

Compilers

Compilers

Principles, Techniques, & Tools

Second Edition



Aho
Lam
Sethi
Ullman

Second
Edition


Addison
Wesley

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



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 L^AT_EX.

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>.

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 Ke, Wei Li, Mike Smith, Art Stanness, 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
2	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 ϵ -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	109
3.1	The Role of the Lexical Analyzer	109
3.1.1	Lexical Analysis Versus Parsing	110
3.1.2	Tokens, Patterns, and Lexemes	111
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 <i>nullable</i> , <i>firstpos</i> , and <i>lastpos</i>	176
3.9.4	Computing <i>followpos</i>	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 <i>getReg</i>	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 Reg- isters	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
12.7.4	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.

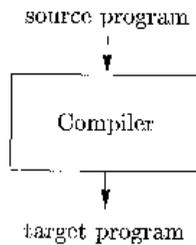


Figure 1.1: A compiler

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.

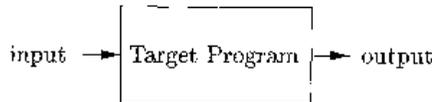


Figure 1.2: Running the target program

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.

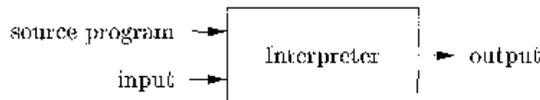


Figure 1.3: An interpreter

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. □

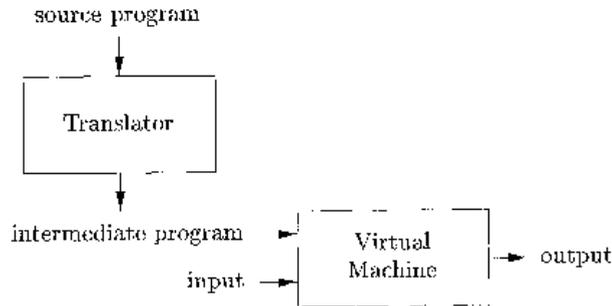


Figure 1.4: A hybrid compiler

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.

