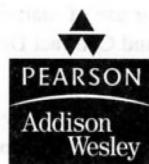


THIRD EDITION

An Introduction to the
Theory of Computer Science

Languages and Machines

Thomas A. Sudkamp
WRIGHT STATE UNIVERSITY



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

Acquisitions Editor Matt Goldstein
Project Editor Katherine Harutunian
Production Supervisor Marilyn Lloyd
Marketing Manager Michelle Brown
Marketing Coordinator Jake Zavracky
Project Management Windfall Software
Composition Windfall Software
Copyeditor Yonie Overton
Technical Illustration Horizon Design
Proofreader Jennifer McClain
Indexer Thomas Sudkamp
Cover Design Manager Joyce Cosentino Wells
Cover Designer Alison R. Paddock
Cover Image © 2005 Nova Development
Prepress and Manufacturing Caroline Fell
Printer Hamilton Printing

Access the latest information about Addison-Wesley titles from our World Wide Web site:
<http://www.aw-bc.com/computing>

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.

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Library of Congress Cataloging-in-Publication Data

Sudkamp, Thomas A.

Languages and machines : an introduction to the theory of computer science / Thomas A. Sudkamp.—3rd ed.

p. cm.

Includes bibliographical references and index.

ISBN 0-321-32221-5 (alk. paper)

1. Formal languages. 2. Machine theory. 3. Computational complexity. I. Title.

QA267.3.S83 2005

511.3—dc22

2004030342

Copyright © 2006 by Pearson Education, Inc.

For information on obtaining permission for use of material in this work, please submit a written request to Pearson Education, Inc., Rights and Contract Department, 75 Arlington Street, Suite 300, Boston, MA 02116 or fax your request to (617) 848-7047.

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 any other media embodiments now known or hereafter to become known, without the prior written permission of the publisher. Printed in the United States of America.

ISBN 0-321-32221-5

1 2 3 4 5 6 7 8 9 10-HAM-08 07 06 05

$\langle dedication \rangle \rightarrow \langle parents \rangle$
 $\langle parents \rangle \rightarrow \langle first\ name \rangle \langle last\ name \rangle$
 $\langle first\ name \rangle \rightarrow Donald \mid Mary$
 $\langle last\ name \rangle \rightarrow Sudkamp$

Contents

2.1	Strings and Languages	42
2.2	Finite Specification of Languages	45
2.3	Regular Sets and Expressions	49
2.4	Regular Expressions and Text Searching	54
	Exercises	58
	Bibliographic Notes	61
Preface		xiii
Introduction		1
PART I Foundations		
Chapter 1		
Mathematical Preliminaries		7
1.1	Set Theory	8
1.2	Cartesian Product, Relations, and Functions	11
1.3	Equivalence Relations	14
1.4	Countable and Uncountable Sets	16
1.5	Diagonalization and Self-Reference	21
1.6	Recursive Definitions	23
1.7	Mathematical Induction	27
1.8	Directed Graphs	32
	Exercises	36
	Bibliographic Notes	40
Chapter 2		
Languages		41
2.1	Strings and Languages	42
2.2	Finite Specification of Languages	45
2.3	Regular Sets and Expressions	49
2.4	Regular Expressions and Text Searching	54
	Exercises	58
	Bibliographic Notes	61

PART II Grammars, Automata, and Languages

Chapter 3		Chapt Proper
Context-Free Grammars	65	6.1
3.1 Context-Free Grammars and Languages	68	6.2
3.2 Examples of Grammars and Languages	76	6.3
3.3 Regular Grammars	81	6.4
3.4 Verifying Grammars	83	6.5
3.5 Leftmost Derivations and Ambiguity	89	6.6
3.6 Context-Free Grammars and Programming Language Definition	93	6.7
Exercises	97	
Bibliographic Notes	102	
Chapter 4		Chapte Pushdo
Normal Forms for Context-Free Grammars	103	7.1
4.1 Grammar Transformations	104	7.2
4.2 Elimination of λ -Rules	106	7.3
4.3 Elimination of Chain Rules	113	7.4
4.4 Useless Symbols	116	7.5
4.5 Chomsky Normal Form	121	
4.6 The CYK Algorithm	124	
4.7 Removal of Direct Left Recursion	129	
4.8 Greibach Normal Form	131	
Exercises	138	
Bibliographic Notes	143	
Chapter 5		PART
Finite Automata	145	Chapter Turing M
5.1 A Finite-State Machine	145	8.1
5.2 Deterministic Finite Automata	147	8.2
5.3 State Diagrams and Examples	151	8.3
5.4 Nondeterministic Finite Automata	159	8.4
5.5 λ -Transitions	165	8.5
5.6 Removing Nondeterminism	170	8.6
5.7 DFA Minimization	178	8.7
Exercises	184	8.8
Bibliographic Notes	190	

	Chapter 6	
	Properties of Regular Languages	191
65	6.1 Finite-State Acceptance of Regular Languages	191
	6.2 Expression Graphs	193
	6.3 Regular Grammars and Finite Automata	196
	6.4 Closure Properties of Regular Languages	200
	6.5 A Nonregular Language	203
	6.6 The Pumping Lemma for Regular Languages	205
	6.7 The Myhill-Nerode Theorem	211
	Exercises	217
	Bibliographic Notes	220
	Chapter 7	
	Pushdown Automata and Context-Free Languages	221
103	7.1 Pushdown Automata	221
	7.2 Variations on the PDA Theme	227
	7.3 Acceptance of Context-Free Languages	232
	7.4 The Pumping Lemma for Context-Free Languages	239
	7.5 Closure Properties of Context-Free Languages	243
	Exercises	247
	Bibliographic Notes	251

PART III Computability

	Chapter 8	
	Turing Machines	255
145	8.1 The Standard Turing Machine	255
	8.2 Turing Machines as Language Acceptors	259
	8.3 Alternative Acceptance Criteria	262
	8.4 Multitrack Machines	263
	8.5 Two-Way Tape Machines	265
	8.6 Multitape Machines	268
	8.7 Nondeterministic Turing Machines	274
	8.8 Turing Machines as Language Enumerators	282
	Exercises	288
	Bibliographic Notes	293

Chapter 9		12.7
Turing Computable Functions	295	
9.1 Computation of Functions	295	
9.2 Numeric Computation	299	
9.3 Sequential Operation of Turing Machines	301	
9.4 Composition of Functions	308	
9.5 Uncomputable Functions	312	
9.6 Toward a Programming Language	313	
Exercises	320	
Bibliographic Notes	323	
Chapter 10		13.1
The Chomsky Hierarchy	325	13.2
10.1 Unrestricted Grammars	325	13.3
10.2 Context-Sensitive Grammars	332	13.4
10.3 Linear-Bounded Automata	334	13.5
10.4 The Chomsky Hierarchy	338	13.6
Exercises	339	13.7
Bibliographic Notes	341	13.8
Part		
Chapter 11		Chapter
Decision Problems and the Church-Turing Thesis	343	Time C
11.1 Representation of Decision Problems	344	14.1
11.2 Decision Problems and Recursive Languages	346	14.2
11.3 Problem Reduction	348	14.3
11.4 The Church-Turing Thesis	352	14.4
11.5 A Universal Machine	354	14.5
Exercises	358	14.6
Bibliographic Notes	360	14.7
Chapter 12		Chapter
Undecidability	361	P, NP,
12.1 The Halting Problem for Turing Machines	362	15.1
12.2 Problem Reduction and Undecidability	365	15.2
12.3 Additional Halting Problem Reductions	368	15.3
12.4 Rice's Theorem	371	15.4
12.5 An Unsolvable Word Problem	373	15.5
12.6 The Post Correspondence Problem	377	

295	12.7 Undecidable Problems in Context-Free Grammars 382 Exercises 386 Bibliographic Notes 388
Chapter 13	
Mu-Recursive Functions 389	
325	13.1 Primitive Recursive Functions 389 13.2 Some Primitive Recursive Functions 394 13.3 Bounded Operators 398 13.4 Division Functions 404 13.5 Gödel Numbering and Course-of-Values Recursion 406 13.6 Computable Partial Functions 410 13.7 Turing Computability and Mu-Recursive Functions 415 13.8 The Church-Turing Thesis Revisited 421 Exercises 424 Bibliographic Notes 430
PART IV Computational Complexity	
Chapter 14	
Time Complexity 433	
343	14.1 Measurement of Complexity 434 14.2 Rates of Growth 436 14.3 Time Complexity of a Turing Machine 442 14.4 Complexity and Turing Machine Variations 446 14.5 Linear Speedup 448 14.6 Properties of Time Complexity of Languages 451 14.7 Simulation of Computer Computations 458 Exercises 462 Bibliographic Notes 464
Chapter 15	
\mathcal{P}, \mathcal{NP}, and Cook's Theorem 465	
361	15.1 Time Complexity of Nondeterministic Turing Machines 466 15.2 The Classes \mathcal{P} and \mathcal{NP} 468 15.3 Problem Representation and Complexity 469 15.4 Decision Problems and Complexity Classes 472 15.5 The Hamiltonian Circuit Problem 474

X Contents

15.6	Polynomial-Time Reduction	477	18.5
15.7	$P = NP?$	479	
15.8	The Satisfiability Problem	481	
15.9	Complexity Class Relations	492	
	Exercises	493	
	Bibliographic Notes	496	
Chapter 16			
NP-Complete Problems			497
16.1	Reduction and NP-Complete Problems	497	19.1
16.2	The 3-Satisfiability Problem	498	19.2
16.3	Reductions from 3-Satisfiability	500	19.3
16.4	Reduction and Subproblems	513	19.4
16.5	Optimization Problems	517	19.5
16.6	Approximation Algorithms	519	19.6
16.7	Approximation Schemes	523	19.7
	Exercises	526	19.8
	Bibliographic Notes	528	
Chapter 17			
Additional Complexity Classes			529
17.1	Derivative Complexity Classes	529	20.1
17.2	Space Complexity	532	20.2
17.3	Relations between Space and Time Complexity	535	20.3
17.4	P -Space, NP -Space, and Savitch's Theorem	540	20.4
17.5	P -Space Completeness	544	20.5
17.6	An Intractable Problem	548	
	Exercises	550	
	Bibliographic Notes	551	

PART V Deterministic Parsing

Chapter 18

Parsing: An Introduction

555

18.1	The Graph of a Grammar	555
18.2	A Top-Down Parser	557
18.3	Reductions and Bottom-Up Parsing	561
18.4	A Bottom-Up Parser	563

	18.5 Parsing and Compiling	567	
	Exercises	568	
	Bibliographic Notes	569	
	Chapter 19		
	LL(k) Grammars		571
497	19.1 Lookahead in Context-Free Grammars	571	
	19.2 FIRST, FOLLOW, and Lookahead Sets	576	
	19.3 Strong LL(k) Grammars	579	
	19.4 Construction of FIRST _{k} Sets	580	
	19.5 Construction of FOLLOW _{k} Sets	583	
	19.6 A Strong LL(1) Grammar	585	
	19.7 A Strong LL(k) Parser	587	
	19.8 LL(k) Grammars	589	
	Exercises	591	
	Bibliographic Notes	593	
	Chapter 20		
	LR(k) Grammars		595
529	20.1 LR(0) Contexts	595	
	20.2 An LR(0) Parser	599	
	20.3 The LR(0) Machine	601	
	20.4 Acceptance by the LR(0) Machine	606	
	20.5 LR(1) Grammars	612	
	Exercises	620	
	Bibliographic Notes	621	
	Appendix I		
	Index of Notation		623
	Appendix II		
	The Greek Alphabet		627
	Appendix III		
	The ASCII Character Set		629
555	Appendix IV		
	Backus-Naur Form Definition of Java		631
	Bibliography		641
	Subject Index		649

Preface

Introduction to Computer and Information Science, Second Edition, by Michael Sipser, is now available in its third edition. This book is intended for introductory courses in computer science, particularly those that focus on the theory of computation. It is also suitable for self-study or as a reference for anyone interested in the mathematical foundations of computing. The book covers a wide range of topics, including automata theory, formal languages, computability, complexity, and algorithms. It includes many examples and exercises, and provides a comprehensive treatment of the subject matter.

The objective of the third edition of *Languages and Machines: An Introduction to the Theory of Computer Science* remains the same as that of the first two editions, to provide a mathematically sound presentation of the theory of computer science at a level suitable for junior- and senior-level computer science majors. The impetus for the third edition was threefold: to enhance the presentation by providing additional motivation and examples; to expand the selection of topics, particularly in the area of computational complexity; and to provide additional flexibility to the instructor in the design of an introductory course in the theory of computer science.

While many applications-oriented students question the importance of studying theoretical foundations, it is this subject that addresses the "big picture" issues of computer science. When today's programming languages and computer architectures are obsolete and solutions have been found for problems currently of interest, the questions considered in this book will still be relevant. What types of patterns can be algorithmically detected? How can languages be formally defined and analyzed? What are the inherent capabilities and limitations of algorithmic computation? What problems have solutions that require so much time or memory that they are realistically intractable? How do we compare the relative difficulty of two problems? Each of these questions will be addressed in this text.

Organization

Since most computer science students at the undergraduate level have little or no background in abstract mathematics, the presentation is intended not only to introduce the foundations of computer science but also to increase the student's mathematical sophistication. This is accomplished by a rigorous presentation of the concepts and theorems of the subject accompanied by a generous supply of examples. Each chapter ends with a set of exercises that reinforces and augments the material covered in the chapter.

To make the topics accessible, no special mathematical prerequisites are assumed. Instead, Chapter 1 introduces the mathematical tools of the theory of computing: naive set

precise and unambiguous definitions of the concepts, structures, and operations. The following notational conventions will be used throughout the book:

Items	Description	Examples
Elements and strings	Italic lowercase letters from the beginning of the alphabet	<i>a, b, abc</i>
Functions	Italic lowercase letters	<i>f, g, h</i>
Sets and relations	Capital letters	X, Y, Z, Σ , Γ
Grammars	Capital letters	G, G_1 , G_2
Variables of grammars	Italic capital letters	A, B, C, S
Abstract machines	Capital letters	M, M_1 , M_2

The use of roman letters for sets and mathematical structures is somewhat nonstandard but was chosen to make the components of a structure visually identifiable. For example, a context-free grammar is a structure $G = (\Sigma, V, P, S)$. From the fonts alone it can be seen that G consists of three sets and a variable S .

A three-part numbering system is used throughout the book; a reference is given by chapter, section, and item. One numbering sequence records definitions, lemmas, theorems, corollaries, and algorithms. A second sequence is used to identify examples. Tables, figures, and exercises are referenced simply by chapter and number.

The end of a proof is marked by ■ and the end of an example by □. An index of symbols, including descriptions and the numbers of the pages on which they are introduced, is given in Appendix I.

Supplements

Solutions to selected exercises are available only to qualified instructors. Please contact your local Addison-Wesley sales representative or send email to aw.cse@aw.com for information on how to access them.

Acknowledgments

First and foremost, I would like to thank my wife Janice and daughter Elizabeth, whose kindness, patience, and consideration made the successful completion of this book possible. I would also like to thank my colleagues and friends at the Institut de Recherche en Informatique de Toulouse, Université Paul Sabatier, Toulouse, France. The first draft of this revision was completed while I was visiting IRIT during the summer of 2004. A special thanks to Didier Dubois and Henri Prade for their generosity and hospitality.

The number of people who have made contributions to this book increases with each edition. I extend my sincere appreciation to all the students and professors who have

l-
used this book and have sent me critiques, criticisms, corrections, and suggestions for improvement. Many of the suggestions have been incorporated into this edition. Thank you for taking the time to send your comments and please continue to do so. My email address is *tsudkamp@cs.wright.edu*.

This book, in its various editions, has been reviewed by a number of distinguished computer scientists including Professors Andrew Astromoff (San Francisco State University), Dan Cooke (University of Texas-El Paso), Thomas Fernandez, Sandeep Gupta (Arizona State University), Raymond Gumb (University of Massachusetts-Lowell), Thomas F. Hain (University of South Alabama), Michael Harrison (University of California at Berkeley), David Hemmendinger (Union College), Steve Homer (Boston University), Dan Jurca (California State University-Hayward), Klaus Kaiser (University of Houston), C. Kim (University of Oklahoma), D. T. Lee (Northwestern University), Karen Lemone (Worcester Polytechnic Institute), C. L. Liu (University of Illinois at Urbana-Champaign), Richard J. Lorentz (California State University-Northridge), Fletcher R. Norris (The University of North Carolina at Wilmington), Jeffery Shallit (University of Waterloo), Frank Stomp (Wayne State University), William Ward (University of South Alabama), Dan Ventura (Brigham Young University), Charles Wallace (Michigan Technological University), Kenneth Williams (Western Michigan University), and Hsu-Chun Yen (Iowa State University). Thank you all.

I would also like to gratefully acknowledge the assistance received from the people at the Computer Science Education Division of the Addison-Wesley Publishing Company and Windfall Software who were members of the team that successfully completed this project.

Thomas A. Sudkamp
Dayton, Ohio

Introduction

The theory of computer science began with the questions that spur most scientific endeavors: *how* and *what*. After these had been answered, the question that motivates many economic decisions, *how much*, came to the forefront. The objective of this book is to explain the significance of these questions for the study of computer science and provide answers whenever possible.

Formal language theory was initiated by the question, “How are languages defined?” In an attempt to capture the structure and nuances of natural language, linguist Noam Chomsky developed formal systems called *grammars* for defining and generating syntactically correct sentences. At approximately the same time, computer scientists were grappling with the problem of explicitly and unambiguously defining the syntax of programming languages. These two studies converged when the syntax of the programming language ALGOL was defined using a formalism equivalent to a context-free grammar.

The investigation of computability was motivated by two fundamental questions: “What is an algorithm?” and “What are the capabilities and limitations of algorithmic computation?” An answer to the first question requires a formal model of computation. It may seem that the combination of a computer and high-level programming language, which clearly constitute a computational system, would provide the ideal framework for the study of computability. Only a little consideration is needed to see difficulties with this approach. What computer? How much memory should it have? What programming language? Moreover, the selection of a particular computer or language may have inadvertent and unwanted consequences on the answer to the second question. A problem that may be solved on one computer configuration may not be solvable on another.

The question of whether a problem is algorithmically solvable should be independent of the model computation used: Either there is an algorithmic solution to a problem or there is no such solution. Consequently, a system that is capable of performing all possible algorithmic computations is needed to appropriately address the question of computability. The characterization of general algorithmic computation has been a major area of research for mathematicians and logicians since the 1930s. Many different systems have been proposed as models of computation, including recursive functions, the lambda calculus of Alonzo

2 Introduction

Church, Markov systems, and the abstract machines developed by Alan Turing. All of these systems, and many others designed for this purpose, have been shown to be capable of solving the same set of problems. One interpretation of the Church-Turing Thesis, which will be discussed in Chapter 11, is that a problem has an algorithmic solution only if it can be solved in any (and hence all) of these computational systems.

Because of its simplicity and the similarity of its components to those of a modern day computer, we will use the Turing machine as our framework for the study of computation. The Turing machine has many features in common with a computer: It processes input, writes to memory, and produces output. Although Turing machine instructions are primitive compared with those of a computer, it is not difficult to see that the computation of a computer can be simulated by an appropriately defined sequence of Turing machine instructions. The Turing machine model does, however, avoid the physical limitations of conventional computers; there is no upper bound on the amount of memory or time that may be used in a computation. Consequently, any problem that can be solved on a computer can be solved with a Turing machine, but the converse of this is not guaranteed.

After accepting the Turing machine as a universal model of effective computation, we can address the question, "What are the capabilities and limitations of algorithmic computation?" The Church-Turing Thesis assures us that a problem is solvable only if there is a suitably designed Turing machine that solves it. To show that a problem has no solution reduces to demonstrating that no Turing machine can be designed to solve the problem. Chapter 12 follows this approach to show that several important questions concerning our ability to predict the outcome of a computation are unsolvable.

Once a problem is known to be solvable, one can begin to consider the efficiency or optimality of a solution. The question *how much* initiates the study of computational complexity. Again the Turing machine provides an unbiased platform that permits the comparison of the resource requirements of various problems. The time complexity of a Turing machine measures the number of instructions required by a computation. Time complexity is used to partition the set of solvable problems into two classes: tractable and intractable. A problem is considered tractable if it is solvable by a Turing machine in which the number of instructions executed during a computation is bounded by a polynomial function of length of the input. A problem that is not solvable in polynomial time is considered intractable because of the excessive amount of computational resources required to solve all but the simplest cases of the problem.

The Turing machine is not the only abstract machine that we will consider; rather, it is the culmination of a series of increasingly powerful machines whose properties will be examined. The analysis of effective computation begins with an examination of the properties of deterministic finite automata. A deterministic finite automaton is a read-once machine in which the instruction to be executed is determined by the state of the machine and the input symbol being processed. Although structurally simple, deterministic finite automata have applications in many disciplines including pattern recognition, the design of switching circuits, and the lexical analysis of programming languages.

A more powerful family of machines, known as pushdown automata, are created by adding an external stack memory to finite automata. The addition of the stack extends the

computational capabilities of a finite automaton. As with the Turing machines, our study of computability will characterize the computational capabilities of both of these families of machines.

Language definition and computability, the dual themes of this book, are not two unrelated topics that fall under the broad heading of computer science theory, but rather they are inextricably intertwined. The computations of a machine can be used to recognize a language; an input string is accepted by the machine if the computation initiated with the string indicates its syntactic correctness. Thus each machine has an associated language, the set of strings accepted by the machine. The computational capabilities of each family of abstract machines is characterized by the languages accepted by the machines in the family. With this in mind, we begin our investigations into the related topics of language definition and effective computation.

PART I

Foundations

Theoretical computer science includes the study of language definition, pattern recognition, the capabilities and limitations of algorithmic computation, and the analysis of the complexity of problems and their solutions. These topics are built on the foundations of set theory and discrete mathematics. Chapter 1 reviews the mathematical concepts, operations, and notation required for the study of formal language theory and the theory of computation.

Formal language theory has its roots in linguistics, mathematical logic, and computer science. A set-theoretic definition of language is given in Chapter 2. This definition is sufficiently broad to include both natural (spoken and written) languages and formal languages, but the generality is gained at the expense of not providing an effective method for generating the strings of a language. To overcome this shortcoming, recursive definitions and set operations are used to give finite specifications of languages. This is followed by the introduction of regular sets, a family of languages that arises in automata theory, formal language theory, switching circuits, and neural networks. The section ends with an example of the use of regular expressions—a shorthand notation for regular sets—in describing patterns for searching text.

Mathematical Preliminaries

Set theory and discrete mathematics provide the mathematical foundation for formal language theory, computability theory, and the analysis of computational complexity. We begin our study of these topics with a review of the notation and basic operations of set theory. Cardinality measures the size of a set and provides a precise definition of an infinite set. One of the interesting results of the investigations into the properties of sets by German mathematician Georg Cantor is that there are different sizes of infinite sets. While Cantor's work showed that there is a complete hierarchy of sizes of infinite sets, it is sufficient for our purposes to divide infinite sets into two classes: countable and uncountable. A set is countably infinite if it has the same number of elements as the set of natural numbers. Sets with more elements than the natural numbers are uncountable.

In this chapter we will use a construction known as the *diagonalization argument* to show that the set of functions defined on the natural numbers is uncountably infinite. After we have agreed upon what is meant by the terms *effective procedure* and *computable function* (reaching this consensus is a major goal of Part III of this book), we will be able to determine the size of the set of functions that can be algorithmically computed. A comparison of the sizes of these two sets will establish the existence of functions whose values cannot be computed by any algorithmic process.

While a set may consist of an arbitrary collection of objects, we are interested in sets whose elements can be mechanically produced. Recursive definitions are introduced to generate the elements of a set. The relationship between recursively generated sets and mathematical induction is developed, and induction is shown to provide a general proof technique for establishing properties of elements in recursively generated infinite sets.

This chapter ends with a review of directed graphs and trees, structures that will be used throughout the book to graphically illustrate the concepts of formal language theory and the theory of computation.

1.1 Set Theory

We assume that the reader is familiar with the notions of elementary set theory. In this section, the concepts and notation of that theory are briefly reviewed. The symbol \in signifies membership; $x \in X$ indicates that x is a member or element of the set X . A slash through a symbol represents *not*, so $x \notin X$ signifies that x is not a member of X . Two sets are equal if they contain the same members. Throughout this book, sets are denoted by capital letters. In particular, X , Y , and Z are used to represent arbitrary sets. Italics are used to denote the elements of a set. For example, symbols and strings of the form a , b , A , B , $aaaa$, and abc represent elements of sets.

Brackets { } are used to indicate a set definition. Sets with a small number of members can be defined explicitly; that is, their members can be listed. The sets

$$X = \{1, 2, 3\}$$

$$Y = \{a, b, c, d, e\}$$

are defined in an explicit manner. Sets having a large finite or infinite number of members must be defined implicitly. A set is defined implicitly by specifying conditions that describe the elements of the set. The set consisting of all perfect squares is defined by

$$\{n \mid n = m^2 \text{ for some natural number } m\}.$$

The vertical bar $|$ in an implicit definition is read “such that.” The entire definition is read “the set of n such that n equals m squared for some natural number m .”

The previous example mentioned the set of **natural numbers**. This important set, denoted N , consists of the numbers $0, 1, 2, 3, \dots$. The **empty set**, denoted \emptyset , is the set that has no members and can be defined explicitly by $\emptyset = \{\}$.

A set is determined completely by its membership; the order in which the elements are presented in the definition is immaterial. The explicit definitions

$$X = \{1, 2, 3\}, Y = \{2, 1, 3\}, Z = \{1, 3, 2, 2, 2\}$$

describe the same set. The definition of Z contains multiple instances of the number 2. Repetition in the definition of a set does not affect the membership. Set equality requires that the sets have exactly the same members, and this is the case; each of the sets X , Y , and Z has the natural numbers 1, 2, and 3 as its members.

A set Y is a **subset** of X , written $Y \subseteq X$, if every member of Y is also a member of X . The empty set is trivially a subset of every set. Every set X is a subset of itself. If Y is a

will be
theory

subset of X and $Y \neq X$, then Y is called a **proper subset** of X . The set of all subsets of X is called the **power set** of X and is denoted $\mathcal{P}(X)$.

Example 1.1.1

Let $X = \{1, 2, 3\}$. The subsets of X are

$$\begin{array}{cccc} \emptyset & \{1\} & \{2\} & \{3\} \\ \{1, 2\} & \{2, 3\} & \{3, 1\} & \{1, 2, 3\}. \end{array}$$

□

Set operations are used to construct new sets from existing ones. The **union** of two sets is defined by

$$X \cup Y = \{z \mid z \in X \text{ or } z \in Y\}.$$

The *or* is inclusive. This means that z is a member of $X \cup Y$ if it is a member of X or Y or both. The **intersection** of two sets is the set of elements common to both. This is defined by

$$X \cap Y = \{z \mid z \in X \text{ and } z \in Y\}.$$

Two sets whose intersection is empty are said to be **disjoint**. The union and intersection of n sets, X_1, X_2, \dots, X_n , are defined by

$$\bigcup_{i=1}^n X_i = X_1 \cup X_2 \cup \dots \cup X_n = \{x \mid x \in X_i, \text{ for some } i = 1, 2, \dots, n\}$$

$$\bigcap_{i=1}^n X_i = X_1 \cap X_2 \cap \dots \cap X_n = \{x \mid x \in X_i, \text{ for all } i = 1, 2, \dots, n\},$$

respectively.

Subsets X_1, X_2, \dots, X_n of a set X are said to **partition** X if

- i) $X = \bigcup_{i=1}^n X_i$
- ii) $X_i \cap X_j = \emptyset$, for $1 \leq i, j \leq n$, and $i \neq j$.

For example, the set of even natural numbers (zero is considered even) and the set of odd natural numbers partition \mathbb{N} .

The **difference** of sets X and Y , $X - Y$, consists of the elements of X that are not in Y :

$$X - Y = \{z \mid z \in X \text{ and } z \notin Y\}.$$

Let X be a subset of a universal set U . The **complement** of X with respect to U is the set of elements in U but not in X . In other words, the complement of X with respect to U is the set $U - X$. When the universe U is known, the complement of X with respect to U is denoted \bar{X} . The following identities, known as *DeMorgan's Laws*, exhibit the relationships

between union, intersection, and complement when X and Y are subsets of a set U and complementation is taken with respect to U :

- i) $\overline{(X \cup Y)} = \overline{X} \cap \overline{Y}$
- ii) $\overline{(X \cap Y)} = \overline{X} \cup \overline{Y}$.

Example 1.1.2

Let $X = \{0, 1, 2, 3\}$, $Y = \{2, 3, 4, 5\}$, and let \overline{X} and \overline{Y} denote the complement of X and Y with respect to \mathbb{N} . Then

$$\begin{array}{ll} X \cup Y = \{0, 1, 2, 3, 4, 5\} & \overline{X} = \{n \mid n > 3\} \\ X \cap Y = \{2, 3\} & \overline{Y} = \{0, 1\} \cup \{n \mid n > 5\} \\ X - Y = \{0, 1\} & \overline{X} \cap \overline{Y} = \{n \mid n > 5\} \\ Y - X = \{4, 5\} & \overline{(X \cup Y)} = \{n \mid n > 5\} \end{array}$$

The final two sets in the right-hand column exhibit the equality required by DeMorgan's Law. \square

The definition of subset provides the method for proving that a set X is a subset of Y ; we must show that every element of X is also an element of Y . When X is finite, we can explicitly check each element of X for membership in Y . When X contains infinitely many elements, a different approach is needed. The strategy is to show that an arbitrary element of X is in Y .

Example 1.1.3

We will show that $X = \{8n - 1 \mid n > 0\}$ is a subset of $Y = \{2m + 1 \mid m \text{ is odd}\}$. To gain a better understanding of the sets X and Y , it is useful to generate some of the elements of X and Y :

$$\begin{aligned} X: 8 \cdot 1 - 1 &= 7, 8 \cdot 2 - 1 = 15, 8 \cdot 3 - 1 = 23, 8 \cdot 4 - 1 = 31, \dots \\ Y: 2 \cdot 1 + 1 &= 3, 2 \cdot 3 + 1 = 7, 2 \cdot 5 + 1 = 11, 2 \cdot 7 + 1 = 13, \dots \end{aligned}$$

To establish the inclusion, we must show that every element of X is also an element of Y . An arbitrary element x of X has the form $8n - 1$, for some $n > 0$. Let $m = 4n - 1$. Then m is an odd natural number and

$$\begin{aligned} 2m + 1 &= 2(4n - 1) + 1 \\ &= 8n - 2 + 1 \\ &= 8n - 1 \\ &= x. \end{aligned}$$

Thus x is also in Y and $X \subseteq Y$. \square

$Y \subseteq$
Wh
and

Exa

We p

are e
 $x = n$

Lettin
of the
W
Factor
and the
Si

1.2

The Ca
elemen
defined

A b
can be u
of $N \times$

The nota
 $[1, 1] \not\subseteq 1$

t U and

Set equality can be defined using set inclusion; sets X and Y are equal if $X \subseteq Y$ and $Y \subseteq X$. This simply states that every element of X is also an element of Y and vice versa. When establishing the equality of two sets, the two inclusions are usually proved separately and combined to yield the equality.

Example 1.1.4

X and Y

We prove that the sets

$$X = \{n \mid n = m^2 \text{ for some natural number } m > 0\}$$

$$Y = \{n^2 + 2n + 1 \mid n \geq 0\}$$

are equal. First, we show that every element of X is also an element of Y. Let $x \in X$; then $x = m^2$ for some natural number $m > 0$. Let m_0 be that number. Then x can be written

$$\begin{aligned} x &= (m_0)^2 \\ &= (m_0 - 1 + 1)^2 \\ &= (m_0 - 1)^2 + 2(m_0 - 1) + 1. \end{aligned}$$

Morgan's

□

subset of Y;
e, we can
tely many
y element

Letting $n = m_0 - 1$, we see that $x = n^2 + 2n + 1$ with $n \geq 0$. Consequently, x is a member of the set Y.

We now establish the opposite inclusion. Let $y = (n_0)^2 + 2n_0 + 1$ be an element of Y. Factoring yields $y = (n_0 + 1)^2$. Thus y is the square of a natural number greater than zero and therefore an element of X.

Since $X \subseteq Y$ and $Y \subseteq X$, we conclude that $X = Y$.

□

To gain a
nents of X

1.2 Cartesian Product, Relations, and Functions

The **Cartesian product** is a set operation that builds a set consisting of ordered pairs of elements from two existing sets. The Cartesian product of sets X and Y, denoted $X \times Y$, is defined by

$$X \times Y = \{(x, y) \mid x \in X \text{ and } y \in Y\}.$$

ment of Y.
1. Then m

A **binary relation** on X and Y is a subset of $X \times Y$. The ordering of the natural numbers can be used to generate a relation LT (less than) on the set $N \times N$. This relation is the subset of $N \times N$ defined by

$$LT = \{(i, j) \mid i < j \text{ and } i, j \in N\}.$$

□

The notation $[i, j] \in LT$ indicates that i is less than j , for example, $[0, 1], [0, 2] \in LT$ and $[1, 1] \notin LT$.

