top-down parsers in general. The grammar in Fig. 2.16 generates a subset of the statements of C or Java. We use the boldface terminals **if** and **for** for the keywords "if" and "for", respectively, to emphasize that these character sequences are treated as units, i.e., as single terminal symbols. Further, the terminal **expr** represents expressions; a more complete grammar would use a nonterminal *expr* and have productions for nonterminal *expr.* Similarly, **other** is a terminal representing other statement constructs.

The top-down construction of a parse tree like the one in Fig. 2.17, is done by starting with the root, labeled with the starting nonterminal *stmt,* and repeatedly performing the following two steps.

1. At node JV, labeled with nonterminal *A,* select one of the productions for *A* and construct children at *N* for the symbols in the production body.

2. Find the next node at which a subtree is to be constructed, typically the leftmost unexpanded nonterminal of the tree.

For some grammars, the above steps can be implemented during a single left-to-right scan of the input string. The current terminal being scanned in the

stmt	—y	**expr ;**
		if (expr) *stmt*
		for (*optexpr ; optexpr ; optexpr)* *stmt*
		other
optexpr	—>	*e*
		expr

Figure 2.16: A grammar for some statements in C and Java

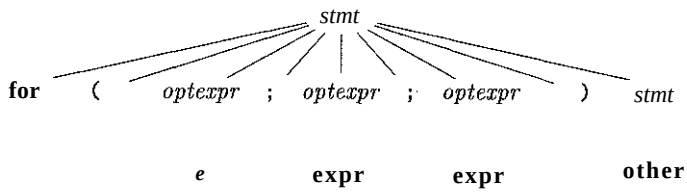


Figure 2.17: A parse tree according to the grammar in Fig. 2.16

input is frequently referred to as the *lookahead* symbol. Initially, the lookahead symbol is the first, i.e., leftmost, terminal of the input string. Figure 2.18 illustrates the construction of the parse tree in Fig. 2.17 for the input string

for (; expr ; expr) other

Initially, the terminal **for** is the lookahead symbol, and the known part of the parse tree consists of the root, labeled with the starting nonterminal *stmt* in Fig. 2.18(a). The objective is to construct the remainder of the parse tree in such a way that the string generated by the parse tree matches the input string.

For a match to occur, the nonterminal *stmt* in Fig. 2.18(a) must derive a string that starts with the lookahead symbol **for.** In the grammar of Fig. 2.16, there is just one production for *stmt that* can derive such a string, so we select it, and construct the children of the root labeled with the symbols in the production body. This expansion of the parse tree is shown in Fig. 2.18(b).

Each of the three snapshots in Fig. 2.18 has arrows marking the lookahead symbol in the input and the node in the parse tree that is being considered. Once children are constructed at a node, we next consider the leftmost child. In Fig. 2.18(b), children have just been constructed at the root, and the leftmost child labeled with **for** is being considered.

When the node being considered in the parse tree is for a terminal, and the terminal matches the lookahead symbol, then we advance in both the parse tree and the input. The next terminal in the input becomes the new lookahead symbol, and the next child in the parse tree is considered. In Fig. 2.18(c), the arrow in the parse tree has advanced to the next child of the root, and the arrow

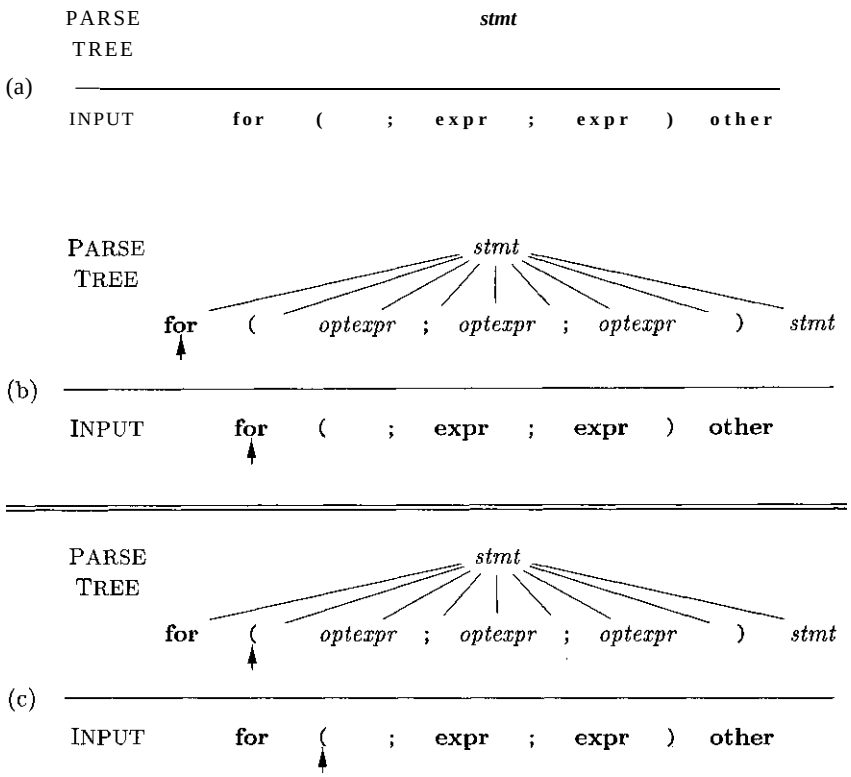


Figure 2.18: Top-down parsing while scanning the input from left to right

in the input has advanced to the next terminal, which is (. A further advance will take the arrow in the parse tree to the child labeled with nonterminal *optexpr* and take the arrow in the input to the terminal ;.

At the nonterminal node labeled *optexpr,* we repeat the process of selecting a production for a nonterminal. Productions with e as the body ("e-productions") require special treatment. For the moment, we use them as a default when no other production can be used; we return to them in Section 2.4.3. With nonterminal *optexpr* and lookahead ;, the e-production is used, since ; does not match the only other production for *optexpr,* which has terminal **expr** as its body.

In general, the selection of a production for a nonterminal may involve trial-and-error; that is, we may have to try a production and backtrack to try another production if the first is found to be unsuitable. A production is unsuitable if, after using the production, we cannot complete the tree to match the input

string. Backtracking is not needed, however, in an important special case called predictive parsing, which we discuss next.

2.4.2 Predictive Parsing

Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input. One procedure is associated with each nonterminal of a grammar. Here, we consider a simple form of recursive-descent parsing, called *predictive parsing,* in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal. The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input, and can be used to build an explicit parse tree, if desired.

The predictive parser in Fig. **2.19** consists of procedures for the nonterminals *stmt* and *optexpr* of the grammar in Fig. **2.16** and an additional procedure *match,* used to simplify the code for *stmt* and *optexpr.* Procedure *match(t)* compares its argument *t* with the lookahead symbol and advances to the next input terminal if they match. Thus *match* changes the value of variable *lookahead,* a global variable that holds the currently scanned input terminal.

Parsing begins with a call of the procedure for the starting nonterminal *stmt.* With the same input as in Fig. **2.18,** *lookahead* is initially the first terminal **for.** Procedure *stmt* executes code corresponding to the production

stmt -> **for** **(** *optexpr* ; *optexpr* ; *optexpr*) *stmt*

In the code for the production body — that is, the **for** case of procedure *stmt* — each terminal is matched with the lookahead symbol, and each nonterminal leads to a call of its procedure, in the following sequence of calls:

```
match(for);     matched);
optexprQ;    match(';');    optexprQ;    match(';');    optexprQ;
match(')');     stmtQ;
```

Predictive parsing relies on information about the first symbols that can be generated by a production body. More precisely, let *a* be a string of grammar symbols (terrninals and/or nonterminals). We define **FIRST**(a) to be the set of terminals that appear as the first symbols of one or more strings of terminals generated from *a.* If *a* is e or can generate e, then e is also in FIRST *(a).*

The details of how one computes FIRST(o;) are in Section 4.4.2. Here, we shall just use ad hoc reasoning to deduce the symbols in FIRST (a); typically, *a* will either begin with a terminal, which is therefore the only symbol in FIRST(Q;), or *a* will begin with a nonterminal whose production bodies begin with terminals, in which case these terminals are the only members of **FIRST**(a).

For example, with respect to the grammar of Fig. **2.16,** the following are correct calculations of FIRST.

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr);    match(';');    break;
        case if:
                match(it);    match(' (');    match (expr);    match(')');    stmtQ;
                break;
        case for:
                match(for);    match(' (');
                optexprQ;    match(';');    optexpr();    match(';');    optexprQ;
                match(')');    stmtQ;    break;
        case other;
                match (other);  break;
        default:
                report ("syntax error");
        }
}

void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match (terminal £) {
        if ( lookahead == £ ) lookahead — next Terminal;
        else report ("syntax error");
}
```

Figure 2.19: Pseudocode for a predictive parser

FIRST(stmt)	=	**{expr, if, for, other}**
FIRST(expr ;)	=	**{expr}**

The **FIRST** sets must be considered if there are two productions *A -» a* and *A —> 0.* Ignoring e-productions for the moment, predictive parsing requires **FlRST**(a) and **FIRST**(/3) to be disjoint. The lookahead symbol can then be used to decide which production to use; if the lookahead symbol is in **FIRST(OJ**), then *a* is used. Otherwise, if the lookahead symbol is in **FIRST**(/3), then ***0*** is used.

2.4.3 When to Use e-Productions

Our predictive parser uses an e-production as a default when no other production can be used. With the input of Fig. 2.18, after the terminals **for** and (are matched, the lookahead symbol is ;. At this point procedure *optexpr* is called, and the code

if (*lookahead* === **expr**) *match* **(expr);**

in its body is executed. Nonterminal *optexpr* has two productions, with bodies **expr** and e. The lookahead symbol ";" does not match the terminal **expr,** so the production with body **expr** cannot apply. In fact, the procedure returns without changing the lookahead symbol or doing anything else. Doing nothing corresponds to applying an e-production.

More generally, consider a variant of the productions in Fig. 2.16 where *optexpr* generates an expression nonterminal instead of the terminal **expr:**

optexpr —>• *expr*
I e

Thus, *optexpr* either generates an expression using nonterminal *expr* or it generates e. While parsing *optexpr,* if the lookahead symbol is not in FIRST (espr), then the e-production is used.

For more on when to use e-productions, see the discussion of LL(1) grammars in Section 4.4.3.

2.4.4 Designing a Predictive Parser

We can generalize the technique introduced informally in Section 2.4.2, to apply to any grammar that has disjoint FIRST sets for the production bodies belonging to any nonterminal. We shall also see that when we have a translation scheme — that is, a grammar with embedded actions — it is possible to execute those actions as part of the procedures designed for the parser.

Recall that a *predictive parser* is a program consisting of a procedure for every nonterminal. The procedure for nonterminal *A* does two things.

1. It decides which A-production to use by examining the lookahead symbol. The production with body *a* (where *a* is not e, the empty string) is used if the lookahead symbol is in FiRST(a). If there is a conflict between two nonempty bodies for any lookahead symbol, then we cannot use this parsing method on this grammar. In addition, the e-production for *A,* if it exists, is used if the lookahead symbol is not in the FIRST set for any other production body for *A.*

2. The procedure then mimics the body of the chosen production. That is, the symbols of the body are "executed" in turn, from the left. A nonterminal is "executed" by a call to the procedure for that nonterminal, and a terminal matching the lookahead symbol is "executed" by reading the next input symbol. If at some point the terminal in the body does not match the lookahead symbol, a syntax error is reported.

Figure 2.19 is the result of applying these rules to the grammar in Fig. 2.16.

Just as a translation scheme is formed by extending a grammar, a syntax-directed translator can be formed by extending a predictive parser. An algorithm for this purpose is given in Section 5.4. The following limited construction suffices for the present:

1. Construct a predictive parser, ignoring the actions in productions.
2. Copy the actions from the translation scheme into the parser. If an action appears after grammar symbol *X* in production *p*, then it is copied after the implementation of *X* in the code for *p*. Otherwise, if it appears at the beginning of the production, then it is copied just before the code for the production body.

We shall construct such a translator in Section 2.5.

2.4.5 Left Recursion

It is possible for a recursive-descent parser to loop forever. A problem arises with "left-recursive" productions like

expr —y expr + term

where the leftmost symbol of the body is the same as the nonterminal at the head of the production. Suppose the procedure for *expr* decides to apply this production. The body begins with *expr* so the procedure for *expr* is called recursively. Since the lookahead symbol changes only when a terminal in the body is matched, no change to the input took place between recursive calls of *expr*. As a result, the second call to *expr* does exactly what the first call did, which means a third call to *expr*, and so on, forever.

A left-recursive production can be eliminated by rewriting the offending production. Consider a nonterminal *A* with two productions

A -> Aa | /3

where *a* and */3* are sequences of terminals and nonterminals that do not start with *A*. For example, in

expr —y expr + term \ term

nonterminal *A — expr*, string *a = + term*, and string *(3 = term*.

The nonterminal *A* and its production are said to be *left recursive*, because the production *A -> Aa* has *A* itself as the leftmost symbol on the right side.[4] Repeated application of this production builds up a sequence of a's to the right of *A*, as in Fig. 2.20(a). When *A* is finally replaced by */3*, we have a */3* followed by a sequence of zero or more a's.

The same effect can be achieved, as in Fig. 2.20(b), by rewriting the productions for *A* in the following manner, using a new nonterminal *R:*

[4] In a general left-recursive grammar, instead of a production *A A a* , the nonterminal *A* may derive *Aa* through intermediate productions.

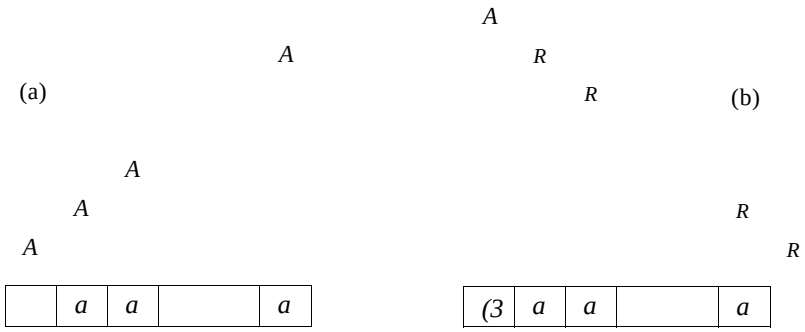


Figure 2.20: Left- and right-recursive ways of generating a string

$$A \rightarrow /3R$$
$$R \rightarrow \bullet\ aR \mid e$$

Nonterminal R and its production R -» aR are *recursive* because this production for R has R itself as the last symbol on the right side. Right-recursive productions lead to trees that grow down towards the right, as in Fig. 2.20(b). Trees growing down to the right make it harder to translate expressions containing left-associative operators, such as minus. In Section 2.5.2, however, we shall see that the proper translation of expressions into postfix notation can still be attained by a careful design of the translation scheme.

In Section 4.3.3, we shall consider more general forms of left recursion and show how all left recursion can be eliminated from a grammar.

2.4.6 Exercises for Section 2.4

Exercise **2.4.1**: Construct recursive-descent parsers, starting with the following grammars:

a) $S \rightarrow + SS \mid -SS \mid \mathbf{a}$

b) $S \rightarrow 5\ (\ 5\ )\ 5 \mid e$

c) $S \quad 0\ 5\ 1 \mid 0\ 1$

2.5 A Translator for Simple Expressions

Using the techniques of the last three sections, we now construct a syntax-directed translator, in the form of a working Java program, that translates arithmetic expressions into postfix form. To keep the initial program manageably small, we start with expressions consisting of digits separated by binary plus and minus signs. We extend the program in Section 2.6 to translate expressions that include numbers and other operators. It is worth studying the

translation of expressions in detail, since they appear as a construct in so many languages.

A syntax-directed translation scheme often serves as the specification for a translator. The scheme in Fig. 2.21 (repeated from Fig. 2.15) defines the translation to be performed here.

```
expr  ->  expr + term   { print('+') }
      |   expr - term   { print('-') }
      |   term

term  ->  0             { print('O') }
      |   1             { print('l') }

      |   9             { print('9') }
```

Figure 2.21: Actions for translating into postfix notation

Often, the underlying grammar of a given scheme has to be modified before it can be parsed with a predictive parser. In particular, the grammar underlying the scheme in Fig. 2.21 is left recursive, and as we saw in the last section, a predictive parser cannot handle a left-recursive grammar.

We appear to have a conflict: on the one hand we need a grammar that facilitates translation, on the other hand we need a significantly different grammar that facilitates parsing. The solution is to begin with the grammar for easy translation and carefully transform it to facilitate parsing. By eliminating the left recursion in Fig. 2.21, we can obtain a grammar suitable for use in a predictive recursive-descent translator.

2.5.1 Abstract and Concrete Syntax

A useful starting point for designing a translator is a data structure called an abstract syntax tree. In an *abstract syntax tree* for an expression, each interior node represents an operator; the children of the node represent the operands of the operator. More generally, any programming construct can be handled by making up an operator for the construct and treating as operands the semantically meaningful components of that construct.

In the abstract syntax tree for `9-5+2` in Fig. 2.22, the root represents the operator +. The subtrees of the root represent the subexpressions **9 - 5** and **2.** The grouping of **9 - 5** as an operand reflects the left-to-right evaluation of operators at the same precedence level. Since - and + have the same precedence, `9-5+2` is equivalent to `(9-5)+2`.

Abstract syntax trees, or simply *syntax trees,* resemble parse trees to an extent. However, in the syntax tree, interior nodes represent programming constructs while in the parse tree, the interior nodes represent nonterminals. Many nonterminals of a grammar represent programming constructs, but others

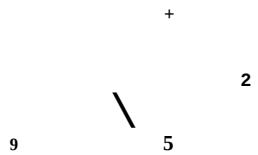


Figure 2.22: Syntax tree for 9-5+2

are "helpers" of one sort of another, such as those representing terms, factors, or other variations of expressions. In the syntax tree, these helpers typically are not needed and are hence dropped. To emphasize the contrast, a parse tree is sometimes called a *concrete syntax tree,* and the underlying grammar is called a *concrete syntax* for the language.

In the syntax tree in Fig. 2.22, each interior node is associated with an operator, with no "helper" nodes for *single productions* (a production whose body consists of a single nonterminal, and nothing else) like *expr —»• term* or for e-productions like *rest* -> e.

It is desirable for a translation scheme to be based on a grammar whose parse trees are as close to syntax trees as possible. The grouping of subexpressions by the grammar in Fig. 2.21 is similar to their grouping in syntax trees. For example, subexpressions of the addition operator are given by *expr* and *term* in the production body *expr+ term.*

2.5.2 Adapting the Translation Scheme

The left-recursion-elimination technique sketched in Fig. 2.20 can also be applied to productions containing semantic actions. First, the technique extends to multiple productions for *A.* In our example, *A* is *expr,* and there are two left-recursive productions for *expr* and one that is not left recursive. The technique transforms the productions *A* ->• *Aa* | *A/3* | 7 into

A -»• 7.R
R *aR\/3R\e*

Second, we need to transform productions that have embedded actions, not just terminals and nonterminals. Semantic actions embedded in the productions are simply carried along in the transformation, as if they were terminals.

Example 2.13: Consider the translation scheme of Fig. 2.21. Let

A *expr*
a + *term* { print('+') }
- *term* { print('-') }
1 *term*

Then the left-recursion-eliminating transformation produces the translation scheme in Fig. **2.23.** The *expr* productions in Fig. **2.21** have been transformed into the productions for *expr,* and a new nonterminal *rest* plays the role of *R.* The productions for *term* are repeated from Fig. **2.21.** Figure **2.24** shows how **9-5+2** is translated using the grammar in Fig. **2.23.** •

```
expr  —y  term rest
rest  —y  + term { print('+') } rest
      |   - term { print('-') } rest
      I   c
term  -»  0 { print('0') }
      |   1 { print('1') }

      |   9 { print('9') }
```

Figure **2.23:** Translation scheme after left-recursion elimination

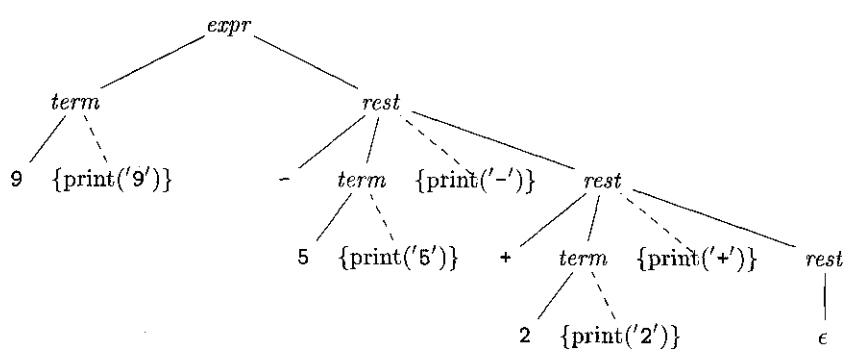


Figure **2.24:** Translation of **9-5+2** to **95-2+**

Left-recursion elimination must be done carefully, to ensure that we preserve the ordering of semantic actions. For example, the transformed scheme in Fig. **2.23** has the actions { print('+') } and { print('-') } in the middle of a production body, in each case between nonterminals *term* and *rest.* If the actions were to be moved to the end, after *rest,* then the translations would become incorrect. We leave it to the reader to show that **9-5+2** would then be translated incorrectly into **952+-,** the postfix notation for **9-(5+2),** instead of the desired **95-2+,** the postfix notation for **(9-5)+2.**

2.5.3 Procedures for the Nonterminals

Functions *expr*, *rest*, and *term* in Fig. 2.25 implement the syntax-directed translation scheme in Fig. 2.23. These functions mimic the production bodies of the corresponding nonterminals. Function *expr* implements the production *expr*-\ *term rest* by the calls *termQ* followed by *restQ*.