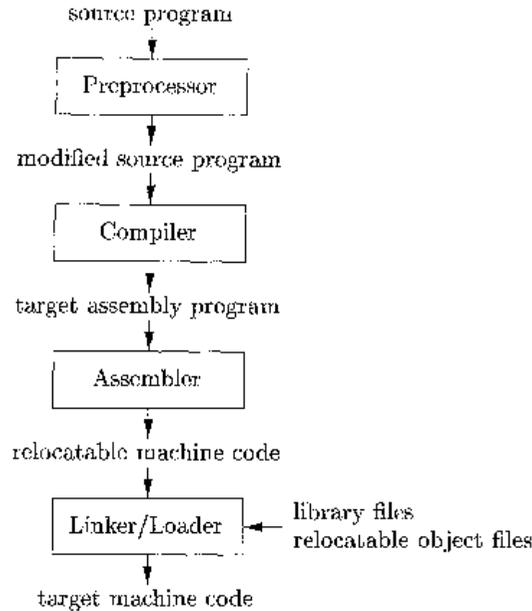


Figure 1.5: A language-processing system

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

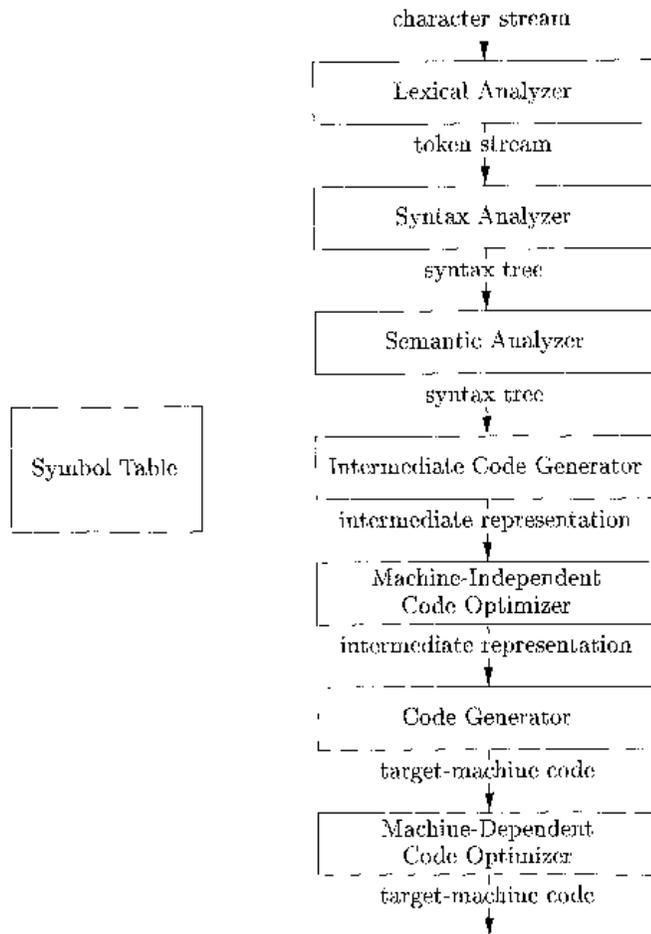


Figure 1.6: Phases of a compiler

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

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

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

$$\textit{position} = \textit{initial} + \textit{rate} * 60 \quad (1.1)$$

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 $\langle \textit{id}, 1 \rangle$, 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 $\langle = \rangle$. 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 $\langle \textit{id}, 2 \rangle$, where 2 points to the symbol-table entry for *initial*.
4. + is a lexeme that is mapped into the token $\langle + \rangle$.
5. *rate* is a lexeme that is mapped into the token $\langle \textit{id}, 3 \rangle$, where 3 points to the symbol-table entry for *rate*.
6. * is a lexeme that is mapped into the token $\langle * \rangle$.
7. 60 is a lexeme that is mapped into the token $\langle 60 \rangle$.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

$$\langle \textit{id}, 1 \rangle \langle = \rangle \langle \textit{id}, 2 \rangle \langle + \rangle \langle \textit{id}, 3 \rangle \langle * \rangle \langle 60 \rangle \quad (1.2)$$

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

¹Technically speaking, for the lexeme 60 we should make up a token like $\langle \textit{number}, 4 \rangle$, 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.

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

```

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = 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. Chapter 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 **id1** so the optimizer can transform (1.3) into the shorter sequence

$$\begin{aligned} \text{t1} &= \text{id3} * 60.0 \\ \text{id1} &= \text{id2} + \text{t1} \end{aligned} \tag{1.4}$$

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 **R1** and **R2**, the intermediate code in (1.4) might get translated into the machine code

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

(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 `R1` and the fourth adds to it the value previously computed in register `R2`. Finally, the value in register `R1` is stored into the address of `id1`, 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 0's and 1'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 Tcl 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. \square

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. \square

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.

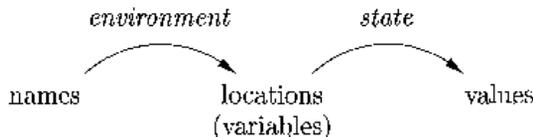


Figure 1.8: Two-stage mapping from names to values

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 (“l-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 l-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 f . When f 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 f , 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 f is executing, uses of i cannot refer to the i that is local to f . 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. \square

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
```

²Technically, 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 `ARRAYSIZE` 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. \square

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 B , 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 B_1, B_2, \dots, B_k be all the blocks that surround this use of x , with B_k the smallest, nested within B_{k-1} , which is nested within B_{k-2} , and so on. Search for the largest i such that there is a declaration of x belonging to B_i . This use of x refers to the declaration in B_i . Alternatively, this use of x is within the scope of the declaration in B_i .