The Cartesian product can be generalized to construct new sets from any finite number of sets. If x_1, x_2, \dots, x_n are n elements, then $[x_1, x_2, \dots, x_n]$ is called an **ordered n -tuple**. An ordered pair is simply another name for an ordered 2-tuple. Ordered 3-tuples, 4-tuples, and 5-tuples are commonly referred to as triples, quadruples, and quintuples, respectively. The Cartesian product of n sets X_1, X_2, \dots, X_n is defined by

$$X_1 \times X_2 \times \cdots \times X_n = \{[x_1, x_2, \dots, x_n] \mid x_i \in X_i, \text{ for } i = 1, 2, \dots, n\}.$$

An **n -ary relation** on X_1, X_2, \dots, X_n is a subset of $X_1 \times X_2 \times \cdots \times X_n$. 1-ary, 2-ary, and 3-ary relations are called *unary*, *binary*, and *ternary*, respectively.

Example 1.2.1

Let $X = \{1, 2, 3\}$ and $Y = \{a, b\}$. Then

- a) $X \times Y = \{[1, a], [1, b], [2, a], [2, b], [3, a], [3, b]\}$
- b) $Y \times X = \{[a, 1], [a, 2], [a, 3], [b, 1], [b, 2], [b, 3]\}$
- c) $Y \times Y = \{[a, a], [a, b], [b, a], [b, b]\}$
- d) $X \times Y \times Y = \{[1, a, a], [1, b, a], [2, a, a], [2, b, a], [3, a, a], [3, b, a], [1, a, b], [1, b, b], [2, a, b], [2, b, b], [3, a, b], [3, b, b]\}$

□

Informally, a **function** from a set X to a set Y is a mapping of elements of X to elements of Y in which each element of X is mapped to at most one element of Y . A function f from X to Y is denoted $f : X \rightarrow Y$. The element of Y assigned by the function f to an element $x \in X$ is denoted $f(x)$. The set X is called the **domain** of the function and the elements of X are the arguments or operands of the function f . The **range** of f is the subset of Y consisting of the members of Y that are assigned to elements of X . Thus the range of a function $f : X \rightarrow Y$ is the set $\{y \in Y \mid y = f(x) \text{ for some } x \in X\}$.

The relationship that assigns to each person his or her age is a function from the set of people to the natural numbers. Note that an element in the range may be assigned to more than one element of the domain—there are many people who have the same age. Moreover, not all natural numbers are in the range of the function; it is unlikely that the number 1000 is assigned to anyone.

The domain of a function is a set, but this set is often the Cartesian product of two or more sets. A function

$$f : X_1 \times X_2 \times \cdots \times X_n \rightarrow Y$$

is said to be an **n -variable function** or operation. The value of the function with variables x_1, x_2, \dots, x_n is denoted $f(x_1, x_2, \dots, x_n)$. Functions with one, two, or three variables are often referred to as *unary*, *binary*, and *ternary* operations. The function $sq : \mathbb{N} \rightarrow \mathbb{N}$ that assigns n^2 to each natural number is a unary operation. When the domain of a function consists of the Cartesian product of a set X with itself, the function is simply said to be a binary operation on X . Addition and multiplication are examples of binary operations on \mathbb{N} .

A function f relates members of the domain to members of the range of f . A natural definition of function is in terms of this relation. A **total function** f from X to Y is a binary relation on $X \times Y$ that satisfies the following two properties:

- i) For each $x \in X$, there is a $y \in Y$ such that $[x, y] \in f$.
- ii) If $[x, y_1] \in f$ and $[x, y_2] \in f$, then $y_1 = y_2$.

Condition (i) guarantees that each element of X is assigned a member of Y , hence the term *total*. The second condition ensures that this assignment is unique. The previously defined relation LT is not a total function since it does not satisfy the second condition. A relation on $\mathbb{N} \times \mathbb{N}$ representing *greater than* fails to satisfy either of the conditions. Why?

Example 1.2.2

Let $X = \{1, 2, 3\}$ and $Y = \{a, b\}$. The eight total functions from X to Y are listed below.

x	$f(x)$	x	$f(x)$	x	$f(x)$	x	$f(x)$
1	a	1	a	1	a	1	b
2	a	2	a	2	b	2	a
3	a	3	b	3	a	3	a

x	$f(x)$	x	$f(x)$	x	$f(x)$	x	$f(x)$
1	a	1	b	1	b	1	b
2	b	2	a	2	b	2	b
3	b	3	b	3	a	3	b

A **partial function** f from X to Y is a relation on $X \times Y$ in which $y_1 = y_2$ whenever $[x, y_1] \in f$ and $[x, y_2] \in f$. A partial function f is defined for an argument x if there is a $y \in Y$ such that $[x, y] \in f$. Otherwise, f is undefined for x . A total function is simply a partial function defined for all elements of the domain.

Although functions have been formally defined in terms of relations, we will use the standard notation $f(x) = y$ to indicate that y is the value assigned to x by the function f , that is, that $[x, y] \in f$. The notation $f(x) \uparrow$ indicates that the partial function f is undefined for the argument x . The notation $f(x) \downarrow$ is used to show that $f(x)$ is defined without explicitly giving its value.

Integer division defines a binary partial function div from $\mathbb{N} \times \mathbb{N}$ to \mathbb{N} . The quotient obtained from the division of i by j , when defined, is assigned to $div(i, j)$. For example, $div(3, 2) = 1$, $div(4, 2) = 2$, and $div(1, 2) = 0$. Using the previous notation, $div(i, 0) \uparrow$ and $div(i, j) \downarrow$ for all values of j other than zero.

A total function $f : X \rightarrow Y$ is said to be **one-to-one** if each element of X maps to a distinct element in the range. Formally, f is one-to-one if $x_1 \neq x_2$ implies $f(x_1) \neq f(x_2)$. A function $f : X \rightarrow Y$ is said to be **onto** if the range of f is the entire set Y . A total function

that is both one-to-one and onto defines a correspondence between the elements of domain and the range.

Example 1.2.3

The functions f , g , and s are defined from \mathbb{N} to $\mathbb{N} - \{0\}$, the set of positive natural numbers.

- i) $f(n) = 2n + 1$
- ii) $g(n) = \begin{cases} 1 & \text{if } n = 0 \\ n & \text{otherwise} \end{cases}$
- iii) $s(n) = n + 1$

The function f is one-to-one but not onto; the range of f consists of the odd numbers. The mapping from \mathbb{N} to $\mathbb{N} - \{0\}$ defined by g is clearly onto but not one-to-one since $g(0) = g(1) = 1$. The function s is both one-to-one and onto, defining a correspondence that maps each natural number to its successor. \square

Example 1.2.4

In the preceding example we noted that the function $f(n) = 2n + 1$ is one-to-one, but not onto the set $\mathbb{N} - \{0\}$. It is, however, a mapping from \mathbb{N} to the set of odd natural numbers that is both one-to-one and onto. We will use f to demonstrate how to prove that a function has these properties.

One-to-one: To prove that a function is one-to-one, we show that n and m must be the same whenever $f(n) = f(m)$. The assumption $f(n) = f(m)$ yields,

$$\begin{aligned} 2n + 1 &= 2m + 1 && \text{or} \\ 2n &= 2m, && \text{and finally,} \\ n &= m. \end{aligned}$$

It follows that $n \neq m$ implies $f(n) \neq f(m)$, and f is one-to-one.

Onto: To establish that f maps \mathbb{N} onto the set of odd natural numbers, we must show that every odd natural number is in the range of f . If m is an odd natural number, it can be written $m = 2n + 1$ for some $n \in \mathbb{N}$. Then $f(n) = 2n + 1 = m$ and m is in the range of f . \square

1.3 Equivalence Relations

A binary relation over a set X has been formally defined as a subset of the Cartesian product $X \times X$. Informally, we use a relation to indicate whether a property holds between two elements of a set. An ordered pair is in the relation if its elements satisfy the prescribed condition. For example, the property *is less than* defines a binary relation on the set of natural numbers. The relation defined by this property is the set $LT = \{(i, j) \mid i < j\}$.

ain

Infix notation is often used to express membership in many common binary relations. In this standard usage, $i < j$ indicates that i is less than j and consequently the pair $[i, j]$ is in the relation LT defined above.

We now consider a type of relation, known as an equivalence relation, that can be used to partition the underlying set. Equivalence relations are generally denoted using the infix notation $a \equiv b$ to indicate that a is equivalent to b .

Definition 1.3.1

A binary relation \equiv over a set X is an **equivalence relation** if it satisfies

- i) *Reflexivity*: $a \equiv a$, for all $a \in X$
- ii) *Symmetry*: $a \equiv b$ implies $b \equiv a$, for all $a, b \in X$
- iii) *Transitivity*: $a \equiv b$ and $b \equiv c$ implies $a \equiv c$, for all $a, b, c \in X$.

Definition 1.3.2

Let \equiv be an equivalence relation over X . The **equivalence class** of an element $a \in X$ defined by the relation \equiv is the set $[a]_{\equiv} = \{b \in X \mid a \equiv b\}$.

Example 1.3.1

Let \equiv_p be the parity relation over N defined by $n \equiv_p m$ if, and only if, n and m have the same parity (even or odd). To prove that \equiv_p is an equivalence relation, we must show that it is symmetric, reflexive, and transitive.

- i) *Reflexivity*: For every natural number n , n has the same parity as itself and $n \equiv_p n$.
- ii) *Symmetry*: If $n \equiv_p m$, then n and m have the same parity and $m \equiv_p n$.
- iii) *Transitivity*: If $n \equiv_p m$ and $m \equiv_p k$, then n and m have the same parity and m and k have the same parity. It follows that n and k have the same parity and $n \equiv_p k$.

The two equivalence classes of the parity relation \equiv_p are $[0]_{\equiv_p} = \{0, 2, 4, \dots\}$ and $[1]_{\equiv_p} = \{1, 3, 5, \dots\}$. \square

An equivalence class is usually written $[a]_{\equiv}$, where a is an element in the class. In the preceding example, $[0]_{\equiv_p}$ was used to represent the set of even natural numbers. Lemma 1.3.3 shows that if $a \equiv b$, then $[a]_{\equiv} = [b]_{\equiv}$. Thus the element chosen to represent the class is irrelevant.

Lemma 1.3.3

Let \equiv be an equivalence relation over X and let a and b be elements of X . Then either $[a]_{\equiv} = [b]_{\equiv}$ or $[a]_{\equiv} \cap [b]_{\equiv} = \emptyset$.

Proof. Assume that the intersection of $[a]_{\equiv}$ and $[b]_{\equiv}$ is not empty. Then there is some element c that is in both of the equivalence classes. Using symmetry and transitivity, we show that $[b]_{\equiv} \subseteq [a]_{\equiv}$. Since c is in both $[a]_{\equiv}$ and $[b]_{\equiv}$, we know $a \equiv c$ and $b \equiv c$. By symmetry, $c \equiv b$. Using transitivity, we conclude that $a \equiv b$.

Now let d be any element in $[b]_{\equiv}$. Then $b \equiv d$. The combination of $a \equiv b$, $b \equiv d$, and transitivity yields $a \equiv d$. That is, $d \in [a]_{\equiv}$. We have shown that every element in $[b]_{\equiv}$ is also in $[a]_{\equiv}$, so $[b]_{\equiv} \subseteq [a]_{\equiv}$. By a similar argument, we can establish that $[a]_{\equiv} \subseteq [b]_{\equiv}$. The two inclusions combine to produce the desired set equality. ■

Theorem 1.3.4

Let \equiv be an equivalence relation over X . The equivalence classes of \equiv partition X .

Proof. By Lemma 1.3.3, we know that the equivalence classes form a disjoint family of subsets of X . Let a be any element of X . By reflexivity, $a \in [a]_{\equiv}$. Thus each element of X is in one of the equivalence classes. It follows that the union of the equivalence classes is the entire set X . ■

1.4 Countable and Uncountable Sets

Cardinality is a measure that compares the size of sets. Intuitively, the cardinality of a set is the number of elements in the set. This informal definition is sufficient when dealing with finite sets; the cardinality can be obtained by counting the elements of the set. There are obvious difficulties in extending this approach to infinite sets.

Two finite sets can be shown to have the same number of elements by constructing a one-to-one correspondence between the elements of the sets. For example, the mapping

$$\begin{aligned} a &\longrightarrow 1 \\ b &\longrightarrow 2 \\ c &\longrightarrow 3 \end{aligned}$$

demonstrates that the sets $\{a, b, c\}$ and $\{1, 2, 3\}$ have the same size. This approach, comparing the size of sets using mappings, works equally well for sets with a finite or infinite number of members.

Definition 1.4.1

- i) Two sets X and Y have the same cardinality if there is a total one-to-one function from X onto Y .
- ii) The cardinality of a set X is less than or equal to the cardinality of a set Y if there is a total one-to-one function from X into Y .

Note that the two definitions differ only by the extent to which the mapping covers the set Y . If the range of the one-to-one mapping is all of Y , then the two sets have the same cardinality.

The cardinality of a set X is denoted $card(X)$. The relationships in (i) and (ii) are denoted $card(X) = card(Y)$ and $card(X) \leq card(Y)$, respectively. The cardinality of X is said to be strictly less than that of Y , written $card(X) < card(Y)$, if $card(X) \leq card(Y)$ and $card(X) \neq card(Y)$. The Schröder-Bernstein Theorem establishes the familiar relationship between \leq and $=$ for cardinality. The proof of the Schröder-Bernstein Theorem is left as an exercise.

$\equiv d$, and
 $[a] \equiv [b]$ is
 $[a] \equiv [b]$. The

family of
 element of X
 classes is

of a set is
 dealing with
 There are

structured a
 mapping

ach, com-
 or infinite

ction from

' if there is

rs the set Y.
 cardinality.
 and (ii) are
 ality of X is
 $ard(Y)$ and
 relationship
 :m is left as

Theorem 1.4.2 (Schröder-Bernstein)

If $card(X) \leq card(Y)$ and $card(Y) \leq card(X)$, then $card(X) = card(Y)$.

The cardinality of a finite set is denoted by the number of elements in the set. Thus $card(\{a, b\}) = 2$. A set that has the same cardinality as the set of natural numbers is said to be **countably infinite** or **denumerable**. Intuitively, a set is denumerable if its members can be put into an order and counted. The mapping f that establishes the correspondence with the natural numbers provides such an ordering; the first element is $f(0)$, the second $f(1)$, the third $f(2)$, and so on. The term **countable** refers to sets that are either finite or denumerable. A set that is not countable is said to be **uncountable**.

The set $N - \{0\}$ is countably infinite; the function $s(n) = n + 1$ defines a one-to-one mapping from N onto $N - \{0\}$. It may seem paradoxical that the set $N - \{0\}$, obtained by removing an element from N , has the same number of elements of N . Clearly, there is no one-to-one mapping of a finite set onto a proper subset of itself. It is this property that differentiates finite and infinite sets.

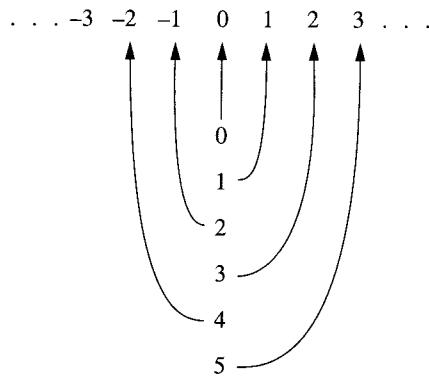
Definition 1.4.3

A set is **infinite** if it has a proper subset of the same cardinality.

Example 1.4.1

The set of odd natural numbers is countably infinite. The function $f(n) = 2n + 1$ from Example 1.2.4 establishes the one-to-one correspondence between N and the odd numbers. \square

A set is countably infinite if its elements can be put in a one-to-one correspondence with the natural numbers. A diagram of a mapping from N onto a set graphically illustrates the countability of the set. The one-to-one correspondence between the natural numbers and the set of all integers

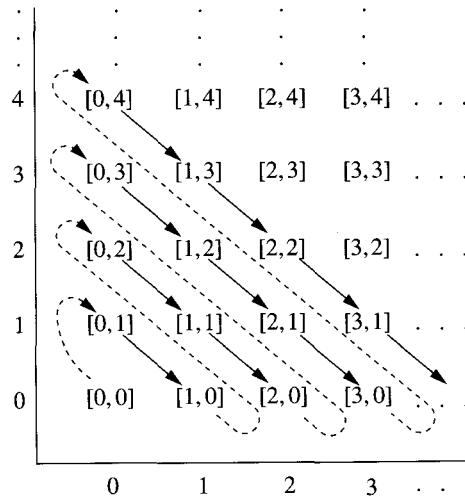


exhibits the countability of the set of integers. This correspondence is defined by the function

$$f(n) = \begin{cases} \text{div}(n, 2) + 1 & \text{if } n \text{ is odd} \\ -\text{div}(n, 2) & \text{if } n \text{ is even.} \end{cases}$$

Example 1.4.2

The points of an infinite two-dimensional grid can be used to show that $\mathbb{N} \times \mathbb{N}$, the set of ordered pairs of natural numbers, is denumerable. The grid is constructed by labeling the axes with the natural numbers. The position defined by the i th entry on the horizontal axis and the j th entry on the vertical axis represents the ordered pair $[i, j]$.



The elements of the grid can be listed sequentially by following the arrows in the diagram. This creates the correspondence

0	1	2	3	4	5	6	7	...
↓	↓	↓	↓	↓	↓	↓	↓	
[0, 0]	[0, 1]	[1, 0]	[0, 2]	[1, 1]	[2, 0]	[0, 3]	[1, 2]	...

that demonstrates the countability of $\mathbb{N} \times \mathbb{N}$. The one-to-one correspondence outlined above maps the ordered pair $[i, j]$ to the natural number $((i+j)(i+j+1)/2) + i$. \square

The sets of interest in language theory and computability are almost exclusively finite or denumerable. We state, without proof, several closure properties of countable sets.

Theorem 1.4.4

- i) The union of two countable sets is countable.
- ii) The Cartesian product of two countable sets is countable.

- iii) The set of finite subsets of a countable set is countable.
- iv) The set of finite-length sequences consisting of elements of a nonempty countable set is countably infinite.

The preceding theorem indicates that the property of countability is retained under many standard set-theoretic operations. Each of these closure results can be established by constructing a one-to-one correspondence between the new set and a subset of the natural numbers.

A set is uncountable if it is impossible to sequentially list its members. The following proof technique, known as *Cantor's diagonalization argument*, is used to show that there is an uncountable number of total functions from \mathbb{N} to \mathbb{N} . Two total functions $f : \mathbb{N} \rightarrow \mathbb{N}$ and $g : \mathbb{N} \rightarrow \mathbb{N}$ are equal if they have the same value for every element in the domain. That is, $f = g$ if $f(n) = g(n)$ for all $n \in \mathbb{N}$. To show that two functions are distinct, it suffices to find a single input value for which the functions differ.

Assume that the set of total functions from the natural numbers to the natural numbers is denumerable. Then there is a sequence f_0, f_1, f_2, \dots that contains all the functions. The values of the functions are exhibited in the two-dimensional grid with the input values on the horizontal axis and the functions on the vertical axis.

	0	1	2	3	4	...
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$	$f_0(4)$...
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$...
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$...
f_3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$...
f_4	$f_4(0)$	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$...
:	:	:	:	:	:	

f
e
s

am.

bove
□
finite

Consider the function $f : \mathbb{N} \rightarrow \mathbb{N}$ defined by $f(n) = f_n(n) + 1$. The values of f are obtained by adding 1 to the values on the diagonal of the grid, hence the name diagonalization. By the definition of f , $f(i) \neq f_i(i)$ for every i . Consequently, f is not in the sequence f_0, f_1, f_2, \dots . This is a contradiction since the sequence was assumed to contain all the total functions. The assumption that the number of functions is countably infinite leads to a contradiction. It follows that the set is uncountable.

Diagonalization is a general proof technique for demonstrating that a set is not countable. As seen in the preceding example, establishing uncountability using diagonalization is a proof by contradiction. The first step is to assume that the set is countable and therefore its members can be exhaustively listed. The contradiction is achieved by producing a member of the set that cannot occur anywhere in the list. No conditions are put on the listing of the elements other than that it must contain all the elements of the set. Producing a contradiction by diagonalization shows that there is no possible exhaustive listing of the elements and consequently that the set is uncountable. This technique is exhibited again in the following examples.

Example 1.4.3

A function f from \mathbb{N} to \mathbb{N} has a *fixed point* if there is some natural number i such that $f(i) = i$. For example, $f(n) = n^2$ has fixed points 0 and 1, while $f(n) = n^2 + 1$ has no fixed points. We will show that the number of functions that do not have fixed points is uncountable. The argument is similar to the proof that the number of all functions from \mathbb{N} to \mathbb{N} is uncountable, except that we now have an additional condition that must be met when constructing an element that is not in the listing.

Assume that the number of the functions without fixed points is countable. Then these functions can be listed f_0, f_1, f_2, \dots . To obtain a contradiction to our assumption that the set is countable, we construct a function that has no fixed points and is not in the list. Consider the function $f(n) = f_n(n) + n + 1$. The addition of $n + 1$ in the definition of f ensures that $f(n) > n$ for all n . Thus f has no fixed points. By an argument similar to that given above, $f(i) \neq f_i(i)$ for all i . Consequently, the listing f_0, f_1, f_2, \dots is not exhaustive, and we conclude that the number of functions without fixed points is uncountable. \square

Example 1.4.4

$\mathcal{P}(\mathbb{N})$, the set of subsets of \mathbb{N} , is uncountable. Assume that the set of subsets of \mathbb{N} is countable. Then they can be listed N_0, N_1, N_2, \dots . Define a subset D of \mathbb{N} as follows: For every natural number j ,

$$j \in D \text{ if, and only if, } j \notin N_j.$$

By our construction, $0 \in D$ if $0 \notin N_0$, $1 \in D$ if $1 \notin N_1$, and so on. The set D is clearly a set of natural numbers. By our assumption, N_0, N_1, N_2, \dots is an exhaustive listing of the subsets of \mathbb{N} . Hence, $D = N_i$ for some i . Is the number i in the set D ? By definition of D ,

$$i \in D \text{ if, and only if, } i \notin N_i.$$

But since $D = N_i$, this becomes

$$i \in D \text{ if, and only if, } i \notin D,$$

which is a contradiction. Thus, our assumption that $\mathcal{P}(\mathbb{N})$ is countable must be false and we conclude that $\mathcal{P}(\mathbb{N})$ is uncountable.

To appreciate the “diagonal” technique, consider a two-dimensional grid with the natural numbers on the horizontal axis and the vertical axis labeled by the sets N_0, N_1, N_2, \dots . The position of the grid designated by row N_i and column j contains *yes* if $j \in N_i$. Otherwise, the position defined by N_i and column j contains *no*. The set D is constructed by considering the relationship between the entries along the diagonal of the grid: the number j and the set N_j . By the way that we have defined D , the number j is an element of D if, and only if, the entry in the position labeled by N_j and j is *no*. \square

In ad
strati
used i
agona
self-re
parade
of the
referen

T
on the
between
type th
operator
of Rus

Th
told th
(a man
consid
the set
the sha
the tow

where th
will have
himself c
he is p_i f
self-refer
operator (

Who
he shaves

1.5 Diagonalization and Self-Reference

uch that
has no
oints is
from N
et when

en these
that the
Consider
ures that
n above,
, and we

□

ts of N is
llows: For

rly a set of
the subsets
D,

alse and we

rid with the
sets $N_0, N_1,$
es if $j \in N_i.$
onstructed by
the number
ment of D if,

□

In addition to its use in cardinality proofs, diagonalization provides a method for demonstrating that certain properties or relations are inherently contradictory. These results are used in nonexistence proofs since there can be no object that satisfies such a property. Diagonalization proofs of nonexistence frequently depend upon contradictions that arise from self-reference—an object analyzing its own actions, properties, or characteristics. Russell's paradox, the undecidability of the Halting Problem for Turing Machines, and Gödel's proof of the undecidability of number theory are all based on contradictions associated with self-reference.

The diagonalization proofs in the preceding section used a table with operators listed on the vertical axis and their arguments on the horizontal axis to illustrate the relationship between the operators and arguments. In each example, the operators were of a different type than their arguments. In self-reference, the same family of objects comprises the operators and their arguments. We will use the *barber's paradox*, an amusing simplification of Russell's paradox, to illustrate diagonalization and self-reference.

The barber's paradox is concerned with who shaves whom in a mythical town. We are told that every man who is able to shave himself does so and that the barber of the town (a man himself) shaves all and only the people who cannot shave themselves. We wish to consider the possible truth of such a statement and the existence of such a town. In this case, the set of males in the town make up both the operators and the arguments; they are doing the shaving and being shaved. Let $M = \{p_1, p_2, p_3, \dots, p_i, \dots\}$ be the set of all males in the town. A tabular representation of the shaving relationship has the form

	p_1	p_2	p_3	\dots	p_i	\dots
p_1	-	-	-	\dots	-	\dots
p_2	-	-	-	\dots	-	\dots
p_3	-	-	-	\dots	-	\dots
\vdots						
p_i	-	-	-	\dots	-	\dots
\vdots						

where the i, j th position of the table has a 1 if p_i shaves p_j and a 0 otherwise. Every column will have one entry with a 1 and all the other entries will be 0; each person either shaves himself or is shaved by the barber. The barber must be one of the people in the town, so he is p_i for some value i . What is the value of the position i, i in the table? This is classic self-reference; we are asking what occurs when a particular object is simultaneously the operator (the person doing the shaving) and the operand (the person being shaved).

Who shaves the barber? If the barber is able to shave himself, then he cannot do so since he shaves only people who are unable to shave themselves. If he is unable to shave himself,

then he must shave himself since he shaves everyone who cannot shave themselves. We have shown that the properties describing the shaving habits of the town are contradictory so such a town cannot exist.

Russell's paradox follows the same pattern, but its consequences were much more significant than the nonexistence of a mythical town. One of the fundamental tenets of set theory as proposed by Cantor in the late 1800s was that any property or condition that can be described defines a set—the set of objects that satisfy the condition. There may be no objects, finitely many, or infinitely many that satisfy the property, but regardless of the number or the type of elements, the objects form a set. Russell devised an argument based on self-reference to show that this claim cannot be true.

The relationship examined by Russell's paradox is that of the membership of one set in another. For each set X we ask the question, "Is a set Y an element of X ?" This is not an unreasonable question, since one set can certainly be an element of another. The table below gives both some negative and positive examples of this question.

X	Y	$Y \in X$?
$\{a\}$	$\{a\}$	no
$\{\{a\}, b\}$	$\{a\}$	yes
$\{\{a\}, a, \emptyset\}$	\emptyset	yes
$\{\{a, b\}, \{a\}\}$	$\{\{a\}\}$	no
$\{\{\{a\}, b\}, b\}$	$\{\{a\}, b\}$	yes

It is important to note that the question is not whether Y is a subset of X , but whether it is an element of X .

The membership relation can be depicted by the table

	X_1	X_2	X_3	...	X_i	...
X_1	-	-	-	...	-	...
X_2	-	-	-	...	-	...
X_3	-	-	-	...	-	...
\vdots	\vdots	\vdots	\vdots	\ddots	-	...
X_i	-	-	-	...	-	...
\vdots						

where axes are labeled by the sets. A table entry $[i, j]$ is 1 if X_j is an element of X_i and 0 if X_j is not an element of X_i .

A question of self-reference can be obtained by identifying the operator and the operand in the membership question. That is, we ask if a set X_i is an element of itself. The diagonal entry $[i, i]$ in the preceding table contains the answer to the question, "Is X_i an element of X_i ?" Now consider the property that a set is not an element of itself. Does this property define a set? There are clearly examples of sets that satisfy the property; the set $\{a\}$ is not

an ele
diag
A
not in
an ele
our as
W
Canto
Russe
on nai
formal
a fund
Proble

1.6

Many, i
an infi
generat
describ
since ev
after 0, 1
with our
no idea
number.

In t
automata
theory. T
A metho
that has r

A re
of the se
basis con
The opera
members.
the basis e

The K
process ca
obtained b
This intuit
idea is for

We
ctory

more
ets of
n that
ay be
of the
based

ne set
is not
table

er it is

ζ_i and 0
operand
diagonal
lement of
property
 $a\}$ is not

an element of itself. The satisfaction of the property is indicated by the complement of the diagonal. A set X_i is not an element of itself if, and only if, entry $[i, i]$ is 0.

Assume that $S = \{X \mid X \notin X\}$ is a set. Is S in S ? If S is an element of itself, then it is not in S by the definition of S . Moreover, if S is not in S , then it must be in S since it is not an element of itself. This is an obvious contradiction. We were led to this contradiction by our assumption that the collection of sets that satisfy the property $X \notin X$ form a set.

We have constructed a describable property that cannot define a set. This shows that Cantor's assertion about the universality of sets is demonstrably false. The ramifications of Russell's paradox were far-reaching. The study of set theory moved from a foundation based on naive definitions to formal systems of axioms and inference rules and helped initiate the formalist philosophy of mathematics. In Chapter 12 we will use self-reference to establish a fundamental result in the theory of computer science, the undecidability of the Halting Problem.

1.6 Recursive Definitions

Many, in fact most, of the sets of interest in formal language and automata theory contain an infinite number of elements. Thus it is necessary that we develop techniques to describe, generate, or recognize the elements that belong to an infinite set. In the preceding section we described the set of natural numbers utilizing ellipsis dots (. . .). This seemed reasonable since everyone reading this text is familiar with the natural numbers and knows what comes after 0, 1, 2, 3. However, this description would be totally inadequate for an alien unfamiliar with our base 10 arithmetic system and numeric representations. Such a being would have no idea that the symbol 4 is the next element in the sequence or that 1492 is a natural number.

In the development of a mathematical theory, such as the theory of languages or automata, the theorems and proofs may utilize only the definitions of the concepts of that theory. This requires precise definitions of both the objects of the domain and the operations. A method of definition must be developed that enables our friend the alien, or a computer that has no intuition, to generate and "understand" the properties of the elements of a set.

A **recursive definition** of a set X specifies a method for constructing the elements of the set. The definition utilizes two components: a basis and a set of operations. The basis consists of a finite set of elements that are explicitly designated as members of X . The operations are used to construct new elements of the set from the previously defined members. The recursively defined set X consists of all elements that can be generated from the basis elements by a finite number of applications of the operations.

The key word in the process of recursively defining a set is *generate*. Clearly, no process can list the complete set of natural numbers. Any particular number, however, can be obtained by beginning with zero and constructing an initial sequence of the natural numbers. This intuitively describes the process of recursively defining the set of natural numbers. This idea is formalized in the following definition.

Definition 1.6.1

A recursive definition of \mathbf{N} , the set of natural numbers, is constructed using the successor function s .

- i) Basis: $0 \in \mathbf{N}$.
- ii) Recursive step: If $n \in \mathbf{N}$, then $s(n) \in \mathbf{N}$.
- iii) Closure: $n \in \mathbf{N}$ only if it can be obtained from 0 by a finite number of applications of the operation s .

The basis explicitly states that 0 is a natural number. In (ii), a new natural number is defined in terms of a previously defined number and the successor operation. The closure section guarantees that the set contains only those elements that can be obtained from 0 using the successor operator. Definition 1.6.1 generates an infinite sequence 0, $s(0)$, $s(s(0))$, $s(s(s(0)))$, . . . This sequence is usually abbreviated 0, 1, 2, 3, . . . However, anything that can be done with the familiar Arabic numerals could also be done with the more cumbersome unabbreviated representation.

The essence of a recursive procedure is to define complicated processes or structures in terms of simpler instances of the same process or structure. In the case of the natural numbers, “simpler” often means smaller. The recursive step of Definition 1.6.1 defines a number in terms of its predecessor.

The natural numbers have now been defined, but what does it mean to understand their properties? We usually associate operations of addition, multiplication, and subtraction with the natural numbers. We may have learned these by brute force, either through memorization or tedious repetition. For the alien or a computer to perform addition, the meaning of “add” must be appropriately defined. One cannot memorize the sum of all possible combinations of natural numbers, but we can use recursion to establish a method by which the sum of any two numbers can be mechanically calculated. The successor function is the only operation on the natural numbers that has been introduced. Thus the definition of addition may use only 0 and s .

Definition 1.6.2

In the following recursive definition of the sum of m and n , the recursion is done on n , the second argument of the sum.

- i) Basis: If $n = 0$, then $m + n = m$.
- ii) Recursive step: $m + s(n) = s(m + n)$.
- iii) Closure: $m + n = k$ only if this equality can be obtained from $m + 0 = m$ using finitely many applications of the recursive step.

The closure step is often omitted from a recursive definition of an operation on a given domain. In this case, it is assumed that the operation is defined for all the elements of the domain. The operation of addition given above is defined for all elements of $\mathbf{N} \times \mathbf{N}$.