```
void exprQ {
        termi);    restQ;
}

void restQ {
        if ( lookahead == '+' ) {
                match('+');    termQ;    print('+');    restQ;
        }
        else if ( lookahead == '-' ) {
                match('-');    termQ;    print('-');    restQ;
        }
        else { } /* do nothing with the input */ ;
}

void termQ {
        if ( lookahead is a digit ) {
                t = lookahead;    match(lookahead);    print(t);
        }
        else report (" syntax error");
}
```

Figure 2.25: Pseudocode for nonterminals *expr*, *rest*, and *term*.

Function *rest* implements the three productions for nonterminal *rest* in Fig. 2.23. It applies the first production if the lookahead symbol is a plus sign, the second production if the lookahead symbol is a minus sign, and the production *rest* e in all other cases. The first two productions for *rest* are implemented by the first two branches of the if-statement in procedure *rest*. If the lookahead symbol is +, the plus sign is matched by the call *match('+')*. After the call *termQ*, the semantic action is implemented by writing a plus character. The second production is similar, with - instead of +. Since the third production for *rest* has e as its right side, the last else-clause in function *rest* does nothing.

The ten productions for *term* generate the ten digits. Since each of these productions generates a digit and prints it, the same code in Fig. 2.25 implements them all. If the test succeeds, variable *t* saves the digit represented by *lookahead* so it can be written after the call to *match*. Note that *match* changes

the lookahead symbol, so the digit needs to be saved for later printing.[5]

2.5.4 Simplifying the Translator

Before showing a complete program, we shall make two simplifying transformations to the code in Fig. 2.25. The simplifications will fold procedure *rest* into procedure *expr*. When expressions with multiple levels of precedence are translated, such simplifications reduce the number of procedures needed.

First, certain recursive calls can be replaced by iterations. When the last statement executed in a procedure body is a recursive call to the same procedure, the call is said to be *tail recursive*. For example, in function *rest*, the calls of *rest()* with lookahead + and - are tail recursive because in each of these branches, the recursive call to *rest is* the last statement executed by the given call of *rest*.

For a procedure without parameters, a tail-recursive call can be replaced simply by a jump to the beginning of the procedure. The code for *rest* can be rewritten as the pseudocode of Fig. 2.26. As long as the lookahead symbol is a plus or a minus sign, procedure *rest* matches the sign, calls *term* to match a digit, and continues the process. Otherwise, it breaks out of while loop and returns from *rest*.

```
void restQ {
    while( true ) {
        if( lookahead == '+' ) {
            match('+'); termQ; print('+'); continue;
        }
        else if ( lookahead —= '-' ) {
            match('-'); termQ; print('-'); continue;
        }
        break ;
    }
}
```

Figure 2.26: Eliminating tail recursion in the procedure *rest* of Fig. 2.25.

Second, the complete Java program will include one more change. Once the tail-recursive calls to *rest* in Fig. 2.25 are replaced by iterations, the only remaining call to *rest* is from within procedure *expr*. The two procedures can therefore be integrated into one, by replacing the call *restQ* by the body of procedure *rest*.

[5] As a minor optimization, we could print before calling ***match*** to avoid the need to save the digit. In general, changing the order of actions and grammar symbols is risky, since it could change what the translation does.

2.5.5 The Complete Program

The complete Java program for our translator appears in Fig. 2.27. The first line of Fig. 2.27, beginning with **import,** provides access to the package j**ava. io** for system input and output. The rest of the code consists of the two classes **Parser** and **Postfix.** Class **Parser** contains variable **lookahead** and functions **Parser, expr, term,** and **match.**

Execution begins with function **main,** which is defined in class **Postfix.** Function **main** creates an instance **parse** of class **Parser** and calls its function **expr** to parse an expression.

The function **Parser,** with the same name as its class, is a *constructor,* it is called automatically when an object of the class is created. Notice from its definition at the beginning of class **Parser** that the constructor **Parser** initializes variable **lookahead** by reading a token. Tokens, consisting of single characters, are supplied by the system input routine **read,** which reads the next character from the input file. Note that **lookahead** is declared to be an integer, rather than a character, to anticipate the fact that additional tokens other than single characters will be introduced in later sections.

Function **expr** is the result of the simplifications discussed in Section 2.5.4; it implements nonterminals *expr* and *rest* in Fig. 2.23. The code for **expir** in Fig. 2.27 calls **term** and then has a while-loop that forever tests whether **lookahead** matches either '+' or '-'. Control exits from this while-loop when it reaches the return statement. Within the loop, the input/output facilities of the **System** class are used to write a character.

Function **term** uses the routine **isDigit** from the Java class **Character** to test if the lookahead symbol is a digit. The routine **isDigit** expects to be applied to a character; however, **lookahead** is declared to be an integer, anticipating future extensions. The construction **(char) lookahead** *casts* or coerces **lookahead** to be a character. In a small change from Fig. 2.25, the semantic action of writing the lookahead character occurs before the call to **match.**

The function **match** checks terminals; it reads the next input terminal if the lookahead symbol is matched and signals an error otherwise by executing

```
throw new Error("syntax error");
```

This code creates a new exception of class Error and supplies it the string **syntax error** as an error message. Java does not require Error exceptions to be declared in a **throws** clause, since they are meant to be used only for abnormal events that should never occur.[6]

[6] Error handling can be streamlined using the exception-handling facilities of Java. One approach is to define a new exception, say **SyntaxError,** that extends the system class **Exception.** Then, throw **SyntaxError** instead of **Error** when an error is detected in either **term** or **match.** Further, handle the exception in **main** by enclosing the call **parse.expr**() within a **try** statement that catches exception **SyntaxError,** writes a message, and terminates. We would need to add a class **SyntaxError** to the program in Fig. 2.27. To complete the extension, in addition to **IOException,** functions **match** and **term** must now declare that they can throw **SyntaxError.** Function **expr,** which calls them, must also declare that it can throw **SyntaxError.**

```
import java.io.*;
class Parser {
    static int lookahead;

    public Parser() throws IOException {
        lookahead = System.in.read();
    }

    void expr() throws IOException {
        term();
        while(true) {
            if( lookahead ==     ) {
                match('+');termO;System,out.write('+' ) ;
            }
            else if( lookahead == '-' ) {
                match('-'); term(); System.out.write('-');
            }
            else return;
        }
    }

    void term() throws IOException {
        if( Character.isDigit((char)lookahead) ) {
            System.out.write((char)lookahead); match(lookahead);
        }
        else throw new Error("syntax error");
    }

    void match(int t) throws IOException {
        if( lookahead == t ) lookahead = System.in.read();
        else throw new Error("syntax error");
    }
>

public class Postfix {
    public static void main(String[] args) throws IOException {
        Parser parse = new ParserO;
        parse.expr();System.out.write('\n');
    >
}
```

Figure 2.27: Java program to translate infix expressions into postfix form

A Few Salient Features of Java

Those unfamiliar with Java may find the following notes on Java helpful in reading the code in Fig. 2.27:

- A class in Java consists of a sequence of variable and function definitions.
- Parentheses enclosing function parameter lists are needed even if there are no parameters; hence we write **`expr()`** and **`term()`**. These functions are actually procedures, because they do not return values, signified by the keyword **`void`** before the function name.
- Functions communicate either by passing parameters "by value" or by accessing shared data. For example, the functions **`expr()`** and **`term()`** examine the lookahead symbol using the class variable **`lookahead`** that they can all access since they all belong to the same class **`Parser`**.
- Like C, Java uses = for assignment, == for equality, and != for inequality.
- The clause **`"throws IOException"`** in the definition of **`term()`** declares that an exception called **`IOException`** can occur. Such an exception occurs if there is no input to be read when the function **`match`** uses the routine **`read`**. Any function that calls **`match`** must also declare that an **`IOException`** can occur during its own execution.

2.6 Lexical Analysis

A lexical analyzer reads characters from the input and groups them into "token objects." Along with a terminal symbol that is used for parsing decisions, a token object carries additional information in the form of attribute values. So far, there has been no need to distinguish between the terms "token" and "terminal," since the parser ignores the attribute values that are carried by a token. In this section, a token is a terminal along with additional information.

A sequence of input characters that comprises a single token is called a *lexeme*. Thus, we can say that the lexical analyzer insulates a parser from the lexeme representation of tokens.

The lexical analyzer in this section allows numbers, identifiers, and "white space" (blanks, tabs, and newlines) to appear within expressions. It can be used to extend the expression translator of the previous section. Since the expression grammar of Fig. 2.21 must be extended to allow numbers and identifiers, we

shall take this opportunity to allow multiplication and division as well. The extended translation scheme appears in Fig. 2.28.

expr	-»	*expr* + *term*	{ print('+') }
	|	*expr* - *term*	{ print('-') }
	|	*term*	
term	-»•	*term* * *factor*	{ print('*') }
	|	*term* I *factor*	{ print('/') }
	|	*factor*	
factor	-»	(*expr*)	
	|	**num**	{ print(**num.** *value)* }
	|	**id**	{ print(**id**.*lexeme)* }

Figure 2.28: Actions for translating into postfix notation

In Fig. 2.28, the terminal **num** is assumed to have an attribute *imm.value,* which gives the integer value corresponding to this occurrence of **num.** Terminal **id** has a string-valued attribute written as **id.***lexeme;* we assume this string is the actual lexeme comprising this instance of the token **id.**

The pseudocode fragments used to illustrate the workings of a lexical analyzer will be assembled into Java code at the end of this section. The approach in this section is suitable for hand-written lexical analyzers. Section 3.5 describes a tool called Lex that generates a lexical analyzer from a specification. Symbol tables or data structures for holding information about identifiers are considered in Section 2.7.

2.6.1 Removal of White Space and Comments

The expression translator in Section 2.5 sees every character in the input, so extraneous characters, such as blanks, will cause it to fail. Most languages allow arbitrary amounts of white space to appear between tokens. Comments are likewise ignored during parsing, so they may also be treated as white space.

If white space is eliminated by the lexical analyzer, the parser will never have to consider it. The alternative of modifying the grammar to incorporate white space into the syntax is not nearly as easy to implement.

The pseudocode in Fig. 2.29 skips white space by reading input characters as long as it sees a blank, a tab, or a newline. Variable *peek* holds the next input character. Line numbers and context are useful within error messages to help pinpoint errors; the code uses variable *line* to count newline characters in the input.

```
for ( ; ; peek = next input character ) {
        if ( peek is a blank or a tab ) do nothing;
        else if ( peek is a newline ) line = line+1;
        else break;
}
```

Figure 2.29: Skipping white space

2.6.2 Reading Ahead

A lexical analyzer may need to read ahead some characters before it can decide on the token to be returned to the parser. For example, a lexical analyzer for C or Java must read ahead after it sees the character >. If the next character is =, then > is part of the character sequence >=, the lexeme for the token for the "greater than or equal to" operator. Otherwise > itself forms the "greater than" operator, and the lexical analyzer has read one character too many.

A general approach to reading ahead on the input, is to maintain an input buffer from which the lexical analyzer can read and push back characters. Input buffers can be justified on efficiency grounds alone, since fetching a block of characters is usually more efficient than fetching one character at a time. A pointer keeps track of the portion of the input that has been analyzed; pushing back a character is implemented by moving back the pointer. Techniques for input buffering are discussed in Section 3.2.

One-character read-ahead usually suffices, so a simple solution is to use a variable, say *peek,* to hold the next input character. The lexical analyzer in this section reads ahead one character while it collects digits for numbers or characters for identifiers; e.g., it reads past 1 to distinguish between 1 and 10, and it reads past **t** to distinguish between **t** and **true.**

The lexical analyzer reads ahead only when it must. An operator like * can be identified without reading ahead. In such cases, *peek* is set to a blank, which will be skipped when the lexical analyzer is called to find the next token. The invariant assertion in this section is that when the lexical analyzer returns a token, variable *peek* either holds the character beyond the lexeme for the current token, or it holds a blank.

2.6.3 Constants

Anytime a single digit appears in a grammar for expressions, it seems reasonable to allow an arbitrary integer constant in its place. Integer constants can be allowed either by creating a terminal symbol, say **num,** for such constants or by incorporating the syntax of integer constants into the grammar. The job of collecting characters into integers and computing their collective numerical value is generally given to a lexical analyzer, so numbers can be treated as single units during parsing and translation.

When a sequence of digits appears in the input stream, the lexical analyzer passes to the parser a token consisting of the terminal **num** along with an integer-valued attribute computed from the digits. If we write tokens as tuples enclosed between ( ), the input **31 + 28 + 59** is transformed into the sequence

(num, 31) (+) (num, 28) (+) (num, 59)

Here, the terminal symbol + has no attributes, so its tuple is simply (+). The pseudocode in Fig. **2.30** reads the digits in an integer and accumulates the value of the integer using variable *v*.

```
if ( peek holds a digit ) {
        v = 0;
        do {
                v = v * 10 + integer value of digit peek;
                peek = next input character;
        } while ( peek holds a digit );
        return token (num, v);
}
```

Figure **2.30:** Grouping digits into integers

2.6.4 Recognizing Keywords and Identifiers

Most languages use fixed character strings such as **for, do,** and **if,** as punctuation marks or to identify constructs. Such character strings are called *keywords.*

Character strings are also used as identifiers to name variables, arrays, functions, and the like. Grammars routinely treat identifiers as terminals to simplify the parser, which can then expect the same terminal, say **id,** each time any identifier appears in the input. For example, on input

count = count + increment; **(2.6)**

the parser works with the terminal stream **id = id + id.** The token for **id** has an attribute that holds the lexeme. Writing tokens as tuples, we see that the tuples for the input stream **(2.6)** are

(id, "count") (=) (id, "count") (+) (id, "increment") (;)

Keywords generally satisfy the rules for forming identifiers, so a mechanism is needed for deciding when a lexeme forms a keyword and when it forms an identifier. The problem is easier to resolve if keywords are *reserved;* i.e., if they cannot be used as identifiers. Then, a character string forms an identifier only if it is not a keyword.

The lexical analyzer in this section solves two problems by using a table to hold character strings:

- *Single Representation. A* string table can insulate the rest of the compiler from the representation of strings, since the phases of the compiler can work with references or pointers to the string in the table. References can also be manipulated more efficiently than the strings themselves.

- *Reserved Words.* Reserved words can be implemented by initializing the string table with the reserved strings and their tokens. When the lexical analyzer reads a string or lexeme that could form an identifier, it first checks whether the lexeme is in the string table. If so, it returns the token from the table; otherwise, it returns a token with terminal **id.**

In Java, a string table can be implemented as a hash table using a class called *Hashtable.* The declaration

Hashtable words = **new** *HashtableQ;*

sets up *words* as a default hash table that maps keys to values. We shall use it to map lexemes to tokens. The pseudocode in Fig. 2.31 uses the operation *get* to look up reserved words.

```
if ( peek holds a letter ) {
        collect letters or digits into a buffer 6;
        s = string formed from the characters in 6;
        w = token returned by words.get(s);
        if ( w is not null) return w;
        else {
                Enter the key-value pair (s, (id, s)) into words
                return token (id, s);
        }
}
```

Figure 2.31: Distinguishing keywords from identifiers

This pseudocode collects from the input a string *s* consisting of letters and digits beginning with a letter. We assume that s is made as long as possible; i.e., the lexical analyzer will continue reading from the input as long as it encounters letters and digits. When something other than a letter or digit, e.g., white space, is encountered, the lexeme is copied into a buffer 6. If the table has an entry for *s*, then the token retrieved by *words.get* is returned. Here, *s* could be either a keyword, with which the *words* table was initially seeded, or it could be an identifier that was previously entered into the table. Otherwise, token **id** and attribute *s* are installed in the table and returned.

2.6.5 A Lexical Analyzer

The pseudocode fragments so far in this section fit together to form a function *scan* that returns token objects, as follows:

```
Token scanQ {
        skip white space, as in Section 2.6.1;
        handle numbers, as in Section 2.6.3;
        handle reserved words and identifiers, as in Section 2.6.4;
        /* if we get here, treat read-ahead character peek as a token */
        Token t = new Token(peek);
        peek = blank /* initialization, as discussed in Section 2.6.2 */ ;
        return i;
}
```

The rest of this section implements function *scan* as part of a Java package for lexical analysis. The package, called lexer has classes for tokens and a class Lexer containing function scan.

The classes for tokens and their fields are illustrated in Fig. 2.32; their methods are not shown. Class Token has a field tag that is used for parsing decisions. Subclass Num adds a field value for an integer value. Subclass Word adds a field lexeme that is used for reserved words and identifiers.

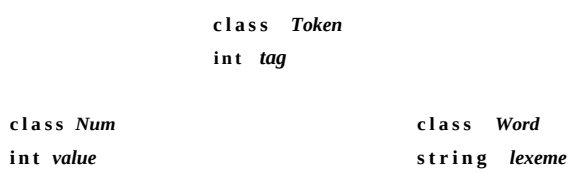


Figure 2.32: Class *Token* and subclasses *Num* and *Word*

Each class is in a file by itself. The file for class Token is as follows:

```
1) package lexer;                    // File Token.java
2) public class Token {
3)     public final int tag;
4)     public Token(int t) { tag = t; }
5) >
```

Line 1 identifies the package lexer. Field tag is declared on line 3 to be final so it cannot be changed once it is set. The constructor Token on line 4 is used to create token objects, as in

```
new Token('+')
```

which creates a new object of class Token and sets its field tag to an integer representation of '+'. (For brevity, we omit the customary method toString, which would return a string suitable for printing.)

Where the pseudocode had terminals like num and id, the Java code uses integer constants. Class Tag implements such constants:

```
package lexer;                    // File Tag.java
public class Tag {
    public final static int
        NUM = 256, ID = 257, TRUE = 258, FALSE = 259;
>
```

In addition to the integer-valued fields NUM and ID, this class defines two additional fields, TRUE and FALSE, for future use; they will be used to illustrate the treatment of reserved keywords.[7]

The fields in class Tag are public, so they can be used outside the package. They are static, so there is just one instance or copy of these fields. The fields are final, so they can be set just once. In effect, these fields represent constants. A similar effect is achieved in C by using define-statements to allow names such as NUM to be used as symbolic constants, e.g.:

```
#define NUM 256
```

The Java code refers to Tag. NUM and Tag. ID in places where the pseudocode referred to terminals num and id. The only requirement is that Tag.NUM and Tag. ID must be initialized with distinct values that differ from each other and from the constants representing single-character tokens, such as '+' or '*'.

```
1) package lexer;                    // File Num.Java
2) public class Num extends Token {
3)     public final int value;
4)     public Num(int v) { super(Tag.NUM); value = v; }
5) >
```

```
1) package lexer;                    // File Word.java
2) public class Word extends Token {
3)     public final String lexeme;
4)     public WordQnt t, String s) {
5)         super(t); lexeme = new String(s);
6)     >
7) >
```

Figure 2.33: Subclasses Num and Word of Token

Classes Num and Word appear in Fig. 2.33. Class Num extends Token by declaring an integer field value on line 3. The constructor Num on line 4 calls super (Tag. NUM), which sets field tag in the superclass Token to Tag. NUM.

[7] ASCII characters are typically converted into integers between 0 and 255. We therefore use integers greater than 255 for terminals.

```
 1) package lexer;                    // File Lexer.Java
 2) import java.io.*; import java.util.*;
 3) public class Lexer {
 4)    public int line = 1;
 5)    private char peek = ' ';
 6)    private Hashtable words = new HashtableO;
 7)    void reserve(Word t) { words.put(t.lexeme, t); }
 8)    public Lexer() {
 9)       reserve( new Word(Tag.TRUE,  "true")  );
10)       reserve( new Word(Tag.FALSE, "false") );
11)    >
12)    public Token scan() throws IOException {
13)       for( ; ; peek = (char)System.in.read() ) {
14)          if( peek == ' ' II peek == '\t' ) continue;
15)          else if( peek == '\n' ) line = line + 1;
16)          else break;
17)       }
          /* continues in Fig. 2.35 */
```

Figure **2.34:** Code for a lexical analyzer, part **1** of **2**

Class **Word** is used for both reserved words and identifiers, so the constructor **Word** on line **4** expects two parameters: a lexeme and a corresponding integer value for **tag.** An object for the reserved word **true** can be created by executing

```
new Word(Tag.TRUE, "true")
```

which creates a new object with field **tag** set to **Tag.TRUE** and field **lexeme** set to the string **"true".**

Class **Lexer** for lexical analysis appears in Figs. **2.34** and **2.35.** The integer variable **line** on line **4** counts input lines, and character variable **peek** on line **5** holds the next input character.

Reserved words are handled on lines **6** through **11.** The table **words** is declared on line **6.** The helper function **reserve** on line **7** puts a string-word pair in the table. Lines **9** and **10** in the constructor **Lexer** initialize the table. They use the constructor **Word** to create word objects, which are passed to the helper function **reserve.** The table is therefore initialized with reserved words **"true"** and **"false"** before the first call of **scan.**

The code for **scan** in Fig. **2.34-2.35** implements the pseudocode fragments in this section. The for-statement on lines **13** through **17** skips blank, tab, and newline characters. Control leaves the for-statement with **peek** holding a non-white-space character.

The code for reading a sequence of digits is on lines **18** through **25.** The function **isDigit** is from the built-in Java class **Character.** It is used on line **18** to check whether **peek** is a digit. If so, the code on lines **19** through **24**

```
18)         if( Character.isDigit(peek) ) {
19)            int v = 0;
20)            do {
21)               v = 10*v + Character.digit(peek, 10);
22)               peek = (char)System.in.read();
23)            } while( Character.isDigit(peek) );
24)            return new Num(v);
25)         }
26)         if( Character.isLetter(peek) ) {
27)            StringBuffer b = new StringBuffer();
28)            do {
29)               b.append(peek);
30)               peek = (char)System.in.read();
31)            } while( Character.isLetterOrDigit(peek) );
32)            String s = b.toStringO ;
33)            Word w = (Word)words.get(s);
34)            if( w != null ) return w;
35)            w = new Word(Tag.ID, s);
36)            words.put(s, w);
37)            return w;
38)         }
39)         Token t = new Token(peek);
40)         peek = ' ';
41)         return t;
42)      }
43)   }
```

Figure **2.35:** Code for a lexical analyzer, part **2** of **2**

accumulates the integer value of the sequence of digits in the input and returns a new Num object.

Lines **26** through **38** analyze reserved words and identifiers. Keywords **true** and **false** have already been reserved on lines **9** and **10.** Therefore, line **35** is reached if string **s** is not reserved, so it must be the lexeme for an identifier. Line **35** therefore returns a new word object with **lexeme** set to **s** and **tag** set to **Tag. ID.** Finally, lines **39** through **41** return the current character as a token and set **peek** to a blank that will be stripped the next time **scan** is called.

2.6.6 Exercises for Section 2.6

Exercise 2.6.1: Extend the lexical analyzer in Section **2.6.5** to remove comments, defined as follows:

a) A comment begins with // and includes all characters until the end of that line.

b) A comment begins with /* and includes all characters through the next occurrence of the character sequence */.

Exercise **2.6.2**: Extend the lexical analyzer in Section 2.6.5 to recognize the relational operators <, <=, ==, !=, >=, >.

Exercise **2.6.3**: Extend the lexical analyzer in Section 2.6.5 to recognize floating point numbers such as 2., 3.14, and .5.

2.7 Symbol Tables

Symbol tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phases of a compiler and used by the synthesis phases to generate the target code. Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program.

From Section 1.6.1, the scope of a declaration is the portion of a program to which the declaration applies. We shall implement scopes by setting up a separate symbol table for each scope. A program block with declarations[8] will have its own symbol table with an entry for each declaration in the block. This approach also works for other constructs that set up scopes; for example, a class would have its own table, with an entry for each field and method.

This section contains a symbol-table module suitable for use with the Java translator fragments in this chapter. The module will be used as is when we put together the translator in Appendix A. Meanwhile, for simplicity, the main example of this section is a stripped-down language with just the key constructs that touch symbol tables; namely, blocks, declarations, and factors. All of the other statement and expression constructs are omitted so we can focus on the symbol-table operations. A program consists of blocks with optional declarations and "statements" consisting of single identifiers. Each such statement represents a use of the identifier. Here is a sample program in this language:

$$\{\ \text{int x; char } \mathbf{y;}\ \{\ \text{bool } \mathbf{y;}\ \text{x; } \mathbf{y;}\ \}\ \text{x; } \mathbf{y;}\ \mathbf{\}} \tag{2.7}$$

The examples of block structure in Section 1.6.3 dealt with the definitions and uses of names; the input (2.7) consists solely of definitions and uses of names.

The task we shall perform is to print a revised program, in which the declarations have been removed and each "statement" has its identifier followed by a colon and its type.

[8] In C, for instance, program blocks are either functions or sections of functions that are separated by curly braces and that have one or more declarations within them.

Who Creates Symbol-Table Entries?

Symbol-table entries are created and used during the analysis phase by the lexical analyzer, the parser, and the semantic analyzer. In this chapter, we have the parser create entries. With its knowledge of the syntactic structure of a program, a parser is often in a better position than the lexical analyzer to distinguish among different declarations of an identifier.

In some cases, a lexical analyzer can create a symbol-table entry as soon as it sees the characters that make up a lexeme. More often, the lexical analyzer can only return to the parser a token, say **id,** along with a pointer to the lexeme. Only the parser, however, can decide whether to use a previously created symbol-table entry or create a new one for the identifier.

Example 2.14: On the above input (2.7), the goal is to produce:

```
{ { x:int; yrbool; } x:int; y:char; }
```

The first x and y are from the inner block of input (2.7). Since this use of x refers to the declaration of x in the outer block, it is followed by int, the type of that declaration. The use of y in the inner block refers to the declaration of y in that very block and therefore has boolean type. We also see the uses of x and y in the outer block, with their types, as given by declarations of the outer block: integer and character, respectively. •

2.7.1 Symbol Table Per Scope

The term "scope of identifier really refers to the scope of a particular declaration of *x*. The term *scope* by itself refers to a portion of a program that is the scope of one or more declarations.

Scopes are important, because the same identifier can be declared for different purposes in different parts of a program. Common names like i and x often have multiple uses. As another example, subclasses can redeclare a method name to override a method in a superclass.

If blocks can be nested, several declarations of the same identifier can appear within a single block. The following syntax results in nested blocks when *stmts* can generate a block:

block -> '{' decls stmts '}'

(We quote curly braces in the syntax to distinguish them from curly braces for semantic actions.) With the grammar in Fig. 2.38, *decls* generates an optional sequence of declarations and *stmts* generates an optional sequence of statements.

Optimization of Symbol Tables for Blocks

Implementations of symbol tables for blocks can take advantage of the most-closely nested rule. Nesting ensures that the chain of applicable symbol tables forms a stack. At the top of the stack is the table for the current block. Below it in the stack are the tables for the enclosing blocks. Thus, symbol tables can be allocated and deallocated in a stack-like fashion.

Some compilers maintain a single hash table of accessible entries; that is, of entries that are not hidden by a declaration in a nested block. Such a hash table supports essentially constant-time lookups, at the expense of inserting and deleting entries on block entry and exit. Upon exit from a block *B*, the compiler must undo any changes to the hash table due to declarations in block *B*. It can do so by using an auxiliary stack to keep track of changes to the hash table while block *B* is processed.

Moreover, a statement can be a block, so our language allows nested blocks, where an identifier can be redeclared.

The *most-closely nested* rule for blocks is that an identifier x is in the scope of the most-closely nested declaration of x; that is, the declaration of x found by examining blocks inside-out, starting with the block in which x appears.

Example 2.15: The following pseudocode uses subscripts to distinguish a¬ mong distinct declarations of the same identifier:

```
1)  {    int xi;  int y3;
2)       {    int w2;  bool y2;  int z2;
3)            • • • w2 ••;••• xi          y2          z2   •••;
4)       }
5)        • • • w0 • • •;  ••• xi      ••• yi • • •;
6)  }
```

The subscript is not part of an identifier; it is in fact the line number of the declaration that applies to the identifier. Thus, all occurrences of x are within the scope of the declaration on line 1. The occurrence of y on line 3 is in the scope of the declaration of y on line 2 since y is redeclared within the inner block. The occurrence of y on line 5, however, is within the scope of the declaration of y on line 1.

The occurrence of w on line 5 is presumably within the scope of a declaration of w outside this program fragment; its subscript 0 denotes a declaration that is global or external to this block.

Finally, z is declared and used within the nested block, but cannot be used on line 5, since the nested declaration applies only to the nested block. •

The most-closely nested rule for blocks can be implemented by chaining symbol tables. That is, the table for a nested block points to the table for its enclosing block.

Example 2.16: Figure 2.36 shows symbol tables for the pseudocode in Example 2.15. *B*\ is for the block starting on line 1 and is for the block starting at line 2. At the top of the figure is an additional symbol table *Bo* for any global or default declarations provided by the language. During the time that we are analyzing lines 2 through 4, the environment is represented by a reference to the lowest symbol table — the one for *B%.* When we move to line 5, the symbol table for *B*^ becomes inaccessible, and the environment refers instead to the symbol table for *B*\, from which we can reach the global symbol table, but not the table for B_2. •

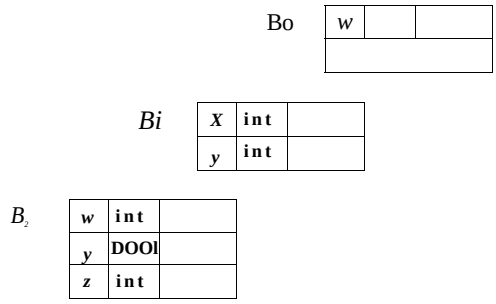


Figure 2.36: Chained symbol tables for Example 2.15

The Java implementation of chained symbol tables in Fig. 2.37 defines a class Env, short for *environment.*[9] Class Env supports three operations:

- *Create a new symbol table.* The constructor Env(p) on lines 6 through 8 of Fig. 2.37 creates an Env object with a hash table named table. The object is chained to the environment-valued parameter p by setting field next to p. Although it is the Env objects that form a chain, it is convenient to talk of the tables being chained.

- *Put* a new entry in the current table. The hash table holds key-value pairs, where:

  — The *key* is a string, or rather a reference to a string. We could alternatively use references to token objects for identifiers as keys.

  - The *value* is an entry of class Symbol. The code on lines 9 through 11 does not need to know the structure of an entry; that is, the code is independent of the fields and methods in class Symbol.

[9] "Environment" is another term for the collection of symbol tables that are relevant at a point in the program.

```
1) package symbols;                    // File Env.java
2) import java.util.*;
3) public class Env {
4)    private Hashtable table;
5)    protected Env prev;
6)    public Env(Env p) {
7)       table = new HashtableO; prev = p;
8)    >
9)    public void put(String s, Symbol sym) {
10)      table.put(s, sym);
11)   >
12)   public Symbol get(String s) {
13)      for( Env e = this; e != null; e = e.prev ) {
14)         Symbol found = (Symbol)(e.table.get(s));
15)         if( found != null ) return found;
16)      }
17)      return null;
18)   }
19) }
```

Figure 2.37: Class *Env* implements chained symbol tables

- *Get* an entry for an identifier by searching the chain of tables, starting with the table for the current block. The code for this operation on lines 12 through 18 returns either a symbol-table entry or null.

Chaining of symbol tables results in a tree structure, since more than one block can be nested inside an enclosing block. The dotted lines in Fig. 2.36 are a reminder that chained symbol tables can form a tree.

2.7.2 The Use of Symbol Tables

In effect, the role of a symbol table is to pass information from declarations to uses. A semantic action "puts" information about identifier *x* into the symbol table, when the declaration of *x* is analyzed. Subsequently, a semantic action associated with a production such as *factor* -> **id** "gets" information about the identifier from the symbol table. Since the translation of an expression *Ei* **op** E_2, for a typical operator **op,** depends only on the translations of *E*\ and E_2, and does not directly depend on the symbol table, we can add any number of operators without changing the basic flow of information from declarations to uses, through the symbol table.

Example 2.17: The translation scheme in Fig. 2.38 illustrates how class *Env* can be used. The translation scheme concentrates on scopes, declarations, and

uses. It implements the translation described in Example 2.14. As noted earlier, on input

```
program  ->•                          { top = null; }
            block

  block  -»                           { saved = top;
                                        top = new Env(top);
                                        print("{ "); }
            decls stmts '}'           { top = saved;
                                        printO'} "); }

  decls  —>  decls decl
         I   e

   decl  -»•  type id ;               { s = new Symbol;
                                        s.type = type, lexeme
                                        top.put (id. lexeme, s); }

  stmts  -»   stmts stmt


   stmt  ->•  6/ocA;
         |    factor ;                { print(";"); }

 factor  ->   id                      { ' — top.get(id.lexeme);
                                        print (id. lexeme);
                                        print(":"); }
                                        print (s.type);
```

Figure 2.38: The use of symbol tables for translating a language with blocks

```
{ int x; char y; { bool y; x; y; } x; y; }
```

the translation scheme strips the declarations and produces

```
{ { x:int; y:bool; } x:int; y:char; }
```

Notice that the bodies of the productions have been aligned in Fig. 2.38 so that all the grammar symbols appear in one column, and all the actions in a second column. As a result, components of the body are often spread over several lines.

Now, consider the semantic actions. The translation scheme creates and discards symbol tables upon block entry and exit, respectively. Variable *top* denotes the top table, at the head of a chain of tables. The first production of

the underlying grammar is *program* -> *block*. The semantic action before *block* initializes *top* to **null,** with no entries.

The second production, *block* -» '{' *decls stmts*'}', has actions upon block entry and exit. On block entry, before *decls,* a semantic action saves a reference to the current table using a local variable *saved.* Each use of this production has its own local variable *saved,* distinct from the local variable for any other use of this production. In a recursive-descent parser, *saved* would be local to the procedure for *block.* The treatment of local variables of a recursive function is discussed in Section 7.2. The code

top = **new** *Env (top);*

sets variable *top* to a newly created new table that is chained to the previous value of *top* just before block entry. Variable *top* is an object of class *Env;* the code for the constructor *Env* appears in Fig. 2.37.

On block exit, after '}', a semantic action restores *top* to its value saved on block entry. In effect, the tables form a stack; restoring *top* to its saved value pops the effect of the declarations in the block.[10] Thus, the declarations in the block are not visible outside the block.

A declaration, *decls* -> **type id** results in a new entry for the declared identifier. We assume that tokens **type** and **id** each have an associated attribute, which is the type and lexeme, respectively, of the declared identifier. We shall not go into all the fields of a symbol object *s,* but we assume that there is a field *type* that gives the type of the symbol. We create a new symbol object *s* and assign its type properly by *s.type* — **type.***lexeme*. The complete entry is put into the top symbol table by *top.put(\d.lexeme, s).*

The semantic action in the production *factor* -> **id** uses the symbol table to get the entry for the identifier. The *get* operation searches for the first entry in the chain of tables, starting with *top.* The retrieved entry contains any information needed about the identifier, such as the type of the identifier. •

2.8 Intermediate Code Generation

The front end of a compiler constructs an intermediate representation of the source program from which the back end generates the target program. In this section, we consider intermediate representations for expressions and statements, and give tutorial examples of how to produce such representations.

2.8.1 Two Kinds of Intermediate Representations

As was suggested in Section 2.1 and especially Fig. 2.4, the two most important intermediate representations are:

[10]Instead of explicitly saving and restoring tables, we could alternatively add static operations ***push*** and ***pop*** to class **Env.**

- Trees, including parse trees and (abstract) syntax trees.
- Linear representations, especially "three-address code."

Abstract-syntax trees, or simply syntax trees, were introduced in Section 2.5.1, and in Section 5.3.1 they will be reexamined more formally. During parsing, syntax-tree nodes are created to represent significant programming constructs. As analysis proceeds, information is added to the nodes in the form of attributes associated with the nodes. The choice of attributes depends on the translation to be performed.

Three-address code, on the other hand, is a sequence of elementary program steps, such as the addition of two values. Unlike the tree, there is no hierarchical structure. As we shall see in Chapter 9, we need this representation if we are to do any significant optimization of code. In that case, we break the long sequence of three-address statements that form a program into "basic blocks," which are sequences of statements that are always executed one-after-the-other, with no branching.

In addition to creating an intermediate representation, a compiler front end checks that the source program follows the syntactic and semantic rules of the source language. This checking is called *static checking;* in general "static" means "done by the compiler."[11] Static checking assures that certain kinds of programming errors, including type mismatches, are detected and reported during compilation.

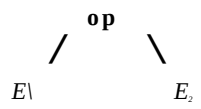
It is possible that a compiler will construct a syntax tree at the same time it emits steps of three-address code. However, it is common for compilers to emit the three-address code while the parser "goes through the motions" of constructing a syntax tree, without actually constructing the complete tree data structure. Rather, the compiler stores nodes and their attributes needed for semantic checking or other purposes, along with the data structure used for parsing. By so doing, those parts of the syntax tree that are needed to construct the three-address code are available when needed, but disappear when no longer needed. We take up the details of this process in Chapter 5.

2.8.2 Construction of Syntax Trees

We shall first give a translation scheme that constructs syntax trees, and later, in Section 2.8.4, show how the scheme can be modified to emit three-address code, along with, or instead of, the syntax tree.

Recall from Section 2.5.1 that the syntax tree

[11] Its opposite, "dynamic," means "while the program is running." Many languages also make certain dynamic checks. For instance, an object-oriented language like Java sometimes must check types during program execution, since the method applied to an object may depend on the particular subclass of the object.



represents an expression formed by applying the operator **op** to the subexpressions represented by E_1 and E_2. Syntax trees can be created for any construct, not just expressions. Each construct is represented by a node, with children for the semantically meaningful components of the construct. For example, the semantically meaningful components of a C while-statement:

while (*expr*) *stmt*

are the expression *expr* and the statement *stmt*.[12] The syntax-tree node for such a while-statement has an operator, which we call **while,** and two children—the syntax trees for the *expr* and the *stmt*.

The translation scheme in Fig. 2.39 constructs syntax trees for a representative, but very limited, language of expressions and statements. All the nonterminals in the translation scheme have an attribute *n*, which is a node of the syntax tree. Nodes are implemented as objects of class *Node*.

Class *Node* has two immediate subclasses: *Expr* for all kinds of expressions, and *Stmt* for all kinds of statements. Each type of statement has a corresponding subclass of *Stmt;* for example, operator **while** corresponds to subclass *While*. A syntax-tree node for operator **while** with children *x* and *y* is created by the pseudocode

new *While (x,y)*

which creates an object of class *While* by calling constructor function *While*, with the same name as the class. Just as constructors correspond to operators, constructor parameters correspond to operands in the abstract syntax.

When we study the detailed code in Appendix A, we shall see how methods are placed where they belong in this hierarchy of classes. In this section, we shall discuss only a few of the methods, informally.

We shall consider each of the productions and rules of Fig. 2.39, in turn. First, the productions defining different types of statements are explained, followed by the productions that define our limited types of expressions.

Syntax Trees for Statements

For each statement construct, we define an operator in the abstract syntax. For constructs that begin with a keyword, we shall use the keyword for the operator. Thus, there is an operator **while** for while-statements and an operator **do** for do-while statements. Conditionals can be handled by defining two operators

[12] The right parenthesis serves only to separate the expression from the statement. The left parenthesis actually has no meaning; it is there only to please the eye, since without it, C would allow unbalanced parentheses.

```
program  -»  block                  return block.n; }

block  —»•  '{'  stmts  '}'         block.n = stmts.n; }

stmts  -y   stmtsi   stmt           stmts.n = new Seq(stmtsi.n, stmt.n)] }
        I e                         stmts.n = null; }

stmt   -»   expr ;                  stmt.n = new Eval(expr.n); }
        |     if ( expr ) stmti
                                    stmt.n = new If(expr.n,stmti.n); }
            while ( expr ) stmti
                                    stmt.n = new While (expr.n, stmti.n); }
            do stmt\ while ( expr );
                                    stmt.n = new Do (stmti.n, expr.n); }
            block                   stmt.n = block.n; }

expr   —»•  rel - expr.             expr.n = new Assign ('=', re/.n, errpr^.n); }
        I   rel                     expr.n = rein; }

rel    ->   rel\ < a                rein = new Rel('<', reli.n, add.n); }
        |   rek <=                  rein = new Rel('<', reli.n, add.n); }
                                    re/.n = add.n; }

add    -» ftrfrfi + ierw            add.n = new 0p('+', addi.n, term.n); }
        |   term                       .n = term.n; }

term   ->  iermi * factor           term.n = new 0p('*',termi.n,factor.n); }
        |   factor                  term.n = factor.n; }

factor     ( ezpr )                 factor.n = expr.n; }
           num                      factor.n = new Num (num.value); }
```

Figure 2.39: Construction of syntax trees for expressions and statements

ifelse and **if** for if-statements with and without an else part, respectively. In our simple example language, we do not use **else,** and so have only an if-statement. Adding **else** presents some parsing issues, which we discuss in Section 4.8.2.

Each statement operator has a corresponding class of the same name, with a capital first letter; e.g., class *If* corresponds to **if.** In addition, we define the subclass *Seq,* which represents a sequence of statements. This subclass corresponds to the nonterminal *stmts* of the grammar. Each of these classes are subclasses of *Stmt,* which in turn is a subclass of *Node.*

The translation scheme in Fig. 2.39 illustrates the construction of syntax-tree nodes. A typical rule is the one for if-statements:

stmt -> **if** (*expr*) *stmti* { *stmt.n* = **new** *If (expr.n, stmt\.n);* }

The meaningful components of the if-statement are *expr* and *stmt\.* The semantic action defines the node *stmt.n* as a new object of subclass *If.* The code for the constructor // is not shown. It creates a new node labeled **if** with the nodes *expr.n* and *stmt\.n* as children.

Expression statements do not begin with a keyword, so we define a new operator **eval** and class *Eval,* which is a subclass of *Stmt,* to represent expressions that are statements. The relevant rule is:

stmt -> *expr* ; { *stmt.n* = **new** *Eval (expr.n);* }

Representing Blocks in Syntax Trees

The remaining statement construct in Fig. 2.39 is the block, consisting of a sequence of statements. Consider the rules:

stmt -» *block* { *stmt.n* = *block.n;* }
block -» '{' *stmts* '}' { *block.n* = *stmts.n;* }

The first says that when a statement is a block, it has the same syntax tree as the block. The second rule says that the syntax tree for nonterminal *block* is simply the syntax tree for the sequence of statements in the block.

For simplicity, the language in Fig. 2.39 does not include declarations. Even when declarations are included in Appendix A, we shall see that the syntax tree for a block is still the syntax tree for the statements in the block. Since information from declarations is incorporated into the symbol table, they are not needed in the syntax tree. Blocks, with or without declarations, therefore appear to be just another statement construct in intermediate code.

A sequence of statements is represented by using a leaf **null** for an empty statement and a operator **seq** for a sequence of statements, as in

stmts -*• *stmtsi stmt* { *stmts.n* = **new** *Seq(stmts,.n, stmt.n);* }

Example 2.18: In Fig. 2.40 we see part of a syntax tree representing a block or statement list. There are two statements in the list, the first an if-statement and the second a while-statement. We do not show the portion of the tree above this statement list, and we show only as a triangle each of the necessary subtrees: two expression trees for the conditions of the if- and while-statements, and two statement trees for their substatements. •

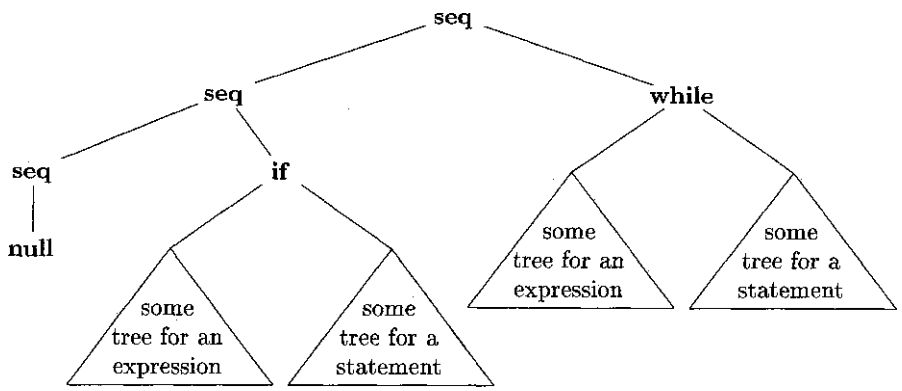


Figure 2.40: Part of a syntax tree for a statement list consisting of an if-statement and a while-statement

Syntax Trees for Expressions

Previously, we handled the higher precedence of * over + by using three nonterminals *expr, term,* and *factor.* The number of nonterminals is precisely one plus the number of levels of precedence in expressions, as we suggested in Section 2.2.6. In Fig. 2.39, we have two comparison operators, < and <= at one precedence level, as well as the usual + and * operators, so we have added one additional nonterminal, called *add.*

Abstract syntax allows us to group "similar" operators to reduce the number of cases and subclasses of nodes in an implementation of expressions. In this chapter, we take "similar" to mean that the type-checking and code-generation rules for the operators are similar. For example, typically the operators + and * can be grouped, since they can be handled in the same way — their requirements regarding the types of operands are the same, and they each result in a single three-address instruction that applies one operator to two values. In general, the grouping of operators in the abstract syntax is based on the needs of the later phases of the compiler. The table in Fig. 2.41 specifies the correspondence between the concrete and abstract syntax for several of the operators of Java.

In the concrete syntax, all operators are left associative, except the assignment operator =, which is right associative. The operators on a line have the

CONCRETE SYNTAX	ABSTRACT SYNTAX
=	**assign**
11	**cond**
&&	**cond**
== !=	**rel**
<= >= >	**rel**
+ -	**op**
* / %	**op**
j	**not**
~ *unary*	**minus**
[]	**access**

Figure 2.41: Concrete and abstract syntax for several Java operators

same precedence; that is, == and != have the same precedence. The lines are in order of increasing precedence; e.g., == has higher precedence than the operators && and =. The subscript *unary* in - *unary* is solely to distinguish a leading unary minus sign, as in -2, from a binary minus sign, as in 2-**a.** The operator [] represents array access, as in **a**[i].

The abstract-syntax column specifies the grouping of operators. The assignment operator = is in a group by itself. The group **cond** contains the conditional boolean operators && and **I I**. The group **rel** contains the relational comparison operators on the lines for == and <. The group **op** contains the arithmetic operators like + and *. Unary minus, boolean negation, and array access are in groups by themselves.

The mapping between concrete and abstract syntax in Fig. 2.41 can be implemented by writing a translation scheme. The productions for nonterminals *expr, rel, add, term,* and *factor* in Fig. 2.39 specify the concrete syntax for a representative subset of the operators in Fig. 2.41. The semantic actions in these productions create syntax-tree nodes. For example, the rule

*term —b termi * factor { term.n* = **new Op(V**,*term\.n,factor.n); }*

creates a node of class *Op,* which implements the operators grouped under **op** in Fig. 2.41. The constructor *Op* has a parameter '*' to identify the actual operator, in addition to the nodes *termy.n* and *factor.n* for the subexpressions.

2.8.3 Static Checking

Static checks are consistency checks that are done during compilation. Not only do they assure that a program can be compiled successfully, but they also have the potential for catching programming errors early, before a program is run. Static checking includes:

- *Syntactic Checking.* There is more to syntax than grammars. For example, constraints such as an identifier being declared at most once in a

scope, or that a break statement must have an enclosing loop or switch statement, are syntactic, although they are not encoded in, or enforced by, a grammar used for parsing.

- *Type Checking.* The type rules of a language assure that an operator or function is applied to the right number and type of operands. If conversion between types is necessary, e.g., when an integer is added to a float, then the type-checker can insert an operator into the syntax tree to represent that conversion. We discuss type conversion, using the common term "coercion," below.

L-values and R-values

We now consider some simple static checks that can be done during the construction of a syntax tree for a source program. In general, complex static checks may need to be done by first constructing an intermediate representation and then analyzing it.

There is a distinction between the meaning of identifiers on the left and right sides of an assignment. In each of the assignments