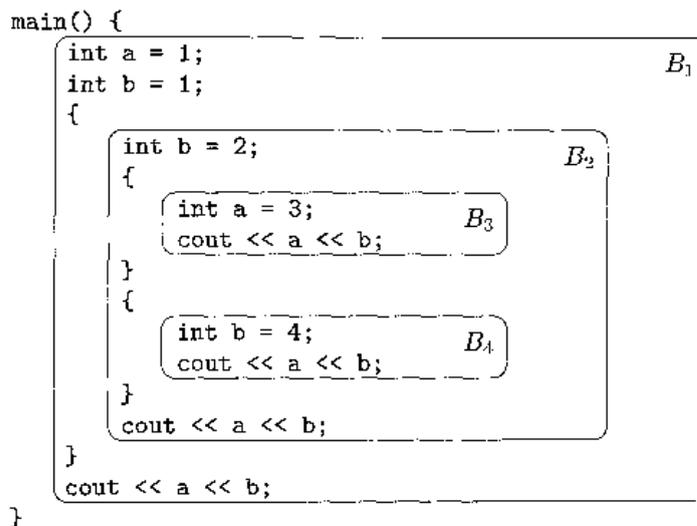


Figure 1.10: Blocks in a C++ program

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 B_1 . Its scope is all of B_1 , except for those blocks nested (perhaps deeply) within B_1 that have their own declaration of a . B_2 , nested immediately within B_1 , does not have a declaration of a , but B_3 does. B_4 does not have a declaration of a , so block B_3 is the only place in the entire program that is outside the scope of the declaration of the name a that belongs to B_1 . That is, this scope includes B_4 and all of B_2 except for the part of B_2 that is within B_3 . 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 B_4 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_4, B_2, B_1 . Note that B_3 does not surround the point in question. B_4 has a declaration of b , so it is to this declaration that this use of b refers, and the value of b printed is 4. However, B_4 does not have a declaration of a , so we next look at B_2 . That block does not have a declaration of a either, so we proceed to B_1 . Fortunately,

DECLARATION	SCOPE
<code>int a = 1;</code>	$B_1 - B_3$
<code>int b = 1;</code>	$B_1 - B_2$
<code>int b = 2;</code>	$B_2 - B_4$
<code>int a = 3;</code>	B_3
<code>int b = 4;</code>	B_4

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. \square

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 `(x + 1)`. But what is `x`? 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 main() { 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 `x`. It is to this declaration that the use of `x` refers.

In the example of Fig. 1.12, the function `main` first calls function `b`. 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=1` 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. \square

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 $x.m()$ 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 $m()$.
3. There is a use of m of the form $x.m()$, 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 . \square

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. \square

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;

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;

```

(a) Code for Exercise 1.6.1

(b) Code for Exercise 1.6.2

Figure 1.13: Block-structured code

```

{   int w, x, y, z;    /* Block B1 */
  {   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 main() { 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.

- ◆ *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.
- ◆ *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.
- ◆ *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.
- ◆ *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.
- ◆ *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.
- ◆ *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.
- ◆ *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*.
- ◆ *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.
- ◆ *Block Structure.* Languages that allow blocks to be nested are said to have *block structure*. A name x 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.
- ◆ *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:  t1 = a [ i ]
3:  if t1 < 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.