The sum of m and the successor of n is defined in terms of the simpler case, the sum of m and n , and the successor operation. The choice of n as the recursive operand was arbitrary; the operation could also have been defined in terms of m , with n fixed.

num
num

Exam

The r
recur

This f

Fr
the co
and th
defined
fewer c
a count
of the
means
operato
the ope
entire s
is a fun

The
Cartesi
pairs. F
of the o
the ord
in Sectio

Example

The rela

- i) Bas
- ii) Rec
- iii) Clos
- appl

Following the construction given in Definition 1.6.2, the sum of any two natural numbers can be computed using 0 and s , the primitives used in the definition of the natural numbers. Example 1.6.1 traces the recursive computation of $3 + 2$.

Example 1.6.1

The numbers 3 and 2 abbreviate $s(s(s(0)))$ and $s(s(0))$, respectively. The sum is computed recursively by

$$\begin{aligned} & s(s(s(0))) + s(s(0)) \\ &= s(s(s(s(0)))) + s(0) \\ &= s(s(s(s(s(0)))) + 0)) \\ &= s(s(s(s(s(0))))) \quad (\text{basis case}). \end{aligned}$$

This final value is the representation of the number 5. \square

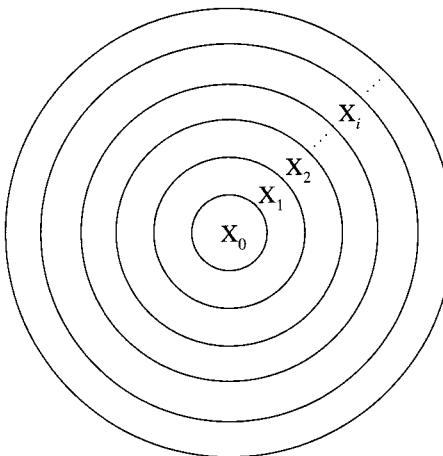
Figure 1.1 illustrates the process of recursively generating a set X from basis X_0 . Each of the concentric circles represents a stage of the construction. X_1 represents the basis elements and the elements that can be obtained from them using a single application of an operation defined in the recursive step. X_i contains the elements that can be constructed with i or fewer operations. The generation process in the recursive portion of the definition produces a countably infinite sequence of nested sets. The set X can be thought of as the infinite union of the X_i 's. Let x be an element of X and let X_j be the first set in which x occurs. This means that x can be constructed from the basis elements using exactly j applications of the operators. Although each element of X can be generated by a finite number of applications of the operators, there is no upper bound on the number of applications needed to generate the entire set X . This property, generation using a finite but unbounded number of operations, is a fundamental property of recursive definitions.

The successor operator can be used recursively to define relations on the set $N \times N$. The Cartesian product $N \times N$ is often portrayed by the grid of points representing the ordered pairs. Following the standard conventions, the horizontal axis represents the first component of the ordered pair and the vertical axis the second. The shaded area in Figure 1.2(a) contains the ordered pairs $[i, j]$ in which $i < j$. This set is the relation LT, less than, that was described in Section 1.2.

Example 1.6.2

The relation LT is defined as follows:

- i) Basis: $[0, 1] \in LT$.
- ii) Recursive step: If $[m, n] \in LT$, then $[m, s(n)] \in LT$ and $[s(m), s(n)] \in LT$.
- iii) Closure: $[m, n] \in LT$ only if it can be obtained from $[0, 1]$ by a finite number of applications of the operations in the recursive step.



9
8
7
6
5
4
3
2
1
0

Recursive generation of X :

$$X_0 = \{x \mid x \text{ is a basis element}\}$$

$$X_{i+1} = X_i \cup \{x \mid x \text{ can be generated by } i+1 \text{ operations}\}$$

$$X = \{x \mid x \in X_j \text{ for some } j \geq 0\}$$

FIGURE 1.1 Nested sequence of sets in recursive definition.

Using the infinite union description of recursive generation, the definition of LT generates the sequence LT_i of nested sets where

$$LT_0 = \{[0, 1]\}$$

$$LT_1 = LT_0 \cup \{[0, 2], [1, 2]\}$$

$$LT_2 = LT_1 \cup \{[0, 3], [1, 3], [2, 3]\}$$

$$LT_3 = LT_2 \cup \{[0, 4], [1, 4], [2, 4], [3, 4]\}$$

⋮

$$LT_i = LT_{i-1} \cup \{[j, i+1] \mid j = 0, 1, \dots, i\}$$

⋮

□

The construction of LT shows that the generation of an element in a recursively defined set may not be unique. The ordered pair $[1, 3] \in LT_2$ is generated by the two distinct sequences of operations:

Basis:	$[0, 1]$	\backslash	$[0, 1]$
1:	$[0, s(1)] = [0, 2]$		$[s(0), s(1)] = [1, 2]$
2:	$[s(0), s(2)] = [1, 3]$		$[1, s(2)] = [1, 3]$.

Example

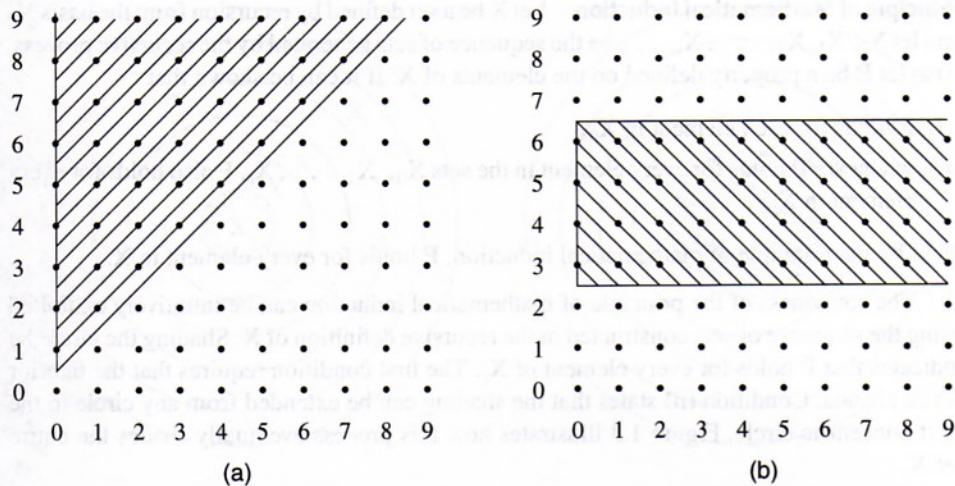
The sha
4, 5, or t

- i) Bas
- ii) Rec
- iii) Clo
- of a

The sequ

1.7 M

Establish
the ability
prove that
individual
that a prop
of nested
entire set.

FIGURE 1.2 Relations on $\mathbb{N} \times \mathbb{N}$.**Example 1.6.3**

The shaded area in Figure 1.2(b) contains all the ordered pairs with second component 3, 4, 5, or 6. A recursive definition of this set, call it X , is given below.

- i) Basis: $[0, 3], [0, 4], [0, 5]$, and $[0, 6]$ are in X .
- ii) Recursive step: If $[m, n] \in X$, then $[s(m), n] \in X$.
- iii) Closure: $[m, n] \in X$ only if it can be obtained from the basis elements by a finite number of applications of the operation in the recursive step.

The sequence of sets X_i generated by this recursive process is defined by

$$X_i = \{[j, 3], [j, 4], [j, 5], [j, 6] \mid j = 0, 1, \dots, i\}.$$

□

1.7 Mathematical Induction

Establishing relationships between the elements of sets and operations on the sets requires the ability to construct proofs that verify the hypothesized properties. It is impossible to prove that a property holds for every member in an infinite set by considering each element individually. The principle of mathematical induction gives sufficient conditions for proving that a property holds for every element in a recursively defined set. Induction uses the family of nested sets generated by the recursive process to extend a property from the basis to the entire set.

Principle of Mathematical Induction Let X be a set defined by recursion from the basis X_0 and let $X_0, X_1, X_2, \dots, X_i, \dots$ be the sequence of sets generated by the recursive process. Also let P be a property defined on the elements of X . If it can be shown that

- i) P holds for each element in X_0 ,
- ii) whenever P holds for every element in the sets X_0, X_1, \dots, X_i , P also holds for every element in X_{i+1} ,

then, by the principle of mathematical induction, P holds for every element in X .

The soundness of the principle of mathematical induction can be intuitively exhibited using the sequence of sets constructed in the recursive definition of X . Shading the circle X_i indicates that P holds for every element of X_i . The first condition requires that the interior set be shaded. Condition (ii) states that the shading can be extended from any circle to the next concentric circle. Figure 1.3 illustrates how this process eventually shades the entire set X .

The justification for the principle of mathematical induction should be clear from the preceding argument. Another justification can be obtained by assuming that conditions (i) and (ii) are satisfied but P is not true for every element in X . If P does not hold for all elements of X , then there is at least one set X_i for which P does not universally hold. Let X_j be the first such set. Since condition (i) asserts that P holds for all elements of X_0 , j cannot be zero. Now P holds for all elements of X_{j-1} by our choice of j . Condition (ii) then requires that P hold for all elements in X_j . This implies that there is no first set in the sequence for which the property P fails. Consequently, P must be true for all the X_i 's, and therefore for X .

An inductive proof consists of three distinct steps. The first step is proving that the property P holds for each element of a basis set. This corresponds to establishing condition (i) in the definition of the principle of mathematical induction. The second is the statement of the inductive hypothesis. The inductive hypothesis is the assumption that the property P holds for every element in the sets X_0, X_1, \dots, X_n . The inductive step then proves, using the inductive hypothesis, that P can be extended to each element in X_{n+1} . Completing the inductive step satisfies the requirements of the principle of mathematical induction. Thus, it can be concluded that P is true for all elements of X .

In Example 1.6.2, a recursive definition was given to generate the relation LT , which consists of ordered pairs $[i, j]$ that satisfy $i < j$. Does every ordered pair generated by the definition satisfy this inequality? We will use this question to illustrate the steps of an inductive proof on a recursively defined set.

The first step is to explicitly show that the inequality is satisfied for all elements in the basis. The basis of the recursive definition of LT is the set $\{[0, 1]\}$. The basis step of the inductive proof is satisfied since $0 < 1$.

The inductive hypothesis states the assumption that $x < y$ for all ordered pairs $[x, y] \in LT_n$. In the inductive step we must prove that $i < j$ for all ordered pairs $[i, j] \in LT_{n+1}$. The recursive step in the definition of LT relates the sets LT_{n+1} and LT_n . Let $[i, j]$ be an ordered

pair in LT_{n+1} . By the in-

Similarly,

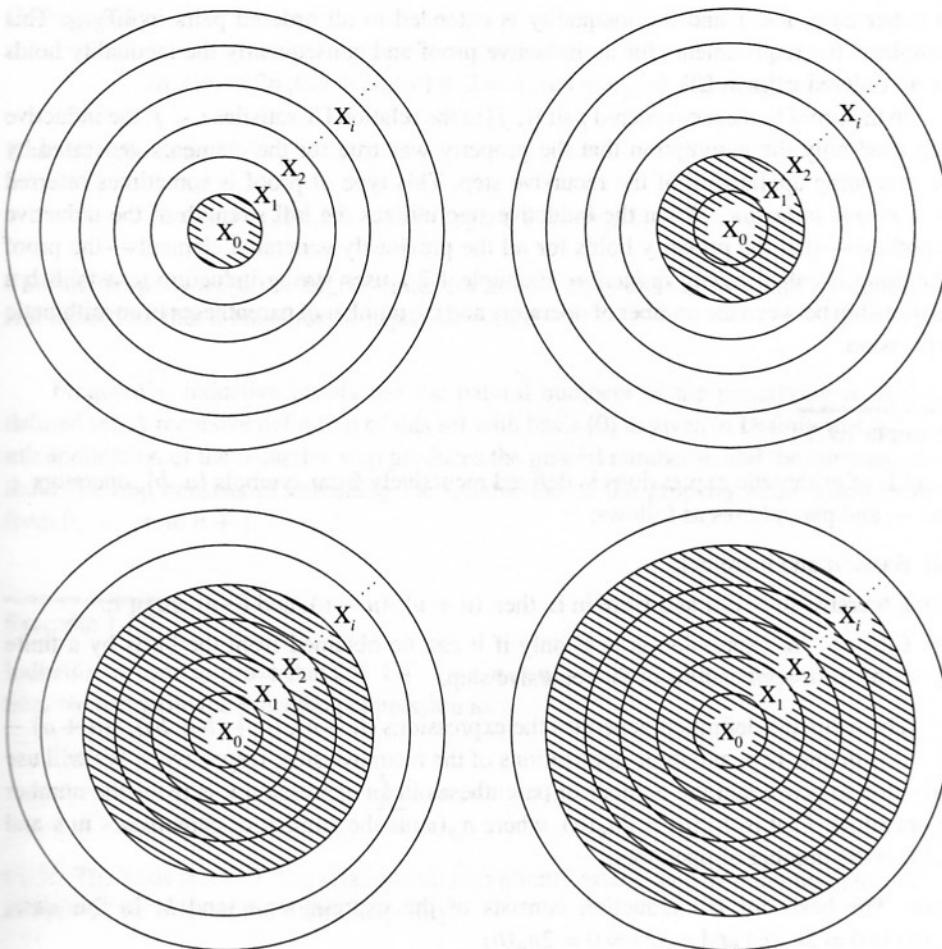


FIGURE 1.3 Principle of mathematical induction.

pair in LT_{n+1} . Then either $[i, j] = [x, s(y)]$ or $[i, j] = [s(x), s(y)]$ for some $[x, y] \in LT_n$. By the inductive hypothesis, $x < y$. If $[i, j] = [x, s(y)]$, then

$$i = x < y < s(y) = j.$$

Similarly, if $[i, j] = [s(x), s(y)]$, then

$$i = s(x) < s(y) = j.$$

In either case, $i < j$ and the inequality is extended to all ordered pairs in LT_{n+1} . This completes the requirements for an inductive proof and consequently the inequality holds for all ordered pairs in LT .

In the proof that every ordered pair $[i, j]$ in the relation LT satisfies $i < j$, the inductive step used only the assumption that the property was true for the elements generated by the preceding application of the recursive step. This type of proof is sometimes referred to as *simple induction*. When the inductive step utilizes the full strength of the inductive hypothesis—that the property holds for all the previously generated elements—the proof technique is called *strong induction*. Example 1.7.1 uses strong induction to establish a relationship between the number of operators and the number of parentheses in an arithmetic expression.

Example 1.7.1

A set E of arithmetic expressions is defined recursively from symbols $\{a, b\}$, operators $+$ and $-$, and parentheses as follows:

- i) Basis: a and b are in E .
- ii) Recursive step: If u and v are in E , then $(u + v)$, $(u - v)$, and $(-v)$ are in E .
- iii) Closure: An expression is in E only if it can be obtained from the basis by a finite number of applications of the recursive step.

The recursive definition generates the expressions $(a + b)$, $(a + (b + b))$, $((a + a) - (b - a))$ in one, two, and three applications of the recursive step, respectively. We will use induction to prove that the number of parentheses in an expression u is twice the number of operators. That is, $n_p(u) = 2n_o(u)$, where $n_p(u)$ is the number of parentheses in u and $n_o(u)$ is the number of operators.

Basis: The basis for the induction consists of the expressions a and b . In this case, $n_p(a) = 0 = 2n_o(a)$ and $n_p(b) = 0 = 2n_o(b)$.

Inductive Hypothesis: Assume that $n_p(u) = 2n_o(u)$ for all expressions generated by n or fewer iterations of the recursive step, that is, for all u in E_n .

Inductive Step: Let w be an expression generated by $n + 1$ applications of the recursive step. Then $w = (u + v)$, $w = (u - v)$, or $w = (-v)$ where u and v are strings in E_n . By the inductive hypothesis,

$$\begin{aligned} n_p(u) &= 2n_o(u) \\ n_p(v) &= 2n_o(v). \end{aligned}$$

If $w = (u + v)$ or $w = (u - v)$,

$$\begin{aligned} n_p(w) &= n_p(u) + n_p(v) + 2 \\ n_o(w) &= n_o(u) + n_o(v) + 1. \end{aligned}$$

his
lds

ive
by
red
ive
oof
h a
etic

s +

inite

z) —
I use
nber
and

case,

n or

ursive
y the

Consequently,

$$2n_o(w) = 2n_o(u) + 2n_o(v) + 2 = n_p(u) + n_p(v) + 2 = n_p(w).$$

If $w = (-v)$, then

$$2n_o(w) = 2(n_o(v) + 1) = 2n_o(v) + 2 = n_p(v) + 2 = n_p(w).$$

Thus the property $n_p(w) = 2n_o(w)$ holds for all $w \in E_{n+1}$ and we conclude, by mathematical induction, that it holds for all expressions in E . \square

Frequently, inductive proofs use the natural numbers as the underlying recursively defined set. A recursive definition of this set with basis $\{0\}$ is given in Definition 1.6.1. The n th application of the recursive step produces the natural number n , and the corresponding inductive step consists of extending the satisfaction of the property under consideration from $0, \dots, n$ to $n + 1$.

Example 1.7.2

Induction is used to prove that $0 + 1 + \dots + n = n(n + 1)/2$. Using the summation notation, we can write the preceding expression as

$$\sum_{i=0}^n i = n(n + 1)/2.$$

Basis: The basis is $n = 0$. The relationship is explicitly established by computing the values of each of the sides of the desired equality.

$$\sum_{i=0}^0 i = 0 = 0(0 + 1)/2.$$

Inductive Hypothesis: Assume for all values $k = 1, 2, \dots, n$ that

$$\sum_{i=0}^k i = k(k + 1)/2.$$

Inductive Step: We need to prove that

$$\sum_{i=0}^{n+1} i = (n + 1)(n + 1 + 1)/2 = (n + 1)(n + 2)/2.$$

The inductive hypothesis establishes the result for the sum of the sequence containing n or fewer integers. Combining the inductive hypothesis with the properties of addition, we obtain

$$\begin{aligned}\sum_{i=0}^{n+1} i &= \sum_{i=0}^n i + (n+1) && \text{(associativity of +)} \\ &= n(n+1)/2 + (n+1) && \text{(inductive hypothesis)} \\ &= (n+1)(n/2 + 1) && \text{(distributive property)} \\ &= (n+1)(n+2)/2.\end{aligned}$$

Since the conditions of the principle of mathematical induction have been established, we conclude that the result holds for all natural numbers. \square

Each step in the proof must follow from previously established properties of the operators or the inductive hypothesis. The strategy of an inductive proof is to manipulate the formula to contain an instance of the property applied to a simpler case. When this is accomplished, the inductive hypothesis may be invoked. After the application of the inductive hypothesis, the remainder of the proof often consists of algebraic manipulation to produce the desired result.

1.8 Directed Graphs

A mathematical structure consists of a set or sets, distinguished elements from the sets, and functions and relations on the sets. A *distinguished element* is an element of a set that has special properties that differentiate it from the other elements. The natural numbers, as defined in Definition 1.6.1, can be expressed as a structure $(\mathbb{N}, s, 0)$. The set \mathbb{N} contains the natural numbers, s is a unary function on \mathbb{N} , and 0 is a distinguished element of \mathbb{N} . Zero is distinguished because of its explicit role in the definition of the natural numbers.

Graphs are frequently used to portray the essential features of a mathematical entity in a diagram, which aids the intuitive understanding of the concept. Formally, a **directed graph** is a mathematical structure consisting of a set N and a binary relation A on N . The elements of N are called the *nodes*, or *vertices*, of the graph and the elements of A are called *arcs* or *edges*. The relation A is referred to as the *adjacency relation*. A node y is said to be *adjacent* to x when $[x, y] \in A$. An arc from x to y in a directed graph is depicted by an arrow from x to y . Using the arrow metaphor, y is called the head of the arc and x the tail. The *in-degree* of a node x is the number of arcs with x as the head. The *out-degree* of x is the number of arcs with x as the tail. Node a in Figure 1.4 has in-degree two and out-degree one.

A **path** from a node x to a node y in a directed graph $G = (N, A)$ is a sequence of nodes and arcs $x_0, [x_0, x_1], x_1, [x_1, x_2], x_2, \dots, x_{n-1}, [x_{n-1}, x_n], x_n$ of G with $x = x_0$ and $y = x_n$. The node x is the initial node of the path and y is the terminal node. Each pair

of nodes
number
its arcs.

The
of length
is simple
Figure 1
said to b

The
labeled c
on $N \times$
on an ar
Figure 1
San Fran

An e
connecte
has in-de
where N
The term
the arbore
omitted in

A no
Accomp
is draw
manner ac

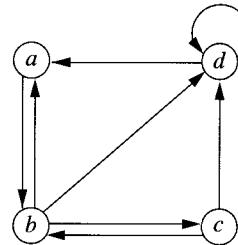
ing n
n, we

ed, we
□

of the
nipulate
hen this
n of the
pulation

the sets,
a set that
mbers, as
[contains
f N. Zero
rs.
ical entity
a directed
on N. The
are called
is said to
cted by an
x the tail.
ree of x is
out-degree

equence of
 $x = x_0$ and
. Each pair



$N = \{a, b, c, d\}$	Node	In-degree	Out-degree
$A = \{[a, b], [b, a], [b, c],$	a	2	1
$[b, d], [c, b], [c, d],$	b	2	3
$[d, a], [d, d]\}$	c	1	2
	d	3	2

FIGURE 1.4 Directed graph.

of nodes x_i, x_{i+1} in the path is connected by the arc $[x_i, x_{i+1}]$. The length of a path is the number of arcs in the path. We will frequently describe a path simply by sequentially listing its arcs.

There is a path of length zero from any node to itself called the **null path**. A path of length one or more that begins and ends with the same node is called a **cycle**. A cycle is *simple* if it does not contain a cyclic subpath. The path $[a, b], [b, c], [c, d], [d, a]$ in Figure 1.4 is a simple cycle of length four. A directed graph containing at least one cycle is said to be *cyclic*. A graph with no cycles is said to be *acyclic*.

The arcs of a directed graph often designate more than the adjacency of the nodes. A labeled directed graph is a structure (N, L, A) where L is the set of labels and A is a relation on $N \times N \times L$. An element $[x, y, v] \in A$ is an arc from x to y labeled by v . The label on an arc specifies a relationship between the adjacent nodes. The labels on the graph in Figure 1.5 indicate the distances of the legs of a trip from Chicago to Minneapolis, Seattle, San Francisco, Dallas, St. Louis, and back to Chicago.

An **ordered tree**, or simply a tree, is an acyclic directed graph in which each node is connected by a unique path from a distinguished node called the **root** of the tree. The root has in-degree zero and all other nodes have in-degree one. A tree is a structure (N, A, r) where N is the set of nodes, A is the adjacency relation, and $r \in N$ is the root of the tree. The terminology of trees combines a mixture of references to family trees and to those of the arboreal nature. Although a tree is a directed graph, the arrows on the arcs are usually omitted in the illustrations of trees. Figure 1.6(a) gives a tree T with root x_1 .

A node y is called a **child** of a node x , and x the **parent** of y , if y is adjacent to x . Accompanying the adjacency relation is an order on the children of any node. When a tree is drawn, this ordering is usually indicated by listing the children of a node in a left-to-right manner according to the ordering. The order of the children of x_2 in T is x_4, x_5 , and x_6 .

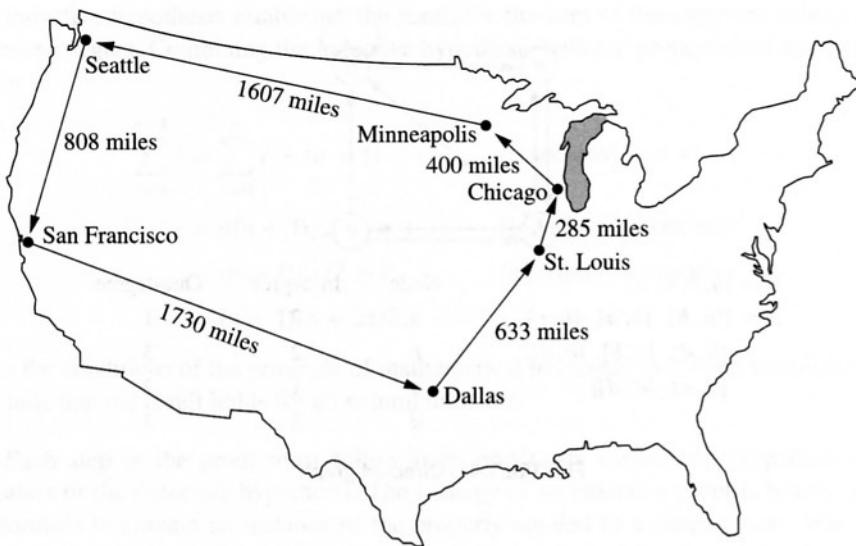


FIGURE 1.5 Labeled directed graph.

A node with out-degree zero is called a **leaf**. All other nodes are referred to as internal nodes. The *depth* of the root is zero; the depth of any other node is the depth of its parent plus one. The height or depth of a tree is the maximum of the depths of the nodes in the tree.

A node y is called a *descendant* of a node x , and x an *ancestor* of y , if there is a path from x to y . With this definition, each node is an ancestor and descendant of itself. The ancestor and descendant relations can be defined recursively using the adjacency relation (Exercises 43 and 44). The *minimal common ancestor* of two nodes x and y is an ancestor of both and a descendant of all other common ancestors. In the tree in Figure 1.6(a), the minimal common ancestor of x_{10} and x_{11} is x_5 , of x_{10} and x_6 is x_2 , and of x_{10} and x_{14} is x_1 .

A subtree of a tree T is a subgraph of T that is a tree in its own right. The set of descendants of a node x and the restriction of the adjacency relation to this set form a subtree with root x . This tree is called the subtree generated by x .

The ordering of siblings in the tree can be extended to a relation LEFTOF on $N \times N$. LEFTOF attempts to capture the property of one node being to the left of another in the diagram of a tree. For two nodes x and y , neither of which is an ancestor of the other, the relation LEFTOF is defined in terms of the subtrees generated by the minimal common ancestor of the nodes. Let z be the minimal common ancestor of x and y and let z_1, z_2, \dots, z_n be the children of z in their correct order. Then x is in the subtree generated by one of the children of z , call it z_i . Similarly, y is in the subtree generated by z_j for some j . Since z is the minimal common ancestor of x and y , $i \neq j$. If $i < j$, then $[x, y] \in \text{LEFTOF}$; $[y, x] \in \text{LEFTOF}$ otherwise. With this definition, no node is LEFTOF one of its ancestors. If x_{13} were to the left of x_{12} , then x_{10} must also be to the left of x_5 , since they are both the first

child of
of the d
The
a tree is
frontier
Wh
can be us
demonstr
binary tr

Example

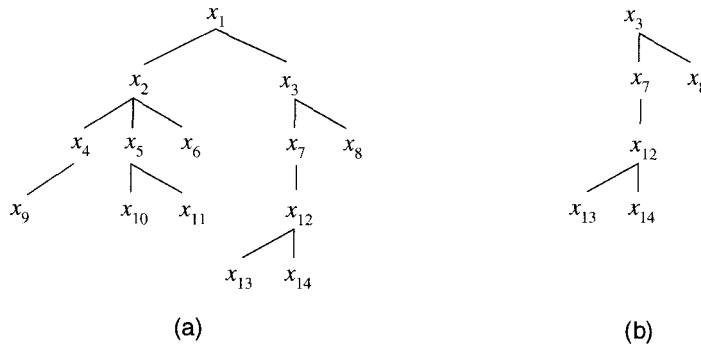
A tree in
a leaf or
binary tre

- i) Basis
- ii) Recur
where

is a st

- iii) Closu
a finit

A strictly
binary tree
denote the
 $2/lv(T) - 1$

FIGURE 1.6 (a) Tree with root x_1 . (b) Subtree generated by x_3 .

child of their parent. The appearance of being to the left or right of an ancestor is a feature of the diagram, not a property of the ordering of the nodes.

The relation LEFTOF can be used to order the set of leaves of a tree. The **frontier** of a tree is constructed from the leaves in the order generated by the relation LEFTOF. The frontier of T is the sequence $x_9, x_{10}, x_{11}, x_6, x_{13}, x_{14}, x_8$.

When a family of graphs is defined recursively, the principle of mathematical induction can be used to prove that properties hold for all graphs in the family. We will use induction to demonstrate a relationship between the number of leaves and the number of arcs in strictly binary trees, trees in which each node is either a leaf or has two children.

Example 1.8.1

A tree in which each node has at most two children is called a **binary tree**. If each node is a leaf or has exactly two children, the tree is called *strictly binary*. The family of strictly binary trees can be defined recursively as follows:

- Basis: A directed graph $T = (\{r\}, \emptyset, r)$ is a strictly binary tree.
- Recursive step: If $T_1 = (N_1, A_1, r_1)$ and $T_2 = (N_2, A_2, r_2)$ are strictly binary trees, where N_1 and N_2 are disjoint and $r \notin N_1 \cup N_2$, then

$$T = (N_1 \cup N_2 \cup \{r\}, A_1 \cup A_2 \cup \{\{r, r_1\}, [r, r_2]\}, r)$$

is a strictly binary tree.

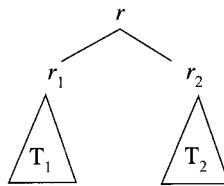
- Closure: T is a strictly binary tree only if it can be obtained from the basis elements by a finite number of applications of the construction given in the recursive step.

A strictly binary tree is either a single node or is constructed from two distinct strictly binary trees by the addition of a root and arcs to the two subtrees. Let $lv(T)$ and $arc(T)$ denote the number of leaves and arcs in a strictly binary tree T . We prove by induction that $2lv(T) - 2 = arc(T)$ for all strictly binary trees.

Basis: The basis consists of strictly binary trees of the form $(\{r\}, \emptyset, r)$. The equality clearly holds in this case since a tree of this form has one leaf and no arcs.

Inductive Hypothesis: Assume that every strictly binary tree T generated by n or fewer applications of the recursive step satisfies $2lv(T) - 2 = arc(T)$.

Inductive Step: Let T be a strictly binary tree generated by $n + 1$ applications of the recursive step in the definition of the family of strictly binary trees. T is built from a node r and two previously constructed strictly binary trees T_1 and T_2 with roots r_1 and r_2 , respectively.



The node r is not a leaf since it has arcs to the roots of T_1 and T_2 . Consequently, $lv(T) = lv(T_1) + lv(T_2)$. The arcs of T consist of the arcs of the component trees plus the two arcs from r .

Since T_1 and T_2 are strictly binary trees generated by n or fewer applications of the recursive step, we may employ the inductive hypothesis to establish the desired equality. By the inductive hypothesis,

$$2lv(T_1) - 2 = arc(T_1)$$

$$2lv(T_2) - 2 = arc(T_2).$$

Now,

$$\begin{aligned} arc(T) &= arc(T_1) + arc(T_2) + 2 \\ &= 2lv(T_1) - 2 + 2lv(T_2) - 2 + 2 \\ &= 2(lv(T_1) + lv(T_2)) - 2 \\ &= 2(lv(T)) - 2, \end{aligned}$$

as desired. □

Exercises

1. Let $X = \{1, 2, 3, 4\}$ and $Y = \{0, 2, 4, 6\}$. Explicitly define the sets described in parts (a) to (e).

- a) $X \cup Y$
- b) $X \cap Y$
- c) $X - Y$

- d) $Y - X$
- e) $\mathcal{P}(X)$

2. L
a)
b)
c)
3. L
4. L
* 5. P
6. Gi
a)
b)
c)
d)
7. Pro
8. Let
pos
9. Give
a)
b)
c)
10. Let
≡ is
11. Let
an e
12. Show
13. Let
 $p \geq$
 \equiv_p .
14. Let X
equiva
15. A bi
 $[m, n$
in \mathbb{N}
16. Prove
17. Prove

clearly

fewer

cursive
nd two
ely. $v(T) =$
wo arcss of the
quality.

□

in parts

2. Let $X = \{a, b, c\}$ and $Y = \{1, 2\}$.
 - a) List all the subsets of X .
 - b) List the members of $X \times Y$.
 - c) List all total functions from Y to X .
3. Let $X = \{3^n \mid n > 0\}$ and $Y = \{3n \mid n \geq 0\}$. Prove that $X \subseteq Y$.
4. Let $X = \{n^3 + 3n^2 + 3n \mid n \geq 0\}$ and $Y = \{n^3 - 1 \mid n > 0\}$. Prove that $X = Y$.
- * 5. Prove DeMorgan's Laws. Use the definition of set equality to establish the identities.
6. Give functions $f : N \rightarrow N$ that satisfy the following.
 - a) f is total and one-to-one but not onto.
 - b) f is total and onto but not one-to-one.
 - c) f is total, one-to-one, and onto but not the identity.
 - d) f is not total but is onto.
7. Prove that the function $f : N \rightarrow N$ defined by $f(n) = n^2 + 1$ is one-to-one but not onto.
8. Let $f : R^+ \rightarrow R^+$ be the function defined by $f(x) = 1/x$, where R^+ denotes the set of positive real numbers. Prove that f is one-to-one and onto.
9. Give an example of a binary relation on $N \times N$ that is
 - a) reflexive and symmetric but not transitive.
 - b) reflexive and transitive but not symmetric.
 - c) symmetric and transitive but not reflexive.
10. Let \equiv be the binary relation on N defined by $n \equiv m$ if, and only if, $n = m$. Prove that \equiv is an equivalence relation. Describe the equivalence classes of \equiv .
11. Let \equiv be the binary relation on N defined by $n \equiv m$ for all $n, m \in N$. Prove that \equiv is an equivalence relation. Describe the equivalence classes of \equiv .
12. Show that the binary relation LT, less than, is not an equivalence relation.
13. Let \equiv_p be the binary relation on N defined by $n \equiv_p m$ if $n \bmod p = m \bmod p$. For $p \geq 2$, prove that \equiv_p is an equivalence relation. Describe the equivalence classes of \equiv_p .
14. Let X_1, \dots, X_n be a partition of a set X . Define an equivalence relation \equiv on X whose equivalence classes are precisely the sets X_1, \dots, X_n .
15. A binary relation \equiv is defined on ordered pairs of natural numbers as follows: $[m, n] \equiv [j, k]$ if, and only if, $m + k = n + j$. Prove that \equiv is an equivalence relation in $N \times N$.
16. Prove that the set of even natural numbers is denumerable.
17. Prove that the set of even integers is denumerable.