```
i = 5;
i = i + 1;
```

the right side specifies an integer value, while the left side specifies where the value is to be stored. The terms *l-value* and *r-value* refer to values that are appropriate on the left and right sides of an assignment, respectively. That is, r-values are what we usually think of as "values," while /-values are locations.

Static checking must assure that the left side of an assignment denotes an /-value. An identifier like i has an /-value, as does an array access like **a[2].** But a constant like **2** is not appropriate on the left side of an assignment, since it has an r-value, but not an /-value.

Type Checking

Type checking assures that the type of a construct matches that expected by its context. For example, in the if-statement

if (*expr* **)** *stmt*

the expression *expr* is expected to have type **boolean.**

Type checking rules follow the operator/operand structure of the abstract syntax. Assume the operator **rel** represents relational operators such as <=. The type rule for the operator group **rel** is that its two operands must have the same type, and the result has type boolean. Using attribute *type* for the type of an expression, let *E* consist of **rel** applied to *Ei* and *E*. The type of *E* can be checked when its node is constructed, by executing code like the following:

if (*Ei.type* == E_2*.type*) *E.type* = boolean;
else error;

The idea of matching actual with expected types continues to apply, even in the following situations:

- *Coercions.* A *coercion* occurs if the type of an operand is automatically converted to the type expected by the operator. In an expression like 2 * 3.14, the usual transformation is to convert the integer 2 into an equivalent floating-point number, 2.0, and then perform a floating-point operation on the resulting pair of floating-point operands. The language definition specifies the allowable coercions. For example, the actual rule for **rel** discussed above might be that *Ei.type* and E_2*.type* are convertible to the same type. In that case, it would be legal to compare, say, an integer with a float.

- *Overloading.* The operator + in Java represents addition when applied to integers; it means concatenation when applied to strings. A symbol is said to be *overloaded* if it has different meanings depending on its context. Thus, + is overloaded in Java. The meaning of an overloaded operator is determined by considering the known types of its operands and results. For example, we know that the + in **z** = x + y is concatenation if we know that any of x, y, or **z** is of type string. However, if we also know that another one of these is of type integer, then we have a type error and there is no meaning to this use of +.

2.8.4 Three-Address Code

Once syntax trees are constructed, further analysis and synthesis can be done by evaluating attributes and executing code fragments at nodes in the tree. We illustrate the possibilities by walking syntax trees to generate three-address code. Specifically, we show how to write functions that process the syntax tree and, as a side-effect, emit the necessary three-address code.

Three-Address Instructions

Three-address code is a sequence of instructions of the form

x - *y* **op** *z*

where *x*, *y*, and *z* are names, constants, or compiler-generated temporaries; and **op** stands for an operator.

Arrays will be handled by using the following two variants of instructions:

x [*y*] - *z*
x = *y* [*z 1*

The first puts the value of *z* in the location *x[y],* and the second puts the value of *y[z]* in the location *x.*

Three-address instructions are executed in numerical sequence unless forced to do otherwise by a conditional or unconditional jump. We choose the following instructions for control flow:

`if False` *x* **`goto L`**	if £ is false, next execute the instruction labeled **`L`**
`if` True *x* **`goto L`**	if rc is true, next execute the instruction labeled **`L`**
`goto L`	next execute the instruction labeled **`L`**

A label L can be attached to any instruction by prepending a prefix L:. An instruction can have more than one label.

Finally, we need instructions that copy a value. The following three-address instruction copies the value of **y** into x:

$$x = y$$

Translation of Statements

Statements are translated into three-address code by using jump instructions to implement the flow of control through the statement. The layout in Fig. 2.42 illustrates the translation of **if** *expr* **then** *stmti.* The jump instruction in the layout

`if False` *x* **`goto`** *after*

jumps over the translation of *stmti* if *expr* evaluates to **false.** Other statement constructs are similarly translated using appropriate jumps around the code for their components.

code to compute
expr into *x*

`if False` *x* **`goto`** *after*

code for *stmti*

after—*- ;

Figure 2.42: Code layout for if-statements

For concreteness, we show the pseudocode for class *If* in Fig. 2.43. Class *If is* a subclass of *Stmt,* as are the classes for the other statement constructs. Each subclass of *Stmt* has a constructor — // in this case — and a function *gen* that is called to generate three-address code for this kind of statement.

```
class // extends Stmt {
        Expr E; Stmt S;
        public If(Expr x, Stmt y) { E = x; S = y; after = newlabelQ; }
        public void gen() {
                Expr n = E.rvalueQ;
                emit( "ifFalse " + n.toStringQ + " goto " + after);
                S.genQ;
                emit (after + ":");
        }
```

Figure 2.43: Function *gen* in class //generates three-address code

The constructor // in Fig. 2.43 creates syntax-tree nodes for if-statements. It is called with two parameters, an expression node *x* and a statement node *y*, which it saves as attributes *E* and 5. The constructor also assigns attribute *after* a unique new label, by calling function *newlabelQ*. The label will be used according to the layout in Fig. 2.42.

Once the entire syntax tree for a source program is constructed, the function *gen* is called at the root of the syntax tree. Since a program is a block in our simple language, the root of the syntax tree represents the sequence of statements in the block. All statement classes contain a function *gen*.

The pseudocode for function *gen* of class //in Fig. 2.43 is representative. It calls *E.rvalueQ* to translate the expression *E* (the boolean-valued expression that is part of the if-statements) and saves the result node returned by *E*. Translation of expressions will be discussed shortly. Function *gen* then emits a conditional jump and calls *S.genQ* to translate the substatement *S*.

Translation of Expressions

We now illustrate the translation of expressions by considering expressions containing binary operators **op,** array accesses, and assignments, in addition to constants and identifiers. For simplicity, in an array access *y[z],* we require that *y* be an identifier.[13] For a detailed discussion of intermediate code generation for expressions, see Section 6.4.

We shall take the simple approach of generating one three-address instruction for each operator node in the syntax tree for an expression. No code is generated for identifiers and constants, since they can appear as addresses in instructions. If a node *x* of class *Expr* has operator **op,** then an instruction is emitted to compute the value at node *x* into a compiler generated "temporary" name, say *t*. Thus, i-j+k translates into two instructions

[13]This simple language supports **a[a[n]],** but not **a**[m] **[n]**. Note that **a[a[n]]** has the form **a** [£?], where ***E*** is **a [n].**

```
t1 = i - j
t2 = t1 + k
```

With array accesses and assignments comes the need to distinguish between /-values and r-values. For example, 2***a**[i] can be translated by computing the r-value of **a**[i] into a temporary, as in

```
t1 = a [ i ]
t2 = 2 * t1
```

But, we cannot simply use a temporary in place of **a**[i], if **a**[i] appears on the left side of an assignment.

The simple approach uses the two functions *lvalue* and *rvalue,* which appear in Fig. 2.44 and 2.45, respectively. When function *rvalue* is applied to a nonleaf node *x,* it generates instructions to compute *x* into a temporary, and returns a new node representing the temporary. When function *lvalue* is applied to a nonleaf, it also generates instructions to compute the subtrees below *x,* and returns a node representing the "address" for *x.*

We describe function *lvalue* first, since it has fewer cases. When applied to a node *x,* function *lvalue* simply returns *x* if it is the node for an identifier (i.e., if *x* is of class *Id).* In our simple language, the only other case where an expression has an /-value occurs when *x* represents an array access, such as **a**[i]. In this case, *x* will have the form *Access(y, z),* where class *Access* is a subclass of *Expr, y* represents the name of the accessed array, and *z* represents the offset (index) of the chosen element in that array. From the pseudo-code in Fig. 2.44, function *lvalue* calls *rvalue(z)* to generate instructions, if needed, to compute the r-value of *z.* It then constructs and returns a new *Access* node with children for the array name *y* and the r-value of *z.*

```
Expr lvalue(x : Expr) {
        if ( x is an Id node ) return x;
        else if ( x is an Access (y, z) node and y is an Id node ) {
                return new Access(y, rvalue(z));
        }
        else error;
}
```

Figure 2.44: Pseudocode for function *lvalue*

Example 2.19: When node *x* represents the array access **a**[2***k**],* the call *lvalue(x)* generates an instruction

```
t = 2 * k
```

and returns a new node *x'* representing the /-value **a**[t], where t is a new temporary name.

In detail, the code fragment

return new *Access(y, rvalue(z));*

is reached with *y* being the node for **a** and *z* being the node for expression 2*k. The cail *rvalue(z)* generates code for the expression 2*k (i.e., the three-address statement t = 2 * k) and returns the new node *z'* representing the temporary name t. That node *z'* becomes the value of the second field in the new *Access* node *x'* that is created. •

```
Expr rvalue(x : Expr) {
        if ( x is an Id or a Constant node ) return x;
        else if ( x is an Op(op,y,z) or a Rel(op,y,z) node ) {
                t = new temporary;
                emit string for t = rvalue(y) op rvalue(z);
                return a new node for t;
        }
        else if ( x is an Access (y,z) node ) {
                t — new temporary;
                call lvalue(x), which returns Access(y, z');
                emit string for t = Access(y, z')\
                return a new node for t;
        }
        else if ( x is an Assign (y, z) node ) {
                z' = rvalue(z)]
                emit string for lvalue(y) = z';
                return z';
        }
}
```

Figure 2.45: Pseudocode for function *rvalue*

Function *rvalue* in Fig. 2.45 generates instructions and returns a possibly new node. When *x* represents an identifier or a constant, *rvalue* returns *x* itself. In all other cases, it returns an *Id* node for a new temporary *t*. The cases are as follows:

- When *x* represents *y* **op** *z*, the code first computes *y'* = *rvalue(y)* and *z'* = *rvalue(z)*. It creates a new temporary *t* and generates an instruction *t* = *y'* **op** *z'* (more precisely, an instruction formed from the string representations of *t*, *y'*, **op,** and *z'*). It returns a node for identifier *t*.

- When *x* represents an array access *ylz]*, we can reuse function *lvalue*. The call *lvalue(x)* returns an access *y* **[2 ']** , where *z'* represents an identifier holding the offset for the array access. The code creates a new temporary *t*, generates an instruction based on *t* = *y tz'l*, and returns a node for *t*.

- When ***x*** represents *y = z*, then the code first computes *z' = rvalue(z).* It generates an instruction based on *lvalue(y) = z'* and returns the node *z'*.

Example 2.20: When applied to the syntax tree for

a[i] = 2*a[j-k]

function *rvalue* generates

```
t3 = j - k
t2 = a [ t3 ]
tl = 2 * t2
a [ i ] = tl
```

That is, the root is an *Assign* node with first argument **a[i]** and second argument **2*a**[j-k] . Thus, the third case applies, and function *rvalue* recursively evaluates **2*a**[j-k]. The root of this subtree is the *Op* node for *, which causes a new temporary **tl** to be created, before the left operand, **2** is evaluated, and then the right operand. The constant **2** generates no three-address code, and its r-value is returned as a *Constant* node with value 2.

The right operand **a**[j-k] is an *Access* node, which causes a new temporary **t2** to be created, before function *lvalue* is called on this node. Recursively, *rvalue* is called on the expression j-k. As a side-effect of this call, the three-address statement **t3** = j - k is generated, after the new temporary **t3** is created. Then, returning to the call of *lvalue* on **a**[j-k], the temporary **t2** is assigned the r-value of the entire access-expression, that is, **t2 = a [t3].**

Now, we return to the call of *rvalue* on the *Op* node **2*a**[j-k], which earlier created temporary **tl.** A three-address statement **11 = 2 * 12** is generated as a side-effect, to evaluate this multiplication-expression. Last, the call to *rvalue* on the whole expression completes by calling *lvalue* on the left side **a[i]** and then generating a three-address instruction **a [i] = t l ,** in which the right side of the assignment is assigned to the left side. •

Better Code for Expressions

We can improve on function *rvalue* in Fig. 2.45 and generate fewer three-address instructions, in several ways:

- Reduce the number of copy instructions in a subsequent optimization phase. For example, the pair of instructions **t = i+1** and **i = t** can be combined into **i = i+1,** if there are no subsequent uses of **t.**

- Generate fewer instructions in the first place by taking context into account. For example, if the left side of a three-address assignment is an array access **a[t],** then the right side must be a name, a constant, or a temporary, all of which use just one address. But if the left side is a name **x,** then the right side can be an operation **y op z** that uses two addresses.

We can avoid some copy instructions by modifying the translation functions to generate a partial instruction that computes, say j+k, but does not commit to where the result is to be placed, signified by a **null** address for the result:

$$\textbf{null} = j + k \tag{2.8}$$

The null result address is later replaced by either an identifier or a temporary, as appropriate. It is replaced by an identifier if j+k is on the right side of an assignment, as in i=j+k;, in which case (2.8) becomes

$$i = j + k$$

But, if j+k is a subexpression, as in j+k+1, then the null result address in (2.8) is replaced by a new temporary **t,** and a new partial instruction is generated

$$\begin{aligned} \textbf{t} &= j + k \\ \textbf{null} &= \textbf{t} + 1 \end{aligned}$$

Many compilers make every effort to generate code that is as good as or better than hand-written assembly code produced by experts. If code-optimization techniques, such as the ones in Chapter 9 are used, then an effective strategy may well be to use a simple approach for intermediate code generation, and rely on the code optimizer to eliminate unnecessary instructions.

2.8.5 Exercises for Section 2.8

Exercise **2.8.1**: For-statements in C and Java have the form:

$$\texttt{for}\ (\ expr_1\ ;\ expr_2\ ;\ expr_3\ )\ stmt$$

The first expression is executed before the loop; it is typically used for initializing the loop index. The second expression is a test made before each iteration of the loop; the loop is exited if the expression becomes 0. The loop itself can be thought of as the statement *{stmt* $expr_3$*;}*. The third expression is executed at the end of each iteration; it is typically used to increment the loop index. The meaning of the for-statement is similar to

$$expr_1;\ \ \text{while}\ (\ expr_2\ )\ \{stmt\ \ expr3;\ \ \}$$

Define a class *For* for for-statements, similar to class //in Fig. 2.43.

Exercise **2.8.2**: The programming language C does not have a boolean type. Show how a C compiler might translate an if-statement into three-address code.

2.9 Summary of Chapter 2

The syntax-directed techniques in this chapter can be used to construct compiler front ends, such as those illustrated in Fig. 2.46.

```
if( peek == '\n' ) line = line + 1;
```

Lexical Analyzer

```
(if) {(> (id, "peek") (eq> (const, »\n») ())
(id, "line") (assign) (id, "line") (+) (num, 1) (;)
```

Syntax-Directed Translator

or

if

assign

peek **(int)** +

An'

```
tl = (int) '\n'
ifFalse peek == tl goto 4
line = line + 1
```

Figure 2.46: Two possible translations of a statement

- The starting point for a syntax-directed translator is a grammar for the source language. A *grammar* describes the hierarchical structure of programs. It is defined in terms of elementary symbols called *terminals* and variable symbols called *nonterminals.* These symbols represent language constructs. The rules or *productions* of a grammar consist of a nonterminal called the *head* or *left side* of a production and a sequence of terminals and nonterminals called the *body* or *right side* of the production. One nonterminal is designated as the *start* symbol.

- In specifying a translator, it is helpful to attach attributes to programming construct, where an *attribute* is any quantity associated with a construct. Since constructs are represented by grammar symbols, the concept of attributes extends to grammar symbols. Examples of attributes include an integer value associated with a terminal **num** representing numbers, and a string associated with a terminal **id** representing identifiers.

- A *lexical analyzer* reads the input one character at a time and produces as output a stream Of *tokens,* where a token consists of a terminal symbol along with additional information in the form of attribute values. In Fig. 2.46, tokens are written as tuples enclosed between ( ). The token **(id,** "peek") consists of the terminal **id** and a pointer to the symbol-table entry containing the string "peek". The translator uses the table to keep

track of reserved words and identifiers that have already been seen.

4- *Parsing* is the problem of figuring out how a string of terminals can be derived from the start symbol of the grammar by repeatedly replacing a nonterminal by the body of one of its productions. Conceptually, a parser builds a parse tree in which the root is labeled with the start symbol, each nonleaf corresponds to a production, and each leaf is labeled with a terminal or the empty string e. The parse tree derives the string of terminals at the leaves, read from left to right.

4* Efficient parsers can be built by hand, using a top-down (from the root to the leaves of a parse tree) method called predictive parsing. A *predictive parser* has a procedure for each nonterminal; procedure bodies mimic the productions for nonterminals; and, the flow of control through the procedure bodies can be determined unambiguously by looking one symbol ahead in the input stream. See Chapter 4 for other approaches to parsing.

- Syntax-directed translation is done by attaching either rules or program fragments to productions in a grammar. In this chapter, we have considered only *synthesized* attributes — the value of a synthesized attribute at any node *x* can depend only on attributes at the children of re, if any. A *syntax-directed definition* attaches rules to productions; the rules compute attribute vales. A *translation scheme* embeds program fragments called *semantic actions* in production bodies. The actions are executed in the order that productions are used during syntax analysis.

- The result of syntax analysis is a representation of the source program, called *intermediate code.* Two primary forms of intermediate code are illustrated in Fig. 2.46. An *abstract syntax tree* has nodes for programming constructs; the children of a node give the meaningful sub constructs. Alternatively, *three-address code* is a sequence of instructions in which each instruction carries out a single operation.

- *Symbol tables* are data structures that hold information about identifiers. Information is put into the symbol table when the declaration of an identifier is analyzed. A semantic action gets information from the symbol table when the identifier is subsequently used, for example, as a factor in an expression.

Chapter 3

Lexical Analysis

In this chapter we show how to construct a lexical analyzer. To implement a lexical analyzer by hand, it helps to start with a diagram or other description for the lexemes of each token. We can then write code to identify each occurrence of each lexeme on the input and to return information about the token identified.

We can also produce a lexical analyzer automatically by specifying the lexeme patterns to a *lexical-analyzer generator* and compiling those patterns into code that functions as a lexical analyzer. This approach makes it easier to modify a lexical analyzer, since we have only to rewrite the affected patterns, not the entire program. It also speeds up the process of implementing the lexical analyzer, since the programmer specifies the software at the very high level of patterns and relies on the generator to produce the detailed code. We shall introduce in Section **3**.5 a lexical-analyzer generator called *Lex* (or *Flex* in a more recent embodiment).

We begin the study of lexical-analyzer generators by introducing regular expressions, a convenient notation for specifying lexeme patterns. We show how this notation can be transformed, first into nondeterministic automata and then into deterministic automata. The latter two notations can be used as input to a "driver," that is, code which simulates these automata and uses them as a guide to determining the next token. This driver and the specification of the automaton form the nucleus of the lexical analyzer.

3.1 The Role of the Lexical Analyzer

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the

kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

These interactions are suggested in Fig. 3.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the *getNextToken* command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

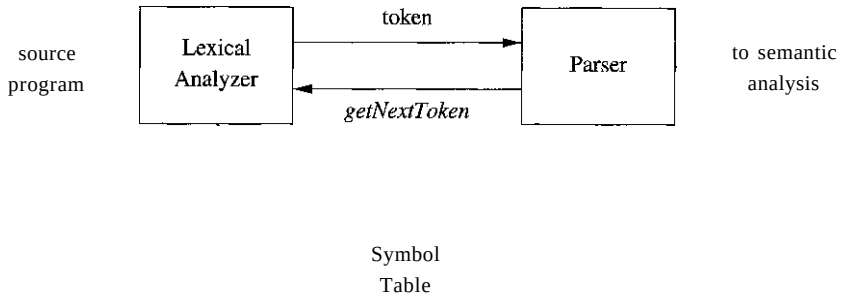


Figure 3.1: Interactions between the lexical analyzer and the parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and *whitespace* (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Sometimes, lexical analyzers are divided into a cascade of two processes:

a) *Scanning* consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

b) *Lexical analysis* proper is the more complex portion, where the scanner produces the sequence of tokens as output.

3.1.1 Lexical Analysis Versus Parsing

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

1. Simplicity of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer. If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.

2. Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

3. Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

3.1.2 Tokens, Patterns, and Lexemes

When discussing lexical analysis, we use three related but distinct terms:

- A *token* is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

- A *pattern* is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.

- A *lexeme* is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Example 3.1: Figure 3.2 gives some typical tokens, their informally described patterns, and some sample lexemes. To see how these concepts are used in practice, in the C statement