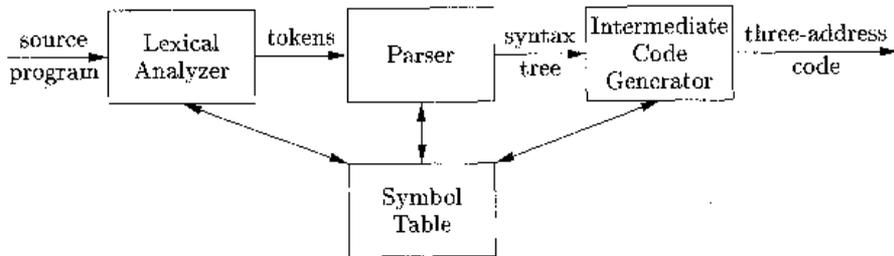
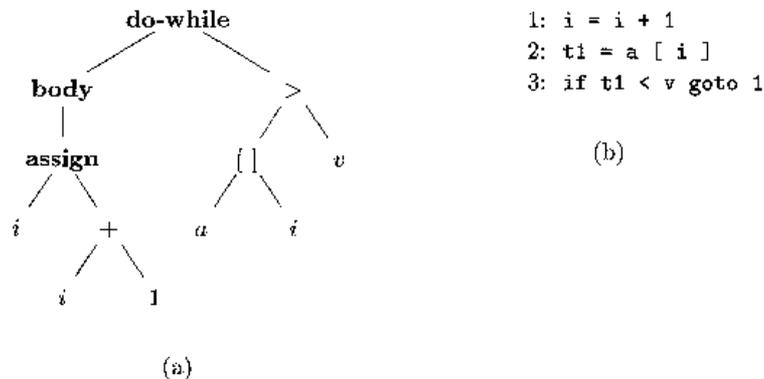


Figure 2.3: A model of a compiler front end

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 $x = y \text{ op } z$, where **op** is a binary operator, y and z 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

$$\mathbf{if} (\text{expression}) \text{statement} \mathbf{else} \text{statement}$$

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

$$\text{stmt} \rightarrow \mathbf{if} (\text{expr}) \text{stmt} \mathbf{else} \text{stmt}$$

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:

$$list \rightarrow list + digit \quad (2.1)$$

$$list \rightarrow list - digit \quad (2.2)$$

$$list \rightarrow digit \quad (2.3)$$

$$digit \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \quad (2.4)$$

¹Individual 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:

$$\mathit{list} \rightarrow \mathit{list} + \mathit{digit} \mid \mathit{list} - \mathit{digit} \mid \mathit{digit}$$

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. \square

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 ϵ , 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*.

\square

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:

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

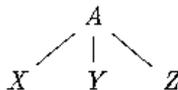
$$\begin{aligned}
 \text{call} &\rightarrow \text{id (optparams)} \\
 \text{optparams} &\rightarrow \text{params} \mid \epsilon \\
 \text{params} &\rightarrow \text{params , param} \mid \text{param}
 \end{aligned}$$

Note that the second possible body for *optparams* (“optional parameter list”) is ϵ , 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. \square

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 $A \rightarrow XYZ$, 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 ϵ .
3. Each interior node is labeled by a nonterminal.
4. If A is the nonterminal labeling some interior node and X_1, X_2, \dots, X_n are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1 X_2 \dots X_n$. Here, X_1, X_2, \dots, 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 $A \rightarrow \epsilon$ is a production, then a node labeled A may have a single child labeled ϵ .

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

$$\textit{list} \rightarrow \textit{list} + \textit{digit}$$

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. \square

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

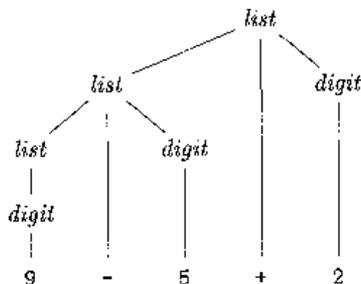


Figure 2.5: Parse tree for 9-5+2 according to the grammar in Example 2.1

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

$$\text{string} \rightarrow \text{string} + \text{string} \mid \text{string} - \text{string} \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

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. \square

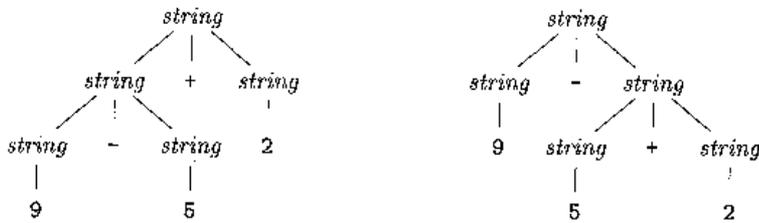


Figure 2.6: Two parse trees for 9-5+2

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:

$$\begin{aligned} \textit{right} &\rightarrow \textit{letter} = \textit{right} \mid \textit{letter} \\ \textit{letter} &\rightarrow \textit{a} \mid \textit{b} \mid \dots \mid \textit{z} \end{aligned}$$

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.