- * 18. Prove that the set of nonnegative rational numbers is denumerable.
19. Prove that the union of two disjoint countable sets is countable.
20. Prove that there are an uncountable number of total functions from \mathbb{N} to $\{0, 1\}$.
21. A total function f from \mathbb{N} to \mathbb{N} is said to be *repeating* if $f(n) = f(n + 1)$ for some $n \in \mathbb{N}$. Otherwise, f is said to be *nonrepeating*. Prove that there are an uncountable number of repeating functions. Also prove that there are an uncountable number of nonrepeating functions.
22. A total function f from \mathbb{N} to \mathbb{N} is *monotone increasing* if $f(n) < f(n + 1)$ for all $n \in \mathbb{N}$. Prove that there are an uncountable number of monotone increasing functions.
23. Prove that there are uncountably many total functions from \mathbb{N} to \mathbb{N} that have a fixed point. See Example 1.4.3 for the definition of a fixed point.
24. A total function f from \mathbb{N} to \mathbb{N} is *nearly identity* if $f(n) = n - 1$, n , or $n + 1$ for every n . Prove that there are uncountably many nearly identity functions.
- * 25. Prove that the set of real numbers in the interval $[0, 1]$ is uncountable. Hint: Use the diagonalization argument on the decimal expansion of real numbers. Be sure that each number is represented by only one infinite decimal expansion.
26. Let F be the set of total functions of the form $f : \{0, 1\} \rightarrow \mathbb{N}$ (functions that map from $\{0, 1\}$ to the natural numbers). Is the set of such functions countable or uncountable? Prove your answer.
27. Prove that the binary relation on sets defined by $X \equiv Y$ if, and only if, $\text{card}(X) = \text{card}(Y)$ is an equivalence relation.
- * 28. Prove the Schröder-Bernstein Theorem.
29. Give a recursive definition of the relation *is equal to* on $\mathbb{N} \times \mathbb{N}$ using the operator s .
30. Give a recursive definition of the relation *greater than* on $\mathbb{N} \times \mathbb{N}$ using the successor operator s .
31. Give a recursive definition of the set of points $[m, n]$ that lie on the line $n = 3m$ in $\mathbb{N} \times \mathbb{N}$. Use s as the operator in the definition.
32. Give a recursive definition of the set of points $[m, n]$ that lie on or under the line $n = 3m$ in $\mathbb{N} \times \mathbb{N}$. Use s as the operator in the definition.
33. Give a recursive definition of the operation of multiplication of natural numbers using the operations s and addition.
34. Give a recursive definition of the predecessor operation

$$\text{pred}(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$$

using the operator s .

35. Su

The
sub36. Le
as* 37. Gi
s a

38. Pro

39. Pro

40. Pro

41. Pro

42. Let
set I
recu

i)

ii)

iii)

a)

b)

o

a

d

c)

n

43. Give
paths
This

44. Give

45. List t

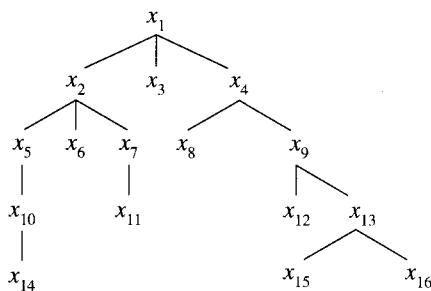
35. Subtraction on the set of natural numbers is defined by

$$n - m = \begin{cases} n - m & \text{if } n > m \\ 0 & \text{otherwise.} \end{cases}$$

This operation is often called *proper subtraction*. Give a recursive definition of proper subtraction using the operations s and $pred$.

36. Let X be a finite set. Give a recursive definition of the set of subsets of X . Use union as the operator in the definition.
- * 37. Give a recursive definition of the set of finite subsets of N . Use union and the successor s as the operators in the definition.
38. Prove that $2 + 5 + 8 + \dots + (3n - 1) = n(3n + 1)/2$ for all $n > 0$.
39. Prove that $1 + 2 + 2^2 + \dots + 2^n = 2^{n+1} - 1$ for all $n \geq 0$.
40. Prove $1 + 2^n < 3^n$ for all $n > 2$.
41. Prove that 3 is a factor of $n^3 - n + 3$ for all $n \geq 0$.
42. Let $P = \{A, B\}$ be a set consisting of two proposition letters (Boolean variables). The set E of well-formed conjunctive and disjunctive Boolean expressions over P is defined recursively as follows:
- i) Basis: $A, B \in E$.
 - ii) Recursive step: If $u, v \in E$, then $(u \vee v) \in E$ and $(u \wedge v) \in E$.
 - iii) Closure: An expression is in E only if it is obtained from the basis by a finite number of iterations of the recursive step.
- a) Explicitly give the Boolean expressions in the sets E_0, E_1 , and E_2 .
- b) Prove by mathematical induction that for every Boolean expression in E , the number of occurrences of proposition letters is one more than the number of operators. For an expression u , let $n_p(u)$ denote the number of proposition letters in u and $n_o(u)$ denote the number of operators in u .
- c) Prove by mathematical induction that, for every Boolean expression in E , the number of left parentheses is equal to the number of right parentheses.
43. Give a recursive definition of all the nodes in a directed graph that can be reached by paths from a given node x . Use the adjacency relation as the operation in the definition. This definition also defines the set of descendants of a node in a tree.
44. Give a recursive definition of the set of ancestors of a node x in a tree.
45. List the members of the relation LEFTOF for the tree in Figure 1.6(a).

46. Using the tree below, give the values of each of the items in parts (a) to (e).



- a) the depth of the tree
 - b) the ancestors of x_{11}
 - c) the minimal common ancestor of x_{14} and x_{11} , of x_{15} and x_{11}
 - d) the subtree generated by x_2
 - e) the frontier of the tree
47. Prove that a strictly binary tree with n leaves contains $2n - 1$ nodes.
48. A **complete binary tree** of depth n is a strictly binary tree in which every node on levels $1, 2, \dots, n - 1$ is a parent and each node on level n is a leaf. Prove that a complete binary tree of depth n has $2^{n+1} - 1$ nodes.

Bibliographic Notes

The topics presented in this chapter are normally covered in a first course in discrete mathematics. A comprehensive presentation of the discrete mathematical structures important to the foundations of computer science can be found in Bobrow and Arbib [1974].

There are a number of classic books that provide detailed presentations of the topics introduced in this chapter. An introduction to set theory can be found in Halmos [1974], Stoll [1963], and Fraenkel, Bar-Hillel, and Levy [1984]. The latter begins with an excellent description of Russell's paradox and other antinomies arising in set theory. The diagonalization argument was originally presented by Cantor in 1874 and is reproduced in Cantor [1947]. The texts by Wilson [1985], Ore [1963], Bondy and Murty [1977], and Busacker and Saaty [1965] introduce the theory of graphs. Induction, recursion, and their relationship to theoretical computer science are covered in Wand [1980].

CHAPTER 2

Languages

The concept of language includes a variety of seemingly distinct categories including natural languages, computer languages, and mathematical languages. A general definition of language must encompass all of these various types of languages. In this chapter, a purely set-theoretic definition of language is given: A language is a set of strings over an alphabet. The alphabet is the set of symbols of the language and a string over the alphabet is a finite sequence of symbols from the alphabet.

Although strings are inherently simple structures, their importance in communication and computation cannot be overemphasized. The sentence “The sun did not shine” is a string of English words. The alphabet of the English language is the set of words and punctuation symbols that can occur in sentences. The mathematical equation

$$p = (n \times r \times t)/v$$

is a string consisting of variable names, operators, and parentheses. A digital photograph is stored as a bit string, a sequence of 0's and 1's. In fact, all data stored and manipulated by computers are represented as bit strings. As computer users, we frequently input information to the computer and receive output in the form of text strings. The source code of a computer program is a text string made up of the keywords, identifiers, and special symbols that constitute the alphabet of the programming language. Because of the importance of strings, we begin this chapter by formally defining the notion of string and studying the properties of operations on strings.

Languages of interest are not made up of arbitrary strings; not all strings of English words are sentences and not all strings of source code are legitimate computer programs. Languages consist of strings that satisfy certain requirements and restrictions that define the

syntax of the language. In this chapter, we will use recursive definitions and set operations to enforce syntactic restrictions on the strings of a language.

We will also introduce the family of languages defined by regular expressions. A regular expression describes a pattern and the language associated with the regular expression consists of all strings that match the pattern. Although we introduce the regular expressions via a set-theoretic construction, as we progress we will see that these languages occur naturally as the languages generated by regular grammars and accepted by finite-state machines. The chapter concludes by examining the use of regular expressions in searching and pattern matching.

2.1 Strings and Languages

The description of a language begins with the identification of its alphabet, the set of symbols that occur in the language. The elements of the language are finite-length strings of alphabet symbols. Consequently, the study of languages requires an understanding of the operations that generate and manipulate strings. In this section we give precise definitions of a string over an alphabet and of the basic string operations.

The sole requirement for an alphabet is that it consists of a finite number of indivisible objects. The alphabet of a natural language, like English or French, consists of the words and punctuation marks of the language. The symbols in the alphabet of the language are considered to be indivisible objects. The word *language* cannot be divided into *lang* and *uage*. The word *format* has no relation to the words *for* and *mat*; these are all distinct members of the alphabet. A string over this alphabet is a sequence of words and punctuation symbols. The sentence that you have just read is such a string. The alphabet of a computer language consists of the permissible keywords, identifiers, and symbols of the language. A string over this alphabet is a sequence of source code.

Because the elements of the alphabet of a language are indivisible, we will generally denote them by single characters. Letters a , b , c , d , e , with or without subscripts, are used to represent the elements of an alphabet and Σ is used to denote an alphabet. Strings over an alphabet are represented by letters occurring near the end of the alphabet. In particular, p , q , u , v , w , x , y , z are used to denote strings. The notation used for natural languages and computer languages provides an exception to this convention. In these cases, the alphabet consists of the indivisible elements of the particular language.

A string has been defined informally as a sequence of elements from an alphabet. In order to establish the properties of strings, the set of strings over an alphabet is defined recursively. The basis consists of the string containing no elements. This string is called the **null string** and denoted λ . The primitive operator used in the definition consists of adjoining a single element from the alphabet to the right-hand side of an existing string.

Definition 2.1.1

Let Σ be an alphabet. Σ^* , the set of strings over Σ , is defined recursively as follows:

- i) Basis: $\lambda \in \Sigma^*$.
- ii) Recursive step: If $w \in \Sigma^*$ and $a \in \Sigma$, then $wa \in \Sigma^*$.
- iii) Closure: $w \in \Sigma^*$ only if it can be obtained from λ by a finite number of applications of the recursive step.

For any nonempty alphabet Σ , Σ^* contains infinitely many elements. If $\Sigma = \{a\}$, Σ^* contains the strings $\lambda, a, aa, aaa, \dots$. The length of a string w , intuitively the number of elements in the string or formally the number of applications of the recursive step needed to construct the string from the elements of the alphabet, is denoted $\text{length}(w)$. If Σ contains n elements, there are n^k strings of length k in Σ^* .

Example 2.1.1

Let $\Sigma = \{a, b, c\}$. The elements of Σ^* include

Length 0:	λ
Length 1:	$a \ b \ c$
Length 2:	$aa \ ab \ ac \ ba \ bb \ bc \ ca \ cb \ cc$
Length 3:	$aaa \ aab \ aac \ aba \ abb \ abc \ aca \ acb \ acc$ $baa \ bab \ bac \ bba \ bbb \ bbc \ bca \ bcb \ bcc$ $caa \ cab \ cac \ cba \ cbb \ cbc \ cca \ ccb \ ccc$

□

By our informal definition, a language consists of strings over an alphabet. For example, the English language consists of those strings of words that we call sentences. Not all strings of words form sentences, only those satisfying certain conditions on the order and type of the constituent words. The collection of rules, requirements, and restrictions that specify the correctly formed sentences defines the syntax of the language. These observations lead to our formal definition of language; a language consists of a subset of the set of all possible strings over the alphabet.

Definition 2.1.2

A language over an alphabet Σ is a subset of Σ^* .

Since strings are the elements of a language, we must examine the properties of strings and the operations on them. Concatenation, taking two strings and "gluing them together," is the fundamental operation in the generation of strings. A formal definition of concatenation is given by recursion on the length of the second string in the concatenation. At this point, the primitive operation of adjoining a single member of the alphabet to the right-hand side of a string is the only operation on strings that has been introduced. Thus any new operation must be defined in terms of it.

Definition 2.1.3

Let $u, v \in \Sigma^*$. The **concatenation** of u and v , written uv , is a binary operation on Σ^* defined

- i) Basis: If $\text{length}(v) = 0$, then $v = \lambda$ and $uv = u$.
- ii) Recursive step: Let v be a string with $\text{length}(v) = n > 0$. Then $v = wa$, for some string w with length $n - 1$ and $a \in \Sigma$, and $uv = (uw)a$.

Example 2.1.2

Let $u = ab$, $v = ca$, and $w = bb$. Then

$$\begin{array}{ll} uv = abca & vw = cabb \\ (uv)w = abcabb & u(vw) = abcabb. \end{array}$$
□

The result of the concatenation of u , v , and w is independent of the order in which the operations are performed. Mathematically, this property is known as *associativity*. Theorem 2.1.4 proves that concatenation is an associative binary operation.

Theorem 2.1.4

Let $u, v, w \in \Sigma^*$. Then $(uv)w = u(vw)$.

Proof. The proof is by induction on the length of the string w . The string w was chosen for compatibility with the recursive definition of strings, which builds on the right-hand side of an existing string.

Basis: $\text{length}(w) = 0$. Then $w = \lambda$, and $(uv)w = uv$ by the definition of concatenation. On the other hand, $u(vw) = u(v) = uv$.

Inductive Hypothesis: Assume that $(uv)w = u(vw)$ for all strings w of length n or less.

Inductive Step: We need to prove that $(uv)w = u(vw)$ for all strings w of length $n + 1$. Let w be such a string. Then $w = xa$ for some string x of length n and $a \in \Sigma$ and

$$\begin{aligned} (uv)w &= (uv)(xa) && \text{(substitution, } w = xa\text{)} \\ &= ((uv)x)a && \text{(definition of concatenation)} \\ &= (u(vx))a && \text{(inductive hypothesis)} \\ &= u((vx)a) && \text{(definition of concatenation)} \\ &= u(v(xa)) && \text{(definition of concatenation)} \\ &= u(vw) && \text{(substitution, } xa = w\text{).} \end{aligned}$$
■

Since associativity guarantees the same result regardless of the order of the operations, parentheses are omitted from a sequence of applications of concatenation. Exponents are used to abbreviate the concatenation of a string with itself. Thus uu may be written u^2 , uuu may be written u^3 , and so on. For completeness, u^0 , which represents concatenating u with itself zero times, is defined to be the null string. The operation of concatenation is not commutative. For strings $u = ab$ and $v = ba$, $uv = abba$ and $vu = baab$. Note that $u^2 = abab$ and not $aabb = a^2b^2$.

Substrings can be defined using the operation of concatenation. Intuitively, u is a substring of v if u “occurs inside of” v . Formally, u is a *substring* of v if there are strings

x and y such that $v = xuy$. A *prefix* of v is a substring u in which x is the null string in the decomposition of v . That is, $v = uy$. Similarly, u is a *suffix* of v if $v = xu$.

The reversal of a string is the string written backward. The reversal of $abbc$ is $cbba$. Like concatenation, this unary operation is also defined recursively on the length of the string. Removing an element from the right-hand side of a string constructs a smaller string that can then be used in the recursive step of the definition. Theorem 2.1.6 establishes the relationship between the operations of concatenation and reversal.

Definition 2.1.5

Let u be a string in Σ^* . The **reversal** of u , denoted u^R , is defined as follows:

- i) Basis: If $\text{length}(u) = 0$, then $u = \lambda$ and $\lambda^R = \lambda$.
- ii) Recursive step: If $\text{length}(u) = n > 0$, then $u = wa$ for some string w with length $n - 1$ and some $a \in \Sigma$, and $u^R = aw^R$.

Theorem 2.1.6

Let $u, v \in \Sigma^*$. Then $(uv)^R = v^R u^R$.

Proof. The proof is by induction on the length of the string v .

Basis: If $\text{length}(v) = 0$, then $v = \lambda$ and $(uv)^R = u^R$. Similarly, $v^R u^R = \lambda^R u^R = u^R$.

Inductive Hypothesis: Assume $(uv)^R = v^R u^R$ for all strings v of length n or less.

Inductive Step: We must prove that, for any string v of length $n + 1$, $(uv)^R = v^R u^R$. Let v be a string of length $n + 1$. Then $v = wa$, where w is a string of length n and $a \in \Sigma$. The inductive step is established by

$$\begin{aligned}
 (uv)^R &= (u(wa))^R \\
 &= ((uw)a)^R && \text{(associativity of concatenation)} \\
 &= a(uw)^R && \text{(definition of reversal)} \\
 &= a(w^R u^R) && \text{(inductive hypothesis)} \\
 &= (aw^R)u^R && \text{(associativity of concatenation)} \\
 &= (wa)^R u^R && \text{(definition of reversal)} \\
 &= v^R u^R.
 \end{aligned}$$

■

2.2 Finite Specification of Languages

A language has been defined as a set of strings over an alphabet. Languages of interest do not consist of arbitrary sets of strings but rather of strings that satisfy some prescribed syntactic requirements. The specification of a language requires an unambiguous description of the strings of the language. A finite language can be explicitly defined by enumerating its elements. Several infinite languages with simple syntactic requirements are defined recursively in the examples that follow.

Example 2.2.1

The language L of strings over $\{a, b\}$ in which each string begins with an a and has even length is defined by

- i) Basis: $aa, ab \in L$.
- ii) Recursive step: If $u \in L$, then $ua, uab, uba, ubb \in L$.
- iii) Closure: A string $u \in L$ only if it can be obtained from the basis elements by a finite number of applications of the recursive step.

The strings in L are built by adjoining two elements to the right-hand side of a previously constructed string. The basis ensures that each string in L begins with an a . Adding substrings of length two maintains the even parity. \square

Example 2.2.2

The language L over the alphabet $\{a, b\}$ defined by

- i) Basis: $\lambda \in L$;
- ii) Recursive step: If $u \in L$, then $ua, uab \in L$;
- iii) Closure: A string $u \in L$ only if it can be obtained from the basis element by a finite number of applications of the recursive step;

consists of strings in which each occurrence of b is immediately preceded by an a . For example, $\lambda, a, abaab$ are in L and bb, bab, abb are not in L. \square

The recursive step in the preceding examples concatenated elements to the end of an existing string. Breaking a string into substrings permits the addition of elements anywhere within the original string. This technique is illustrated in the following example.

Example 2.2.3

Let L be the language over the alphabet $\{a, b\}$ defined by

- i) Basis: $\lambda \in L$.
- ii) Recursive step: If $u \in L$ and u can be written $u = xyz$, then $xaybz \in L$ and $xaybz \in L$.
- iii) Closure: A string $u \in L$ only if it can be obtained from the basis element by a finite number of applications of the recursive step.

The language L consists of all strings with the same number of a 's and b 's. The first construction in the recursive step, $xaybz \in L$, consists of the following three actions:

1. Select a string u that is already in L.
2. Divide u into three substrings x, y, z such that $u = xyz$. Note that any of the substrings may be λ .
3. Insert an a between x and y and a b between y and z .

Taken together, the two rules can be intuitively interpreted as “insert one a and one b anywhere in the string u .¹” \square

Recursive definitions provide a tool for defining the strings of a language. Examples 2.2.1, 2.2.2, and 2.2.3 have shown that requirements on order, positioning, and parity can be obtained using a recursive generation of strings. The process of generating strings using a single recursive definition, however, is unsuitable for enforcing the complex syntactic requirements of natural or computer languages.

Another technique for constructing languages is to use set operations to construct complex sets of strings from simpler ones. An operation defined on strings can be extended to an operation on sets, hence on languages. Descriptions of infinite languages can then be constructed from finite sets using the set operations. The next two definitions introduce operations on sets of strings that will be used for both language definition and pattern specification.

Definition 2.2.1

The concatenation of languages X and Y , denoted XY , is the language

$$XY = \{uv \mid u \in X \text{ and } v \in Y\}.$$

The concatenation of X with itself n times is denoted X^n . X^0 is defined as $\{\lambda\}$.

Example 2.2.4

Let $X = \{a, b, c\}$ and $Y = \{abb, ba\}$. Then

$$XY = \{aabb, babb, cabb, aba, bba, cba\}$$

$$X^0 = \{\lambda\}$$

$$X^1 = X = \{a, b, c\}$$

$$X^2 = XX = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$$

$$\begin{aligned} X^3 = X^2 X = & \{aaa, aab, aac, aba, abb, abc, aca, acb, acc, \\ & baa, bab, bac, bba, bbb, bbc, bca, bcb, bcc, \\ & caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc\}. \end{aligned}$$

\square

The sets in the previous example should look familiar. For each i , X^i contains the strings of length i in Σ^* given in Example 2.1.1. This observation leads to another set operation, the Kleene star of a set X , denoted X^* . Using the $*$ operator, the strings over a set can be defined with the operations of concatenation and union rather than with the primitive operation of Definition 2.1.1.

Definition 2.2.2

Let X be a set. Then

$$X^* = \bigcup_{i=0}^{\infty} X^i \quad \text{and} \quad X^+ = \bigcup_{i=1}^{\infty} X^i.$$

The set X^* contains all strings that can be built from the elements of X . If X is an alphabet, X^+ is the set of all nonnull strings over X . An alternative definition of X^+ using concatenation and the Kleene star is $X^+ = XX^*$.

The definition of a formal language requires an unambiguous specification of the strings that belong to the language. Describing languages informally lacks the rigor required for a precise definition. Consider the language over $\{a, b\}$ consisting of all strings that contain the substring bb . Does this mean that a string in the language contains exactly one occurrence of bb , or are multiple substrings bb permitted? This could be answered by specifically describing the strings as containing exactly one or at least one occurrence of bb . However, these types of questions are inherent in the imprecise medium provided by natural languages.

The precision afforded by set operations can be used to give an unambiguous description of the strings of a language. Example 2.2.5 gives a set theoretic definition of the strings that contain the substring bb . In this definition it is clear that the language contains all strings in which bb occurs at least once.

Example 2.2.5

The language $L = \{a, b\}^*\{bb\}\{a, b\}^*$ consists of the strings over $\{a, b\}$ that contain the substring bb . The concatenation of $\{bb\}$, which contains the single string bb , ensures the presence of bb in every string in L . The sets $\{a, b\}^*$ permit any number of a 's and b 's, in any order, to precede and follow the occurrence of bb . In particular, additional copies of the substring bb may occur before or after the occurrence ensured by the concatenation of $\{bb\}$. \square

Example 2.2.6

Concatenation can be used to specify the order of components of strings. Let L be the language that consists of all strings that begin with aa or end with bb . The set $\{aa\}\{a, b\}^*$ describes the strings with prefix aa . Similarly, $\{a, b\}^*\{bb\}$ is the set of strings with suffix bb . Thus $L = \{aa\}\{a, b\}^* \cup \{a, b\}^*\{bb\}$. \square

Example 2.2.7

Let $L_1 = \{bb\}$ and $L_2 = \{\lambda, bb, bbbb\}$ be languages over $\{b\}$. The languages L_1^* and L_2^* both contain precisely the strings consisting of an even number of b 's. Note that λ , with length zero, is an element of both L_1^* and L_2^* . \square

Example 2.2.8

The set $\{aa, bb, ab, ba\}^*$ consists of all even-length strings over $\{a, b\}$. The repeated concatenation constructs strings by adding two elements at a time. The set of strings of odd length can be defined by $\{a, b\}^* - \{aa, bb, ab, ba\}^*$. This set can also be obtained by concatenating a single element to the even-length strings. Thus the odd-length strings are also defined by $\{aa, bb, ab, ba\}^* \{a, b\}$. \square

2.3 Regular Sets and Expressions

In the previous section we used set operations to construct new languages from existing ones. The operators were selected to ensure that certain patterns occurred in the strings of the language. In this section we follow the approach of constructing languages from set operations but limit the sets and operations that are allowed in the construction process.

A set of strings is regular if it can be generated from the empty set, the set containing the null string, and sets containing a single element of the alphabet using union, concatenation, and the Kleene star operation. The regular sets, defined recursively in Definition 2.3.1, comprise a family of languages that play an important role in formal languages, pattern recognition, and the theory of finite-state machines.

Definition 2.3.1

Let Σ be an alphabet. The **regular sets** over Σ are defined recursively as follows:

- Basis: $\emptyset, \{\lambda\}$ and $\{a\}$, for every $a \in \Sigma$, are regular sets over Σ .
- Recursive step: Let X and Y be regular sets over Σ . The sets

$$X \cup Y$$

$$XY$$

$$X^*$$

are regular sets over Σ .

- Closure: X is a regular set over Σ only if it can be obtained from the basis elements by a finite number of applications of the recursive step.

A language is called **regular** if it is defined by a regular set. The following examples show how regular sets can be used to describe the strings of a language.

Example 2.3.1

The language from Example 2.2.5, the set of strings containing the substring bb , is a regular set over $\{a, b\}$. From the basis of the definition, $\{a\}$ and $\{b\}$ are regular sets. The union of $\{a\}$ and $\{b\}$ and the Kleene star operation produce $\{a, b\}^*$, the set of all strings over

$\{a, b\}$. By concatenation, $\{b\}\{b\} = \{bb\}$ is regular. Applying concatenation twice yields $\{a, b\}^*\{bb\}\{a, b\}^*$. \square

Example 2.3.2

The set of strings that begin and end with an a and contain at least one b is regular over $\{a, b\}$. The strings in this set could be described intuitively as “an a , followed by any string, followed by a b , followed by any string, followed by an a .” The concatenation

$$\{a\}\{a, b\}^*\{b\}\{a, b\}^*\{a\}$$

exhibits the regularity of the set. \square

By definition, regular sets are those that can be built from the empty set, the set containing the null string, and the sets containing a single element of the alphabet using the operations of union, concatenation, and Kleene star. Regular expressions are used to abbreviate the descriptions of regular sets. The regular sets \emptyset , $\{\lambda\}$, and $\{a\}$ are represented by \emptyset , λ , and a , removing the need for the set brackets $\{ \}$. The set operations of union, Kleene star, and concatenation are designated by \cup , $*$, and juxtaposition, respectively. Parentheses are used to indicate the order of the operations.

Definition 2.3.2

Let Σ be an alphabet. The **regular expressions** over Σ are defined recursively as follows:

- i) Basis: \emptyset , λ , and a , for every $a \in \Sigma$, are regular expressions over Σ .
- ii) Recursive step: Let u and v be regular expressions over Σ . The expressions

$$(u \cup v)$$

$$(uv)$$

$$(u^*)$$

are regular expressions over Σ .

- iii) Closure: u is a regular expression over Σ only if it can be obtained from the basis elements by a finite number of applications of the recursive step.

Since union and concatenation are associative, parentheses can be omitted from expressions consisting of a sequence of one of these operations. To further reduce the number of parentheses, a precedence is assigned to the operators. The priority designates the Kleene star as the most binding operation, followed by concatenation and union. Employing these conventions, regular expressions for the sets in Examples 2.3.1 and 2.3.2 are $(a \cup b)^*bb(a \cup b)^*$ and $a(a \cup b)^*b(a \cup b)^*a$, respectively. The notation u^+ is used to abbreviate the expression uu^* . Similarly, u^2 denotes the regular expression uu , u^3 denotes u^2u , and so on.

Example 2.3.3

The set $\{bawab \mid w \in \{a, b\}^*\}$ is regular over $\{a, b\}$. The following table demonstrates the recursive generation of a regular set and the corresponding regular expression definition of the language. The column on the right gives the justification for the regularity of each of the components used in the recursive operations.

Set	Expression	Justification
1. $\{a\}$	a	Basis
2. $\{b\}$	b	Basis
3. $\{a\}\{b\} = \{ab\}$	ab	1, 2, concatenation
4. $\{a\} \cup \{b\} = \{a, b\}$	$a \cup b$	1, 2, union
5. $\{b\}\{a\} = \{ba\}$	ba	2, 1, concatenation
6. $\{a, b\}^*$	$(a \cup b)^*$	4, Kleene star
7. $\{ba\}\{a, b\}^*$	$ba(a \cup b)^*$	5, 6, concatenation
8. $\{ba\}\{a, b\}^*\{ab\}$	$ba(a \cup b)^* ab$	7, 3, concatenation

□

The preceding example illustrates how regular sets and regular expressions are generated from the basic regular sets. Every regular set can be obtained by a finite sequence of operations in the manner shown in Example 2.3.3.

A regular expression defines a pattern and a string is in the language of the expression only if it matches the pattern. Concatenation specifies order; a string w is in uv only if it consists of a string from u followed by one from v . The Kleene star permits repetition and \cup selection. The pattern specified by the regular expression in Example 2.3.3 requires ba to begin the string, ab to end it, and any combination of a 's and b 's to occur between the required prefix and suffix. The following examples further illustrate the ability of regular expressions to describe patterns.

Example 2.3.4

The regular expressions $(a \cup b)^*aa(a \cup b)^*$ and $(a \cup b)^*bb(a \cup b)^*$ represent the regular sets with strings containing aa and bb , respectively. Combining these two expressions with the \cup operator yields the expression $(a \cup b)^*aa(a \cup b)^* \cup (a \cup b)^*bb(a \cup b)^*$ representing the set of strings over $\{a, b\}$ that contain the substring aa or bb .

□

Example 2.3.5

A regular expression for the set of strings over $\{a, b\}$ that contain exactly two b 's must explicitly ensure the presence of two b 's. Any number of a 's may occur before, between, and after the b 's. Concatenating the required subexpressions produces $a^*ba^*ba^*$.

□

Example 2.3.6

The regular expressions

- i) $a^*ba^*b(a \cup b)^*$
- ii) $(a \cup b)^*ba^*ba^*$
- iii) $(a \cup b)^*b(a \cup b)^*b(a \cup b)^*$

define the set of strings over $\{a, b\}$ containing two or more b 's. As in Example 2.3.5, the presence of at least two b 's is ensured by the two instances of the expression b in the concatenation. \square

Example 2.3.7

Consider the regular set defined by the expression $a^*(a^*ba^*ba^*)^*$. The expression inside the parentheses is the regular expression from Example 2.3.5 representing the strings with exactly two b 's. The Kleene star generates the concatenation of any number of these strings. The result is the null string (no repetitions of the pattern) and all strings with a positive, even number of b 's. Strings consisting of only a 's are not included in $(a^*ba^*ba^*)^*$. Concatenating a^* to the beginning of the expression produces the set consisting of all strings with an even number of b 's. Another regular expression for this set is $a^*(ba^*ba^*)^*$. \square

Example 2.3.8

The ability of substrings to share elements complicates the construction of a regular expression for the set of strings that begin with ba , end with ab , and contain the substring aa . The expression $ba(a \cup b)^*aa(a \cup b)^*ab$ explicitly inserts each of the three components. Every string represented by this expression must contain at least four a 's. However, the string $baab$ satisfies the specification but only has two a 's. A regular expression for this language is

$$\begin{aligned} & ba(a \cup b)^*aa(a \cup b)^*ab \\ \cup \quad & baa(a \cup b)^*ab \\ \cup \quad & ba(a \cup b)^*aab \\ \cup \quad & baab. \end{aligned}$$

 \square

The construction of a regular expression is a positive process; features of the desired strings are explicitly inserted into the expression using concatenation, union, or the Kleene star. There is no negative operation to omit strings that have a particular property. To construct a regular expression for the set of strings that do not have a property, it is necessary

to formulate the condition in a positive manner and construct the regular expression using the reformulation of the language. The next two examples illustrate this approach.