```
printf("Total = %d\n", score);
```

both printf and score are lexemes matching the pattern for token id, and "Total = °/,d\n" is a lexeme matching literal. •

In many programming languages, the following classes cover most or all of the tokens:

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters **i, f**	**if**
else	characters **e, 1, s, e**	**else**
comparison	< or > or <= or >= or == or ! =	**<=, ! =**
id	letter followed by letters and digits	**pi, score, D2**
number	any numeric constant	**3.14159, 0, 6.02e23**
literal	anything but ", surrounded by "'s	**"core dumped"**

Figure 3.2: Examples of tokens

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.

2. Tokens for the operators, either individually or in classes such as the token comparison rfientioned in Fig. 3.2.

3. One token representing all identifiers.

4. One or more tokens representing constants, such as numbers and literal strings.

5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

3.1.3 Attributes for Tokens

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token **number** matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. The most important example is the token id, where we need to associate with the token a great deal of information. Normally, information about an identifier — e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) — is kept in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Tricky Problems When Recognizing Tokens

Usually, given the pattern describing the lexemes of a token, it is relatively simple to recognize matching lexemes when they occur on the input. However, in some languages it is not immediately apparent when we have seen an instance of a lexeme corresponding to a token. The following example is taken from Fortran, in the fixed-format still allowed in Fortran 90. In the statement

```
DO 5 I = 1.25
```

it is not apparent that the first lexeme is D05I, an instance of the identifier token, until we see the dot following the 1. Note that blanks in fixed-format Fortran are ignored (an archaic convention). Had we seen a comma instead of the dot, we would have had a do-statement

```
DO 5 I = 1,25
```

in which the first lexeme is the keyword DO.

Example 3.2: The token names and associated attribute values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs.

<id, pointer to symbol-table entry for E>
<assign_op>
<id, pointer to symbol-table entry for M>
<mult_op>
<id, pointer to symbol-table entry for C>
<exp_op>
<number, integer value 2>

Note that in certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value. In this example, the token **number** has been given an integer-valued attribute. In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for **number** a pointer to that string. •

3.1.4 Lexical Errors

It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error. For instance, if the string fi is encountered for the first time in a C program in the context:

```
fi ( a == f(x)) ...
```

a lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier. Since fi is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters.

However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. The simplest recovery strategy is "panic mode" recovery. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

Transformations like these may be tried in an attempt to repair the input. The simplest such strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation. This strategy makes sense, since in practice most lexical errors involve a single character. A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes, but this approach is considered too expensive in practice to be worth the effort.

3.1.5 Exercises for Section 3.1

Exercise **3.1.1**: Divide the following C++ program:

```
float limitedSquare(x) float x {
    /* returns x-squared, but never more than 100 */
    return (x<=-10.0||x>=10.0)?100:x*x;
}
```

into appropriate lexemes, using the discussion of Section 3.1.2 as a guide. Which lexemes should get associated lexical values? What should those values be?

! Exercise **3.1.2**: Tagged languages like HTML or XML are different from conventional programming languages in that the punctuation (tags) are either very numerous (as in HTML) or a user-definable set (as in XML). Further, tags can often have parameters. Suggest how to divide the following HTML document:

```
Here is a photo of <B>my house</B>:
<PXIMG SRC = "house.gif"><BR>
See <A HREF = "morePix.html">More Pictures</A> if you
liked that one.<P>
```

into appropriate lexemes. Which lexemes should get associated lexical values, and what should those values be?

3.2 Input Buffering

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. The box on "Tricky Problems When Recognizing Tokens" in Section 3.1 gave an extreme example, but there are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id. In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large lookaheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

3.2.1 Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded, as suggested in Fig. 3.3.

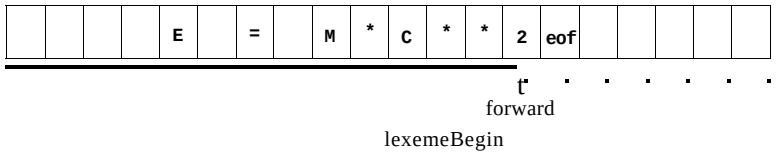


Figure 3.3: Using a pair of input buffers

Each buffer is of the same size *N*, and *N* is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read *N* characters into a buffer, rather than using one system call per character. If fewer than *N* characters remain in the input file, then a special character, represented by **eof,**

marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained:

1. Pointer **lexemeBegin,** marks the beginning of the current lexeme, whose extent we are attempting to determine.

2. Pointer **forward** scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, **forward** is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, **lexemeBegin** is set to the character immediately after the lexeme just found. In Fig. **3.3,** we see **forward** has passed the end of the next lexeme, ** (the Fortran exponentiation operator), and must be retracted one position to its left.

Advancing **forward** requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move **forward** to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than *N,* we shall never overwrite the lexeme in its buffer before determining it.

3.2.2 Sentinels

If we use the scheme of Section **3.2.1** as described, we must check, each time we advance **forward,** that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a *sentinel* character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof.**

Figure **3.4** shows the same arrangement as Fig. **3.3,** but with the sentinels added. Note that **eof** retains its use as a marker for the end of the entire input. Any **eof** that appears other than at the end of a buffer means that the input is at an end. Figure **3.5** summarizes the algorithm for advancing **forward.** Notice how the first test, which can be part of a multiway branch based on the character pointed to by **forward,** is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

3.3 Specification of Tokens

Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in spec-

Can We Run Out of Buffer Space?

In most modern languages, lexemes are short, and one or two characters of lookahead is sufficient. Thus a buffer size N in the thousands is ample, and the double-buffer scheme of Section **3.2.1** works without problem. However, there are some risks. For example, if character strings can be very long, extending over many lines, then we could face the possibility that a lexeme is longer than N. To avoid problems with long character strings, we can treat them as a concatenation of components, one from each line over which the string is written. For instance, in Java it is conventional to represent long strings by writing a piece on each line and concatenating pieces with a + operator at the end of each piece.

A more difficult problem occurs when arbitrarily long lookahead may be needed. For example, some languages like PL/I do not treat keywords as *reserved;* that is, you can use identifiers with the same name as a keyword like **DECLARE.** If the lexical analyzer is presented with text of a PL/I program that begins **DECLARE (ARG1, ARG2,...** it cannot be sure whether **DECLARE** is a keyword, and **ARG1** and so on are variables being declared, or whether **DECLARE** is a procedure name with its arguments. For this reason, modern languages tend to reserve their keywords. However, if not, one can treat a keyword like **DECLARE** as an ambiguous identifier, and let the parser resolve the issue, perhaps in conjunction with symbol-table lookup.

ifying those types of patterns that we actually need for tokens. In this section we shall study the formal notation for regular expressions, and in Section 3.5 we shall see how these expressions are used in a lexical-analyzer generator. Then, Section 3.7 shows how to build the lexical analyzer by converting regular expressions to automata that perform the recognition of the specified tokens.

3.3.1 Strings and Languages

An *alphabet* is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set {0,1} is the *binary alphabet.* ASCII is an important example of an alphabet; it is used in many software systems. Uni-

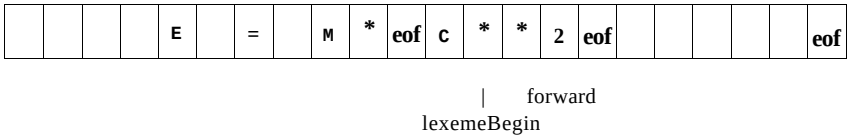


Figure **3.4:** Sentinels at the end of each buffer

```
switch ( *forward++ ) {
    case eof:
        if (forward is at end of first buffer ) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer ) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}
```

Figure 3.5: Lookahead code with sentinels

Implementing Multiway Branches

We might imagine that the switch in Fig. 3.5 requires many steps to execute, and that placing the case **eof** first is not a wise choice. Actually, it doesn't matter in what order we list the cases for each character. In practice, a multiway branch depending on the input character is be made in one step by jumping to an address found in an array of addresses, indexed by characters.

code, which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet.

A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string *s*, usually written |s|, is the number of occurrences of symbols in *s*. For example, banana is a string of length six. The *empty string*, denoted e, is the string of length zero.

A *language* is any countable set of strings over some fixed alphabet. This definition is very broad. Abstract languages like 0, the *empty set*, or {e}, the set containing only the empty string, are languages under this definition. So too are the set of all syntactically well-formed C programs and the set of all grammatically correct English sentences, although the latter two languages are difficult to specify exactly. Note that the definition of "language" does not require that any meaning be ascribed to the strings in the language. Methods for defining the "meaning" of strings are discussed in Chapter 5.

Terms for Parts of Strings

The following string-related terms are commonly used:

1. A *prefix* of string *s* is any string obtained by removing zero or more symbols from the end of *s*. For example, ban, banana, and e are prefixes of banana.

2. A *suffix* of string *s* is any string obtained by removing zero or more symbols from the beginning of *s*. For example, nana, banana, and e are suffixes of banana.

3. A *substring* of *s* is obtained by deleting any prefix and any suffix from *s*. For instance, banana, nan, and e are substrings of banana.

4. The *proper* prefixes, suffixes, and substrings of a string *s* are those, prefixes, suffixes, and substrings, respectively, of *s* that are not e or not equal to *s* itself.

5. A *subsequence* of *s* is any string formed by deleting zero or more not necessarily consecutive positions of *s*. For example, baan is a subsequence of banana.

If *x* and *y* are strings, then the *concatenation* of *x* and *y*, denoted *xy*, is the string formed by appending *y* to *x*. For example, if *x* = dog and *y* = house, then *xy* — doghouse. The empty string is the identity under concatenation; that is, for any string *s*, *es* = *se* — *s*.

If we think of concatenation as a product, we can define the "exponentiation" of strings as follows. Define s° to be e, and for all $i > 0$, define s^{i} to be $s^{i\sim 1}s$. Since es = *s*, it follows that s^{1} = *s*. Then s^{2} = *ss*, s^{3} = *sss*, and so on.

3.3.2 Operations on Languages

In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in Fig. 3.6. Union is the familiar operation on sets. The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them. The *(Kleene) closure* of a language *L*, denoted *L**, is the set of strings you get by concatenating *L* zero or more times. Note that L°, the "concatenation of *L* zero times," is defined to be {e}, and inductively, *U* is $L^{i\sim 1}L$. Finally, the positive closure, denoted L^{+}, is the same as the Kleene closure, but without the term L°. That is, e will not be in L^{+} unless it is in *L* itself.

OPERATION ,	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s$ is in L or s is in $M\}$
Concatenation of L and M	$LM = \{st \mid s$ is in L and t is in $M\}$
Kleene closure of L	$L^* = \cup L^i$
Positive closure of L	

Figure 3.6: Definitions of operations on languages

Example **3.3** Let L be the set of letters $\{A, B, \ldots, Z, a, b, \ldots, z\}$ and let D be the set of digits $\{0, 1, \ldots 9\}$. We may think of L and D in two, essentially equivalent, ways. One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that L and D are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages L and D, using the operators of Fig. 3.6:

1. $L \cup D$ is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.
3. L^4 is the set of all 4-letter strings.
4. L^* is the set of all strings of letters, including ϵ, the empty string.
5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
6. D^+ is the set of all strings of one or more digits.

•

3.3.3 Regular Expressions

Suppose we wanted to describe the set of valid C identifiers. It is almost exactly the language described in item (5) above; the only difference is that the underscore is included among the letters.

In Example 3.3, we were able to describe identifiers by giving names to sets of letters and digits and using the language operators union, concatenation, and closure. This process is so useful that a notation called *regular expressions* has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet. In this notation, if *letter-* is established to stand for any letter or the underscore, and *digit-* is

established to stand for any digit, then we could describe the language of C identifiers by:

*letter- (letter- \ digit)**

The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of," and the juxtaposition of *letter,* with the remainder of the expression signifies concatenation.

The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language *L(r),* which is also defined recursively from the languages denoted by r's subexpressions. Here are the rules that define the regular expressions over some alphabet £ and the languages that those expressions denote.

BASIS: There are two rules that form the basis:

1. e is a regular expression, and *L(e)* is {e}, that is, the language whose sole member is the empty string.

2. If *a* is a symbol in E, then **a** is a regular expression, and L**(a)** = *{a},* that is, the language with one string, of length one, with *a* in its one position. Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.[1]

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and *s* are regular expressions denoting languages *L(r)* and *L(s),* respectively.

1. (r)|(s) is a regular expression denoting the language *L(r)* U *L(s).*

2. (r)(s) is a regular expression denoting the language *L(r)L(s).*

3. (r)* is a regular expression denoting (L(r))*.

4. (r) is a regular expression denoting *L(r).* This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:

a) The unary operator * has highest precedence and is left associative.

b) Concatenation has second highest precedence and is left associative.

[1] however, when talking about specific characters from the ASCII character set, we shall generally use teletype font for both the character and its regular expression.

c) | has lowest precedence and is left associative.

Under these conventions, for example, we may replace the regular expression **(a)|((b)*(c))** by **a|b***c. Both expressions denote the set of strings that are either a single *a* or are zero or more 6's followed by one c.

Example 3.4: Let £ = {a, 6}.

1. The regular expression **a|b** denotes the language *{a, b}.*

2. **(a|b)(a|b)** denotes *{aa, ah, ba, bb},* the language of all strings of length two over the alphabet E. Another regular expression for the same language is **aa|ab|ba|bb.**

3. **a*** denotes the language consisting of all strings of zero or more a's, that is, {e,*a,aa,aaa,...}*.

4. **(a|b)*** denotes the set of all strings consisting of zero or more instances of *a* or *b,* that is, all strings of a's and 6's: {e,*a, b,aa, ab, ba, bb,aaa,...}.* Another regular expression for the same language is **(a*b*)*.**

5. **a|a*b** denotes the language *{a, b, ab, aab, aaab,...*}, that is, the string *a* and all strings consisting of zero or more a's and ending in *b*.

•

A language that can be defined by a regular expression is called a *regular set.* If two regular expressions r and *s* denote the same regular set, we say they are *equivalent* and write *r = s*. For instance, **(a|b) = (b|a).** There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent. Figure 3.7 shows some of the algebraic laws that hold for arbitrary regular expressions *r, s,* and *t.*

LAW	DESCRIPTION
r\s = s\r	| is commutative
r\(s\t) = (r\s)\t	| is associative
r(st) = (rs)t	Concatenation is associative
r(s\t) = rs\rt; (s\t)r = sr\tr	Concatenation distributes over |
er = re = r	e is the identity for concatenation
*r** = (r|e)*	e is guaranteed in a closure
	* is idempotent

Figure 3.7: Algebraic laws for regular expressions

3.3.4 Regular Definitions

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If £ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

$$d_i \rightarrow r_i$$
$$d_2 \rightarrow r_2$$
$$\vdots$$
$$d_n \rightarrow r_n$$

where:

1. Each d_i is a new symbol, not in E and not the same as any other of the cTs, and
2. Each r_i is a regular expression over the alphabet E U $\{d_1, d_2, \ldots, d_{i-1}\}$

By restricting to E and the previously defined **GTS**, we avoid recursive definitions, and we can construct a regular expression over E alone, for each r_i. We do so by first replacing uses of d_1 in r_2 (which cannot use any of the d's except for d_1), then replacing uses of d_1 and d_2 in r_i by r_1 and (the substituted) r_2, and so on. Finally, in r_n we replace each d_i, for $i = 1, 2, \ldots, n-1$, by the substituted version of r_i, each of which has only symbols of E.

Example 3.5: C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$\begin{aligned} \textit{letter} &\rightarrow \mathbf{A} \mid \mathbf{B} \mid \cdots \mid \mathbf{Z} \mid \mathbf{a} \mid \mathbf{b} \mid \cdots \mid \mathbf{z} \mid \_ \\ \textit{digit} &\rightarrow \mathbf{0} \mid \mathbf{1} \mid \cdots \mid \mathbf{9} \\ \textit{id} &\rightarrow \textit{letter} \left(\textit{letter} \mid \textit{digit} \right)^* \end{aligned}$$

Example 3.6: Unsigned numbers (integer or floating point) are strings such as **5280, 0.01234, 6.336E4,** or **1.89E-4.** The regular definition

$$\begin{aligned} \textit{digit} &\rightarrow \mathbf{0} \mid \mathbf{1} \mid \cdots \mid \mathbf{9} \\ \textit{digits} &\rightarrow \textit{digit}\ \textit{digit}^* \\ \textit{optionalFraction} &\rightarrow \mathbf{.}\ \textit{digits} \mid \epsilon \\ \textit{optionalExponent} &\rightarrow \left(\mathbf{E} \left(\mathbf{+} \mid \mathbf{-} \mid \epsilon \right) \textit{digits} \right) \mid \epsilon \\ \textit{number} &\rightarrow \textit{digits}\ \textit{optionalFraction}\ \textit{optionalExponent} \end{aligned}$$

is a precise specification for this set of strings. That is, an *optionalFraction* is either a decimal point (dot) followed by one or more digits, or it is missing (the empty string). An *optionalExponent*, if not missing, is the letter E followed by an optional + or - sign, followed by one or more digits. Note that at least one digit must follow the dot, so *number* does not match **1.**, but does match **1.0.**

3.3.5 Extensions of Regular Expressions

Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns. Here we mention a few notational extensions that were first incorporated into Unix utilities such as Lex that are particularly useful in the specification lexical analyzers. The references to this chapter contain a discussion of some regular-expression variants in use today.

1. *One or more instances.* The unary, postfix operator + represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then $(r)^+$ denotes the language $(L(r))^+$. The operator + has the same precedence and associativity as the operator *. Two useful algebraic laws, $r^* = r^+ | e$ and $r^+ = rr^* = r^*r$ relate the Kleene closure and positive closure.

2. *Zero or one instance.* The unary postfix operator ? means "zero or one occurrence." That is, r? is equivalent to r|e, or put another way, $L(r?) = L(r) \cup \{e\}$. The ? operator has the same precedence and associativity as * and +.

3. *Character classes.* A regular expression $a_1|a_2|\cdots|a_n$, where the a_is are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2\cdots a_n]$. More importantly, when $a_1, a_2, \ldots, a_n$ form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a_1-a_n, that is, just the first and last separated by a hyphen. Thus, **[abc]** is shorthand for **a|b|c,** and **[a-z]** is shorthand for **a|b|.--|z.**

Example 3.7: Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

letter. -> **[A-Za-z_]**
digit -> **[0-9]**
id -> *letter- (letter \ digit)**

The regular definition of Example 3.6 can also be simplified:

digit -> **[0-9]**
digits —>• *digit*+
number -»• *digits (. digits)?* **(E [+-]?** *digits)?*

•

3.3.6 Exercises for Section 3.3

Exercise 3.3.1: Consult the language reference manuals to determine *(i)* the sets of characters that form the input alphabet (excluding those that may only appear in character strings or comments), *(ii)* the lexical form of numerical constants, and *(Hi)* the lexical form of identifiers, for each of the following languages: (a) C (b) C++ (c) C# (d) Fortran (e) Java (f) Lisp (g) SQL.

! **Exercise 3.3.2**: Describe the languages denoted by the following regular expressions:

a) **a(a|b)*a.**

b) ((e|**a)b*)*.**

c) **(a|b)*a(a|b)(a|b).**

d) **a*ba*ba*ba*.**

!! e) **(aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*.**

Exercise 3.3.3: In a string of length n, how many of the following are there?

a) Prefixes.

b) Suffixes.

c) Proper prefixes.

! d) Substrings.

! e) Subsequences.

Exercise 3.3.4: Most languages are *case sensitive,* so keywords can be written only one way, and the regular expressions describing their lexeme is very simple. However, some languages, like SQL, are *case insensitive,* so a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Thus, the SQL keyword **SELECT** can also be written **select, Select,** or **sElEcT,** for instance. Show how to write a regular expression for a keyword in a case-insensitive language. Illustrate the idea by writing the expression for "select" in SQL.

! **Exercise 3.3.5:** Write regular definitions for the following languages:

a) All strings of lowercase letters that contain the five vowels in order.

b) All strings of lowercase letters in which the letters are in ascending lexicographic order.

c) Comments, consisting of a string surrounded by /* and */, without an intervening */, unless it is inside double-quotes (").

!! d) All strings of digits with no repeated digits. *Hint:* Try this problem first with a few digits, such as {0,1,2}.

!! e) All strings of digits with at most one repeated digit.

!! f) All strings of a's and 6's with an even number of a's and an odd number of 6's.

g) The set of Chess moves, in the informal notation, such as p-k4 or kbpxqn.

!! h) All strings of a's and 6's that do not contain the substring a66.

i) All strings of a's and 6's that do not contain the subsequence a66.

Exercise **3.3.6:** Write character classes for the following sets of characters:

a) The first ten letters (up to "j") in either upper or lower case.

b) The lowercase consonants.

c) The "digits" in a hexadecimal number (choose either upper or lower case for the "digits" above 9).

d) The characters that can appear at the end of a legitimate English sentence (e.g., exclamation point).

The following exercises, up to and including Exercise 3.3.10, discuss the extended regular-expression notation from Lex (the lexical-analyzer generator that we shall discuss extensively in Section 3.5). The extended notation is listed in Fig. 3.8.

Exercise **3.3.7:** Note that these regular expressions give all of the following symbols *(operator characters)* a special meaning:

\ » . ~ $ [ ] * + ? { } I /

Their special meaning must be turned off if they are needed to represent themselves in a character string. We can do so by quoting the character within a string of length one or more; e.g., the regular expression "**" matches the string **. We can also get the literal meaning of an operator character by preceding it by a backslash. Thus, the regular expression ** also matches the string **. Write a regular expression that matches the string "\.

Exercise **3.3.8:** In Lex, a *complemented character class* represents any character except the ones listed in the character class. We denote a complemented class by using " as the first character; this symbol (caret) is not itself part of the class being complemented, unless it is listed within the class itself. Thus, [~A-Za-z] matches any character that is not an uppercase or lowercase letter, and [~\~] represents any character but the caret (or newline, since newline cannot be in any character class). Show that for every regular expression with complemented character classes, there is an equivalent regular expression without complemented character classes.

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	**a**
V	character c literally	*
IIgII	string s literally	**"**"**
	any character but newline	**a.** *b
	beginning of a line	~abc
$	end of a line	abc$
[*]	any one of the characters in string *s*	[abc]
	any one character not in string *s*	[~abc]
r*	zero or more strings matching *r*	**a***
r+	one or more strings matching *r*	**a+**
r?	zero or one r	**a?**
r{m, n}	between *m* and *n* occurrences of r	**a**[l**,5]**
rir$_2$	an ?*1 followed by an *r*$_2$	ab
r\ | *r*$_2$	an ri or an *r*$_2$	**a**|b
(r)	same as r	(alb)
ri/r$_2$	7*1 when followed by *r*$_2$	abc/**123**

Figure **3.8: Lex** regular expressions

! Exercise 3.3.9: The regular expression *r{m, n]* matches from *m* to n occurrences of the pattern r. For example, **a[1,5]** matches a string of one to five a's. Show that for every regular expression containing repetition operators of this form, there is an equivalent regular expression without repetition operators.

! Exercise 3.3.10: The operator " matches the left end of a line, and $ matches the right end of a line. The operator " is also used to introduce complemented character classes, but the context always makes it clear which meaning is intended. For example, **~[~aeiou]*$** matches any complete line that does not contain a lowercase vowel.

a) How do you tell which meaning of ^ is intended?

b) Can you always replace a regular expression using the " and $ operators by an equivalent expression that does not use either of these operators?

! Exercise 3.3.11: The UNIX shell command **sh** uses the operators in Fig. **3.9** in filename expressions to describe sets of file names. For example, the filename expression ***.o** matches all file names ending in **.o; sortl.?** matches all filenames of the form **sort**.c, where c is any character. Show how **sh** filename

EXPRESSION	MATCHES	EXAMPLE
V	string *s* literally	'V
V	character c literally	V
*	any string	*.o
?	any character	sortl.?
[s]	any character in *s*	sortl.[cso]

Figure 3.9: Filename expressions used by the shell command sh

expressions can be replaced by equivalent regular expressions using only the basic union, concatenation, and closure operators.

! **Exercise 3.3.12:** SQL allows a rudimentary form of patterns in which two characters have special meaning: underscore (_) stands for any one character and percent-sign (%) stands for any string of 0 or more characters. In addition, the programmer may define any character, say e, to be the escape character, so e preceding an e preceding _, % or another e gives the character that follows its literal meaning. Show how to express any SQL pattern as a regular expression, given that we know which character is the escape character.

3.4 Recognition of Tokens

In the previous section we learned how to express patterns using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns. Our discussion will make use of the following running example.

stmt —> **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
I *e*
expr —»• *term* **relop** *term*
| *term*
term -> **id**
| **number**

Figure 3.10: A grammar for branching statements

Example 3.8: The grammar fragment of Fig. 3.10 describes a simple form of branching statements and conditional expressions. This syntax is similar to that of the language Pascal, in that **then** appears explicitly after conditions.

For **relop,** we use the comparison operators of languages like Pascal or SQL, where = is "equals" and <> is "not equals," because it presents an interesting structure of lexemes.

The terminals of the grammar, which are **if, then, else, relop, id,** and **number,** are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions, as in Fig. 3.11. The patterns for *id* and *number* are similar to what we saw in Example 3.7.

digit	->	[0-9]
digits		*digit*$^+$
number		*digits (. digits)?* **(E [+-]?** *digits)?*
letter		[A-Za-z]
id		*letter (letter \ digit)**
if	->	if
then		**then**
else		**else**
relop		$<$ $1>1 \Leftarrow 1 \geq 1 = 1 \Leftrightarrow$

Figure 3.11: Patterns for tokens of Example 3.8

For this language, the lexical analyzer will recognize the keywords if, **then,** and **else,** as well as lexemes that match the patterns for *relop, id,* and *number.* To simplify matters, we make the common assumption that keywords are also *reserved words:* that is, they are not identifiers, even though their lexemes match the pattern for identifiers.

In addition, we assign the lexical analyzer the job of stripping out white-space, by recognizing the "token" *ws* defined by:

ws -» **(blank | tab** j **newline)**$^+$

Here, **blank, tab,** and **newline** are abstract symbols that we use to express the ASCII characters of the same names. Token *ws* is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. It is the following token that gets returned to the parser.

Our goal for the lexical analyzer is summarized in Fig. 3.12. That table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value, as discussed in Section 3.1.3, is returned. Note that for the six relational operators, symbolic constants **LT, LE,** and so on are used as the attribute value, in order to indicate which instance of the token **relop** we have found. The particular operator found will influence the code that is output from the compiler. •

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	-	-
if	**if**	-
then	**then**	-
else	**else**	-
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	

Figure 3.12: Tokens, their patterns, and attribute values

3.4.1 Transition Diagrams

As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "transition diagrams." In this section, we perform the conversion from regular-expression patterns to transition diagrams by hand, but in Section 3.6, we shall see that there is a mechanical way to construct these diagrams from collections of regular expressions.

Transition diagrams have a collection of nodes or circles, called *states.* Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of a state as summarizing all we need to know about what characters we have seen between the *lexemeBegin* pointer and the *forward* pointer (as in the situation of Fig. 3.3).

Edges are directed from one state of the transition diagram to another. Each edge is *labeled* by a symbol or set of symbols. If we are in some state **5,** and the next input symbol is *a,* we look for an edge out of state *s* labeled by *a* (and perhaps by other symbols, as well). If we find such an edge, we advance the *forward* pointer arid enter the state of the transition diagram to which that edge leads. We shall assume that all our transition diagrams are *deterministic,* meaning that there is never more than one edge out of a given state with a given symbol among its labels. Starting in Section 3.5, we shall relax the condition of determinism, making life much easier for the designer of a lexical analyzer, although trickier for the implementer. Some important conventions about transition diagrams are:

1. Certain states are said to be *accepting,* or *final.* These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the *lexemeBegin* and *forward* pointers. We always

indicate an accepting state by a double circle, and if there is an action to be taken — typically returning a token and an attribute value to the parser — we shall attach that action to the accepting state.

2. In addition, if it is necessary to retract the *forward* pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract *forward* by more than one position, but if it were, we could attach any number of *'s to the accepting state.

3. One state is designated the *start state,* or *initial state;* it is indicated by an edge, labeled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

Example 3.9: Figure 3.13 is a transition diagram that recognizes the lexemes matching the token **relop.** We begin in state 0, the start state. If we see < as the first input symbol, then among the lexemes that match the pattern for **relop** we can only be looking at <, <>, or <=. We therefore go to state 1, and look at the next character. If it is =, then we recognize lexeme <=, enter state 2, and return the token **relop** with attribute LE, the symbolic constant representing this particular comparison operator. If in state 1 the next character is >, then instead we have lexeme <>, and enter state 3 to return an indication that the not-equals operator has been found. On any other character, the lexeme is <, and we enter state 4 to return that information. Note, however, that state 4 has a * to indicate that we must retract the input one position.

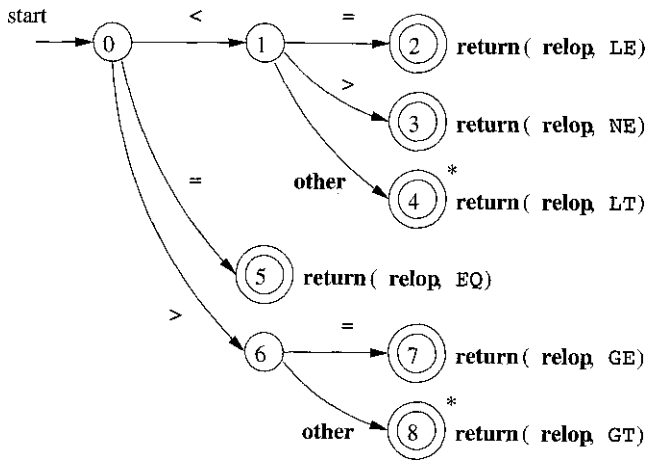


Figure 3.13: Transition diagram for **relop**

On the other hand, if in state 0 the first character we see is =, then this one character must be the lexeme. We immediately return that fact from state 5.

The remaining possibility is that the first character is >. Then, we must enter state 6 and decide, on the basis of the next character, whether the lexeme is >= (if we next see the = sign), or just > (on any other character). Note that if, in state 0, we see any character besides <, =, or >, we can not possibly be seeing a **relop** lexeme, so this transition diagram will not be used. •

3.4.2 Recognition of Reserved Words and Identifiers

Recognizing keywords and identifiers presents a problem. Usually, keywords like if or **then** are reserved (as they are in our running example), so they are not identifiers even though they *look* like identifiers. Thus, although we typically use a transition diagram like that of Fig. 3.14 to search for identifier lexemes, this diagram will also recognize the keywords if, **then,** and **else** of our running example.

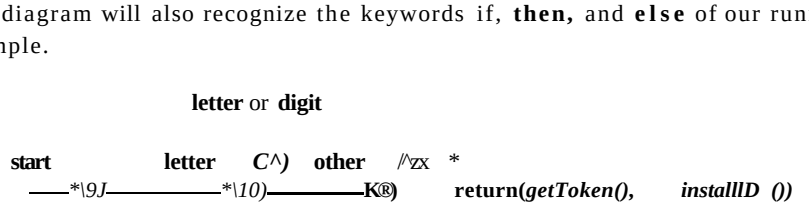


Figure 3.14: A transition diagram for **id**'s and keywords

There are two ways that we can handle reserved words that look like identifiers:

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this method is in use in Fig. 3.14. When we find an identifier, a call to *installID* places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is **id.** The function *getToken* examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents — either **id** or one of the keyword tokens that was initially installed in the table.

2. Create separate transition diagrams for each keyword; an example for the keyword **then** is shown in Fig. 3.15. Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token **then** in situations where the correct token was **id,** with a lexeme like thenextvalue that has **then** as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved-word

tokens are recognized in preference to id, when the lexeme matches both patterns. We *do not* use this approach in our example, which is why the states in Fig. 3.15 are unnumbered.

Figure 3.15: Hypothetical transition diagram for the keyword then

3.4.3 Completion of the Running Example

The transition diagram for id's that we saw in Fig. 3.14 has a simple structure. Starting in state 9, it checks that the lexeme begins with a letter and goes to state 10 if so. We stay in state 10 as long as the input contains letters and digits. When we first encounter anything but a letter or digit, we go to state 11 and accept the lexeme found. Since the last character is not part of the identifier, we must retract the input one position, and as discussed in Section 3.4.2, we enter what we have found in the symbol table and determine whether we have a keyword or a true identifier.

The transition diagram for token number is shown in Fig. 3.16, and is so far the most complex diagram we have seen. Beginning in state 12, if we see a digit, we go to state 13. In that state, we can read any number of additional digits. However, if we see anything but a digit or a dot, we have seen a number in the form of an integer; 123 is an example. That case is handled by entering state 20, where we return token number and a pointer to a table of constants where the found lexeme is entered. These mechanics are not shown on the diagram but are analogous to the way we handled identifiers.

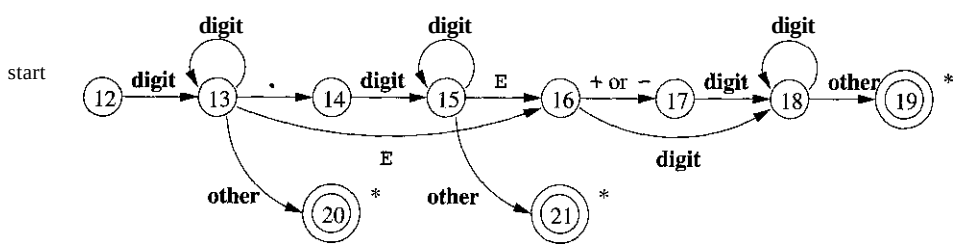


Figure 3.16: A transition diagram for unsigned numbers

If we instead see a dot in state 13, then we have an "optional fraction." State 14 is entered, and we look for one or more additional digits; state 15 is used for that purpose. If we see an **E**, then we have an "optional exponent," whose recognition is the job of states 16 through 19. Should we, in state 15, instead see anything but **E** or a digit, then we have come to the end of the fraction, there is no exponent, and we return the lexeme found, via state 21.

The final transition diagram, shown in Fig. 3.17, is for whitespace. In that diagram, we look for one or more "whitespace" characters, represented by **delim** in that diagram — typically these characters would be blank, tab, newline, and perhaps other characters that are not considered by the language design to be part of any token.

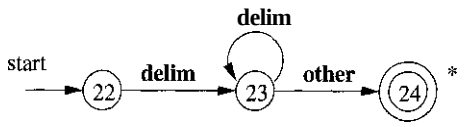


Figure 3.17: A transition diagram for whitespace

Note that in state 24, we have found a block of consecutive whitespace characters, followed by a nonwhitespace character. We retract the input to begin at the nonwhitespace, but we do not return to the parser. Rather, we must restart the process of lexical analysis after the whitespace.

3.4.4 Architecture of a Transition-Diagram-Based Lexical Analyzer

There are several ways that a collection of transition diagrams can be used to build a lexical analyzer. Regardless of the overall strategy, each state is represented by a piece of code. We may imagine a variable state holding the number of the current state for a transition diagram. A switch based on the value of state takes us to code for each of the possible states, where we find the action of that state. Often, the code for a state is itself a switch statement or multiway branch that determines the next state by reading and examining the next input character.

Example 3.10: In Fig. 3.18 we see a sketch of getRelopO, a C++ function whose job is to simulate the transition diagram of Fig. 3.13 and return an object of type TOKEN, that is, a pair consisting of the token name (which must be **relop** in this case) and an attribute value (the code for one of the six comparison operators in this case). getRelopO first creates a new object ret Token and initializes its first component to RELOP, the symbolic code for token **relop.**

We see the typical behavior of a state in case 0, the case where the current state is 0. A function nextChar() obtains the next character from the input and assigns it to local variable c. We then check c for the three characters we expect to find, making the state transition dictated by the transition diagram of Fig. 3.13 in each case. For example, if the next input character is =, we go to state 5.

If the next input character is not one that can begin a comparison operator, then a function fail() is called. What fail() does depends on the global error-recovery strategy of the lexical analyzer. It should reset the forward pointer to lexemeBegin, in order to allow another transition diagram to be applied to