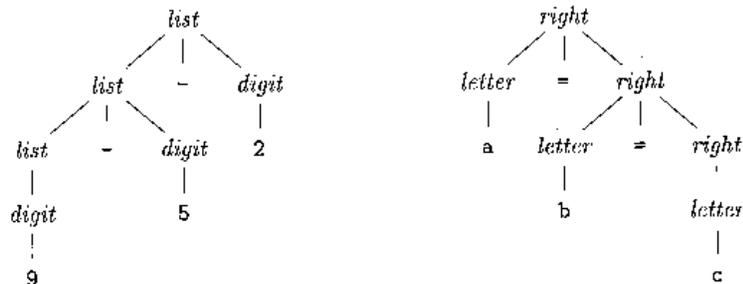


Figure 2.7: Parse trees for left- and right-associative grammars

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.

$$\mathit{factor} \rightarrow \mathbf{digit} \mid (\mathit{expr})$$

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.

$$\begin{aligned} \mathit{term} &\rightarrow \mathit{term} * \mathit{factor} \\ &\mid \mathit{term} / \mathit{factor} \\ &\mid \mathit{factor} \end{aligned}$$

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

$$\begin{aligned} \mathit{expr} &\rightarrow \mathit{expr} + \mathit{term} \\ &\mid \mathit{expr} - \mathit{term} \\ &\mid \mathit{term} \end{aligned}$$

The resulting grammar is therefore

$$\begin{aligned} \mathit{expr} &\rightarrow \mathit{expr} + \mathit{term} \mid \mathit{expr} - \mathit{term} \mid \mathit{term} \\ \mathit{term} &\rightarrow \mathit{term} * \mathit{factor} \mid \mathit{term} / \mathit{factor} \mid \mathit{factor} \\ \mathit{factor} &\rightarrow \mathbf{digit} \mid (\mathit{expr}) \end{aligned}$$

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). \square

Example 2.7: Keywords allow us to recognize statements, since most statements 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 ϵ . 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. \square

$$\begin{array}{l}
 \textit{stmt} \rightarrow \textit{id} = \textit{expression} ; \\
 \quad | \textit{if} (\textit{expression}) \textit{stmt} \\
 \quad | \textit{if} (\textit{expression}) \textit{stmt} \textit{else} \textit{stmt} \\
 \quad | \textit{while} (\textit{expression}) \textit{stmt} \\
 \quad | \textit{do} \textit{stmt} \textit{while} (\textit{expression}) ; \\
 \quad | \{ \textit{stmts} \} \\
 \\
 \textit{stmts} \rightarrow \textit{stmts} \textit{stmt} \\
 \quad | \epsilon
 \end{array}$$

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

$$S \rightarrow S S + \mid S S * \mid a$$

- Show how the string $aa+a*$ can be generated by this grammar.
- Construct a parse tree for this string.
- 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.

- $S \rightarrow 0 S 1 \mid 0 1$
- $S \rightarrow + S S \mid - S S \mid a$
- $S \rightarrow S (S) S \mid \epsilon$
- $S \rightarrow a S b S \mid b S a S \mid \epsilon$
- $S \rightarrow a \mid S + S \mid S S \mid S * \mid (S)$

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.

- Arithmetic expressions in postfix notation.
- Left-associative lists of identifiers separated by commas.
- Right-associative lists of identifiers separated by commas.
- Arithmetic expressions of integers and identifiers with the four binary operators $+$, $-$, $*$, $/$.