Example 2.3.9

To construct a regular expression for the set of strings over $\{a, b\}$ that do not end in aaa , we must ensure that aaa is not a suffix of any string described by the expression. The possible endings for a string with a b in one of the final three positions are b , ba , or baa . The first part of the regular expression

$$(a \cup b)^*(b \cup ba \cup baa) \cup \lambda \cup a \cup aa$$

defines these strings. The final three expressions represent the special case of strings of length zero, one, and two that do not contain a b . \square

Example 2.3.10

The language L defined by $c^*(b \cup ac^*)^*$ consists of all strings over $\{a, b, c\}$ that do not contain the substring bc . The outer c^* and the ac^* inside the parentheses allow any number of a 's and c 's to occur in any order. A b can be followed by another b or a string from ac^* . The a at the beginning of ac^* blocks a b from directly preceding a c . To help develop your understanding of the representation of sets by expressions, convince yourself that both $acabacc$ and $bbaaacc$ are in the set represented by $c^*(b \cup ac^*)^*$. \square

Examples 2.3.6 and 2.3.7 show that the regular expression definition of a language is not unique. Two expressions that represent the same set are called *equivalent*. The identities in Table 2.1 can be used to algebraically manipulate regular expressions to construct equivalent expressions. These identities are the regular expression formulation of properties of union, concatenation, and the Kleene star operation.

Identity 5 follows from the commutativity of the union of sets. Identities 9 and 10 are the distributive laws of union and concatenation translated to the regular expression notation. The final set of expressions provides a number of equivalent representations of all strings made from elements of u and v . The identities in Table 2.1 can be used to simplify or to establish the equivalence of regular expressions.

Example 2.3.11

A regular expression is constructed to represent the set of strings over $\{a, b\}$ that do not contain the substring aa . A string in this set may contain a prefix of any number of b 's. All a 's must be followed by at least one b or terminate the string. The regular expression $b^*(ab^+)^* \cup b^*(ab^+)^*a$ generates the desired set by partitioning it into two disjoint subsets;

TABLE 2.1 Regular Expression Identities

1.	$\emptyset u = u\emptyset = \emptyset$
2.	$\lambda u = u\lambda = u$
3.	$\emptyset^* = \lambda$
4.	$\lambda^* = \lambda$
5.	$u \cup v = v \cup u$
6.	$u \cup \emptyset = u$
7.	$u \cup u = u$
8.	$u^* = (u^*)^*$
9.	$u(v \cup w) = uv \cup uw$
10.	$(u \cup v)w = uw \cup vw$
11.	$(uv)^*u = u(vu)^*$
12.	$\begin{aligned} (u \cup v)^* &= (u^* \cup v)^* \\ &= u^*(u \cup v)^* = (u \cup vu^*)^* \\ &= (u^*v^*)^* = u^*(vu^*)^* \\ &= (u^*v)^*u^* \end{aligned}$

the first consists of strings that end in b and the second of strings that end in a . This expression can be simplified using the identities from Table 2.1 as follows:

$$\begin{aligned} &b^*(ab^+)^* \cup b^*(ab^+)^*a \\ &= b^*(ab^+)^*(\lambda \cup a) \\ &= b^*(abb^*)^*(\lambda \cup a) \\ &= (b \cup ab)^*(\lambda \cup a). \end{aligned} \quad \square$$

While regular expressions allow us to describe many complex patterns, it is important to note that there are languages that cannot be defined by any regular expression. In Chapter 6 we will see that there is no regular expression that defines the language $\{a^i b^i \mid i \geq 0\}$.

2.4 Regular Expressions and Text Searching

A common application of regular expressions, perhaps the most common for the majority of computer users, is the specification of patterns for searching documents and files. In this section we will examine the use of regular expressions in two types of text searching applications.

The major difference between the use of regular expressions for language definition and for text searching is the scope of the desired match. A string is in the language defined by a regular expression if the entire string matches the pattern specified by regular expression.

For example, a string matches ab^+ only if it begins with an *a* and is followed by one or more *b*'s.

In text searching we are looking for the occurrence of a substring in the text that matches the desired pattern. Thus the words

about
abbot
rehabilitate
tabulate
abominable

would all be considered to match the pattern ab^+ . In fact, *abominable* would match it twice!

This brings up a difference between two types of text searching that can be described (somewhat simplistically) as off-line and online searching. By off-line search we mean that a search program is run, the input to the program is a pattern and a file, and the output consists of the lines or the text in the file that match the pattern. Frequently, off-line file searching is done using operating system utilities or programs written in a language designed for searching. GREP and awk are examples of the utilities available for file searching, and Perl is a programming language designed for file searching. We will use GREP, which is an acronym for "Global search for Regular Expression and Print," to illustrate this type of regular expression search.

Online search tools are provided by web browsers, text editors, and word processing systems. The objective is to interactively find the first, the next, or to sequentially find all occurrences of substrings that match the search pattern. The "Find" command in Microsoft Word will be used to demonstrate the differences between online and off-line pattern matching.

Since the desired patterns are generally entered on a keyboard, the regular expression notation used by search utilities should be concise and not contain superscripts. Although there is no uniform syntax for regular expressions in search applications, the notation used in the majority of the applications has many features in common. We will use the extended regular expression notation of GREP to illustrate the description of patterns for text searching.

The alphabet of the file or document frequently consists of the ASCII character set, which is given in Appendix III. This is considerably larger than the two or three element alphabets that we have used in most of our examples of regular expressions. With the alphabet $\{a, b\}$, the regular expression for any string is $(a \cup b)^*$. To write the expression for any string of ASCII characters using this format would require several lines and would be extremely inconvenient to enter on a keyboard. Two notational conventions, bracket expressions and range expressions, were introduced to facilitate the description of patterns over an extended alphabet.

The bracket notation [] is used to represent the union of alphabet symbols. For example, $[abcd]$ is equivalent to the expression $(a \cup b \cup c \cup d)$. Adding a caret immediately

TABLE 2.2 Extended Regular Expression Operations

Operation	Symbol	Example	Regular Expression
concatenation		ab	ab
		[a-c][AB]	$aA \cup aB \cup bA \cup bB \cup cA \cup cB$
Kleene star	*	[ab]*	$(a \cup b)^*$
disjunction		[ab]* A	$(a \cup b)^* \cup A$
zero or more	+	[ab]+	$(a \cup b)^+$
zero or one	?	a?	$(a \cup \lambda)$
one character	.	a.a	$a(a \cup b)a$ if $\Sigma = \{a, b\}$
n-times	{n}	a{4}	$aaaa = a^4$
n or more times	{n,}	a{4,}	$aaaaaa^*$
n to m times	{n,m}	a{4,6}	$aaaa \cup aaaaa \cup aaaaaa$

after the left bracket produces the complement of the union, thus $[\^abcd]$ designates all characters other than a, b, c, and d.

Range expressions use the ordering of the ASCII character set to describe a sequence of characters. For example, A-Z is the range expression that designates all capital letters. In the ASCII table these are the characters numbered from 65 to 90. Range expressions can be arguments in bracket expressions; [a-zA-Z0-9] represents the set of all letters and digits. In addition, certain frequently occurring subsets of characters are given their own mnemonic identifiers. For example, [:digit:], [:alpha:], and [:alnum:] are shorthand for [0-9], [a-zA-Z], and [a-zA-Z0-9]. The extended regular expression notation also includes symbols \< and \> that require the match to occur at the beginning or the end of a word.

Along with the standard operations of \cup , concatenation, and $*$, the extended regular expression notation of GREP contains additional operations on expressions. These operations do not extend the type of patterns that can be expressed, rather they are introduced to simplify the representation of patterns. A description of the extended regular expression operations are given in Table 2.2. A set of priorities and parentheses combine to define the scope of the operations.

The input to GREP is a pattern and file to be searched. GREP performs a line-by-line search on the file. If a line contains a substring that matches the pattern, the line is printed and the search continues with the subsequent line. To demonstrate pattern matching using extended regular expressions, we will search a file caesar containing Caesar's comments to his wife in Shakespeare's *Julius Caesar*, Act 2, Scene 2.

```
Cowards die many times before their deaths;
The valiant never taste of death but once.
Of all the wonders that I yet have heard.
It seems to me most strange that men should fear;
```

Seeing that death, a necessary end,
Will come when it will come.

We begin by looking for matches of the pattern `m[a-z]n`. This is matched by a substring of length three consisting of an `m` and an `n` separated by any single lowercase letter. The result of the search is

```
C:> grep -E "m[a-z]n" caesar
Cowards die many times before their deaths;
It seems to me most strange that men should fear;
```

The option `-E` in the GREP call indicates that the extended regular expression notation is used to describe the pattern, and the quotation marks delimit the pattern. The substring `man` in `many` and the word `men` match this pattern and the lines containing these strings are printed.

The search is now changed to find occurrences of `m` and `n` separated by any number of lowercase letters and blanks.

```
C:> grep -E "m[a-z ]*n" caesar
Cowards die many times before their deaths;
It seems to me most strange that men should fear;
Will come when it will come.
```

The final line is added to the output because the pattern is matched by the substring `me` when. The pattern `m[a-z]*n` is matched six times in the line

```
It seems to me most strange that men should fear;
```

However, GREP does not need to find all matches; finding one is sufficient for a line to be selected for output.

The extended regular expression notation can be used to describe more complicated patterns of interest that may occur in text. Consider the task of finding lines in a text file that contain a person's name. To determine the form of names, we initially consider the potential strings that occur as parts of a name:

- i) First name or initial: `[A-Z] [a-z]+|[A-Z] [.]`
- ii) Middle name, initial, or neither: `([A-Z] [a-z]+|[A-Z] [.])?`
- iii) Family name: `[A-Z] [a-z]+`

A string that can occur in the first position is either a name or an initial. In the former case, the string begins with a capital letter followed by a string of lowercase letters. An initial is simply a capital letter followed by a period. The same expressions can be used for middle names and family names. The `?` indicates that no middle name or initial is required. These expressions are concatenated with blanks

```
(([A-Z] [a-z]+|[A-Z] [.] ) [ ]) ((([A-Z] [a-z]+|[A-Z] [.] ) [ ])?)?(([A-Z] [a-z]+|)
```

to produce a general pattern for matching names.

The preceding expression will match E. B. White, Edgar Allen Poe, and Alan Turing. Since pattern matching is restricted to the form of the strings and not any underlying meaning (that is, pattern matching checks syntax and not semantics), the expression will also match Buckingham Palace and U. S. Mail. Moreover, the pattern will not match Vincent van Gogh, Dr. Watson, or Aristotle. Additional conditions would need to be added to the expression to match these variations of names.

Unlike off-line analysis, search commands in web browsers or word processors interactively find occurrences of strings that match an input pattern. A substring matching a pattern may span several lines. The pattern m^*n in the Microsoft Word “Find” command searches for substrings beginning with m and ending with n ; any string may separate the m and n . The search finds and highlights the first substring beginning at or after the current location of the cursor that matches the pattern. Repeating the search by clicking “next” highlights successive matches of the pattern. The substrings identified as matches of m^*n in the file caesar follow, with the matching substrings highlighted.

Cowards die *many* times before their deaths;

Cowards die many *times before their deaths*;

The valiant never taste of death but once.

It seems to me *most strange* that men should fear;

It seems to me *most strange* that men should fear;

It seems to me *most strange* that men should fear;

It seems to me most strange that *men* should fear;

Will come when it will come.

Notice that not all matching substrings are highlighted. The pattern m^*n is matched by any substring that begins with an occurrence of m and extends to any subsequent occurrence of n . The search only highlights the first matching substring for every m in the file.

In Chapter 6 we will see that a regular expression can be converted into a finite-state machine. The computation of the resulting machine will find the strings or substrings that match the pattern described by the expression. The restrictions on the operations used in regular expressions—intersection and set difference are not allowed—facilitate the automatic conversion from the description of a pattern to the implementation of a search algorithm.

Exercises

1. Give a recursive definition of the length of a string over Σ . Use the primitive operation from the definition of string.

2. Using induction on i , prove that $(w^R)^i = (w^i)^R$ for any string w and all $i \geq 0$.
3. Prove, using induction on the length of the string, that $(w^R)^R = w$ for all strings $w \in \Sigma^*$.
4. Let $X = \{aa, bb\}$ and $Y = \{\lambda, b, ab\}$.
 - a) List the strings in the set XY .
 - b) How many strings of length 6 are there in X^* ?
 - c) List the strings in the set Y^* of length three or less.
 - d) List the strings in the set X^*Y^* of length four or less.
5. Let L be the set of strings over $\{a, b\}$ generated by the recursive definition
 - i) Basis: $b \in L$.
 - ii) Recursive step: if u is in L then $ub \in L$, $uab \in L$, and $uba \in L$, and $bua \in L$.
 - iii) Closure: a string v is in L only if it can be obtained from the basis by a finite number of iterations of the recursive step.
 - a) List the elements in the sets L_0 , L_1 , and L_2 .
 - b) Is the string $bbaaba$ in L ? If so, trace how it is produced. If not, explain why not.
 - c) Is the string $bbaaaabb$ in L ? If so, trace how it is produced. If not, explain why not.
6. Give a recursive definition of the set of strings over $\{a, b\}$ that contain at least one b and have an even number of a 's before the first b . For example, bab , aab , and $aaaabababab$ are in the set, while aa , abb are not.
7. Give a recursive definition of the set $\{a^i b^j \mid 0 \leq i \leq j \leq 2i\}$.
8. Give a recursive definition of the set of strings over $\{a, b\}$ that contain twice as many a 's as b 's.
9. Prove that every string in the language defined in Example 2.2.1 has even length. The proof is by induction on the recursive generation of the strings.
10. Prove that every string in the language defined in Example 2.2.2 has at least as many a 's as b 's. Let $n_a(u)$ denote the number of a 's in the string u and $n_b(u)$ denote the number of b 's in u . The inductive proof should establish the inequality $n_a(u) \geq n_b(u)$.
11. Let L be the language over $\{a, b\}$ generated by the recursive definition
 - i) Basis: $\lambda \in L$.
 - ii) Recursive step: If $u \in L$ then $aaub \in L$.
 - iii) Closure: A string w is in L only if it can be obtained from the basis by a finite number of applications of the recursive step.
 - a) Give the sets L_0 , L_1 , and L_2 generated by the recursive definition.
 - b) Give an implicit definition of the set of strings defined by the recursive definition.
 - c) Prove by mathematical induction that for every string u in L , the number of a 's in u is twice the number b 's in u . Let $n_a(u)$ and $n_b(u)$ denote the number of a 's and the number of b 's in u , respectively.

- * 12. A **palindrome** over an alphabet Σ is a string in Σ^* that is spelled the same forward and backward. The set of palindromes over Σ can be defined recursively as follows:
- Basis: λ and a , for all $a \in \Sigma$, are palindromes.
 - Recursive step: If w is a palindrome and $a \in \Sigma$, then awa is a palindrome.
 - Closure: w is a palindrome only if it can be obtained from the basis elements by a finite number of applications of the recursive step.

The set of palindromes can also be defined by $\{w \mid w = w^R\}$. Prove that these two definitions generate the same set.

13. Let $L_1 = \{aaa\}^*$, $L_2 = \{a, b\}\{a, b\}\{a, b\}\{a, b\}$, and $L_3 = L_2^*$. Describe the strings that are in the languages L_2 , L_3 , and $L_1 \cap L_3$.

For Exercises 14 through 38, give a regular expression that represents the described set.

- The set of strings over $\{a, b, c\}$ in which all the a 's precede the b 's, which in turn precede the c 's. It is possible that there are no a 's, b 's, or c 's.
- The same set as Exercise 14 without the null string.
- The set of strings over $\{a, b, c\}$ with length three.
- The set of strings over $\{a, b, c\}$ with length less than three.
- The set of strings over $\{a, b, c\}$ with length greater than three.
- The set of strings over $\{a, b\}$ that contain the substring ab and have length greater than two.
- The set of strings of length two or more over $\{a, b\}$ in which all the a 's precede the b 's.
- The set of strings over $\{a, b\}$ that contain the substring aa and the substring bb .
- The set of strings over $\{a, b\}$ in which the substring aa occurs at least twice. Hint: Beware of the substring aaa .
- The set of strings over $\{a, b, c\}$ that begin with a , contain exactly two b 's, and end with cc .
- The set of strings over $\{a, b\}$ that contain the substring ab and the substring ba .
- The set of strings over $\{a, b, c\}$ in which every b is immediately followed by at least one c .
- The set of strings over $\{a, b\}$ in which the number of a 's is divisible by three.
- The set of strings over $\{a, b, c\}$ in which the total number of b 's and c 's is three.
- The set of strings over $\{a, b\}$ in which every a is either immediately preceded or immediately followed by b , for example, $baab$, aba , and b .
- The set of strings over $\{a, b, c\}$ that do not contain the substring aa .
- The set of strings over $\{a, b\}$ that do not begin with the substring aaa .
- The set of strings over $\{a, b\}$ that do not contain the substring aaa .
- The set of strings over $\{a, b\}$ that do not contain the substring aba .

33. The set of strings over $\{a, b\}$ in which the substring aa occurs exactly once.
34. The set of strings of odd length over $\{a, b\}$ that contain the substring bb .
35. The set of strings of even length over $\{a, b, c\}$ that contain exactly one a .
36. The set of strings of odd length over $\{a, b\}$ that contain exactly two b 's.
37. The set of strings over $\{a, b\}$ with an even number of a 's or an odd number of b 's.
- * 38. The set of strings over $\{a, b\}$ with an even number of a 's and an even number of b 's.
This is tricky; a strategy for constructing this expression is presented in Chapter 6.
39. Use the regular expression identities in Table 2.1 to establish the following identities:
- $(ba)^+(a^*b^* \cup a^*) = (ba)^*ba^+(b^* \cup \lambda)$
 - $b^+(a^*b^* \cup \lambda)b = b(b^*a^* \cup \lambda)b^+$
 - $(a \cup b)^* = (a \cup b)^*b^*$
 - $(a \cup b)^* = (a^* \cup ba^*)^*$
 - $(a \cup b)^* = (b^*(a \cup \lambda)b^*)^*$
40. Write the output that would be printed by a search of the file caesar described in Section 2.4 with the following extended regular expressions.
- [Cc]
 - [K-Z]
 - \<[a-z]{6}\>
 - \<[a-z]{6}\>|\<[a-z]{7}\>
41. Design an extended regular expression to search for addresses. For this exercise, an address will consist of
- a number,
 - a street name, and
 - a street type identifier or abbreviation.

Your pattern should match addresses of the form 1428 Elm Street, 51095 Tobacco Rd., and 1600 Pennsylvania Avenue. Do not be concerned if your regular expression does not identify all possible addresses.

Bibliographic Notes

Regular expressions were developed by Kleene [1956] for studying the properties of neural networks. McNaughton and Yamada [1960] proved that the regular sets are closed under the operations of intersection and complementation. An axiomatization of the algebra of regular expressions can be found in Salomaa [1966].

PART II

Grammars, Automata, and Languages

The syntax of a language specifies the permissible forms of the strings in the language. In Chapter 2, set-theoretic operations and recursive definitions were used to generate the strings of a language. These string-building tools, although primitive, were adequate for enforcing simple constraints on the order and the number of elements in a string. We now introduce a rule-based approach for defining and generating the strings of a language. This approach to language definition has its origin in both linguistics and computer science: linguistics in the attempt to formally describe natural language and computer science in the need to have precise and unambiguous definitions of high-level programming languages. Using terminology from the linguistic study, the string generation systems are called grammars.

In Chapter 3 we introduce two families of grammars, regular and context-free grammars. A family of grammars is defined by the form of the rules and the conditions under which they are applicable. A rule specifies a string transformation, and the strings of a language are generated by a sequence of rule applications. The flexibility provided by rules has proved to be well suited for defining the syntax of programming languages. The grammar that describes the programming language Java is used to demonstrate the context-free definition of several common programming language constructs.

After defining languages by the generation of strings, we turn our attention to the mechanical verification of whether a string satisfies a desired condition or matches a desired pattern. The family of deterministic finite automata is the first in a series of increasingly powerful abstract machines that we will use for pattern matching and language definition. We refer to the machines as abstract because we are not concerned with constructing hardware or software implementations of them. Instead, we are interested in determining the computational capability of the machines. The input to an abstract machine is a string, and the result of a computation indicates the acceptability of the input string. The language of a machine is the set of strings accepted by the computations of the machine.

A deterministic finite automaton is a read-once machine in which the instruction to be executed is determined by the state of the machine and the input symbol being processed. Finite automata have many applications including the lexical analysis of computer programs, digital circuit design, text searching, and pattern recognition. Kleene's theorem shows that the languages accepted by finite automata are precisely those that can be described by regular expressions and generated by regular grammars. A more powerful class of read-once machines, pushdown automata, is created by augmenting a finite automaton with a stack memory. The addition of the external memory permits pushdown automata to accept the context-free languages.

The correspondence between the generation of language by grammars and their acceptance by machines is a central theme of this book. The relationship between machines and grammars will continue with the families of unrestricted grammars and Turing machines introduced in Part III. The regular, context-free, and unrestricted grammars are members of the Chomsky hierarchy of grammars that will be examined in Chapter 10.

In this chapter, we will learn how to generate sentences in English. We will begin by defining what a sentence is and then we will look at some simple examples. Then we will move on to more complex sentences and see how they are generated. Finally, we will discuss some of the difficulties involved in generating sentences in English.

The sentence "The cat sat on the mat" is a simple example of a sentence. It consists of a subject ("The cat") and a predicate ("sat on the mat"). The subject is a noun phrase, and the predicate is a verb phrase. The subject is the main idea of the sentence, and the predicate tells us what the subject is doing.

Since the sentence "The cat sat on the mat" is a simple sentence, it has a single syntactic structure. This structure is called a terminal structure, because it consists of terminal symbols (the words "The", "cat", "sat", "on", "the", and "mat").

1. $\langle \text{sentence} \rangle$

2. $\langle \text{sentence} \rangle$

An informal introduction to the concept of a sentence follows by a very brief discussion of the components of a sentence. The components of a sentence are the words that make up the sentence. These words are called terminal symbols. There are two types of terminal symbols: nouns and verbs. Nouns are used to name things, and verbs are used to describe actions. A sentence can be formed by combining a noun and a verb. For example, the sentence "The cat sat on the mat" is formed by combining the noun "cat" and the verb "sat".

n to be
sed. Fi-
grams,
ws that
by reg-
d-once
a stack
ept the

: accep-
nes and
achines
mbers of

CHAPTER 3

Context-Free Grammars

In this chapter we present a rule-based approach for generating the strings of a language. Borrowing the terminology of natural languages, we call a syntactically correct string a **sentence** of the language. A small subset of the English language is used to illustrate the components of the string-generation process. The alphabet of our miniature language is the set $\{a, \text{the}, \text{John}, \text{Jill}, \text{hamburger}; \text{car}, \text{drives}, \text{eats}, \text{slowly}, \text{frequently}, \text{big}, \text{juicy}, \text{brown}\}$. The elements of the alphabet are called the **terminal symbols** of the language. Capitalization, punctuation, and other important features of written languages are ignored in this example.

The sentence-generation procedure should construct the strings *John eats a hamburger* and *Jill drives frequently*. Strings of the form *Jill* and *car John slowly* should not result from this process. Additional symbols are used during the construction of sentences to enforce the syntactic restrictions of the language. These intermediate symbols, known as **variables** or **nonterminals**, are represented by enclosing them in angle brackets $\langle \rangle$.

Since the generation procedure constructs sentences, the initial variable is named $\langle \text{sentence} \rangle$. The generation of a sentence consists of replacing variables by strings of a specific form. Syntactically correct replacements are given by a set of transformation rules. Two possible rules for the variable $\langle \text{sentence} \rangle$ are

1. $\langle \text{sentence} \rangle \rightarrow \langle \text{noun-phrase} \rangle \langle \text{verb-phrase} \rangle$
2. $\langle \text{sentence} \rangle \rightarrow \langle \text{noun-phrase} \rangle \langle \text{verb} \rangle \langle \text{direct-object-phrase} \rangle$

An informal interpretation of rule 1 is that a sentence may be formed by a noun phrase followed by a verb phrase. At this point, of course, neither of the variables $\langle \text{noun-phrase} \rangle$ nor $\langle \text{verb-phrase} \rangle$ has been defined. The second rule gives an alternative definition of sentence, a noun phrase followed by a verb followed by a direct object phrase. The existence of multiple transformations indicates that syntactically correct sentences may have several different forms.

A noun phrase may contain either a proper or a common noun. A common noun is preceded by a determiner, while a proper noun stands alone. This feature of the syntax of the English language is represented by rules 3 and 4.

Rules for the variables that generate noun and verb phrases are given below. Rather than rewriting the left-hand side of alternative rules for the same variable, we list the right-hand sides of the rules sequentially. Numbering the rules is not a feature of the generation process, merely a notational convenience.

3. $\langle \text{noun-phrase} \rangle \rightarrow \langle \text{proper-noun} \rangle$
4. $\rightarrow \langle \text{determiner} \rangle \langle \text{common-noun} \rangle$
5. $\langle \text{proper-noun} \rangle \rightarrow \text{John}$
6. $\rightarrow \text{Jill}$
7. $\langle \text{common-noun} \rangle \rightarrow \text{car}$
8. $\rightarrow \text{hamburger}$
9. $\langle \text{determiner} \rangle \rightarrow \text{a}$
10. $\rightarrow \text{the}$
11. $\langle \text{verb-phrase} \rangle \rightarrow \langle \text{verb} \rangle \langle \text{adverb} \rangle$
12. $\rightarrow \langle \text{verb} \rangle$
13. $\langle \text{verb} \rangle \rightarrow \text{drives}$
14. $\rightarrow \text{eats}$
15. $\langle \text{adverb} \rangle \rightarrow \text{slowly}$
16. $\rightarrow \text{frequently}$

With the exception of $\langle \text{direct-object-phrase} \rangle$, rules have been defined for each of the variables that have been introduced.

The application of a rule transforms one string to another. The transformation consists of replacing an occurrence of the variable on the left-hand side of the \rightarrow with the string on the right-hand side. The generation of a sentence consists of repeated rule applications to transform the variable $\langle \text{sentence} \rangle$ into a string of terminal symbols.

For example, the sentence *Jill drives frequently* is generated by the following transformations:

Derivation	Rule Applied
$\langle \text{sentence} \rangle \Rightarrow \langle \text{noun-phrase} \rangle \langle \text{verb-phrase} \rangle$	1
$\Rightarrow \langle \text{proper-noun} \rangle \langle \text{verb-phrase} \rangle$	3
$\Rightarrow \text{Jill} \langle \text{verb-phrase} \rangle$	6
$\Rightarrow \text{Jill} \langle \text{verb} \rangle \langle \text{adverb} \rangle$	11
$\Rightarrow \text{Jill} \text{ drives} \langle \text{adverb} \rangle$	13
$\Rightarrow \text{Jill} \text{ drives} \text{ frequently}$	16

The sy
right g
derivati
The res
The set
by the r

To
given. B
to gener
includin
generati

As can be
capable of
elements o
recursion i

17. $\langle \text{adjec}$
- 18.
19. $\langle \text{adjec}$
- 20.
- 21.

The definit
(adjective-l
The λ on th
(adjective-l
of adjectives

22. $\langle \text{direct-objec}$
- 23.

non noun is
the syntax of
below. Rather
is the right-
e generation

The symbol \Rightarrow , used to designate a rule application, is read “derives.” The column on the right gives the number of the rule that was applied to achieve the transformation. The derivation terminates when all variables have been removed from the derived string. The resulting string, consisting solely of terminal symbols, is a sentence of the language. The set of terminal strings derivable from the variable $\langle \text{sentence} \rangle$ is the language generated by the rules of our example.

To complete the set of rules, the transformations for $\langle \text{direct-object-phrase} \rangle$ must be given. Before designing rules, we must decide upon the form of the strings that we wish to generate. In our language we will allow the possibility of any number of adjectives, including repetitions, to precede the direct object. This requires a set of rules capable of generating each of the following strings:

John eats a hamburger
John eats a big hamburger
John eats a big juicy hamburger
John eats a big brown juicy hamburger
John eats a big big brown juicy hamburger

As can be seen by the potential repetition of the adjectives, the rules of the grammar must be capable of generating strings of arbitrary length. The use of a recursive definition allows the elements of an infinite set to be generated by a finite specification. Following that example, recursion is introduced into the string-generation process, that is, into the rules.

for each of the
generation consists
in the string on
applications to
the following transfor-

17. $\langle \text{adjective-list} \rangle \rightarrow \langle \text{adjective} \rangle \langle \text{adjective-list} \rangle$
18. $\rightarrow \lambda$
19. $\langle \text{adjective} \rangle \rightarrow \text{big}$
20. $\rightarrow \text{juicy}$
21. $\rightarrow \text{brown}$

The definition of $\langle \text{adjective-list} \rangle$ follows the standard recursive pattern. Rule 17 defines $\langle \text{adjective-list} \rangle$ in terms of itself, while rule 18 provides the basis of the recursive definition. The λ on the right-hand side of rule 18 indicates that the application of this rule replaces $\langle \text{adjective-list} \rangle$ with the null string. Repeated applications of rule 17 generate a sequence of adjectives. Rules for $\langle \text{direct-object-phrase} \rangle$ are constructed using $\langle \text{adjective-list} \rangle$:

22. $\langle \text{direct-object-phrase} \rangle \rightarrow \langle \text{adjective-list} \rangle \langle \text{proper-noun} \rangle$
23. $\rightarrow \langle \text{determiner} \rangle \langle \text{adjective-list} \rangle \langle \text{common-noun} \rangle$

The sentence *John eats a big juicy hamburger* can be derived by the following sequence of rule applications:

Derivation	Rule Applied
$\langle \text{sentence} \rangle \Rightarrow \langle \text{noun-phrase} \rangle \langle \text{verb} \rangle \langle \text{direct-object-phrase} \rangle$	2
$\Rightarrow \langle \text{proper-noun} \rangle \langle \text{verb} \rangle \langle \text{direct-object-phrase} \rangle$	3
$\Rightarrow \text{John} \langle \text{verb} \rangle \langle \text{direct-object-phrase} \rangle$	5
$\Rightarrow \text{John eats} \langle \text{direct-object-phrase} \rangle$	14
$\Rightarrow \text{John eats} \langle \text{determiner} \rangle \langle \text{adjective-list} \rangle \langle \text{common-noun} \rangle$	23
$\Rightarrow \text{John eats a} \langle \text{adjective-list} \rangle \langle \text{common-noun} \rangle$	9
$\Rightarrow \text{John eats a} \langle \text{adjective} \rangle \langle \text{adjective-list} \rangle \langle \text{common-noun} \rangle$	17
$\Rightarrow \text{John eats a big} \langle \text{adjective-list} \rangle \langle \text{common-noun} \rangle$	19
$\Rightarrow \text{John eats a big} \langle \text{adjective} \rangle \langle \text{adjective-list} \rangle \langle \text{common-noun} \rangle$	17
$\Rightarrow \text{John eats a big juicy} \langle \text{adjective-list} \rangle \langle \text{common-noun} \rangle$	20
$\Rightarrow \text{John eats a big juicy} \langle \text{common-noun} \rangle$	18
$\Rightarrow \text{John eats a big juicy hamburger}$	8

The generation of sentences is strictly a function of the rules. The string *the car eats slowly* is a sentence in the language since it has the form $\langle \text{noun-phrase} \rangle \langle \text{verb-phrase} \rangle$ outlined by rule 1. This illustrates the important distinction between syntax and semantics; the generation of sentences is concerned with the form of the derived string without regard to any underlying meaning that may be associated with the terminal symbols.

By rules 3 and 4, a noun phrase consists of a proper noun or a common noun preceded by a determiner. The variable $\langle \text{adjective-list} \rangle$ may be incorporated into the $\langle \text{noun-phrase} \rangle$ rules, permitting adjectives to modify a noun:

- 3'. $\langle \text{noun-phrase} \rangle \rightarrow \langle \text{adjective-list} \rangle \langle \text{proper-noun} \rangle$
- 4'. $\rightarrow \langle \text{determiner} \rangle \langle \text{adjective-list} \rangle \langle \text{common-noun} \rangle$

With this modification, the string *big John eats frequently* can be derived from the variable $\langle \text{sentence} \rangle$.

3.1 Context-Free Grammars and Languages

We will now define a formal system, the context-free grammar, that is used to generate the strings of a language. The natural language example was presented to motivate the components and features of string generation using a context-free grammar.

Definition 3.1.1

A **context-free grammar** is a quadruple (V, Σ, P, S) where V is a finite set of variables, Σ (the alphabet) is a finite set of terminal symbols, P is a finite set of rules, and S is a

ing sequence of

Rule Applied

2
3
5
14
23
9
17
19
17
20
18
8

string the car eats
e) *(verb-phrase)*
ix and semantics;
ng without regard
bols.
on noun preceded
he *(noun-phrase)*

from the variable

s used to generate
ed to motivate the
mar.

ite set of variables,
of rules, and S is a

distinguished element of V called the start symbol. The sets V and Σ are assumed to be disjoint.