```
TOKEN getRelopO
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextCharQ;
                    if ( c == '<» ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state - 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...

            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
```

Figure **3.18:** Sketch of implementation of **relop** transition diagram

the true beginning of the unprocessed input. It might then change the value of **state** to be the start state for another transition diagram, which will search for another token. Alternatively, if there is no other transition diagram that remains unused, **fail()** could initiate an error-correction phase that will try to repair the input and find a lexeme, as discussed in Section **3.1.4.**

We also show the action for state **8** in Fig. **3.18.** Because state **8** bears a *, we must retract the input pointer one position (i.e., put c back on the input stream). That task is accomplished by the function **retract** (). Since state **8** represents the recognition of lexeme >=, we set the second component of the returned object, which we suppose is named **attribute,** to GT, the code for this operator. •

To place the simulation of one transition diagram in perspective, let us consider the ways code like Fig. **3.18** could fit into the entire lexical analyzer.

1. We could arrange for the transition diagrams for each token to be tried sequentially. Then, the function **f ail()** of Example **3.10** resets the pointer **forward** and starts the next transition diagram, each time it is called. This method allows us to use transition diagrams for the individual keywords, like the one suggested in Fig. **3.15.** We have oniy to use these before we use the diagram for id, in order for the keywords to be reserved words.

2. We could run the various transition diagrams "in parallel," feeding the next input character to all of them and allowing each one to make whatever transitions it required. If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input. The normal strategy is to take the longest prefix of the input that matches any pattern. That rule allows us to prefer identifier thenext to keyword then, or the operator -> to -, for example.

3. The preferred approach, and the one we shall take up in the following sections, is to combine all the transition diagrams into one. We allow the transition diagram to read input until there is no possible next state, and then take the longest lexeme that matched any pattern, as we discussed in item (2) above. In our running example, this combination is easy, because no two tokens can start with the same character; i.e., the first character immediately tells us which token we are looking for. Thus, we could simply combine states 0, 9, 12, and 22 into one start state, leaving other transitions intact. However, in general, the problem of combining transition diagrams for several tokens is more complex, as we shall see shortly.

3.4.5 Exercises for Section 3.4

Exercise 3.4.1: Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.2.

Exercise 3.4.2: Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.5.

The following exercises, up to Exercise 3.4.12, introduce the Aho-Corasick algorithm for recognizing a collection of keywords in a text string in time proportional to the length of the text and the sum of the length of the keywords. This algorithm uses a special form of transition diagram called a *trie.* A trie is a tree-structured transition diagram with distinct labels on the edges leading from a node to its children. Leaves of the trie represent recognized keywords.

Knuth, Morris, and Pratt presented an algorithm for recognizing a single keyword $\&i\&_2 \cdots K$ in a text string. Here the trie is a transition diagram with n states, 0 through n. State 0 is the initial state, and state n represents acceptance, that is, discovery of the keyword. From each state s from 0 through n - 1, there is a transition to state s + 1, labeled by symbol b_s**.i.** For example, the trie for the keyword ababaa is:

In order to process text strings rapidly and search those strings for a keyword, it is useful to define, for keyword $\&i\&_2 \cdots \&_n$ and position s in that keyword (corresponding to state s of its trie), a *failure function, f(s),* computed as in

Fig. **3.19.** The objective is that &i&2" •&/(*) ^s the longest proper prefix of &1&2 • • • b_s that is also a suffix of $b\backslash b_2$ • • • b_s. The reason *f(s)* is important is that if we are trying to match a text string for 61&2 • • • b_n, and we have matched the first *s* positions, but we then fail (i.e., the next position of the text string does not hold b_{s+1}*i)*, then *f(s)* is the longest prefix of &1&2 • * • b_n that could possibly match the text string up to the point we are at. Of course, the next character of the text string must be &/($_s$)+i, or else we still have problems and must consider a yet shorter prefix, which will be &/(/($_s$)).

```
1)  t = 0;
2)  /(1)=0;
3)  for (s = 1; s < n; s + +) {
4)        while (t > 0 kk b_{s+1} ! =  bt+i) t = fit);
5)        if (6_{s+}i == b_{t+1}i) {
6)              t = t + l;
7)              f(s + 1) = t;
          }
8)        else f(s + 1) = 0;
    }
```

Figure **3.19:** Algorithm to compute the failure function for keyword 6162 • • • b_n

As an example, the failure function for the trie constructed above for **ababaa** is:

s	**1**	2	**3**	**4**	5	6
f(s)	**0**	**0**	**1**	2	**3**	**1**

For instance, states **3** and **1** represent prefixes **aba** and **a,** respectively. **/(3) = 1** because **a** is the longest proper prefix of **aba** that is also a suffix of **aba.** Also, /(2) **= 0,** because the longest proper prefix of **ab** that is also a suffix is the empty string.

Exercise 3.4.3: Construct the failure function for the strings:

a) **abababaab.**

b) **aaaaaa.**

c) **abbaabb.**

! Exercise 3.4.4: Prove, by induction on s, that the algorithm of Fig. **3.19** correctly computes the failure function.

!! Exercise 3.4.5: Show that the assignment *t = f(t)* in line **(4)** of Fig. **3.19** is executed at most *n* times. Show that therefore, the entire algorithm takes only *0(n)* time on a keyword of length *n.*

Having computed the failure function for a keyword $bib_2 \cdots b_n$, we can scan a string $a\backslash a_2 \text{---} a_m$ in time $0(m)$ to tell whether the keyword occurs in the string. The algorithm, shown in Fig. 3.20, slides the keyword along the string, trying to make progress by matching the next character of the keyword with the next character of the string. If it cannot do so after matching s characters, then it "slides" the keyword right $s - f(s)$ positions, so only the first /(s) characters of the keyword are considered matched with the string.

```
1)  s = 0;
2)  for (i = 1; i < m; i++) {
3)          while (s > 0        a_i ! =  b_(s+1)) s = f(s);
4)          if,(a* == b_s+i) s = s + 1;
5)          if (s == n) return "yes";
    }
6)  return "no";
```

Figure 3.20: The KMP algorithm tests whether string $a\backslash a_2$ - ••a_m contains a single keyword $b\backslash b_2 \cdots b_n$ as a substring in $0(m + n)$ time

Exercise 3.4.6: Apply Algorithm KMP to test whether keyword **ababaa** is a substring of:

a) **abababaab.**

b) **abababbaa.**

!! **Exercise 3.4.7:** Show that the algorithm of Fig. 3.20 correctly tells whether the keyword is a substring of the given string. *Hint:* proceed by induction on i. Show that for all i, the value of s after line **(4)** is the length of the longest prefix of the keyword that is a suffix of $a\backslash a_2 \cdots$ a^.

!! **Exercise 3.4.8**: Show that the algorithm of Fig. 3.20 runs in time $0(m + n)$, assuming that function / is already computed and its values stored in an array indexed by s.

Exercise 3.4.9: The *Fibonacci strings* are defined as follows:

1. si = **b.**

2. s_2 = **a.**

3. $Sk = Sk\text{-}iSk\text{-}2$ for $k > 2$.

For example, s_3 = **ab, S4 = aba,** and **S5 = abaab.**

a) What is the length of s_n?

b) Construct the failure function for *SQ*.

c) Construct the failure function for ***sj.***

!! d) Show that the failure function for any s_n can be expressed by /(1) = /(2) = 0, and for 2 < *j* < \$s_n$\, *f(j)* is *j* - \$s_k$-*i*\, where *k* is the largest integer such that < *j* + 1.

!! e) In the KMP algorithm, what is the largest number of consecutive applications of the failure function, when we try to determine whether keyword ***Sk*** appears in text string s^+i?

Aho and Corasick generalized the KMP algorithm to recognize any of a set of keywords in a text string. In this case, the trie is a true tree, with branching from the root. There is one state for every string that is a prefix (not necessarily proper) of any keyword. The parent of a state corresponding to string ***b\b₂ • • • bk*** is the state that corresponds to &1&2 • • • frfc-i- A state is accepting if it corresponds to a complete keyword. For example, Fig. 3.21 shows the trie for the keywords he, she, his, and hers.

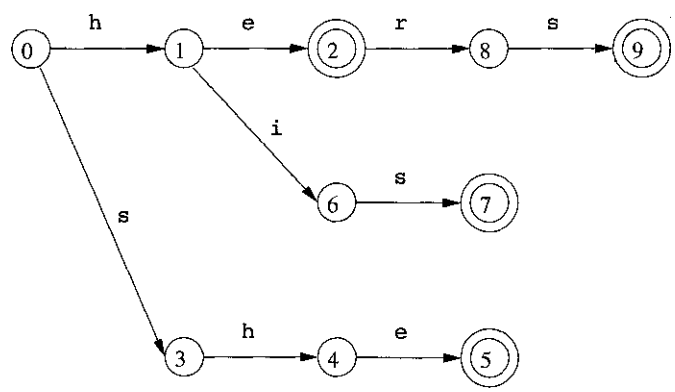


Figure 3.21: Trie for keywords he, she, his, hers

The failure function for the general trie is defined as follows. Suppose ***s*** is the state that corresponds to string &i&₂ • • • ***b***ₙ**.** Then ***f(s)*** is the state that corresponds to the longest proper suffix of &i6₂ • • -***b***ₙ that is also a prefix of ***some*** keyword. For example, the failure function for the trie of Fig. 3.21 is:

s	1	2	3	4	5	6	7	8	9
f(s)	0	0	0	1	2	0	3	0	3

! Exercise 3.4.10: Modify the algorithm of Fig. 3.19 to compute the failure function for general tries. *Hint:* The major difference is that we cannot simply test for equality or inequality of b_{i+1} and $b_i i$ in lines (4) and (5) of Fig. 3.19. Rather, from any state there may be several transitions out on several characters, as there are transitions on both e and i from state 1 in Fig. 3.21. Any of

those transitions could lead to a state that represents the longest suffix that is also a prefix.

Exercise 3.4.11: Construct the tries and compute the failure function for the following sets of keywords:

a) aaa, abaaa, and ababaaa.

b) all, fall, fatal, llama, and lame.

c) pipe, pet, item, temper, and perpetual.

! Exercise 3.4.12: Show that your algorithm from Exercise 3.4.10 still runs in time that is linear in the sum of the lengths of the keywords.

3.5 The Lexical-Analyzer Generator Lex

In this section, we introduce a tool called Lex, or in a more recent implementation Flex, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens. The input notation for the Lex tool is referred to as the *Lex language* and the tool itself is the *Lex compiler.* Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex.yy. c, that simulates this transition diagram. The mechanics of how this translation from regular expressions to transition diagrams occurs is the subject of the next sections; here we only learn the Lex language.

3.5.1 Use of Lex

Figure 3.22 suggests how Lex is used. An input file, which we call lex.l, is written in the Lex language and describes the lexical analyzer to be generated. The Lex compiler transforms lex. 1 to a C program, in a file that is always named lex.yy. c. The latter file is compiled by the C compiler into a file called a.out, as always. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

The normal use of the compiled C program, referred to as a. out in Fig. 3.22, is as a subroutine of the parser. It is a C function that returns an integer, which is a code for one of the possible token names. The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable yylval,[2] which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.

[2] incidentally, the **yy** that appears in **yylval** and **lex.yy.c** refers to the **Yacc** parser-generator, which we shall describe in Section 4.9, and which is commonly used in conjunction with **Lex.**

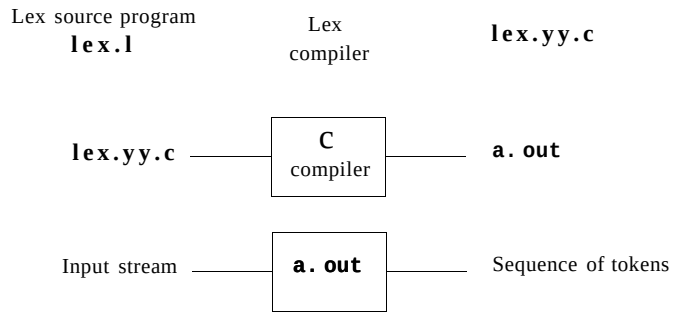


Figure **3.22:** Creating a lexical analyzer with **Lex**

3.5.2 Structure of Lex Programs

A **Lex** program has the following form:

```
declarations
%%
translation rules
%%
auxiliary functions
```

The declarations section includes declarations of variables, *manifest constants* (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions, in the style of Section **3.3.4.**

The translation rules each have the form

Pattern { Action }

Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C, although many variants of **Lex** using other languages have been created.

The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

The lexical analyzer created by **Lex** behaves in concert with the parser as follows. When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns *Pi.* It then executes the associated action *Ai.* Typically, *Ai* will return to the parser, but if it does not (e.g., because *Pi* describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser. The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable yylval to pass additional information about the lexeme found, if needed.

Example 3.11: Figure 3.23 is a Lex program that recognizes the tokens of Fig. 3.12 and returns the token found. A few observations about this code will introduce us to many of the important features of Lex.

In the declarations section we see a pair of special brackets, %{ and %}. Anything within these brackets is copied directly to the file lex.yy.c, and is not treated as a regular definition. It is common to place there the definitions of the manifest constants, using C #define statements to associate unique integer codes with each of the manifest constants. In our example, we have listed in a comment the names of the manifest constants, LT, **IF,** and so on, but have not shown them defined to be particular integers.[3]

Also in the declarations section is a sequence of regular definitions. These use the extended notation for regular expressions described in Section 3.3.5. Regular definitions that are used in later definitions or in the patterns of the translation rules are surrounded by curly braces. Thus, for instance, *delim* is defined to be a shorthand for the character class consisting of the blank, the tab, and the newline; the latter two are represented, as in all UNIX commands, by backslash followed by t or n, respectively. Then, *ws* is defined to be one or more delimiters, by the regular expression {delim}+.

Notice that in the definition of *id* and *number,* parentheses are used as grouping metasymbols and do not stand for themselves. In contrast, E in the definition of *number* stands for itself. If we wish to use one of the Lex metasymbols, such as any of the parentheses, +, *, or ?, to stand for themselves, we may precede them with a backslash. For instance, we see \. in the definition of *number,* to represent the dot, since that character is a metasymbol representing "any character," as usual in UNIX regular expressions.

In the auxiliary-function section, we see two such functions, installID() and installNumO. Like the portion of the declaration section that appears between %{. . .%}, everything in the auxiliary section is copied directly to file lex.yy.c, but may be used in the actions.

Finally, let us examine some of the patterns and rules in the middle section of Fig. 3.23. First, *ws,* an identifier declared in the first section, has an associated empty action. If we find whitespace, we do not return to the parser, but look for another lexeme. The second token has the simple regular expression pattern if. Should we see the two letters if on the input, and they are not followed by another letter or digit (which would cause the lexical analyzer to find a longer prefix of the input matching the pattern for id), then the lexical analyzer consumes these two letters from the input and returns the token name **IF,** that is, the integer for which the manifest constant **IF** stands. Keywords then and else are treated similarly.

The fifth token has the pattern defined by *id.* Note that, although keywords like if match this pattern as well as an earlier pattern, Lex chooses whichever

[3] If **Lex** is used along with **Yacc,** then it would be normal to define the manifest constants in the **Yacc** program and use them without definition in the **Lex** program. Since **lex.yy.c** is compiled with the **Yacc** output, the constants thus will be available to the actions in the **Lex** program.

```
U
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
'/.}

/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

°/y.

{ws}       {/* no action and no return */}
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval = (int) installID(); return(ID);}
{number}   {yylval = (int) installNumO ; return(NUMBER);}
1 I < M    {yylval = LT; return(RELOP);}
"<="       {yylval = LE; return(RELOP);}
II _ II    {yylval = EQ; return(RELOP);}
"<>"       {yylval = NE; return(RELOP);}
           {yylval = GT; return(RELOP);}
           {yylval = GE; return(RELOP)•}


int installlDQ {/* function to install the lexeme, whose
                   first character is pointed to by yytext,
                   arid whose length is yyleng, into the
                   symbol table and return a pointer
                   thereto */


int installNumO {/* similar to installlD, but puts numer-
                    ical constants into a separate table */
}
```

Figure **3.23: Lex** program for the tokens of Fig. **3.12**

pattern is listed first in situations where the longest matching prefix matches two or more patterns. The action taken when *id* is matched is threefold:

1. Function installID() is called to place the lexeme found in the symbol table.

2. This function returns a pointer to the symbol table, which is placed in global variable yylval, where it can be used by the parser or a later component of the compiler. Note that installID() has available to it two variables that are set automatically by the lexical analyzer that Lex generates:

   (a) yytext is a pointer to the beginning of the lexeme, analogous to lexemeBegin in Fig. 3.3.

   (b) yyleng is the length of the lexeme found.

3. The token name ID is returned to the parser.

The action taken when a lexeme matching the pattern *number* is similar, using the auxiliary function installNumO. •

3.5.3 Conflict Resolution in Lex

We have alluded to the two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

1. Always prefer a longer prefix to a shorter prefix.

2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

Example 3.12: The first rule tells us to continue reading letters and digits to find the longest prefix of these characters to group as an identifier. It also tells us to treat <= as a single lexeme, rather than selecting < as one lexeme and = as the next lexeme. The second rule makes keywords reserved, if we list the keywords before id in the program. For instance, if then is determined to be the longest prefix of the input that matches any pattern, and the pattern then precedes {id}, as it does in Fig. 3.23, then the token THEN is returned, rather than ID. •

3.5.4 The Lookahead Operator

Lex automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input. However, sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters. If so, we may use the slash in a pattern to indicate the end of the part of the

pattern that matches the lexeme. What follows / is additional pattern that must be matched before we can decide that the token in question was seen, but what matches this second pattern is not part of the lexeme.

Example 3.13: In Fortran and some other languages, keywords are not reserved. That situation creates problems, such as a statement

```
IF(I,J) = 3
```

where **IF** is the name of an array, not a keyword. This statement contrasts with statements of the form

```
IF( condition ) THEN ...
```

where **IF** is a keyword. Fortunately, we can be sure that the keyword **IF** is always followed by a left parenthesis, some text — the condition —- that may contain parentheses, a right parenthesis and a letter. Thus, we could write a **Lex** rule for the keyword **IF** like:

```
IF / \( .* \) {letter}
```

This rule says that the pattern the lexeme matches is just the two letters **IF.** The slash says that additional pattern follows but does not match the lexeme. In this pattern, the first character is the left parentheses. Since that character is a **Lex** metasymbol, it must be preceded by a backslash to indicate that it has its literal meaning. The dot and star match "any string without a newline." Note that the dot is a **Lex** metasymbol meaning "any character except newline." It is followed by a right parenthesis, again with a backslash to give that character its literal meaning. The additional pattern is followed by the symbol *letter,* which is a regular definition representing the character class of all letters.

Note that in order for this pattern to be foolproof, we must preprocess the input to delete whitespace. We have in the pattern neither provision for whitespace, nor can we deal with the possibility that the condition extends over lines, since the dot will not match a newline character.

For instance, suppose this pattern is asked to match a prefix of input:

```
IF(A<(B+C)*D)THEN...
```

the first two characters match **IF,** the next character matches \(, the next nine characters match .*, and the next two match \) and *letter.* Note the fact that the first right parenthesis (after C) is not followed by a letter is irrelevant; we only need to find some way of matching the input to the pattern. We conclude that the letters **IF** constitute the lexeme, and they are an instance of token **if.**

•

3.5.5 Exercises for Section 3.5

Exercise 3.5.1: Describe how to make the following modifications to the **Lex** program of Fig. **3.23:**

a) Add the keyword while.

b) Change the comparison operators to be the C operators of that kind.

c) Allow the underscore (_) as an additional letter.

! d) Add a new pattern with token **STRING.** The pattern consists of a double-quote ("), any string of characters and a final double-quote. However, if a double-quote appears in the string, it must be escaped by preceding it with a backslash (\), and therefore a backslash in the string must be represented by two backslashes. The lexical value, which is the string without the surrounding double-quotes, and with backslashes used to escape a character removed. Strings are to be installed in a table of strings.

Exercise 3.5.2: Write a **Lex** program that copies a file, replacing each non-empty sequence of white space by a single blank.

Exercise 3.5.3: Write a **Lex** program that copies a C program, replacing each instance of the keyword **float** by **double.**

! **Exercise 3.5.4:** Write a **Lex** program that converts a file to "Pig latin." Specifically, assume the file is a sequence of words (groups of letters) separated by whitespace. Every time you encounter a word:

1. If the first letter is a consonant, move it to the end of the word and then add **ay.**

2. If the first letter is a vowel, just add **ay** to the end of the word.

All nonletters are copied intact to the output.

! **Exercise 3.5.5:** In SQL, keywords and identifiers are case-insensitive. Write a **Lex** program that recognizes the keywords **SELECT, FROM,** and **WHERE** (in any combination of capital and lower-case letters), and token **ID,** which for the purposes of this exercise you may take to be any sequence of letters and digits, beginning with a letter. You need not install identifiers in a symbol table, but tell how the "install" function would differ from that described for case-sensitive identifiers as in Fig. **3.23.**

3.6 Finite Automata

We shall now discover how Lex turns its input program into a lexical analyzer. At the heart of the transition is the formalism known as *finite automata.* These are essentially graphs, like transition diagrams, with a few differences:

1. Finite automata are *recognizers;* they simply say "yes" or "no" about each possible input string.
2. Finite automata come in two flavors:
   (a) *Nondeterministic finite automata* (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and e, the empty string, is a possible label.
   (b) *Deterministic finite automata* (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

Both deterministic and nondeterministic finite automata are capable of recognizing the same languages. In fact these languages are exactly the same languages, called the *regular languages,* that regular expressions can describe.[4]

3.6.1 Nondeterministic Finite Automata

A *nondeterministic finite automaton* (NFA) consists of:

1. A finite set of states 5.
2. A set of input symbols E, the *input alphabet.* We assume that e, which stands for the empty string, is never a member of E.
3. A *transition function* that gives, for each state, and for each symbol in E U {e} a set of *next states.*
4. A state so from *S* that is distinguished as the *start state* (or *initial state).*
5. A set of states *F,* a subset of *S,* that is distinguished as the *accepting states* (or *final states).*

We can represent either an NFA or DFA by a *transition graph,* where the nodes are states and the labeled edges represent the transition function. There is an edge labeled *a* from state *s* to state *t* if and only if *t* is one of the next states for state *s* and input *a.* This graph is very much like a transition diagram, except:

[4]There is a small lacuna: as we defined them, regular expressions cannot describe the empty language, since we would never want to use this pattern in practice. However, finite automata ***can*** define the empty language. In the theory, 0 is treated as an additional regular expression for the sole purpose of defining the empty language.

a) The same symbol can label edges from one state to several different states, and

b) An edge may be labeled by e, the empty string, instead of, or in addition to, symbols from the input alphabet.

Example 3.14: The transition graph for an NFA recognizing the language of regular expression (a|b)*abb is shown in Fig. 3.24. This abstract example, describing all strings of a's and &'s ending in the particular string *abb,* will be used throughout this section. It is similar to regular expressions that describe languages of real interest, however. For instance, an expression describing all files whose name ends in .o is any*.o, where any stands for any printable character.

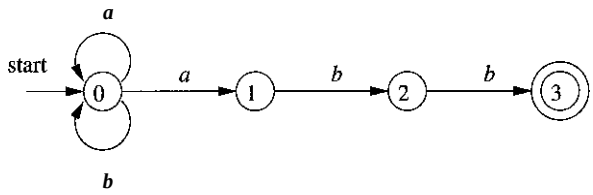


Figure 3.24: A nondeterministic finite automaton

Following our convention for transition diagrams, the double circle around state 3 indicates that this state is accepting. Notice that the only ways to get from the start state 0 to the accepting state is to follow some path that stays in state 0 for a while, then goes to states 1, 2, and 3 by reading *abb* from the input. Thus, the only strings getting to the accepting state are those that end in *abb.* •

3.6.2 Transition Tables

We can also represent an NFA by a *transition table,* whose rows correspond to states, and whose columns correspond to the input symbols and e. The entry for a given state and input is the value of the transition function applied to those arguments. If the transition function has no information about that state-input pair, we put 0 in the table for the pair.

Example 3.15: The transition table for the NFA of Fig. 3.24 is shown in Fig. 3.25. •

The transition table has the advantage that we can easily find the transitions on a given state and input. Its disadvantage is that it takes a lot of space, when the input alphabet is large, yet most states do not have any moves on most of the input symbols.

STATE	*a*	*b*	*e*
0	{0,1}	{0}	Θ
1	Θ	{2}	Θ
2	Θ	{3}	Θ
3	Θ	Θ	Θ

Figure 3.25: Transition table for the NFA of Fig. 3.24

3.6.3 Acceptance of Input Strings by Automata

An NFA *accepts* input string *x* if and only if there is some path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out *x*. Note that e labels along the path are effectively ignored, since the empty string does not contribute to the string constructed along the path.

Example 3.16: The string *aabb* is accepted by the NFA of Fig. 3.24. The path labeled by *aabb* from state 0 to state 3 demonstrating this fact is:

0 —*a*— 0 —*a*— 1 —*b*— 2 —*b*— 3

Note that several paths labeled by the same string may lead to different states. For instance, path

0 —*a*— 0 —*a*— 0 —*b*— 0 —*b*— 0

is another path from state 0 labeled by the string *aabb*. This path leads to state 0, which is not accepting. However, remember that an NFA accepts a string as long as *some* path labeled by that string leads from the start state to an accepting state. The existence of other paths leading to a nonaccepting state is irrelevant. •

The *language defined* (or *accepted)* by an NFA is the set of strings labeling some path from the start to an accepting state. As was mentioned, the NFA of Fig. 3.24 defines the same language as does the regular expression **(a|b)*abb,** that is, all strings from the alphabet *{a, b}* that end in *abb*. We may use *L(A)* to stand for the language accepted by automaton *A*.

Example 3.17: Figure 3.26 is an NFA accepting L**(aa*|bb*).** String *aaa* is accepted because of the path

0 —e— 1 —*a*— 2 —*a*— 2 —*a*— 2

Note that e's "disappear" in a concatenation, so the label of the path is *aaa*. •

3.6.4 Deterministic Finite Automata

A *deterministic finite automaton* (DFA) is a special case of an NFA where:

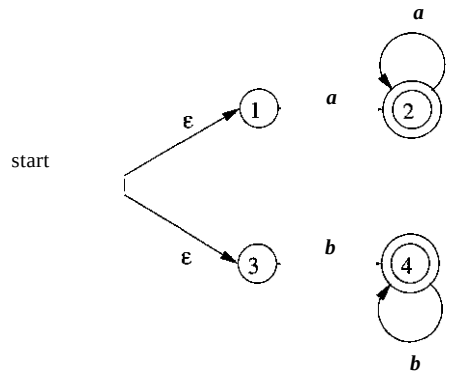


Figure 3.26: NFA accepting **aa*|bb***

1. There are no moves on input e, and

2. For each state *s* and input symbol a, there is exactly one edge out of *s* labeled *a.*

If we are using a transition table to represent a DFA, then each entry is a single state, we may therefore represent this state without the curly braces that we use to form sets.

While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language, the DFA is a simple, concrete algorithm for recognizing strings. It is fortunate indeed that every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers. The following algorithm shows how to apply a DFA to a string.

Algorithm 3.18: Simulating a DFA.

INPUT: An input string *x* terminated by an end-of-file character **eof.** A DFA *D* with start state *so,* accepting states *F,* and transition function *move.*

OUTPUT: Answer "yes" if *D* accepts *x;* "no" otherwise.

METHOD: Apply the algorithm in Fig. 3.27 to the input string *x.* The function *move(s,c)* gives the state to which there is an edge from state *s* on input c. The function *next Char* returns the next character of the input string *x.* •

Example 3.19: In Fig. 3.28 we see the transition graph of a DFA accepting the language **(a|b)*abb,** the same as that accepted by the NFA of Fig. 3.24. Given the input string *ababb,* this DFA enters the sequence of states 0,1,2,1,2,3 and returns "yes." •

```
s = s0
c = nextCharQ;
while ( c != eof) {
        s = move(s,c);
        c = nextCharQ;
}
if ( s is in F ) return "yes";
else return "no";
```

Figure 3.27: Simulating a DFA

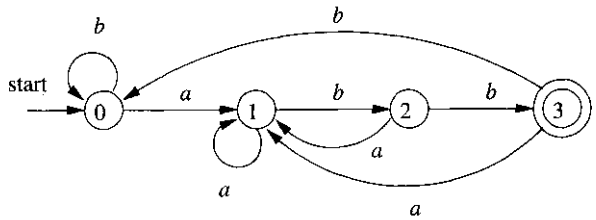


Figure 3.28: DFA accepting (**a**|**b**)***abb**

3.6.5 Exercises for Section 3.6

! **Exercise 3.6.1**: Figure **3.19** in the exercises of Section **3.4** computes the failure function for the KMP algorithm. Show how, given that failure function, we can construct, from a keyword $bib_2 \cdots b_n$ an. n + **1**-state DFA that recognizes $*bib_2 \cdots b_n$, where the dot stands for "any character." Moreover, this DFA can be constructed in $0(n)$ time.

Exercise 3.6.2: Design finite automata (deterministic or nondeterministic) for each of the languages of Exercise **3.3.5.**

Exercise 3.6.3: For the NFA of Fig. **3.29,** indicate all the paths labeled *aabb.* Does the NFA accept *aabb?*

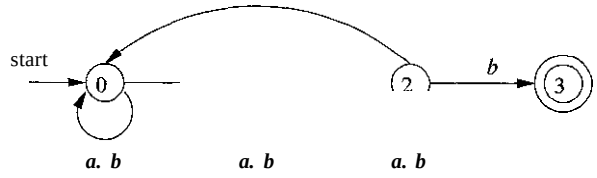


Figure **3.29:** NFA for Exercise **3.6.3**

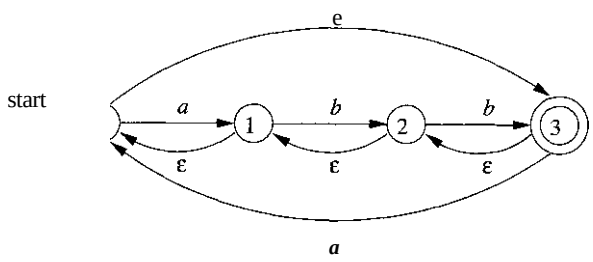


Figure 3.30: NFA for Exercise 3.6.4

Exercise 3.6.4: Repeat Exercise 3.6.3 for the NFA of Fig. 3.30.

Exercise 3.6.5: Give the transition tables for the NFA of:

a) Exercise 3.6.3.

b) Exercise 3.6.4.

c) Figure 3.26.

3.7 Prom Regular Expressions to Automata

The regular expression is the notation of choice for describing lexical analyzers and other pattern-processing software, as was reflected in Section 3.5. However, implementation of that software requires the simulation of a DFA, as in Algorithm 3.18, or perhaps simulation of an NFA. Because an NFA often has a choice of move on an input symbol (as Fig. 3.24 does oh input *a* from state 0) or on e (as Fig. 3.26 does from state 0), or even a choice of making a transition on e or on a real input symbol, its simulation is less straightforward than for a DFA. Thus often it is important to convert an NFA to a DFA that accepts the same language.

In this section we shall first show how to convert NFA's to DFA's. Then, we use this technique, known as "the subset construction," to give a useful algorithm for simulating NFA's directly, in situations (other than lexical analysis) where the NFA-to-DFA conversion takes more time than the direct simulation. Next, we show how to convert regular expressions to NFA's, from which a DFA can be constructed if desired. We conclude with a discussion of the time-space tradeoffs inherent in the various methods for implementing regular expressions, and see how to choose the appropriate method for your application.

3.7.1 Conversion of an NFA to a DFA

The general idea behind the subset construction is that each state of the constructed DFA corresponds to a set of NFA states. After reading input

flifl2 • • • *Q>n,* the DFA is in that state which corresponds to the set of states that the NFA can reach, from its start state, following paths labeled $a\backslash a_2$ • • • a_n.

It is possible that the number of DFA states is exponential in the number of NFA states, which could lead to difficulties when we try to implement this DFA. However, part of the power of the automaton-based approach to lexical analysis is that for real languages, the NFA and DFA have approximately the same number of states, and the exponential behavior is not seen.

Algorithm 3.20: The *subset construction* of a DFA from an NFA.

INPUT: An NFA JV.

OUTPUT: A DFA *D* accepting the same language as *N*.

METHOD: Our algorithm constructs a transition table *Dtran* for *D*. Each state of *D* is a set of NFA states, and we construct *Dtran* so *D* will simulate "in parallel" all possible moves *N* can make on a given input string. Our first problem is to deal with e-transitions of *N* properly. In Fig. 3.31 we see the definitions of several functions that describe basic computations on the states of *N* that are needed in the algorithm. Note that *s* is a single state of *N,* while *T* is a set of states of *N.*

OPERATION	DESCRIPTION
e-closure(s)	Set of NFA states reachable from NFA state *s* on e-transitions alone.
e-closure(T)	Set of NFA states reachable from some NFA state *s* in set *T* on e-transitions alone; = $U_{s\ in\ T}$ *e-closure(s).*
move(T, a)	Set of NFA states to which there is a transition on input symbol *a* from some state *s* in *T.*

Figure 3.31: Operations on NFA states

We must explore those sets of states that *N* can be in after seeing some input string. As a basis, before reading the first input symbol, *N* can be in any of the states of *e-closure(so),* where *SQ* is its start state. For the induction, suppose that *N* can be in set of states *T* after reading input string *x.* If it next reads input a, then *N* can immediately go to any of the states in *move(T, a).* However, after reading a, it may also make several e-transitions; thus *N* could be in any state of *e-closure(move(T, a))* after reading input *xa.* Following these ideas, the construction of the set of Z?'s states, *Dstates,* and its transition function *Dtran,* is shown in Fig. 3.32.

The start state of *D* is *e-closure(so),* and the accepting states of *D* are all those sets of AT's states that include at least one accepting state of *N.* To complete our description of the subset construction, we need only to show how