- ! c) 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.

$$num \rightarrow 11 \mid 1001 \mid num\ 0 \mid num\ num$$

- 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  $expr_1$ ;
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 $E_1 \text{ op } E_2$, where op is any binary operator, then the postfix notation for E is $E_1' E_2' \text{ op}$, where E_1' and E_2' are the postfix notations for E_1 and E_2 , respectively.
3. If E is a parenthesized expression of the form (E_1) , then the postfix notation for E is the same as the postfix notation for E_1 .

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_1 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. \square

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. \square

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 N 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.

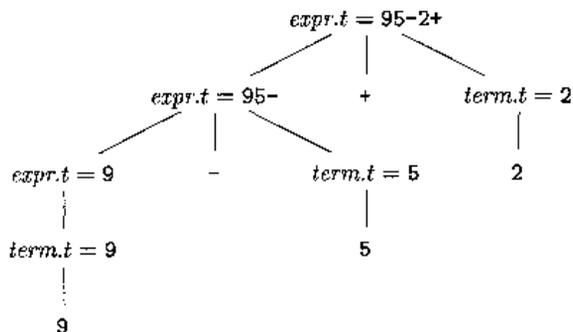


Figure 2.9: Attribute values at nodes in a parse tree

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
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \ term.t \ '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \ term.t \ '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

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 $term \rightarrow 9$ 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 $expr \rightarrow term$ 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

$$expr.t = expr_1.t \| term.t \| '+'$$

associated with this production constructs the value of attribute $expr.t$ by concatenating the postfix forms $expr_1.t$ 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.” \square

³In 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 $A \rightarrow X_1 X_2, \dots, X_n$, 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:

$$\begin{array}{ll}
 \text{PRODUCTION} & \text{SEMANTIC RULE} \\
 \text{expr} \rightarrow \text{expr}_1 + \text{term} & \text{expr.t} = \text{expr}_1.t \parallel \text{term.t} \parallel '+'
 \end{array} \quad (2.5)$$

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. \square

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

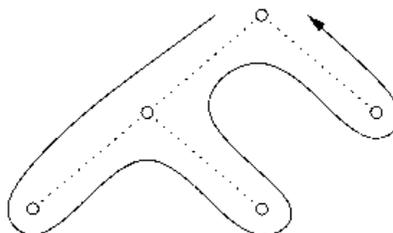


Figure 2.12: Example of a depth-first traversal of a tree

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 N 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 N 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

$$rest \rightarrow + term \{ \text{print}('+') \} rest_1$$

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 $rest_1$ 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

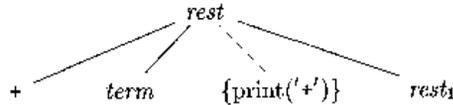


Figure 2.13: An extra leaf is constructed for a semantic action

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.

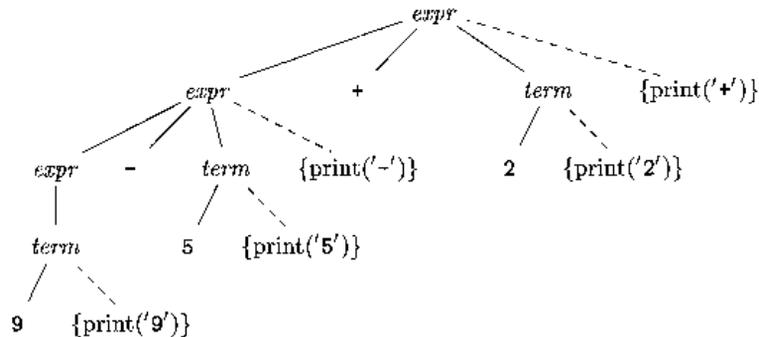


Figure 2.14: Actions translating 9-5+2 into 95-2+

$expr$	\rightarrow	$expr_1 + term$	$\{print('+')\}$
$expr$	\rightarrow	$expr_1 - term$	$\{print('-')\}$
$expr$	\rightarrow	$term$	
$term$	\rightarrow	0	$\{print('0')\}$
$term$	\rightarrow	1	$\{print('1')\}$
		...	
$term$	\rightarrow	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 $expr \rightarrow term$, 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+. \square

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.