A rule is written $A \rightarrow w$ where $A \in V$ and $w \in (V \cup \Sigma)^*$. A rule of this form is called an A rule, referring to the variable on the left-hand side. Since the null string is in $(V \cup \Sigma)^*$, λ may occur on the right-hand side of a rule. A rule of the form $A \rightarrow \lambda$ is called a **null** or **λ -rule**.

Italics are used to denote the variables and terminals of a context-free grammar. Terminals are represented by lowercase letters occurring at the beginning of the alphabet, that is, a, b, c, \dots . Following the conventions introduced for strings, the letters p, q, u, v, w, x, y, z , with or without subscripts, represent arbitrary members of $(V \cup \Sigma)^*$. Variables will be denoted by capital letters. As in the natural language example, variables are referred to as the *nonterminal symbols* of the grammar.

Grammars are used to generate properly formed strings over the prescribed alphabet. The fundamental step in the generation process consists of transforming a string by the application of a rule. The application of $A \rightarrow w$ to the variable A in uAv produces the string uvw . This is denoted $uAv \Rightarrow uvw$. The prefix u and suffix v define the *context* in which the variable A occurs. The grammars introduced in this chapter are called context-free because of the general applicability of the rules. An A rule can be applied to the variable A whenever and wherever it occurs; the context places no limitations on the applicability of a rule.

A string w is derivable from v if there is a finite sequence of rule applications that transforms v to w ; that is, if a sequence of transformations

$$v \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_n = w$$

can be constructed from the rules of the grammar. The derivability of w from v is denoted $v \xrightarrow{*} w$. The set of strings derivable from v , being constructed by a finite but unbounded number of rule applications, can be defined recursively.

Definition 3.1.2

Let $G = (V, \Sigma, P, S)$ be a context-free grammar and $v \in (V \cup \Sigma)^*$. The set of strings derivable from v is defined recursively as follows:

- Basis: v is derivable from v .
- Recursive step: If $u = xAy$ is derivable from v and $A \rightarrow w \in P$, then xwy is derivable from v .
- Closure: A string is derivable from v only if it can be generated from v by a finite number of applications of the recursive step.

Note that the definition of a rule uses the \rightarrow notation, while its application uses \Rightarrow . The symbol $\xrightarrow{*}$ denotes derivability and \xrightarrow{n} designates derivability utilizing one or more rule applications. The length of a derivation is the number of rule applications employed. A derivation of w from v of length n is denoted $v \xrightarrow{n} w$. When more than one grammar is

being considered, the notation $v \xrightarrow[G]{*} w$ will be used to explicitly indicate that the derivation utilizes rules of the grammar G.

A language has been defined as a set of strings over an alphabet. A grammar consists of an alphabet and a method of generating strings. These strings may contain both variables and terminals. The start symbol of the grammar, assuming the role of *< sentence >* in the natural language example, initiates the process of generating acceptable strings. The language of the grammar G is the set of terminal strings derivable from the start symbol. We now state this as a definition.

Definition 3.1.3

Let $G = (V, \Sigma, P, S)$ be a context-free grammar.

- i) A string $w \in (V \cup \Sigma)^*$ is a **sentential form** of G if there is a derivation $S \xrightarrow{*} w$ in G.
- ii) A string $w \in \Sigma^*$ is a **sentence** of G if there is a derivation $S \xrightarrow{*} w$ in G.
- iii) The **language** of G, denoted $L(G)$, is the set $\{w \in \Sigma^* \mid S \xrightarrow{*} w\}$.

A sentential form is a string that is derivable from the start symbol of the grammar. Referring back to the natural language example, the derivation

$$\begin{aligned} \langle \text{sentence} \rangle &\Rightarrow \langle \text{noun-phrase} \rangle \langle \text{verb-phrase} \rangle \\ &\Rightarrow \langle \text{proper-noun} \rangle \langle \text{verb-phrase} \rangle \\ &\Rightarrow \text{Jill} \langle \text{verb-phrase} \rangle \end{aligned}$$

shows that *Jill* *(verb-phrase)* is a sentential form of that grammar. It is not yet a sentence, it still contains variables, but it has the form of a sentence. A sentence is a sentential form that contains only terminal symbols. The language of a grammar consists of the sentences generated by the grammar. A set of strings over an alphabet Σ is said to be a **context-free language** if it is generated by a context-free grammar.

The use of recursion is necessary for a finite set of rules to generate strings of arbitrary length and languages with infinitely many strings. Recursion is introduced into grammars through the rules. A rule of the form $A \rightarrow uAv$ is called **recursive** since it defines the variable A in terms of itself. Rules of the form $A \rightarrow Av$ and $A \rightarrow uA$ are called *left-recursive* and *right-recursive*, respectively, indicating the location of recursion in the rule.

Because of the importance of recursive rules, we examine the form of strings produced by repeated applications of the recursive rules $A \rightarrow aAb$, $A \rightarrow aA$, $A \rightarrow Ab$, and $A \rightarrow AA$:

$$\begin{array}{llll} A \Rightarrow aAb & A \Rightarrow aA & A \Rightarrow Ab & A \Rightarrow AA \\ \Rightarrow aAb & \Rightarrow aA & \Rightarrow Ab & \Rightarrow AAA \\ \Rightarrow aaAbb & \Rightarrow aaA & \Rightarrow Abb & \Rightarrow AAAA \\ \Rightarrow aaaAbbb & \Rightarrow aaaA & \Rightarrow Abbb & \Rightarrow AAAAA \\ \vdots & \vdots & \vdots & \vdots \end{array}$$

A derivation employing the rule $A \rightarrow aAb$ generates any number of a's followed by the same number of b's. Rules of this form are necessary for producing strings that contain symbols in

matched pairs
any number
number of
right-recursi
be termina

A vari
form $A \Rightarrow$
to indirect

A gran
number of
to abbrevia
of the termin
the transfor
and (b) in th
string. Deriv
the rightmos
be more tha

Figure 3
feature of a
which each v
depicted by a
variable but c
tree can be o

Definition 3.1.4

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. The process of generating strings by repeated applications of the rules of G is called a **derivation tree**.

- i) Initialize

derivation

consists of variables and the natural language of the now state

$\xrightarrow{*} w$ in G.

grammar.

t a sentence, potential form
he sentences context-free

s of arbitrary
to grammars
s the variable
recursive and

ngs produced
and $A \rightarrow AA$:

ed by the same
ain symbols in

$G = (V, \Sigma, P, S)$			
$V = \{S, A\}$			
$\Sigma = \{a, b\}$			
P: $S \rightarrow AA$			
$A \rightarrow AAA \mid bA \mid Ab \mid a$			
$S \Rightarrow AA$	$S \Rightarrow AA$	$S \Rightarrow AA$	$S \Rightarrow AA$
$\Rightarrow aA$	$\Rightarrow AAA$	$\Rightarrow Aa$	$\Rightarrow aA$
$\Rightarrow aAAA$	$\Rightarrow aAAA$	$\Rightarrow AAAa$	$\Rightarrow aAAA$
$\Rightarrow abAAA$	$\Rightarrow abAAA$	$\Rightarrow AAbAa$	$\Rightarrow aAAa$
$\Rightarrow abaAA$	$\Rightarrow abaAA$	$\Rightarrow AAbaa$	$\Rightarrow abAAa$
$\Rightarrow ababAA$	$\Rightarrow ababAA$	$\Rightarrow AbAbaa$	$\Rightarrow abAbAa$
$\Rightarrow ababaA$	$\Rightarrow ababaA$	$\Rightarrow Ababaa$	$\Rightarrow ababAa$
$\Rightarrow ababaa$	$\Rightarrow ababaa$	$\Rightarrow ababaa$	$\Rightarrow ababaa$
(a)	(b)	(c)	(d)

FIGURE 3.1 Sample derivations of $ababaa$ in G.

matched pairs, such as left and right parentheses. The right recursive rule $A \rightarrow aA$ generates any number of a 's preceding the variable A , and the left recursive $A \rightarrow Ab$ generates any number of b 's following A . Each application of the rule $A \rightarrow AA$, which is both left- and right-recursive, produces an additional A . The repetitive application of a recursive rule can be terminated at any time by the application of a different A rule.

A variable A is called *recursive* if there is a derivation $A \xrightarrow{*} uAv$. A derivation of the form $A \xrightarrow{*} w \xrightarrow{*} uAv$, where A is not in w , is said to be *indirectly recursive*. Note that, due to indirect recursion, a variable A may be recursive even if there are no recursive A rules.

A grammar G that generates the language consisting of strings with a positive, even number of a 's is given in Figure 3.1. The rules are written using the shorthand $A \rightarrow u \mid v$ to abbreviate $A \rightarrow u$ and $A \rightarrow v$. The vertical bar $|$ is read "or." Four distinct derivations of the terminal string $ababaa$ are shown in Figure 3.1. The definition of derivation permits the transformation of any variable in the string. Each rule application in derivations (a) and (b) in the figure transforms the first variable occurring in a left-to-right reading of the string. Derivations with this property are called *leftmost*. Derivation (c) is *rightmost*, since the rightmost variable has a rule applied to it. These derivations demonstrate that there may be more than one derivation of a string in a context-free grammar.

Figure 3.1 exhibits the flexibility of derivations in a context-free grammar. The essential feature of a derivation is not the order in which the rules are applied, but the manner in which each variable is transformed into a terminal string. The transformation is graphically depicted by a derivation or parse tree. The tree structure indicates the rule applied to each variable but does not designate the order of the rule applications. The leaves of the derivation tree can be ordered to yield the result of a derivation represented by the tree.

Definition 3.1.4

Let $G = (V, \Sigma, P, S)$ be a context-free grammar and let $S \xrightarrow{*} w$ be a derivation in G . The *derivation tree*, DT, of $S \xrightarrow{*} w$ is an ordered tree that can be built iteratively as follows:

- i) Initialize DT with root S .

- ii) If $A \rightarrow x_1x_2 \dots x_n$ with $x_i \in (V \cup \Sigma)$ is the rule in the derivation applied to the string uAv , then add x_1, x_2, \dots, x_n as the children of A in the tree.
- iii) If $A \rightarrow \lambda$ is the rule in the derivation applied to the string uAv , then add λ as the only child of A in the tree.

The ordering of the leaves also follows this iterative process. Initially, the only leaf is S and the ordering is obvious. When the rule $A \rightarrow x_1x_2 \dots x_n$ is used to generate the children of A , each x_i becomes a leaf and A is replaced in the ordering of the leaves by the sequence x_1, x_2, \dots, x_n . The application of a rule $A \rightarrow \lambda$ simply replaces A by the null string. Figure 3.2 traces the construction of the tree corresponding to derivation (a) of Figure 3.1. The ordering of the leaves is given along with each of the trees.

The order of the leaves in a derivation tree is independent of the derivation from which the tree was generated. The ordering provided by the iterative process is identical to the ordering of the leaves given by the relation LEFTOF in Section 1.8. The frontier of the derivation tree is the string generated by the derivation.

Figure 3.3 gives the derivation trees for each of the derivations in Figure 3.1. The trees generated by derivations (a) and (d) are identical, indicating that each variable is transformed into a terminal string in the same manner. The only difference between these derivations is the order of the rule applications.

A derivation tree can be used to produce several derivations that generate the same string. The rule applied to a variable A can be reconstructed from the children of A in the tree. The rightmost derivation

$$\begin{aligned}
 S &\Rightarrow AA \\
 &\Rightarrow AAAA \\
 &\Rightarrow AAAa \\
 &\Rightarrow AAbAa \\
 &\Rightarrow AAbaa \\
 &\Rightarrow AbAbaa \\
 &\Rightarrow Ababaa \\
 &\Rightarrow ababaa
 \end{aligned}$$

is obtained from the derivation tree (a) in Figure 3.3. Notice that this derivation is different from the rightmost derivation (c) in Figure 3.1. In the latter derivation, the second variable in the string AA is transformed using the rule $A \rightarrow a$, while $A \rightarrow AAA$ is used in the preceding derivation. The two trees graphically illustrate the distinct transformations.

As we have seen, the context-free applicability of rules allows a great deal of flexibility in the constructions of derivations. Lemma 3.1.5 shows that a derivation may be broken into subderivations from each variable in the string. Derivability was defined recursively, the length of derivations being finite but unbounded. Consequently, we may use mathematical induction to establish that a property holds for all derivations from a given string.

the string

s the only

only leaf
nerate the
leaves by
 $\Rightarrow A$ by the
tion (a) of

rom which
ical to the
tier of the

. The trees
ansformed
rivation is

e the same
of A in the

n is different
d variable in
he preceding

of flexibility
e broken into
ursively, the
nathematical
ing.

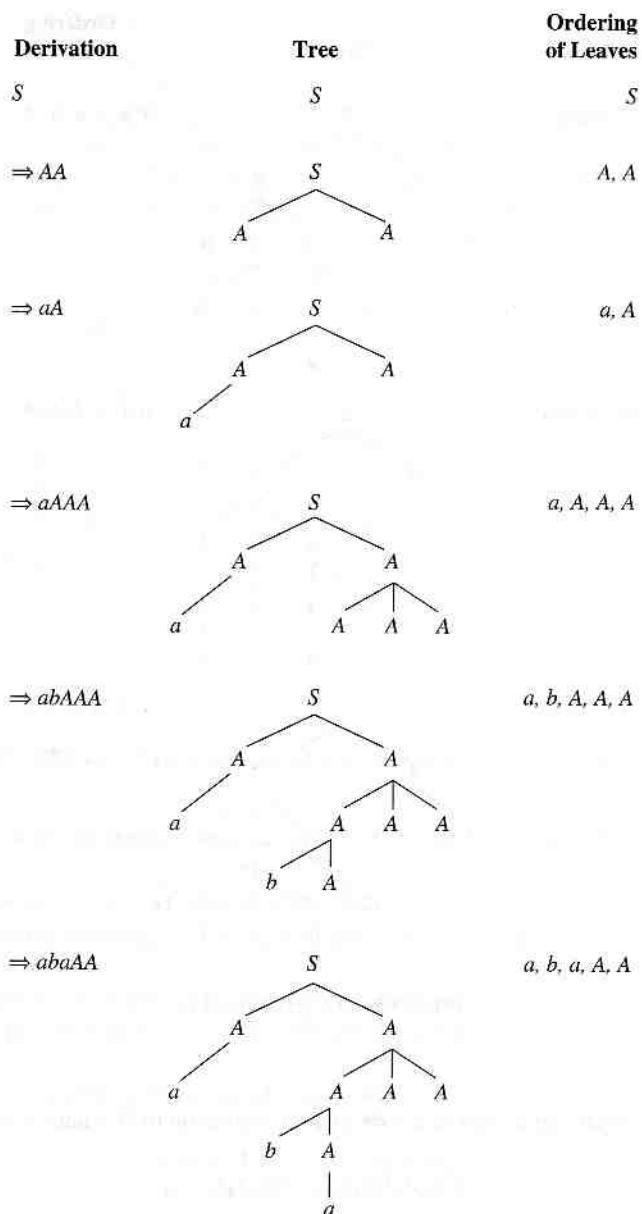


FIGURE 3.2 Construction of derivation tree. (continued on next page)

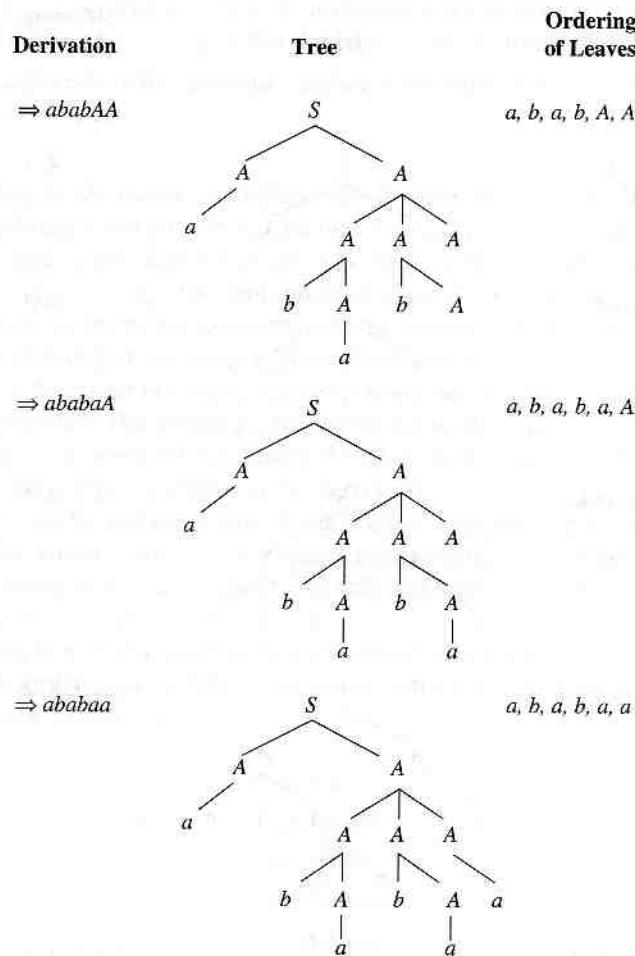


FIGURE 3.2 (continued)

Lemma 3.1.5

Let G be a context-free grammar and $v \xrightarrow{n} w$ be a derivation in G where v can be written

$$v = w_1 A_1 w_2 A_2 \dots w_k A_k w_{k+1},$$

with $w_i \in \Sigma^*$. Then there are strings $p_i \in (\Sigma \cup V)^*$ that satisfy

- i) $A_i \xrightarrow{t_i} p_i$
- ii) $w = w_1 p_1 w_2 p_2 \dots w_k p_k w_{k+1}$
- iii) $\sum_{i=1}^k t_i = n$.

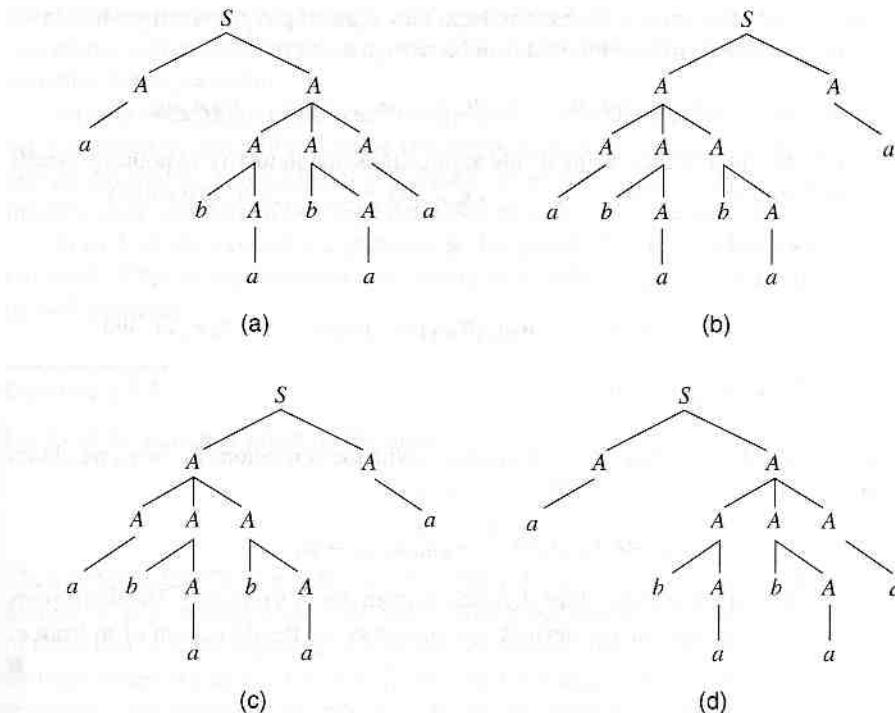


FIGURE 3.3 Trees corresponding to the derivations in Figure 3.1.

Proof. The proof is by induction on the length of the derivation of w from v .

Basis: The basis consists of derivations of the form $v \xrightarrow{0} w$. In this case, $w = v$ and each A_i is equal to the corresponding p_i . The desired derivations have the form $A_i \xrightarrow{0} p_i$.

Inductive Hypothesis: Assume that all derivations $v \xrightarrow{n} w$ can be decomposed into derivations from the A_i 's, the variables of v , which together form a derivation of w from v of length n .

Inductive Step: Let $v \xrightarrow{n+1} w$ be a derivation in G with

$$v = w_1 A_1 w_2 A_2 \dots w_k A_k w_{k+1},$$

where $w_i \in \Sigma^*$. The derivation can be written $v \Rightarrow u \xrightarrow{n} w$. This reduces the original derivation to the application of a single rule and derivation of length n , the latter of which is suitable for the invocation of the inductive hypothesis.

The first rule application in the derivation, $v \Rightarrow u$, transforms one of the variables in v , call it A_j , with a rule of the form

$$A_j \rightarrow u_1 B_1 u_2 B_2 \dots u_m B_m u_{m+1},$$

where each $u_i \in \Sigma^*$. The string u is obtained from v by replacing A_j by the right-hand side of the A_j rule. Making this substitution, u can be written as

$$w_1 A_1 \dots A_{j-1} w_j u_1 B_1 u_2 B_2 \dots u_m B_m u_{m+1} w_{j+1} A_{j+1} \dots w_k A_k w_{k+1}.$$

Since w is derivable from u using n rule applications, the inductive hypothesis asserts that there are strings p_1, \dots, p_{j-1} , q_1, \dots, q_m , and p_{j+1}, \dots, p_k that satisfy

- i) $A_i \xrightarrow{q_i} p_i$ for $i = 1, \dots, j-1, j+1, \dots, k$
 $B_i \xrightarrow{s_i} q_i$ for $i = 1, \dots, m$;
- ii) $w = w_1 p_1 w_2 \dots p_{j-1} w_j u_1 q_1 u_2 \dots u_m q_m u_{m+1} w_{j+1} p_{j+1} \dots w_k p_k w_{k+1}$; and
- iii) $\sum_{i=1}^{j-1} t_i + \sum_{i=j+1}^k t_i + \sum_{i=1}^m s_i = n$.

Combining the rule $A_j \rightarrow u_1 B_1 u_2 B_2 \dots u_m B_m u_{m+1}$ with the derivations $B_i \xrightarrow{*} q_i$, we obtain a derivation

$$A_j \xrightarrow{*} u_1 q_1 u_2 q_2 \dots u_m q_m u_{m+1} = p_j$$

whose length is the sum of lengths of the derivations from the B_i 's plus one. The derivations $A_i \xrightarrow{*} p_i$, $i = 1, \dots, k$, provide the desired decomposition of the derivation of w from v . ■

Lemma 3.1.5 demonstrates the flexibility and modularity of derivations in context-free grammars. Every complex derivation can be broken down into subderivations of the constituent variables. This modularity will be exploited in the design of complex languages by using variables to define smaller and more manageable subsets of the language. These independently defined sublanguages are then combined by additional rules to produce the syntax of the entire language.

3.2 Examples of Grammars and Languages

Context-free grammars have been introduced to generate languages. Formal languages, like computer languages and natural languages, have requirements that the strings must satisfy in order to be syntactically correct. Grammars for these languages must generate precisely the desired strings and no others. There are two natural approaches that we may take to help develop our understanding of the relationship between grammars and languages. One is to begin with an informal specification of a language and then construct a grammar that generates it. This is the approach followed in the design of programming languages—the syntax is selected and the language designer produces a set of rules that defines the correctly formed strings. Conversely, we may begin with the rules of a grammar and analyze them to determine the form of the strings of the language. This is the approach frequently taken when checking the syntax of the source code of a computer program. The syntax of the programming language is specified by a set of grammatical rules, such as the definition of

the program
statement
variables

Initial
With exp
that prod
intuitive

In ea
terminals
of each g

Example

Let G be t

Then L(G)
number of
of the rule
then gener
sentence o
one b.

Example 3.1

The relatio
 $a^n b^m c^m d^{2n}$
The same is

generate str
the A rules
recursion, er

Example 3.2

Recall that a
the set of pal
of palindrom
strings λ , a ,

t-hand side

esis asserts
fy

nd

η_i , we obtain

e derivations
of w from v .

s in context-
uations of the
ex languages
guage. These
o produce the

anguages, like
s must satisfy
erate precisely
ay take to help
uages. One is
grammar that
g languages—
at defines the
ar and analyze
ach frequently
m. The syntax
e definition of

the programming language Java given in Appendix IV. The syntax of constants, identifiers, statements, and entire programs is correct if the source code is derivable from the appropriate variables in the grammar.

Initially, determining the relationship between strings and rules may seem difficult. With experience, you will recognize frequently occurring patterns in strings and the rules that produce them. The goal of this section is to analyze examples to help you develop an intuitive understanding of language definition using context-free grammars.

In each of the examples a grammar is defined by listing its rules. The variables and terminals of the grammar are those occurring in the rules. The variable S is the start symbol of each grammar.

Example 3.2.1

Let G be the grammar given by the rules

$$\begin{aligned} S &\rightarrow aSa \mid aBa \\ B &\rightarrow bB \mid b. \end{aligned}$$

Then $L(G) = \{a^n b^m a^n \mid n > 0, m > 0\}$. The rule $S \rightarrow aSa$ recursively builds an equal number of a 's on each end of the string. The recursion is terminated by the application of the rule $S \rightarrow aBa$, ensuring at least one leading and one trailing a . The recursive B rule then generates any number of b 's. To remove the variable B from the string and obtain a sentence of the language, the rule $B \rightarrow b$ must be applied, forcing the presence of at least one b . \square

Example 3.2.2

The relationship between the number of leading a 's and trailing d 's in the language $\{a^n b^m c^m d^{2n} \mid n \geq 0, m > 0\}$ indicates that a recursive rule is needed to generate them. The same is true of the b 's and c 's. Derivations in the grammar

$$\begin{aligned} S &\rightarrow aSdd \mid A \\ A &\rightarrow bAc \mid bc \end{aligned}$$

generate strings in an outside-to-inside manner. The S rules produce the a 's and d 's while the A rules generate the b 's and c 's. The rule $A \rightarrow bc$, whose application terminates the recursion, ensures the presence of the substring bc in every string in the language. \square

Example 3.2.3

Recall that a string w is a palindrome if $w = w^R$. A grammar is constructed to generate the set of palindromes over $\{a, b\}$. The rules of the grammar mimic the recursive definition of palindromes given in Exercise 2.12. The basis of the set of palindromes consists of the strings λ , a , and b . The S rules

$$S \rightarrow a \mid b \mid \lambda$$

immediately generate these strings. The recursive part of the definition consists of adding the same symbol to each side of an existing palindrome. The rules

$$S \rightarrow aSa \mid bSb$$

capture the recursive generation process. \square

Example 3.2.4

The first recursive rule of

$$S \rightarrow aSb \mid aSbb \mid \lambda$$

generates a trailing b for every a , while the second generates two b 's for each a . Thus there is at least one b for every a and at most two. The language of the grammar is $\{a^n b^m \mid 0 \leq n \leq m \leq 2n\}$. \square

Example 3.2.5

Consider the grammar

$$S \rightarrow abScB \mid \lambda$$

$$B \rightarrow bB \mid b.$$

The recursive S rule generates an equal number of ab 's and cB 's. The B rules generate b^+ . In a derivation each occurrence of B may produce a different number of b 's. For example, in the derivation

$$\begin{aligned} S &\Rightarrow abScB \\ &\Rightarrow ababScBcB \\ &\Rightarrow ababcBcB \\ &\Rightarrow ababcbcB \\ &\Rightarrow ababcbcbB \\ &\Rightarrow ababcbcb, \end{aligned}$$

the first occurrence of B generates a single b and the second occurrence produces bb . The language of the grammar is the set $\{(ab)^n (cb^{m_n})^n \mid n \geq 0, m_n > 0\}$. The superscript m_n indicates that the number of b 's produced by each occurrence of B may be different since b^{m_i} need not equal b^{m_j} when $i \neq j$. \square

Example 3.2.6

Let G_1 and G_2 be the grammars

$$\begin{array}{ll} G_1: S \rightarrow AB & G_2: S \rightarrow aS \mid aA \\ A \rightarrow aA \mid a & A \rightarrow bA \mid \lambda. \\ B \rightarrow bB \mid \lambda & \end{array}$$

Both methods allow in a le

Example

The gr is, the

G_1 requ rules. T for the

Example

The gra

In G_1 , an rule. A d A $\rightarrow bC$ string of

Two 3.2.6, 3.2 by signific forms ma

Example

A grammar The strate so forth. T

sts of adding

Both of these grammars generate the language a^+b^* . The A rules in G_1 provide the standard method of generating a nonnull string of a 's. The use of the λ -rule to terminate the derivation allows the possibility of having no b 's. The rules in grammar G_2 build the strings of a^+b^* in a left-to-right manner.

Example 3.2.7

The grammars G_1 and G_2 generate the strings over $\{a, b\}$ that contain exactly two b 's. That is, the language of the grammars is $a^*ba^*ba^*$.

$$\begin{array}{ll} G_1: S \rightarrow AbAbA & G_2: S \rightarrow aS \mid bA \\ A \rightarrow aA \mid \lambda & A \rightarrow aA \mid bC \\ & C \rightarrow aC \mid \lambda \end{array}$$

G_1 requires only two variables since the three instances of a^* are generated by the same A rules. The second builds the strings in a left-to-right manner, requiring a distinct variable for the generation of each sequence of a 's. \square

Example 3.2.8

The grammars from Example 3.2.7 can be modified to generate strings with at least two b 's.

$$\begin{array}{ll} G_1: S \rightarrow AbAbA & G_2: S \rightarrow aS \mid bA \\ A \rightarrow aA \mid bA \mid \lambda & A \rightarrow aA \mid bC \\ & C \rightarrow aC \mid bC \mid \lambda \end{array}$$

In G_1 , any string can be generated before, between, and after the two b 's produced by the S rule. A derivation in G_2 produces the first b using the rule $S \rightarrow bA$ and the second b with $A \rightarrow bC$. The derivation finishes using applications of the C rules, which can generate any string of a 's and b 's. \square

Two grammars that generate the same language are said to be *equivalent*. Examples 3.2.6, 3.2.7, and 3.2.8 show that equivalent grammars may produce the strings of a language by significantly different derivations. In later chapters we will see that rules having particular forms may facilitate the mechanical determination of the syntactic correctness of strings.

Example 3.2.9

A grammar is given that generates the language consisting of even-length strings over $\{a, b\}$. The strategy can be generalized to construct strings of length divisible by three, by four, and so forth. The variables S and O serve as counters. An S occurs in a sentential form when an

even number of terminals has been generated. An O records the presence of an odd number of terminals.

$$\begin{aligned} S &\rightarrow aO \mid bO \mid \lambda \\ O &\rightarrow aS \mid bS \end{aligned}$$

The application of $S \rightarrow \lambda$ completes the derivation of a terminal string. Until this occurs, a derivation alternates between applications of S and O rules. \square

Example 3.2.10

Let L be the language over $\{a, b\}$ consisting of all strings with an even number of b 's. The grammar

$$\begin{aligned} S &\rightarrow aS \mid bB \mid \lambda \\ B &\rightarrow aB \mid bS \mid bC \\ C &\rightarrow aC \mid \lambda \end{aligned}$$

that generates L combines the techniques presented in the previous examples, Example 3.2.9 for the even number of b 's and Example 3.2.7 for the arbitrary number of a 's. Deleting all rules containing C yields another grammar that generates L . \square

Example 3.2.11

Exercise 2.38 requested a regular expression for the language over $\{a, b\}$ consisting of strings with an even number of a 's and an even number of b 's. It was noted at the time that a regular expression for this language was quite complex. The flexibility provided by string generation with rules makes the construction of a context-free grammar for this language straightforward. The variables are chosen to represent the parities of the number of a 's and b 's in the derived string. The variables of the grammar with their interpretations are

Variable	Interpretation
S	Even number of a 's and even number of b 's
A	Even number of a 's and odd number of b 's
B	Odd number of a 's and even number of b 's
C	Odd number of a 's and odd number of b 's

The application of a rule adds one terminal symbol to the derived string and updates the variable to reflect the new status. The rules of the grammar are

$$\begin{aligned} S &\rightarrow aB \mid bA \mid \lambda \\ A &\rightarrow aC \mid bS \\ B &\rightarrow aS \mid bC \\ C &\rightarrow aA \mid bB. \end{aligned}$$

dd number

is occurs, a

□

of b's. The

ample 3.2.9
Deleting all

□

onsisting of
the time that
led by string
his language
er of a's and
ns are

g and updates

When the variable S is present, the derived string has an even number of a 's and an even number of b 's. The application of $S \rightarrow \lambda$ removes the variable from the sentential form, producing a string that satisfies the language specification. □

Example 3.2.12

The rules of a grammar are designed to impose a structure on the strings in the language. This structure may consist of ensuring the presence or absence of certain combinations of elements of the alphabet. We construct a grammar with alphabet $\{a, b, c\}$ whose language consists of all strings that do not contain the substring abc . The variables are used to determine how far the derivation has progressed toward generating the string abc .

$$S \rightarrow bS \mid cS \mid aB \mid \lambda$$

$$B \rightarrow aB \mid cS \mid bC \mid \lambda$$

$$C \rightarrow aB \mid bS \mid \lambda$$

The strings are built in a left-to-right manner. At most one variable is present in a sentential form. If an S is present, no progress has been made toward deriving abc . The variable B occurs when the previous terminal is an a . The variable C is present only when preceded by ab . Thus, the C rules cannot generate the terminal c . □

3.3 Regular Grammars

Regular grammars are an important subclass of context-free grammars that play a prominent role in the lexical analysis and parsing of programming languages. Regular grammars are obtained by placing restrictions on the form of the right-hand side of the rules. In Chapter 6 we will show that regular grammars generate precisely the languages that are defined by regular expressions or accepted by finite-state machines.