```
initially, e-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = e-closure(move(T,a));
                if ( U is not in Dstates )
                        add U as an unmarked state to Dstates;
                Dtran[T, a] = U;
        }
}
```

Figure 3.32: The subset construction

e-closure(T) is computed for any set of NFA states *T*. This process, shown in Fig. 3.33, is a straightforward search in a graph from a set of states. In this case, imagine that only the e-labeled edges are available in the graph. •

```
push all states of T onto stack;
initialize e~closure(T) to T;
while ( stack is not empty ) {
        pop t, the top element, off stack;
        for ( each state u with an edge from t to u labeled e )
                if ( u is not in e-closure(T) ) {
                        add u to e-closure(T);
                        push u onto stack;
                }
}
```

Figure 3.33: Computing *e-closure(T)*

Example 3.21: Figure 3.34 shows another NFA accepting **(a|b)*abb;** it happens to be the one we shall construct directly from this regular expression in Section 3.7. Let us apply Algorithm 3.20 to Fig. 3.29.

The start state *A* of the equivalent DFA is *e-closure(0),* or *A* = {0,1,2,4,7}, since these are exactly the states reachable from state 0 via a path all of whose edges have label e. Note that a path can have zero edges, so state 0 is reachable from itself by an e-labeled path.

The input alphabet is *{a, b}.* Thus, our first step is to mark *A* and compute *Dtran[A,a] = e-closure(move(A,a))* and *Dtran[A,b] = e-closure(move(A,b)).* Among the states 0, 1, 2, 4, and 7, only 2 and 7 have transitions on a, to 3 and 8, respectively. Thus, *move(A,a)* = {3,8}. Also, *e-closure({3,*8} = {1,2,3,4,6,7,8}, so we conclude

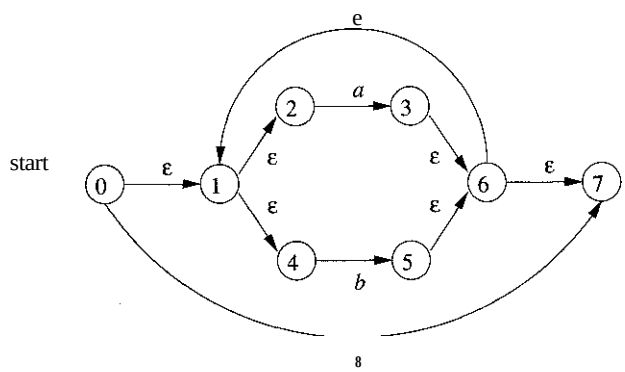


Figure 3.34: NFA *N* for **(a|b)*abb**

$$Dtran[A,a] = e\text{-}closure(move(A,a)) = e\text{-}closure(\{3,8\}) = \{1,2,3,4,6,7,8\}$$

Let us call this set *B*, so *Dtran[A, a] = B*.

Now, we must compute *Dtran[A,b]*. Among the states in *A*, only 4 has a transition on 6, and it goes to 5. Thus,

$$Dtran[A,b] = e\text{-}closure(\{5\}) = \{1,2,4,6,7\}$$

Let us call the above set C, so *Dtran[A, b] — C*.

NFA STATE	DFA STATE	*a*	*b*
{0,1,2,4,7}	*A*	*B*	*C*
{1,2,3,4,6,7,8}	*B*	*B*	*D*
{1,2,4,5,6,7}	*C*	*B*	*C*
{1,2,4,5,6,7,9}	*D*	*B*	*E*
{1,2,3,5,6,7,10}	*E*	*B*	*C*

Figure 3.35: Transition table *Dtran* for DFA *D*

If we continue this process with the unmarked sets *B* and C, we eventually reach a point where all the states of the DFA are marked. This conclusion is guaranteed, since there are "only" 2^{11} different subsets of a set of eleven NFA states. The five different DFA states we actually construct, their corresponding sets of NFA states, and the transition table for the DFA *D* are shown in Fig. 3.35, and the transition graph for *D* is in Fig. 3.36. State *A* is the start state, and state *E*, which contains state 10 of the NFA, is the only accepting state.

Note that *D* has one more state than the DFA of Fig. 3.28 for the same language. States *A* and *C* have the same move function, and so can be merged. We discuss the matter of minimizing the number of states of a DFA in Section 3.9.6.

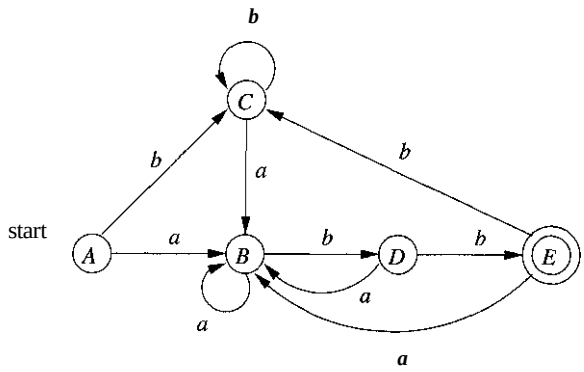


Figure 3.36: Result of applying the subset construction to Fig. 3.34

3.7.2 Simulation of an NFA

A strategy that has been used in a number of text-editing programs is to construct an NFA from a regular expression and then simulate the NFA using something like an on-the-fly subset construction. The simulation is outlined below.

Algorithm 3.22: Simulating an NFA.

INPUT: An input string *x* terminated by an end-of-file character **eof.** An NFA *N* with start state *SQ,* accepting states *F,* and transition function *move.*

OUTPUT: Answer "yes" if *M* accepts *x;* "no" otherwise.

METHOD: The algorithm keeps a set of current states *S,* those that are reached from so following a path labeled by the inputs read so far. If c is the next input character, read by the function *nextCharQ,* then we first compute *move(S,c)* and then close that set using *e-closureQ.* The algorithm is sketched in Fig. 3.37.
•

```
S = e-closure(so);
c = nextCharQ;
while ( c != eof) {
        S = e-closure(move(S,c));
        c = nextCharQ;
}
if ( 5 n F != 0 ) return "yes";
else return "no";
```

Figure 3.37: Simulating an NFA

3.7.3 Efficiency of NFA Simulation

If carefully implemented, Algorithm 3.22 can be quite efficient. As the ideas involved are useful in a number of similar algorithms involving search of graphs, we shall look at this implementation in additional detail. The data structures we need are:

1. Two stacks, each of which holds a set of NFA states. One of these stacks, *oldStates*, holds the "current" set of states, i.e., the value of *S* on the right side of line (4) in Fig. 3.37. The second, *newStates*, holds the "next" set of states — 5 on the left side of line (4). Unseen is a step where, as we go around the loop of lines (3) through (6), *newStates* is transferred to *oldStates*.

2. A boolean array *alreadyOn*, indexed by the NFA states, to indicate which states are in *newStates*. While the array and stack hold the same information, it is much faster to interrogate *alreadyOn[s]* than to search for state *s* on the stack *newStates*. It is for this efficiency that we maintain both representations.

3. A two-dimensional array *move[s,a]* holding the transition table of the NFA. The entries in this table, which are sets of states, are represented by linked lists.

To implement line (1) of Fig. 3.37, we need to set each entry in array *alreadyOn* to **FALSE**, then for each state *s* in *e-closure(so)*, push *s* onto *oldStates* and set *alreadyOn[s]* to **TRUE**. This operation on state *s*, and the implementation of line (4) as well, are facilitated by a function we shall call *addState(s)*. This function pushes state *s* onto *newStates*, sets *alreadyOn[s]* to **TRUE**, and calls itself recursively on the states in *move[s,e]* in order to further the computation of *e-closure(s)*. However, to avoid duplicating work, we must be careful never to call *addState* on a state that is already on the stack *newStates*. Figure 3.38 sketches this function.

```
addState(s)  {
        push s onto newStates;
        alreadyOn[s] = TRUE;
        for ( t on move[s, e] )
                if ( !alreadyOn(t) )
                        addState(t);
}
```

Figure 3.38: Adding a new state *s*, which is known not to be on *newStates*

We implement line (4) of Fig. 3.37 by looking at each state *s* on *oldStates*. We first find the set of states *move[s,* c], where c is the next input, and for each

of those states that is not already on *newStates,* we apply *addState* to it. Note that *addState* has the effect of computing the e-*closure* and adding all those states to *newStates* as well, if they were not already on. This sequence of steps is summarized in Fig. 3.39.

```
16)  for (s on oldStates ) {
17)          for ( t on move[s, c] )
18)                  if ( \alreadyOn[t] )
19)                          addState(t);
20)          pop s from oldStates;
21)  }

22)  for ( s on newStates ) {
23)          pop s from newStates;
24)          push s onto oldStates;
25)          a/readyOn[s] = FALSE;
26)  }
```

Figure 3.39: Implementation of step (4) of Fig. 3.37

Now, suppose that the NFA *N* has *n* states and *m* transitions; i.e., *m* is the sum over all states of the number of symbols (or e) on which the state has a transition out. Not counting the call to *addState* at line (19) of Fig. 3.39, the time spent in the loop of lines (16) through (21) is *0(n).* That is, we can go around the loop at most n times, and each step of the loop requires constant work, except for the time spent in *addState.* The same is true of the loop of lines (22) through (26).

During one execution of Fig. 3.39, i.e., of step (4) of Fig. 3.37, it is only possible to call *addState* on a given state once. The reason is that whenever we call *addState(s),* we set *alreadyOn[s]* to TRUE at line (11) of Fig. 3.39. Once *alreadyOn[s]* is TRUE, the tests at line (13) of Fig. 3.38 and line (18) of Fig. 3.39 prevent another call.

The time spent in one call to *addState,* exclusive of the time spent in recursive calls at line (14), is 0(1) for lines (10) and (11). For lines (12) and (13), the time depends on how many e-transitions there are out of state *s.* We do not know this number for a given state, but we know that there are at most m transitions in total, out of all states. As a result, the aggregate time spent in lines (11) over all calls to *addState* during one execution of the code of Fig. 3.39 is *0(m).* The aggregate for the rest of the steps of *addState* is *0(n),* since it is a constant per call, and there are at most n calls.

We conclude that, implemented properly, the time to execute line (4) of Fig. 3.37 is *0(n* + m). The rest of the while-loop of lines (3) through (6) takes 0(1) time per iteration. If the input *x* is of length *k,* then the total work in that loop is *0((k(n* + m)). Line (1) of Fig. 3.37 can be executed in 0(n + *m)* time, since it is essentially the steps of Fig. 3.39 with *oldStates* containing only

Big-Oh Notation

An expression like *0(n)* is a shorthand for "at most some constant times n." Technically, we say a function /(n), perhaps the running time of some step of an algorithm, is *0(g(n))* if there are constants c and no, such that whenever $n > n_0$, it is true that /(n) < *cg(n).* A useful idiom is "0(1)," which means "some constant." The use of this *big-oh notation* enables us to avoid getting too far into the details of what we count as a unit of execution time, yet lets us express the rate at which the running time of an algorithm grows.

the state so- Lines (2), (7), and (8) each take 0(1) time. Thus, the running time of Algorithm 3.22, properly implemented, is *0((k(n* + m)). That is, the time taken is proportional to the length of the input times the size (nodes plus edges) of the transition graph.

3.7.4 Construction of an NFA from a Regular Expression

We now give an algorithm for converting any regular expression to an NFA that defines the same language. The algorithm is syntax-directed, in the sense that it works recursively up the parse tree for the regular expression. For each subexpression the algorithm constructs an NFA with a single accepting state.

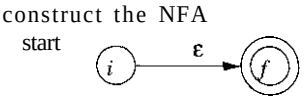
Algorithm 3.23: The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA.

INPUT: A regular expression r over alphabet S.

OUTPUT: An NFA *N* accepting *L(r).*

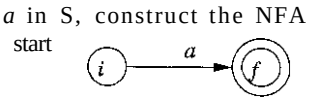
METHOD: Begin by parsing r into its constituent subexpressions. The rules for constructing an NFA consist of basis rules for handling subexpressions with no operators, and inductive rules for constructing larger NFA's from the NFA's for the immediate subexpressions of a given expression.

BASIS: For expression e construct the NFA



Here, *i* is a new state, the start state of this NFA, and / is another new state, the accepting state for the NFA.

For any subexpression *a* in S, construct the NFA



where again *i* and / are new states, the start and accepting states, respectively. Note that in both of the basis constructions, we construct a distinct NFA, with new states, for every occurrence of e or some o as a subexpression of *r*.

INDUCTION: Suppose *N(s)* and *N(t)* are NFA's for regular expressions *s* and *t,* respectively.

a) Suppose *r* = *s\t.* Then *N(r),* the NFA for r, is constructed as in Fig. 3.40. Here, *i* and / are new states, the start and accepting states of *N(r),* respectively. There are e-transitions from *i* to the start states of *N(s)* and *N(t),* and each of their accepting states have e-transitions to the accepting state /. Note that the accepting states of *N(s)* and *N(t)* are not accepting in *N(r).* Since any path from *i* to / must pass through either *N(s)* or *N(t)* exclusively, and since the label of that path is not changed by the e's leaving *i* or entering /, we conclude that *N(r)* accepts *L(s)* U *L(t),* which is the same as *L(r).* That is, Fig. 3.40 is a correct construction for the union operator.

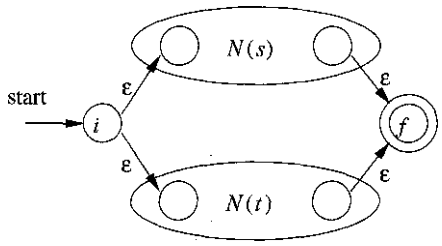


Figure 3.40: NFA for the union of two regular expressions

b) Suppose r = *st.* Then construct *N(r)* as in Fig. 3.41. The start state of *N(s)* becomes the start state of *N(r),* and the accepting state of *N(t)* is the only accepting state of *N(r).* The accepting state of *N(s)* and the start state of *N(t)* are merged into a single state, with all the transitions in or out of either state. A path from *i* to / in Fig. 3.41 must go first through *N(s),* and therefore its label will begin with some string in *L(s).* The path then continues through *N(t),* so the path's label finishes with a string in *L(t).* As we shall soon argue, accepting states never have edges out and start states never have edges in, so it is not possible for a path to re-enter *N(s)* after leaving it. Thus, *N(r)* accepts exactly *L(s)L(i),* and is a correct NFA for *r* = *st.*

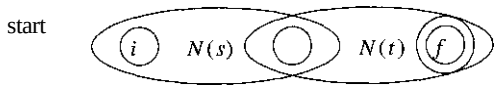


Figure 3.41: NFA for the concatenation of two regular expressions

c) Suppose $r = s^*$. Then for r we construct the NFA $N(r)$ shown in Fig. 3.42. Here, i and / are new states, the start state and lone accepting state of $N(r)$. To get from i to /, we can either follow the introduced path labeled e, which takes care of the one string in $L(s)°$, or we can go to the start state of $N(s)$, through that NFA, then from its accepting state back to its start state zero or more times. These options allow $N(r)$ to accept all the strings in $L(s)^1$, $L(s)^2$, and so on, so the entire set of strings accepted by $N(r)$ is $L(s^*)$.

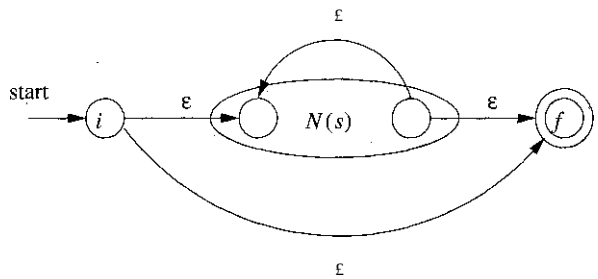


Figure 3.42: NFA for the closure of a regular expression

d) Finally, suppose r = (s). Then $L(r) = L(s)$, and we can use the NFA $N(s)$ as $N(r)$.

•

The method description in Algorithm 3.23 contains hints as to why the inductive construction works as it should. We shall not give a formal correctness proof, but we shall list several properties of the constructed NFA's, in addition to the all-important fact that $N(r)$ accepts language $L(r)$. These properties are interesting in their own right, and helpful in making a formal proof.

1. $N(r)$ has at most twice as many states as there are operators and operands in r. This bound follows from the fact that each step of the algorithm creates at most two new states.

2. $N(r)$ has one start state and one accepting state. The accepting state has no outgoing transitions, and the start state has no incoming transitions.

3. Each state of $N(r)$ other than the accepting state has either one outgoing transition on a symbol in E or two outgoing transitions, both on e.

Example 3.24: Let us use Algorithm 3.23 to construct an NFA for r = (a|b)*abb. Figure 3.43 shows a parse tree for r that is analogous to the parse trees constructed for arithmetic expressions in Section 2.2.3. For subexpression ri, the first a, we construct the NFA:

10

Figure 3.43: Parse tree for **(a|b)*abb**

start

State numbers have been chosen for consistency with what follows. For r_2 we construct:

start

• © -

We can now combine JV(n) and *N(r₂),* using the construction of Fig. 3.40 to obtain the NFA for r_3 = *ri\r₂]* this NFA is shown in Fig. 3.44.

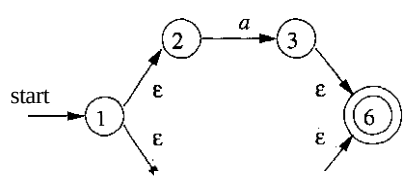


Figure 3.44: NFA for r_3

The NFA for r_4 = (73) is the same as that for 7-3. The NFA for r_5 = (r_3)* is then as shown in Fig. 3.45. We have used the construction in Fig. 3.42 to build this NFA from the NFA in Fig. 3.44.

Now, consider subexpression *r§,* which is another **a.** We use the basis construction for *a* again, but we must use new states. It is not permissible to reuse

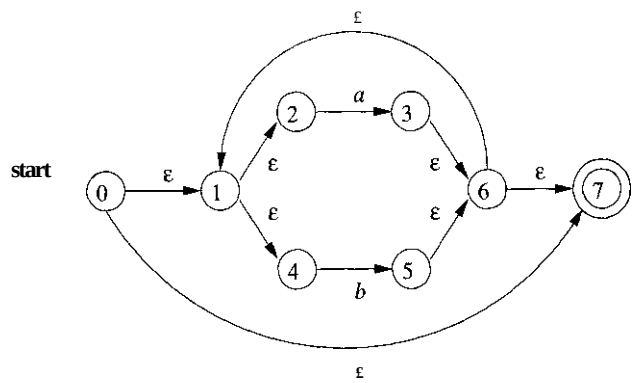
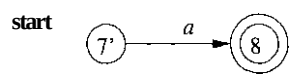


Figure **3.45:** NFA for r_5

the NFA we constructed for *r*\, even though *r*\ and *r%* are the same expression. The NFA for r_6 is:



To obtain the NFA for *rr* = r_5*re,* we apply the construction of Fig. **3.41.** We merge states **7** and **7',** yielding the NFA of Fig. **3.46.** Continuing in this fashion with new NFA's for the two subexpressions **b** called *rg* and rio, we eventually construct the NFA for **(a|b)*abb** that we first met in Fig. **3.34.** •

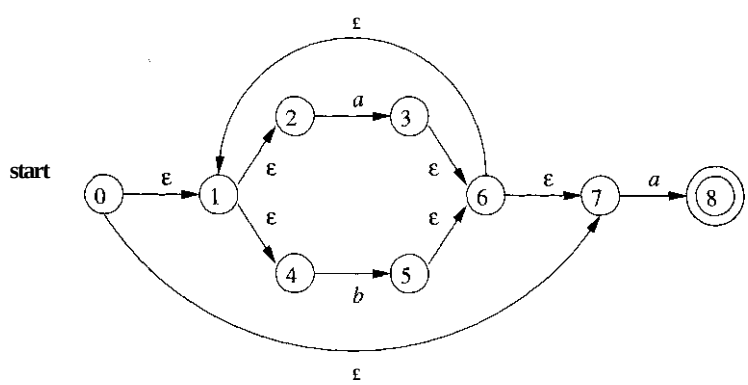


Figure **3.46:** NFA for r_7

3.7.5 Efficiency of String-Processing Algorithms

We observed that Algorithm **3.18** processes a string *x* in time **0(|a?|),** while in Section **3.7.3** we concluded that we could simulate an NFA in time proportional to the product of *x*\ and the size of the NFA's transition graph. Obviously, it