Definition 3.3.1

A **regular grammar** is a context-free grammar in which each rule has one of the following forms:

- i) $A \rightarrow a$,
- ii) $A \rightarrow aB$, or
- iii) $A \rightarrow \lambda$,

where $A, B \in V$, and $a \in \Sigma$.

Derivations in regular grammars have a particularly nice form; there is at most one variable present in a sentential form and that variable, if present, is the rightmost symbol in the string. Each rule application adds a terminal to the derived string until a rule of the

form $A \rightarrow a$ or $A \rightarrow \lambda$ terminates the derivation. These properties are illustrated using the regular grammar G_1

$$\begin{aligned} S &\rightarrow aS \mid aA \\ A &\rightarrow bA \mid \lambda \end{aligned}$$

from Example 3.2.6 that generates the language a^+b^* . The derivation of $aabb$,

$$\begin{aligned} S &\Rightarrow aS \\ &\Rightarrow aaA \\ &\Rightarrow aabA \\ &\Rightarrow aabbA \\ &\Rightarrow aabb, \end{aligned}$$

shows the left-to-right generation of the prefix of terminal symbols. The derivation ends with the application of the rule $A \rightarrow \lambda$.

A language generated by a regular grammar is called a *regular* language. You may recall that the family of regular languages was introduced in Chapter 2 as the set of languages described by regular expressions. There is no conflict with what might appear to be two different definitions of the same term, since we will show that regular expressions and regular grammars define the same family of languages.

A regular language may be generated by both regular and nonregular grammars. The grammars G_1 and G_2 from Example 3.2.6 both generate the language a^+b^* . The grammar G_1 is not regular because the rule $S \rightarrow AB$ does not have the specified form. A language is regular if it is generated by some regular grammar; the existence of nonregular grammars that also generate the language is irrelevant. The grammars constructed in Examples 3.2.9, 3.2.10, 3.2.11, and 3.2.12 provide additional examples of regular grammars.

Example 3.3.1

We will construct a regular grammar that generates the same language as the context-free grammar

$$\begin{aligned} G: S &\rightarrow abSA \mid \lambda \\ A &\rightarrow Aa \mid \lambda. \end{aligned}$$

The language of G is $\lambda \cup (ab)^+a^*$. The equivalent regular grammar

$$\begin{aligned} S &\rightarrow aB \mid \lambda \\ B &\rightarrow bS \mid bA \\ A &\rightarrow aA \mid \lambda \end{aligned}$$

illustrated using the

$aabb$,

The derivation ends

image. You may recall the set of languages that appear to be two similar expressions and

regular grammars. The a^+b^* . The grammar is in form. A language is nonregular grammars and in Examples 3.2.9, 3.2.10, and 3.2.11, we see that the strings generated by the grammar are not regular.

ge as the context-free

generates the strings in a left-to-right manner. The S and B rules generate a prefix from the set $(ab)^*$. If a string has a suffix of a 's, the rule $B \rightarrow bA$ is applied. The A rules are used to generate the remainder of the string. \square

3.4 Verifying Grammars

The grammars in the previous sections were built to generate specific languages. An intuitive argument was given to show that the grammar did indeed generate the correct set of strings. No matter how convincing the argument, the possibility of error exists. A proof is required to guarantee that a grammar generates precisely the desired strings.

To prove that the language of a grammar G is identical to a given language L , the inclusions $L \subseteq L(G)$ and $L(G) \subseteq L$ must be established. To demonstrate the techniques involved, we will prove that the language of the grammar

$$G: S \rightarrow AASB \mid AAB$$

$$A \rightarrow a$$

$$B \rightarrow bbb$$

is the set $L = \{a^{2n}b^{3n} \mid n > 0\}$.

A terminal string is in the language of a grammar if it can be derived from the start symbol using the rules of the grammar. The inclusion $\{a^{2n}b^{3n} \mid n > 0\} \subseteq L(G)$ is established by showing that every string in L is derivable in G . Since L contains an infinite number of strings, we cannot construct a derivation for every string in L . Unfortunately, this is precisely what is required. The apparent dilemma is solved by providing a derivation schema. The schema consists of a pattern that can be followed to construct a derivation for any string in L . A string of the form $a^{2n}b^{3n}$, for $n > 0$, can be derived by the following sequence of rule applications:

Derivation	Rule Applied
$S \xrightarrow{n-1} (AA)^{n-1}SB^{n-1}$	$S \rightarrow AASB$
$\xrightarrow{} (AA)^nB^n$	$S \rightarrow AAB$
$\xrightarrow{2n} (aa)^nB^n$	$A \rightarrow a$
$\xrightarrow{n} (aa)^n(bbb)^n$	$B \rightarrow bbb$
$= a^{2n}b^{3n}$	

where the superscripts on the \Rightarrow specify the number of applications of the rule. The preceding schema provides a “recipe,” that, when followed, can produce a derivation for any string in L .

The opposite inclusion, $L(G) \subseteq \{a^{2n}b^{3n} \mid n > 0\}$, requires each terminal string derivable in G to have the form specified by the set L . The derivation of a string in the language

consists of a finite number of rule applications, indicating the suitability of a proof by induction. The first difficulty is to determine exactly what we need to prove. We wish to establish a relationship between the a 's and b 's in all terminal strings derivable in G . A necessary condition for a string w to be a member of L is that three times the number of a 's in the string be equal to twice the number of b 's. Letting $n_x(u)$ be the number of occurrences of the symbol x in the string u , this relationship can be expressed by $3n_a(u) = 2n_b(u)$.

This numeric relationship between the symbols in a terminal string clearly is not true for every string derivable from S . Consider the derivation

$$\begin{aligned} S &\Rightarrow AASB \\ &\Rightarrow aASB. \end{aligned}$$

The string $aASB$, which is derivable in G , contains one a and no b 's.

To account for the intermediate sentential forms that occur in a derivation, relationships between the variables and terminals that hold for all steps in the derivation must be determined. When a terminal string is derived, no variables will remain and the relationships should yield the required structure of the string.

The interactions of the variables and the terminals in the rules of G must be examined to determine their effect on the derivations of terminal strings. The rule $A \rightarrow a$ guarantees that every A will eventually be replaced by a single a . The number of a 's present at the termination of a derivation consists of those already in the string and the number of A 's in the string. The sum $n_a(u) + n_A(u)$ represents the number of a 's that must be generated in deriving a terminal string from u . Similarly, every B will be replaced by the string bbb . The number of b 's in a terminal string derivable from u is $n_b(u) + 3n_B(u)$. These observations are used to construct condition (i), establishing the correspondence of variables and terminals that holds for each step in the derivation.

i) $3(n_a(u) + n_A(u)) = 2(n_b(u) + 3n_B(u))$.

The string $aASB$, which we have seen is derivable in G , satisfies this condition since $n_a(aASB) + n_A(aASB) = 2$ and $n_b(aASB) + 3n_B(aASB) = 3$.

Conditions (ii) and (iii) are

ii) $n_A(u) + n_a(u) > 1$, and

iii) the a 's and A 's in a sentential form precede the S , which precedes the b 's and B 's.

All strings in $\{a^{2n}b^{3n} \mid n > 0\}$ contain at least two a 's and three b 's. Conditions (i) and (ii) combine to yield this property. Condition (iii) prescribes the order of the symbols in a derivable string. Not all of the symbols must be present in each string; strings derivable from S by one rule application do not contain any terminal symbols.

After the appropriate relationships have been determined, we must prove that they hold for every string derivable from S . The basis of the induction consists of all strings that can be obtained by derivations of length one (the S rules). The inductive hypothesis asserts that the conditions are satisfied for all strings derivable by n or fewer rule applications. The

inductive
relation

The
strings,
(iii) hold

The
derivable
the three

Let
the induc
followed

Written i
inductive
requires t

For
 $3n_B(v)$).
applicatio
table.

Since $j(u)$
inductive l
that each ru
or transfor

We ha
are no vari
(ii) guaran
 $\{a^{2n}b^{3n} \mid n$
of G is $\{a^{2n}b^{3n} \mid n > 0\}$

As illu
language is
only a few
have been c

proof by induction to establish G. A necessary condition for a 's in the occurrences of $= 2n_b(u)$. early is not true

on, relationships must be determined by the relationships

ust be examined $\rightarrow a$ guarantees 's present at the number of A 's in ust be generated ed by the string $+ 3n_B(u)$. These ence of variables

condition since

b 's and B 's.

Conditions (i) and of the symbols in strings derivable

ove that they hold ill strings that can thesis asserts that applications. The

inductive step consists of showing that the application of an additional rule preserves the relationships.

There are two derivations of length one, $S \Rightarrow AASB$ and $S \Rightarrow AAB$. For each of these strings, $3(n_a(u) + n_A(u)) = 2(n_b(u) + 3n_B(u)) = 6$. By observation, conditions (ii) and (iii) hold for the two strings.

The inductive hypothesis asserts that (i), (ii), and (iii) are satisfied by all strings derivable by n or fewer rule applications. We now use the inductive hypothesis to show that the three properties hold for all strings generated by derivations of $n + 1$ rule applications.

Let w be a string derivable from S by a derivation $S \xrightarrow{n+1} w$ of length $n + 1$. To use the inductive hypothesis, we write the derivation of length $n + 1$ as a derivation of length n followed by a single rule application:

$$S \xrightarrow{n} u \Rightarrow w.$$

Written in this form, it is clear that the string u is derivable by n rule applications. The inductive hypothesis asserts that properties (i), (ii), and (iii) hold for u . The inductive step requires that we show that the application of one rule to u preserves these properties.

For any sentential form v , we let $j(v) = 3(n_a(v) + n_A(v))$ and $k(v) = 2(n_b(v) + 3n_B(v))$. By the inductive hypothesis, $j(u) = k(u)$ and $j(u)/3 > 1$. The effects of the application of an additional rule on the constituents of the string w are given in the following table.

Rule	$j(w)$	$k(w)$	$j(w)/3$
$S \rightarrow AASB$	$j(u) + 6$	$k(u) + 6$	$j(u)/3 + 2$
$S \rightarrow AAB$	$j(u) + 6$	$k(u) + 6$	$j(u)/3 + 2$
$A \rightarrow a$	$j(u)$	$k(u)$	$j(u)/3$
$B \rightarrow bbb$	$j(u)$	$k(u)$	$j(u)/3$

Since $j(u) = k(u)$, we conclude that $j(w) = k(w)$. Similarly, $j(w)/3 > 1$ follows from the inductive hypothesis that $j(u)/3 > 1$. The ordering of the symbols is preserved by noting that each rule application either replaces S by an appropriately ordered sequence of variables or transforms a variable to the corresponding terminal.

We have shown that the three conditions hold for every string derivable in G . Since there are no variables in a string $w \in L(G)$, condition (i) implies $3n_a(w) = 2n_b(w)$. Condition (ii) guarantees the existence of a 's and b 's, while (iii) prescribes the order. Thus $L(G) \subseteq \{a^{2n}b^{3n} \mid n > 0\}$. Having established the opposite inclusions, we conclude that the language of G is $\{a^{2n}b^{3n} \mid n > 0\}$.

As illustrated by the preceding argument, proving that a grammar generates a certain language is a complicated process. This, of course, was an extremely simple grammar with only a few rules. The inductive process is straightforward after the correct relationships have been determined. The most challenging part of the inductive proof is determining the

relationships between the variables and the terminals that must hold in the intermediate sentential forms. The relationships are sufficient if, when all references to the variables are removed, they yield the desired structure of the terminal strings.

As seen in the preceding argument, establishing that a grammar G generates a language L requires two distinct arguments:

- i) that all strings of L are derivable in G , and
- ii) that all strings generated by G are in L .

The former is accomplished by providing a derivation schema that can be used to produce a derivation for any string in L . The latter uses induction to show that each sentential form satisfies conditions that lead to the generation of a string in L . The following examples further illustrate the steps involved in these proofs.

Example 3.4.1

Let G be the grammar

$$\begin{aligned} S &\rightarrow aS \mid bB \mid \lambda \\ B &\rightarrow aB \mid bS \mid bC \\ C &\rightarrow aC \mid \lambda \end{aligned}$$

given in Example 3.2.10. We will prove that $L(G) = a^*(a^*ba^*ba^*)^*$, the set of all strings over $\{a, b\}$ with an even number of b 's. It is not true that every string derivable from S has an even number of b 's. The derivation $S \Rightarrow bB$ produces a single b . To derive a terminal string, every B must eventually be transformed into a b . Consequently, we conclude that the desired relationship asserts that $n_b(u) + n_B(u)$ is even. When a terminal string w is derived, $n_B(w) = 0$ and $n_b(w)$ is even.

We will prove that $n_b(u) + n_B(u)$ is even for all strings derivable from S . The proof is by induction on the length of the derivations.

Basis: Derivations of length one. There are three such derivations:

$$\begin{aligned} S &\Rightarrow aS \\ S &\Rightarrow bB \\ S &\Rightarrow \lambda. \end{aligned}$$

By inspection, $n_b(u) + n_B(u)$ is even for these strings.

Inductive Hypothesis: Assume that $n_b(u) + n_B(u)$ is even for all strings u that can be derived with n rule applications.

Inductive Step: To complete the proof, we need to show that $n_b(w) + n_B(w)$ is even whenever w can be obtained by a derivation of the form $S \xrightarrow{n+1} w$. The key step is to reformulate the derivation to apply the inductive hypothesis. A derivation of w of length $n+1$ can be written $S \xrightarrow{n} u \Rightarrow w$.

mediate
ibles are

language

produce
tial form
xamples

all strings
from S has
a terminal
de that the
is derived,
the proof is

be derived

even when
eformulate
+ 1 can be

By the inductive hypothesis, $n_b(u) + n_B(u)$ is even. We show that the result of the application of any rule to u preserves the parity of $n_b(u) + n_B(u)$. The table

Rule	$n_b(w) + n_B(w)$
$S \rightarrow aS$	$n_b(u) + n_B(u)$
$S \rightarrow bB$	$n_b(u) + n_B(u) + 2$
$S \rightarrow \lambda$	$n_b(u) + n_B(u)$
$B \rightarrow aB$	$n_b(u) + n_B(u)$
$B \rightarrow bS$	$n_b(u) + n_B(u)$
$B \rightarrow bC$	$n_b(u) + n_B(u)$
$C \rightarrow aC$	$n_b(u) + n_B(u)$
$C \rightarrow \lambda$	$n_b(u) + n_B(u)$

gives the value of $n_b(w) + n_B(w)$ when the corresponding rule is applied to u . Each of the rules leaves the total number of B 's and b 's fixed except the second, which adds two to the total. Thus the sum of the b 's and B 's in a string obtained from u by the application of a rule is even. Since a terminal string contains no B 's, we have shown that every string in $L(G)$ has an even number of b 's.

To complete the proof, the opposite inclusion, $L(G) \subseteq a^*(a^*ba^*ba^*)^*$, must also be established. To accomplish this, we show that every string in $a^*(a^*ba^*ba^*)^*$ is derivable in G . A string in $a^*(a^*ba^*ba^*)^*$ has the form

$$a^{n_1}ba^{n_2}ba^{n_3}\dots a^{n_{2k}}ba^{n_{2k+1}}, k \geq 0.$$

Any string in a^* can be derived using the rules $S \rightarrow aS$ and $S \rightarrow \lambda$. All other strings in $L(G)$ can be generated by a derivation of the form

Derivation	Rule Applied
$S \xrightarrow{n_1} a^{n_1}S$	$S \rightarrow aS$
$\xrightarrow{n_2} a^{n_1}bB$	$S \rightarrow bB$
$\xrightarrow{n_2} a^{n_1}ba^{n_2}B$	$B \rightarrow aB$
$\xrightarrow{n_3} a^{n_1}ba^{n_2}bS$	$B \rightarrow bS$
\vdots	
$\xrightarrow{n_{2k}} a^{n_1}ba^{n_2}ba^{n_3}\dots a^{n_{2k}}B$	$B \rightarrow aB$
$\xrightarrow{n_{2k+1}} a^{n_1}ba^{n_2}ba^{n_3}\dots a^{n_{2k}}bC$	$B \rightarrow bC$
$\xrightarrow{n_{2k+1}} a^{n_1}ba^{n_2}ba^{n_3}\dots a^{n_{2k}}ba^{n_{2k+1}}C$	$C \rightarrow aC$
$\xrightarrow{n_{2k+1}} a^{n_1}ba^{n_2}ba^{n_3}\dots a^{n_{2k}}ba^{n_{2k+1}}$	$C \rightarrow \lambda$

□

Example 3.4.2

Let G be the grammar

$$S \rightarrow aASB \mid \lambda$$

$$A \rightarrow ad \mid d$$

$$B \rightarrow bb.$$

We show that every string in $L(G)$ has at least as many b 's as a 's. The number of b 's in a terminal string depends upon the b 's and B 's in the intermediate steps of the derivation. Each B generates two b 's, while an A generates at most one a . We will prove, for every sentential form u of G , that $n_a(u) + n_A(u) \leq n_b(u) + 2n_B(u)$. Let $j(u) = n_a(u) + n_A(u)$ and $k(u) = n_b(u) + 2n_B(u)$.

Basis: There are two derivations of length one

Rule	$j(u)$	$k(u)$
$S \Rightarrow aASB$	2	2
$S \Rightarrow \lambda$	0	0

and $j(u) \leq k(u)$ for both of the derivable strings.

Inductive Hypothesis: Assume that $j(u) \leq k(u)$ for all strings u derivable from S in n or fewer rule applications.

Inductive Step: We need to prove that $j(w) \leq k(w)$ whenever $S \xrightarrow{n+1} w$. The derivation of w can be rewritten $S \xrightarrow{n} u \Rightarrow w$ and, by the inductive hypothesis, $j(u) \leq k(u)$. We must show that the inequality is preserved by an additional rule application. The effect of each rule application on j and k is indicated in the following table.

Rule	$j(w)$	$k(w)$
$S \rightarrow aASB$	$j(u) + 2$	$k(u) + 2$
$S \rightarrow \lambda$	$j(u)$	$k(u)$
$B \rightarrow bb$	$j(u)$	$k(u)$
$A \rightarrow ad$	$j(u)$	$k(u)$
$A \rightarrow d$	$j(u) - 1$	$k(u)$

The first rule adds 2 to each side of an inequality, maintaining the inequality. The final rule subtracts 1 from the smaller side, reinforcing the inequality. For a string $w \in L(G)$, the inequality yields $n_a(w) \leq n_b(w)$ as desired. \square

Example 3.4.3

In Example 3.2.2 the grammar

$$G: S \rightarrow aSdd \mid A$$

$$A \rightarrow bAc \mid bc$$

was constructed to generate the language $L = \{a^n b^m c^m d^{2n} \mid n \geq 0, m > 0\}$. We develop relationships among the variables and terminals that are sufficient to prove that $L(G) \subseteq L$. The S and the A rules enforce the numeric relationships between the a 's and d 's and the b 's and c 's. In a derivation of G , the start symbol is removed by an application of the rule $S \rightarrow A$. The presence of an A guarantees that a b will eventually be generated. These observations lead to the following four conditions for every sentential form u of G :

- i) $2n_a(u) = n_d(u)$.
- ii) $n_b(u) = n_c(u)$.
- iii) $n_S(u) + n_A(u) + n_b(u) > 0$.
- iv) The a 's precede the b 's, which precede the S or A , which precede the c 's, which precede the d 's.

The equalities guarantee that the terminals occur in correct numerical relationships. The description of the language also demands that the terminals occur in a specified order. The final condition ensures that the order is maintained at each step in the derivation. \square

3.5 Leftmost Derivations and Ambiguity

The language of a grammar is the set of terminal strings that can be derived, in any manner, from the start symbol. A terminal string may be generated by a number of different derivations. For example, Figure 3.1 gave a grammar and four derivations of the string $ababaa$ using the rules of the grammar. Any one of the derivations is sufficient to exhibit the syntactic correctness of the string.

The derivations using the natural language example that introduced this chapter were all given as leftmost derivations. This is a natural technique for readers of English since the leftmost variable is the first encountered when reading a string. To reduce the number of derivations that must be considered in determining whether a string is in the language of a grammar, we now prove that every string in the language is derivable in a leftmost manner.

Theorem 3.5.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. A string w is in $L(G)$ if, and only if, there is a leftmost derivation of w from S .

Proof. Clearly, $w \in L(G)$ whenever there is a leftmost derivation of w from S . We must establish the “only if” clause of the equivalence, that is, that every string in the $L(G)$ is derivable in a leftmost manner. Let

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \dots \Rightarrow w_n = w$$

be a, not necessarily leftmost, derivation of w in G . The independence of rule applications in a context-free grammar is used to build a leftmost derivation of w . Let w_k be the first sentential form in the derivation to which the rule application is not leftmost. If there is no such k , the derivation is already leftmost and there is nothing to show. We will show that

er of b 's in
derivation.
e, for every
 $u) + n_A(u)$

m S in n or
derivation of
 u). We must
ffect of each

The final rule
 $v \in L(G)$, the
 \square

the rule applications can be reordered so that the first $k + 1$ rule applications are leftmost. This procedure can be repeated, $n - k$ times if necessary, to produce a leftmost derivation.

By the choice of w_k , the derivation $S \xrightarrow{k} w_k$ is leftmost. Assume that A is the leftmost variable in w_k and B is the variable transformed in the $k + 1$ st step of the derivation. Then w_k can be written $u_1 A u_2 B u_3$ with $u_i \in \Sigma^*$. The application of a rule $B \rightarrow v$ to w_k has the form

$$w_k = u_1 A u_2 B u_3 \Rightarrow u_1 A u_2 v u_3 = w_{k+1}.$$

Since w is a terminal string, an A rule must eventually be applied to the leftmost variable in w_k . Let the first rule application that transforms the variable A occur at the $j + 1$ st step in the original derivation. Then the application of the rule $A \rightarrow p$ can be written

$$w_j = u_1 A q \Rightarrow u_1 p q = w_{j+1}.$$

The rules applied in steps $k + 2$ to j transform the string $u_2 v u_3$ into q . The derivation is completed by the subderivation

$$w_{j+1} \xrightarrow{*} w_n = w.$$

The original derivation has been divided into five distinct subderivations. The first k rule applications are already leftmost, so they are left intact. To construct a leftmost derivation, the rule $A \rightarrow p$ is applied to the leftmost variable at step $k + 1$. The context-free nature of rule applications permits this rearrangement. A derivation of w that is leftmost for the first $k + 1$ rule applications is obtained as follows:

$$\begin{aligned} S &\xrightarrow{k} w_k = u_1 A u_2 B u_3 \\ &\Rightarrow u_1 p u_2 B u_3 && (\text{applying } A \rightarrow p) \\ &\Rightarrow u_1 p u_2 v u_3 && (\text{applying } B \rightarrow v) \\ &\xrightarrow{j-k-1} u_1 p q = w_{j+1} && (\text{using the derivation } u_2 v u_3 \xrightarrow{*} q) \\ &\xrightarrow{n-j-1} w_n. && (\text{using the derivation } w_{j+1} \xrightarrow{*} w_n). \end{aligned}$$

Every time this procedure is repeated, the derivation becomes “more” leftmost. If the length of a derivation is n , then at most n iterations are needed to produce a leftmost derivation of w . ■

Theorem 3.5.1 does not guarantee that all sentential forms of the grammar can be generated by a leftmost derivation. Only leftmost derivations of terminal strings are assured. Consider the grammar

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \mid \lambda \\ B &\rightarrow bB \mid \lambda \end{aligned}$$

leftmost derivation. The leftmost derivation. Then w_k has the

st variable
+ 1st step
1

derivation is

the first k rule
derivation,
the nature of
for the first

that generates a^*b^* . The sentential form A can be obtained by the rightmost derivation $S \Rightarrow AB \Rightarrow A$. It is easy to see that there is no leftmost derivation of A .

A similar result (Exercise 31) establishes the sufficiency of using rightmost derivations for the generation of terminal strings. Leftmost and rightmost derivations of w from v are explicitly denoted $v \xrightarrow{L} w$ and $v \xrightarrow{R} w$.

Restricting our attention to leftmost derivations eliminates many of the possible derivations of a string. Is this reduction sufficient to establish a canonical derivation? That is, is there a unique leftmost derivation of every string in the language of a grammar? Unfortunately, the answer is no. Two distinct leftmost derivations of the string $ababaa$ were given in Figure 3.1.

The possibility of a string having several leftmost derivations introduces the notion of ambiguity. Ambiguity in formal languages is similar to ambiguity encountered frequently in natural languages. The sentence *Jack was given a book by Hemingway* has two distinct structural decompositions. The prepositional phrase *by Hemingway* can modify either the verb *was given* or the noun *book*. Each of these structural decompositions represents a syntactically correct sentence.

The compilation of a computer program utilizes the derivation produced by the parser to generate machine-language code. The compilation of a program that has two derivations uses only one of the possible interpretations to produce the executable code. An unfortunate programmer may then be faced with debugging a program that is completely correct according to the language definition but does not perform as expected. To avoid this possibility—and help maintain the sanity of programmers everywhere—the definitions of computer languages should be constructed so that no ambiguity can occur. The preceding discussion of ambiguity leads to the following definition.

Definition 3.5.2

A context-free grammar G is **ambiguous** if there is a string $w \in L(G)$ that can be derived by two distinct leftmost derivations. A grammar that is not ambiguous is called **unambiguous**.

Example 3.5.1

Let G be the grammar

$$S \rightarrow aS \mid Sa \mid a$$

that generates a^+ . G is ambiguous since the string aa has two distinct leftmost derivations:

$$\begin{aligned} S &\Rightarrow aS & S &\Rightarrow Sa \\ &\Rightarrow aa & &\Rightarrow aa. \end{aligned}$$

The language a^+ is also generated by the unambiguous grammar

$$S \rightarrow aS \mid a.$$

This grammar, being regular, has the property that all strings are generated in a left-to-right manner. The variable S remains as the rightmost symbol of the string until the recursion is halted by the application of the rule $S \rightarrow a$. \square

The previous example demonstrates that ambiguity is a property of grammars, not of languages. When a grammar is shown to be ambiguous, it is often possible to construct an equivalent unambiguous grammar. This is not always the case. There are some context-free languages that cannot be generated by any unambiguous grammar. Such languages are called **inherently ambiguous**. The syntax of most programming languages, which require unambiguous derivations, is sufficiently restrictive to avoid inherent ambiguity.

Example 3.5.2

Let G be the grammar

$$S \rightarrow bS \mid Sb \mid a$$

with language b^*ab^* . The leftmost derivations

$$\begin{array}{ll} S \Rightarrow bS & S \Rightarrow Sb \\ \Rightarrow bSb & \Rightarrow bSb \\ \Rightarrow bab & \Rightarrow bab \end{array}$$

exhibit the ambiguity of G . The ability to generate the b 's in either order must be eliminated to obtain an unambiguous grammar. $L(G)$ is also generated by the unambiguous grammars

$$\begin{array}{ll} G_1: S \rightarrow bS \mid aA & G_2: S \rightarrow bS \mid A \\ A \rightarrow bA \mid \lambda & A \rightarrow Ab \mid a. \end{array}$$

In G_1 , the sequence of rule applications in a leftmost derivation is completely determined by the string being derived. The only leftmost derivation of the string b^nab^m has the form

$$\begin{array}{l} S \xrightarrow{n} b^nS \\ \quad \Rightarrow b^n a A \\ \quad \xrightarrow{m} b^n ab^m A \\ \quad \Rightarrow b^n ab^m. \end{array}$$

A derivation in G_2 initially generates the leading b 's, followed by the trailing b 's, and finally the a . \square

A grammar is unambiguous if, at each step in a leftmost derivation, there is only one rule whose application can lead to a derivation of the desired string. This does not mean that there is only one applicable rule, but rather that the application of any other rule makes it impossible to complete a derivation of the string.

stri
init
wit
 $S -$
the
pre
suc
the

Exa
The
is an

A str
a sin
gram

A
natu
trees.
deriva
deriva
in ter
frontie
of the

3.6

In the
ing an
demon
tactic n
syntax,

o-right
sion is
□
not of
nstruct
ontext-
ges are
require

Consider the possibilities encountered in constructing a leftmost derivation of the string $bbabb$ using the grammar G_2 from Example 3.5.2. There are two S rules that can initiate a derivation. Derivations initiated with the rule $S \rightarrow A$ generate strings beginning with a . Consequently, a derivation of $bbabb$ must begin with the application of the rule $S \rightarrow bS$. The second b is generated by another application of the same rule. At this point, the derivation continues using $S \rightarrow A$. Another application of $S \rightarrow bS$ would generate the prefix bbb . The suffix bb is generated by two applications of $A \rightarrow Ab$. The derivation is successfully completed with an application of $A \rightarrow a$. Since the terminal string specifies the exact sequence of rule applications, the grammar is unambiguous.

Example 3.5.3

The grammar from Example 3.2.4 that generates the language $L = \{a^n b^m \mid 0 \leq n \leq m \leq 2n\}$ is ambiguous. The string $aabbb$ can be generated by the derivations

$$\begin{array}{ll} S \Rightarrow aSb & S \Rightarrow aSbb \\ \Rightarrow aaSbbb & \Rightarrow aaSbbb \\ \Rightarrow aabbb & \Rightarrow aabbb. \end{array}$$

A strategy for unambiguously generating the strings of L is to initially produce a 's with a single matching b . This is followed by generating a 's with two b 's. An unambiguous grammar that produces the strings of L in this manner is

$$\begin{array}{l} S \Rightarrow aSb \mid A \mid \lambda \\ A \Rightarrow aAb \mid abb. \end{array}$$

□

A derivation tree depicts the transformation of the variables in a derivation. There is a natural one-to-one correspondence between leftmost (rightmost) derivations and derivation trees. Definition 3.1.4 outlines the construction of a derivation tree directly from a leftmost derivation. Conversely, a unique leftmost derivation of a string w can be extracted from a derivation tree with frontier w . Because of this correspondence, ambiguity is often defined in terms of derivation trees. A grammar G is ambiguous if there is a string in $L(G)$ that is the frontier of two distinct derivation trees. Figure 3.3 shows that the two leftmost derivations of the string $ababaa$ given in Figure 3.1 generate distinct derivation trees.

3.6 Context-Free Grammars and Programming Language Definition

In the preceding sections we used context-free grammars to generate “toy” languages using an alphabet with only a few elements and a small number of rules. These examples demonstrated the ability of context-free rules to produce strings that satisfy particular syntactic requirements. A programming language has a larger alphabet and more complicated syntax, increasing the number and complexity of the rules needed to define the language.

The first formal specification of a high-level programming language was given for the language ALGOL 60 by John Backus [1959] and Peter Naur [1963]. The system employed by Backus and Naur is now referred to as *Backus-Naur form*, or *BNF*. The programming language Java, whose specification was given in BNF, will be used to illustrate principles of the syntactic definition of a programming language. A complete formal definition of Java is given in Appendix IV.

A BNF description of a language is a context-free grammar; the only difference is the notation used to define the rules. We will give the rules using the context-free notation, with one exception. The subscript *opt* after a variable or a terminal indicates that it is optional. This notation reduces the number of rules that need to be written, but rules with optional components can easily be transformed into equivalent context-free rules. For example, $A \rightarrow B_{opt}$ and $A \rightarrow B_{opt}C$ can be replaced by the rules $A \rightarrow B | \lambda$ and $A \rightarrow BC | C$, respectively.

The notational conventions used in the Java rules are the same as the natural language example at the beginning of the chapter. The names of the variables indicate the components of the language that they generate and are enclosed in $\langle \rangle$. Java keywords are given in bold, and other terminal symbols are represented by character strings delimited by blanks.

The design of a programming language, like the design of a complex program, is greatly simplified utilizing modularity to develop subsets of the grammar independently. The techniques you have used in building small rule sets provide the skills needed to design a grammar for larger languages with more complicated syntaxes. These techniques include using rules to ensure the presence or relative position of elements and using recursion to generate sequences and to nest parentheses.

To illustrate the principles of language design, we will examine rules that define literals, identifiers, and arithmetic expressions in Java. Literals, strings that have a fixed type and value, are frequently used to initialize variables, to set the bounds on repetitive statements, and to store standard messages to be output. The rule for the variable *(Literal)* defines the types of Java literals. The Java literals, along with the variables that generate them, are

Literal	Variable	Examples
Boolean	$\langle BooleanLiteral \rangle$	true, false
Character	$\langle CharacterLiteral \rangle$	'a', '\n' (linefeed escape sequence), '\pi',
String	$\langle StringLiteral \rangle$	"" (empty string), "This is a nonempty string"
Integer	$\langle IntegerLiteral \rangle$	0, 356, 1234L (long), 077 (octal), 0x1ab2 (hex)
Floating point	$\langle FloatingPointLiteral \rangle$	2., .2, 2.0, 12.34, 2e3, 6.2e-5
Null	$\langle NullLiteral \rangle$	null

Each floating point literal can have an f, F, d, or D as a suffix to indicate its precision. The definitions for the complete set of Java literals are given in rules 143–167 in Appendix IV.

We will consider the rules that define the floating point literals, since they have the most interesting syntactic variations. The four $\langle \text{FloatingPointLiteral} \rangle$ rules specify the general form of floating point literals.

$$\begin{aligned}\langle \text{FloatingPointLiteral} \rangle \rightarrow & \langle \text{Digits} \rangle . \langle \text{Digits}_{opt} \rangle \langle \text{ExponentPart} \rangle_{opt} \langle \text{FloatTypeSuffix} \rangle_{opt} | \\ & . \langle \text{Digits} \rangle \langle \text{ExponentPart} \rangle_{opt} \langle \text{FloatTypeSuffix} \rangle_{opt} | \\ & \langle \text{Digits} \rangle \langle \text{ExponentPart} \rangle \langle \text{FloatTypeSuffix} \rangle_{opt} | \\ & \langle \text{Digits} \rangle \langle \text{ExponentPart} \rangle_{opt} \langle \text{FloatTypeSuffix} \rangle\end{aligned}$$

The variables $\langle \text{Digits} \rangle$, $\langle \text{ExponentPart} \rangle$, and $\langle \text{FloatTypeSuffix} \rangle$ generate the components that make up the literal. The variable $\langle \text{Digits} \rangle$ generates a string of digits using recursion. The nonrecursive rule ensures the presence of at least one digit.

$$\begin{aligned}\langle \text{Digits} \rangle \rightarrow & \langle \text{Digit} \rangle | \langle \text{Digits} \rangle \langle \text{Digit} \rangle \\ \langle \text{Digit} \rangle \rightarrow & 0 | \langle \text{NonZeroDigit} \rangle \\ \langle \text{NonZeroDigit} \rangle \rightarrow & 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \\ \langle \text{ExponentPart} \rangle \rightarrow & \langle \text{ExponentIndicator} \rangle \langle \text{SignedInteger} \rangle \\ \langle \text{ExponentIndicator} \rangle \rightarrow & e | E \\ \langle \text{SignedInteger} \rangle \rightarrow & \langle \text{Sign} \rangle_{opt} \langle \text{Digits} \rangle \\ \langle \text{Sign} \rangle \rightarrow & + | - \\ \langle \text{FloatTypeSuffix} \rangle \rightarrow & f | F | d | D\end{aligned}$$

The subscript *opt* in the rule $\langle \text{SignedInteger} \rangle \rightarrow \langle \text{Sign} \rangle_{opt} \langle \text{Digits} \rangle$ indicates that a signed integer may begin with $+$ or $-$, but the sign is not necessary.

The first $\langle \text{FloatingPointLiteral} \rangle$ rule generates literals of the form 1., 1.1, 1.1e, 1.e, 1.1ef, 1.f, 1.1f, and 1.ef. The leading string of digits and decimal point are required; all other components are optional. The second rule generates literals that begin with a decimal point, and the last two rules define the floating point literals without decimal points.

Identifiers are used as names of variables, types, methods, and so forth. Identifiers are defined by the rules

$$\begin{aligned}\langle \text{Identifier} \rangle \rightarrow & \langle \text{IdentifierChars} \rangle \\ \langle \text{IdentifierChars} \rangle \rightarrow & \langle \text{JavaLetter} \rangle | \langle \text{JavaLetter} \rangle \langle \text{JavaLetterOrDigit} \rangle\end{aligned}$$

where the Java letters include the letters A to Z and a to z, the underscore $_$, and the dollar sign $$$, along with other characters represented in the Unicode encoding.

The definition of statements in Java begins with the variable $\langle \text{Statement} \rangle$:

$$\begin{aligned}\langle \text{Statement} \rangle \rightarrow & \langle \text{StatementWithoutTrailingSubstatement} \rangle | \langle \text{LabeledStatement} \rangle | \\ & \langle \text{IfThenStatement} \rangle | \langle \text{IfThenElseStatement} \rangle | \\ & \langle \text{WhileStatement} \rangle | \langle \text{ForStatement} \rangle.\end{aligned}$$

Statements without trailing substatements include blocks and the **do** and **switch** statements. The entire set of statements is given in rules 73–75 in Appendix IV. Like the rules for the literals, the statement rules define the high-level structure of a statement. For example, **if-then** and **do** statements are defined by

$$\begin{aligned}\langle \text{IfThenStatement} \rangle &\rightarrow \text{if } (\langle \text{Expression} \rangle) \langle \text{Statement} \rangle \\ \langle \text{DoStatement} \rangle &\rightarrow \text{do } \langle \text{Statement} \rangle \text{ while } (\langle \text{Expression} \rangle).\end{aligned}$$

The occurrence of the variable $\langle \text{Statement} \rangle$ on the right-hand side of the preceding rules generates the statements to be executed after the condition in the **if-then** statement and in the loop in the **do** loop.

The evaluation of expressions is the key to numeric computation and checking the conditions in **if-then**, **do**, **while**, and **switch** statements. The syntax of expressions is defined by the rules 118–142 in Appendix IV. The syntax is complicated because Java has numeric and Boolean expressions that may utilize postfix, prefix, or infix operators. Rather than describing individual rules, we will look at several subderivations that occur in the derivation of a simple arithmetic assignment.

The first steps transform the variable $\langle \text{Expression} \rangle$ to an assignment:

$$\begin{aligned}\langle \text{Expression} \rangle &\Rightarrow \langle \text{AssignmentExpression} \rangle \\ &\Rightarrow \langle \text{Assignment} \rangle \\ &\Rightarrow \langle \text{LeftHandSide} \rangle \langle \text{AssignmentOperator} \rangle \langle \text{AssignmentExpression} \rangle \\ &\Rightarrow \langle \text{ExpressionName} \rangle \langle \text{AssignmentOperator} \rangle \langle \text{AssignmentExpression} \rangle \\ &\Rightarrow \langle \text{Identifier} \rangle \langle \text{AssignmentOperator} \rangle \langle \text{AssignmentExpression} \rangle \\ &\Rightarrow \langle \text{Identifier} \rangle = \langle \text{AssignmentExpression} \rangle.\end{aligned}$$

The next step is to derive $\langle \text{AdditiveExpression} \rangle$ from $\langle \text{AssignmentExpression} \rangle$.

$$\begin{aligned}\langle \text{AssignmentExpression} \rangle &\Rightarrow \langle \text{ConditionalExpression} \rangle \\ &\Rightarrow \langle \text{ConditionalOrExpression} \rangle \\ &\Rightarrow \langle \text{ConditionalAndExpression} \rangle \\ &\Rightarrow \langle \text{InclusiveOrExpression} \rangle \\ &\Rightarrow \langle \text{ExclusionOrExpression} \rangle \\ &\Rightarrow \langle \text{AndExpression} \rangle \\ &\Rightarrow \langle \text{EqualityExpression} \rangle \\ &\Rightarrow \langle \text{RelationalExpression} \rangle \\ &\Rightarrow \langle \text{ShiftExpression} \rangle \\ &\Rightarrow \langle \text{AdditiveExpression} \rangle.\end{aligned}$$

Derivations begin with additive expressions.

($\text{AdditiveExpression}$)

begins such a derivation with variables, or ($\langle \text{Exp} \rangle$)

The rules themselves language is described. The resulting values for higher-level

The start symbol program begins with (Comp)

A string of terminals

Exercises

1. Let G be the grammar

- a) Give a derivation of $\langle \text{AdditiveExpression} \rangle$
- b) Build the parse tree for the expression $2 * 3 + 4 - 5$
- c) Use set notation to show the states of the parser for the expression $2 * 3 + 4 - 5$

2. Let G be the grammar

- a) Give a leftmost derivation of $\langle \text{AdditiveExpression} \rangle$
- b) Give a rightmost derivation of $\langle \text{AdditiveExpression} \rangle$

atements.
es for the
example,

ding rules
ent and in

cking the
is defined
s numeric
ather than
derivation

ion)
ession}

Derivations beginning with $\langle \text{AdditiveExpression} \rangle$ produce correctly formed expressions with additive operators, multiplicative operators, and parentheses. For example,

$$\begin{aligned}\langle \text{AdditiveExpression} \rangle &\Rightarrow \langle \text{AdditiveExpression} \rangle + \langle \text{MultiplicativeExpression} \rangle \\ &\Rightarrow \langle \text{MultiplicativeExpression} \rangle + \langle \text{MultiplicativeExpression} \rangle \\ &\Rightarrow \langle \text{UnaryExpression} \rangle + \langle \text{MultiplicativeExpression} \rangle \\ &\stackrel{*}{\Rightarrow} \langle \text{Identifier} \rangle + \langle \text{MultiplicativeExpression} \rangle \\ &\Rightarrow \langle \text{Identifier} \rangle + \\ &\quad \langle \text{MultiplicativeExpression} \rangle * \langle \text{MultiplicativeExpression} \rangle\end{aligned}$$

begins such a derivation. Derivations from $\langle \text{UnaryExpression} \rangle$ can produce literals, variables, or $((\text{Expression}))$ to obtain nested parentheses.

The rules that define identifiers, literals, and expressions show how the design of a large language is decomposed into creating rules for frequently recurring subsets of the language. The resulting variables $\langle \text{Identifier} \rangle$, $\langle \text{Literal} \rangle$, and $\langle \text{Expression} \rangle$ become the building blocks for higher-level rules.

The start symbol of the grammar is $\langle \text{CompilationUnit} \rangle$ and the derivation of a Java program begins with the rule

$$\begin{aligned}\langle \text{CompilationUnit} \rangle \rightarrow & \langle \text{PackageDeclaration} \rangle_{opt} \langle \text{ImportDeclarations} \rangle_{opt} \\ & \langle \text{TypeDeclarations} \rangle_{opt}.\end{aligned}$$

A string of terminal symbols derivable from this rule is a syntactically correct Java program.

Exercises

1. Let G be the grammar

$$\begin{aligned}S &\rightarrow abSc \mid A \\ A &\rightarrow cAd \mid cd.\end{aligned}$$

- a) Give a derivation of $ababccddcc$.
- b) Build the derivation tree for the derivation in part (a).
- c) Use set notation to define $L(G)$.

2. Let G be the grammar

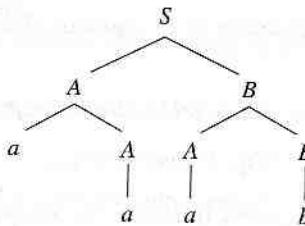
$$\begin{aligned}S &\rightarrow ASB \mid \lambda \\ A &\rightarrow aAb \mid \lambda \\ B &\rightarrow bBa \mid ba.\end{aligned}$$

- a) Give a leftmost derivation of $aabbba$.
- b) Give a rightmost derivation of $abaabbbbabbaa$.

- c) Build the derivation tree for the derivations in parts (a) and (b).
 d) Use set notation to define $L(G)$.
3. Let G be the grammar

$$\begin{aligned} S &\rightarrow SAB \mid \lambda \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid \lambda. \end{aligned}$$

- a) Give a leftmost derivation of $abbaab$.
 b) Give two leftmost derivations of aa .
 c) Build the derivation tree for the derivations in part (b).
 d) Give a regular expression for $L(G)$.
4. Let DT be the derivation tree



- a) Give a leftmost derivation that generates the tree DT .
 b) Give a rightmost derivation that generates the tree DT .
 c) How many different derivations are there that generate DT ?
 5. Give the leftmost and rightmost derivations corresponding to each of the derivation trees given in Figure 3.3.
 6. For each of the following context-free grammars, use set notation to define the language generated by the grammar.
- a) $S \rightarrow aaSB \mid \lambda$
 $B \rightarrow bB \mid b$
- b) $S \rightarrow aSbb \mid A$
 $A \rightarrow cA \mid c$
- c) $S \rightarrow abSdc \mid A$
 $A \rightarrow cdAba \mid \lambda$
- d) $S \rightarrow aSb \mid A$
 $A \rightarrow cAd \mid cBd$
 $B \rightarrow aBb \mid ab$
- e) $S \rightarrow aSB \mid AB$
 $B \rightarrow bb \mid b$
7. Construct a grammar over $\{a, b, c\}$ whose language is $\{a^n b^{2n} c^m \mid n, m > 0\}$.
 8. Construct a grammar over $\{a, b, c\}$ whose language is $\{a^n b^m c^{2n+m} \mid n, m > 0\}$.
 9. Construct a grammar over $\{a, b, c\}$ whose language is $\{a^n b^m c^i \mid 0 \leq n + m \leq i\}$.

10. Con...
 11. Con...
 12. Cons...
 the s...
 * 13. Cons...
 length...
 14. For e...
 genera...
 a) $S \rightarrow$
 $A \rightarrow$
 b) $S \rightarrow$
 $A \rightarrow$
 $B \rightarrow$

- For Exerci...
 15. The se...
 precede...
 16. The se...
 17. The se...
 Beware...
 18. The se...
 19. The se...
 20. The se...
 immedi...
 21. The se...
 22. The se...
 23. The se...
 * 24. The se...
 ab .
 25. The se...
 26. The gr...
 even nu...
 27. Prove th...
 28. Let G be

Prove th...

10. Construct a grammar over $\{a, b\}$ whose language is $\{a^m b^n \mid 0 \leq n \leq m \leq 3n\}$.
11. Construct a grammar over $\{a, b\}$ whose language is $\{a^m b^i a^n \mid i = m + n\}$.
12. Construct a grammar over $\{a, b\}$ whose language contains precisely the strings with the same number of a 's and b 's.
- * 13. Construct a grammar over $\{a, b\}$ whose language contains precisely the strings of odd length that have the same symbol in the first and middle positions.
14. For each of the following regular grammars, give a regular expression for the language generated by the grammar.

a) $S \rightarrow aA$ $A \rightarrow aA \mid bA \mid b$	c) $S \rightarrow aS \mid bA$ $A \rightarrow bB$ $B \rightarrow aB \mid \lambda$
b) $S \rightarrow aA$ $A \rightarrow aA \mid bB$ $B \rightarrow bB \mid \lambda$	d) $S \rightarrow aS \mid bA \mid \lambda$ $A \rightarrow aA \mid bS$

For Exercises 15 through 25, give a regular grammar that generates the described language.

15. The set of strings over $\{a, b, c\}$ in which all the a 's precede the b 's, which in turn precede the c 's. It is possible that there are no a 's, b 's, or c 's.
16. The set of strings over $\{a, b\}$ that contain the substring aa and the substring bb .
17. The set of strings over $\{a, b\}$ in which the substring aa occurs at least twice. (Hint: Beware of the substring aaa .)
18. The set of strings over $\{a, b\}$ that contain the substring ab and the substring ba .
19. The set of strings over $\{a, b\}$ in which the number of a 's is divisible by three.
20. The set of strings over $\{a, b\}$ in which every a is either immediately preceded or immediately followed by b , for example, $baab, aba$, and b .
21. The set of strings over $\{a, b\}$ that do not contain the substring aba .
22. The set of strings over $\{a, b\}$ in which the substring aa occurs exactly once.
23. The set of strings of odd length over $\{a, b\}$ that contain exactly two b 's.
- * 24. The set of strings over $\{a, b, c\}$ with an odd number of occurrences of the substring ab .
25. The set of strings over $\{a, b\}$ with an even number of a 's or an odd number of b 's.
26. The grammar in Figure 3.1 generates $(b^*ab^*ab^*)^+$, the set of all strings with a positive, even number of a 's. Prove this.
27. Prove that the grammar given in Example 3.2.2 generates the prescribed language.
28. Let G be the grammar

$$\begin{aligned} S &\rightarrow aSb \mid B \\ B &\rightarrow bB \mid b. \end{aligned}$$

Prove that $L(G) = \{a^n b^m \mid 0 \leq n < m\}$.

29. Let G be the grammar

$$S \rightarrow aSaa \mid B$$

$$B \rightarrow bbBdd \mid C$$

$$C \rightarrow bd.$$

- a) What is $L(G)$?
- b) Prove that $L(G)$ is the set given in part (a).

* 30. Let G be the grammar

$$S \rightarrow aSbS \mid aS \mid \lambda.$$

Prove that every prefix of a string in $L(G)$ has at least as many a 's as b 's.

31. Let G be a context-free grammar and $w \in L(G)$. Prove that there is a rightmost derivation of w in G .

32. Let G be the grammar

$$S \rightarrow aS \mid Sb \mid ab.$$

- a) Give a regular expression for $L(G)$.
- b) Construct two leftmost derivations of the string $aabb$.
- c) Build the derivation trees for the derivations from part (b).
- d) Construct an unambiguous grammar equivalent to G .

33. For each of the following grammars, give a regular expression or set-theoretic definition for the language of the grammar. Show that the grammar is ambiguous and construct an equivalent unambiguous grammar.

a) $S \rightarrow aaS \mid aaaaaS \mid \lambda$

b) $S \rightarrow aSA \mid \lambda$

$A \rightarrow bA \mid \lambda$

c) $S \rightarrow aSb \mid aAb$

$A \rightarrow cAd \mid B$

$B \rightarrow aBb \mid \lambda$

d) $S \rightarrow AaSbB \mid \lambda$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid \lambda$

* e) $S \rightarrow A \mid B$

$A \rightarrow abA \mid \lambda$

$B \rightarrow aBb \mid \lambda$

34.

35.

36.

37.

38.

* 39.

a)

b)

34. Let G be the grammar

$$\begin{aligned} S &\rightarrow aA \mid \lambda \\ A &\rightarrow aA \mid bB \\ B &\rightarrow bB \mid b. \end{aligned}$$

a) Give a regular expression for $L(G)$.

b) Prove that G is unambiguous.

35. Let G be the grammar

$$\begin{aligned} S &\rightarrow aS \mid aA \mid a \\ A &\rightarrow aAb \mid ab. \end{aligned}$$

a) Give a set-theoretic definition of $L(G)$.

b) Prove that G is unambiguous.

36. Let G be the grammar

$$\begin{aligned} S &\rightarrow aS \mid bA \mid \lambda \\ A &\rightarrow bA \mid aS \mid \lambda. \end{aligned}$$

Give a regular expression for $L(G)$. Is G ambiguous? If so, give an unambiguous grammar that generates $L(G)$. If not, prove it.

37. Construct unambiguous grammars for the languages $L_1 = \{a^n b^n c^m \mid n, m > 0\}$ and $L_2 = \{a^n b^m c^n \mid n, m > 0\}$. Construct a grammar G that generates $L_1 \cup L_2$. Prove that G is ambiguous. This is an example of an inherently ambiguous language. Explain, intuitively, why every grammar generating $L_1 \cup L_2$ must be ambiguous.

38. Use the definition of Java in Appendix IV to construct a derivation of the string `1.3e2` from the variable `(Literal)`.

* 39. Let G_1 and G_2 be the following grammars:

$$\begin{array}{ll} G_1: S \rightarrow aABb & G_2: S \rightarrow AABB \\ & A \rightarrow aA \mid a \\ & B \rightarrow bB \mid b \end{array}$$

a) For each variable X , show that the right-hand side of every X rule of G_1 is derivable from the corresponding variable X using the rules of G_2 . Use this to conclude that $L(G_1) \subseteq L(G_2)$.

b) Prove that $L(G_1) = L(G_2)$.

- *40. A **right-linear grammar** is a context-free grammar, each of whose rules has one of the following forms:

- i) $A \rightarrow w$, or
- ii) $A \rightarrow wB$,

where $w \in \Sigma^*$. Prove that a language L is generated by a right-linear grammar if, and only if, L is generated by a regular grammar.

41. Try to construct a regular grammar that generates the language $\{a^n b^n \mid n \geq 0\}$. Explain why none of your attempts succeed.
42. Try to construct a context-free grammar that generates the language $\{a^n b^n c^n \mid n \geq 0\}$. Explain why none of your attempts succeed.

Bibliographic Notes

Context-free grammars were introduced by Chomsky [1956], [1959]. Backus-Naur form was developed by Backus [1959]. This formalism was used to define the programming language ALGOL; see Naur [1963]. The BNF definition of Java is given in Appendix IV. The equivalence of context-free languages and the languages generated by BNF definitions was noted by Ginsburg and Rice [1962].

Properties of ambiguity are examined in Floyd [1962], Cantor [1962], and Chomsky and Schutzenberger [1963]. Inherent ambiguity was first noted in Parikh [1966]. A proof that the language in Exercise 37 is inherently ambiguous can be found in Harrison [1978]. Closure properties for ambiguous and inherently ambiguous languages were established by Ginsburg and Ullian [1966a, 1966b].

The d
right-
the la
deriva
the fo
proper

- i) Th
of
- ii) Th
int

In this
Choms
context
form. T
each of

The
gramma
normal
the dept
patterns

f the

, and
plain
 ≥ 0 .

CHAPTER 4

Normal Forms for Context-Free Grammars

form
iming
ix IV.
itions

omsky
proof
1978].
ned by

The definition of a context-free grammar permits unlimited flexibility in the form of the right-hand side of a rule. This flexibility is advantageous for designing grammars, but the lack of structure makes it difficult to establish general relationships about grammars, derivations, and languages. Normal forms for context-free grammars impose restrictions on the form of the rules to facilitate the analysis of context-free grammars and languages. Two properties characterize a normal form:

- i) The grammars that satisfy the normal form requirements should generate the entire set of context-free languages.
- ii) There should be an algorithmic transformation of an arbitrary context-free grammar into an equivalent grammar in the normal form.

In this chapter we introduce two important normal forms for context-free grammars, the Chomsky and Greibach normal forms. Transformations are developed to convert an arbitrary context-free grammar into an equivalent grammar that satisfies the conditions of the normal form. The transformations consist of a series of rule modifications, additions, and deletions, each of which preserves the language of the original grammar.

The restrictions imposed on the rules by a normal form ensure that derivations of the grammar have certain desirable properties. The derivation trees for derivations in a Chomsky normal form grammar are binary trees. In Chapter 7 we will use the relationship between the depth and number of leaves of a binary tree to guarantee the existence of repetitive patterns in strings in a context-free language. We will also use the properties of derivations

in Chomsky normal form grammars to develop an efficient algorithm for deciding if a string is in the language of a grammar.

A derivation using the rules of a Greibach normal form grammar builds a string in a left-to-right manner. Each rule application adds one terminal symbol to the derived string. The Greibach normal form will be used in Chapter 7 to establish a machine-based characterization of the languages that can be generated by context-free grammars.

4.1 Grammar Transformations

The transformation of a grammar into a normal form consists of a sequence of rule additions, deletions, or modifications, each of which preserves the language of the original grammar. The objective of each step is to produce rules that satisfy some desirable property. The sequence of transformations is designed to ensure that each successive step maintains the properties produced by the previous transformations.

Our first transformation is quite simple; the goal is to limit the role of the start symbol to the initiation of a derivation. If the start symbol is a recursive variable, a derivation of the form $S \xrightarrow{*} uSv$ permits the start symbol to occur in sentential forms in intermediate steps of a derivation. For any grammar G , we build an equivalent grammar G' in which the start symbol is nonrecursive. The observation that is important for this transformation is that the start symbol of G' need not be the same variable as the start symbol of G . Although this transformation is straightforward, it demonstrates the steps that are required to prove a transformation preserves the language of the original grammar.

Lemma 4.1.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. There is a grammar G' that satisfies

- i) $L(G) = L(G')$.
- ii) The start symbol of G' is not a recursive variable.

Proof. If the start symbol S does not occur on the right-hand side of a rule of G , then there is nothing to change and $G' = G$. If S is a recursive variable, the recursion of the start symbol must be removed. The alteration is accomplished by “taking a step backward” with the start of a derivation. The grammar $G' = (V \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$ is constructed by designating a new start symbol S' and adding $S' \rightarrow S$ to the rules of G . The two grammars generate the same language since any string u derivable in G by a derivation $S \xrightarrow{G} u$ can be obtained by the derivation $S' \xrightarrow{G'} S \xrightarrow{G} u$. Moreover, the only role of the rule added to P' is to initiate a derivation in G' , the remainder of which is identical to a derivation in G . Thus a string derivable in G' is also derivable in G . ■

g if a string

a string in
the derived
chne-based
ars.

le additions,
al grammar.
operty. The
aintains the

start symbol
vation of the
ediate steps
ich the start
ation is that
G. Although
ed to prove a

it satisfies

le of G, then
on of the start
ckward" with
is constructed
wo grammars
 $S \xrightarrow{G} u$ can be
added to P' is
on in G. Thus

Example 4.1.1

The start symbol of the grammar G

$$\begin{array}{ll} G: S \rightarrow aS \mid AB \mid AC & G': S' \rightarrow S \\ A \rightarrow aA \mid \lambda & S \rightarrow aS \mid AB \mid AC \\ B \rightarrow bB \mid bS & A \rightarrow aA \mid \lambda \\ C \rightarrow cC \mid \lambda & B \rightarrow bB \mid bS \\ & C \rightarrow cC \mid \lambda \end{array}$$

is recursive. The technique outlined in Lemma 4.1.1 is used to construct the equivalent grammar G'. The start symbol of G' is S' , which is nonrecursive. The variable S is still recursive in G', but it is not the start symbol of the new grammar. \square

The process of transforming grammars into normal forms consists of removing and adding rules to the grammar. With each alteration, the language generated by the grammar should remain unchanged. Lemma 4.1.2 establishes a simple criterion by which rules may be added to a grammar without altering the language. Lemma 4.1.3 provides a method for removing a rule. Of course, the removal of a rule must be accompanied by the addition of other rules so the language does not change.

Lemma 4.1.2

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. If $A \xrightarrow{G} w$, then the grammar $G' = (V, \Sigma, P \cup \{A \rightarrow w\}, S)$ is equivalent to G.

Proof. Clearly, $L(G) \subseteq L(G')$ since every rule in G is also in G'. The other inclusion follows from the observation that the effect of the application of the rule $A \rightarrow w$ in a derivation in G' can be accomplished in G by employing the derivation $A \xrightarrow{G} w$ to transform A to w. \blacksquare

Lemma 4.1.3

Let $G = (V, \Sigma, P, S)$ be a context-free grammar, $A \rightarrow uBv$ be a rule in P, and $B \rightarrow w_1 | w_2 | \dots | w_n$ be the B rules of P. The grammar $G' = (V, \Sigma, P', S)$ where

$$P' = (P - \{A \rightarrow uBv\}) \cup \{A \rightarrow uw_1v \mid uw_2v \mid \dots \mid uw_nv\}$$

is equivalent to G.

Proof. Since each rule $A \rightarrow uw_nv$ is derivable in G, the inclusion $L(G') \subseteq L(G)$ follows from Lemma 4.1.2.

The opposite inclusion is established by showing that every terminal string derivable in G using the rule $A \rightarrow uBv$ is also derivable in G' . The rightmost derivation of a terminal string that utilizes this rule has the form

$$S \xrightarrow{*} pAq \Rightarrow puBvq \xrightarrow{*} pxBvq \Rightarrow pxw_ivq \xrightarrow{*} w,$$

where $u \xrightarrow{*} x$ transforms u into a terminal string. The same string can be generated in G' using the rule $A \rightarrow uw_iv$:

$$S \xrightarrow{*} pAq \Rightarrow puw_ivq \xrightarrow{*} pxw_ivq \xrightarrow{*} w. \blacksquare$$

4.2 Elimination of λ -Rules

In the derivation of a terminal string, the intermediate sentential forms may contain variables that do not generate terminal symbols. These variables are removed from the sentential form by applications of λ -rules. This property is illustrated by the derivation of the string $aaaa$ in the grammar

$$S \rightarrow SaB \mid aB$$

$$B \rightarrow bB \mid \lambda.$$

The language generated by this grammar is $(ab^*)^+$. The leftmost derivation of $aaaa$ generates four B 's, each of which is removed by the application of the rule $B \rightarrow \lambda$:

$$\begin{aligned} S &\Rightarrow SaB \\ &\Rightarrow SaBaB \\ &\Rightarrow SaBaBaB \\ &\Rightarrow aBaBaBaB \\ &\Rightarrow aaBaBaB \\ &\Rightarrow aaaBaB \\ &\Rightarrow aaaaB \\ &\Rightarrow aaaa. \end{aligned}$$

The objective of our next transformation is to ensure that every variable in a sentential form contributes to the terminal string that is derived. In the preceding example, none of the occurrences of the B 's produced terminals. A more efficient approach would be to avoid the generation of variables that are subsequently removed by λ -rules.

The language $(ab^*)^+$ is also generated by the grammar

$$S \rightarrow SaB \mid Sa \mid aB \mid a$$

$$B \rightarrow bB \mid b$$

ring derivable
of a terminal

enerated in G'

ntain variables
sentential form
he string $aaaa$

vation of $aaaa$
 $B \rightarrow \lambda$:

le in a sentential
ple, none of the
ould be to avoid

that does not have λ -rules. The derivation of the string $aaaa$,

$$\begin{aligned} S &\Rightarrow Sa \\ &\Rightarrow Saa \\ &\Rightarrow Saaa \\ &\Rightarrow aaaa, \end{aligned}$$

uses half the number of rule applications as before. This efficiency is gained at the expense of increasing the number of rules of the grammar.

The effect of a λ -rule $B \rightarrow \lambda$ in a derivation is not limited to the variable B . Consider the grammar

$$\begin{aligned} S &\rightarrow aAb \\ A &\rightarrow aA \mid B \\ B &\rightarrow bB \mid \lambda \end{aligned}$$

that generates the language a^+b^+ . The variable A occurs in the derivation of the string ab ,

$$\begin{aligned} S &\Rightarrow aAb \\ &\Rightarrow aBb \\ &\Rightarrow ab, \end{aligned}$$

but the subderivation beginning with the application of the rule $A \rightarrow B$ does not produce terminal symbols. Whenever a variable can derive the null string, as A does in the preceding example, it is possible that its occurrence in a sentential form may not contribute to the string. We will call a variable that can derive the null string **nullable**. If a sentential form contains a nullable variable, the length of the derived string can be reduced by a sequence of rule applications.

We will now present a technique to remove λ -rules from a grammar. The modification of the grammar consists of three steps:

1. The determination of the set of nullable variables,
2. The addition of rules in which occurrences of the nullable variables are omitted, and
3. The deletion of the λ -rules.

If a grammar has no nullable variables, each variable that occurs in a derivation contributes to the generation of terminal symbols. Consequently, the application of a rule cannot reduce the length of the sentential form. A grammar with this property is called **noncontracting**.

The first step in the removal of λ -rules is the determination of the set of nullable variables. Algorithm 4.2.1 iteratively constructs this set from the λ -rules of the grammar. The algorithm utilizes two sets: the set **NULL** collects the nullable variables and **PREV**, which contains the nullable variables from the previous iteration, triggers the halting condition.

Algorithm 4.2.1**Construction of the Set of Nullable Variables**

input: context-free grammar $G = (V, \Sigma, P, S)$

1. $\text{NULL} := \{A \mid A \rightarrow \lambda \in P\}$
2. **repeat**
 - 2.1. $\text{PREV} := \text{NULL}$
 - 2.2. **for** each variable $A \in V$ **do**
 - if** there is an A rule $A \rightarrow w$ and $w \in \text{PREV}^*$, **then**

$$\text{NULL} := \text{NULL} \cup \{A\}$$
- until** $\text{NULL} = \text{PREV}$

The set NULL is initialized with the variables that derive the null string in one rule application. A variable A is added to NULL if there is an A rule whose right-hand side consists entirely of variables that have previously been determined to be nullable. The algorithm halts when an iteration fails to find a new nullable variable. The repeat-until loop must terminate since the number of variables is finite. The definition of nullable, based on the notion of derivability, is recursive. Thus, induction may be used to show that the set NULL contains exactly the nullable variables of G at the termination of the computation.

Lemma 4.2.2

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. Algorithm 4.2.1 generates the set of nullable variables of G .

Proof. Induction on the number of iterations of the algorithm is used to show that every variable in NULL derives the null string. If A is added to NULL in step 1, then G contains the rule $A \rightarrow \lambda$, and the derivation is obvious.

Assume that all the variables in NULL after n iterations are nullable. We must prove that any variable added in iteration $n + 1$ is nullable. If A is such a variable, then there is a rule

$$A \rightarrow A_1 A_2 \dots A_k$$

with each A_i in PREV at the $n + 1$ st iteration. By the inductive hypothesis, $A_i \xrightarrow{*} \lambda$ for $i = 1, 2, \dots, k$. These derivations can be used to construct the derivation

$$\begin{aligned} A &\Rightarrow A_1 A_2 \dots A_k \\ &\xrightarrow{*} A_2 \dots A_k \\ &\xrightarrow{*} A_3 \dots A_k \\ &\vdots \\ &\xrightarrow{*} A_k \\ &\xrightarrow{*} \lambda, \end{aligned}$$

exhibiting the nullability of A .

Now we show that every nullable variable is eventually added to NULL. If n is the length of the minimal derivation of the null string from the variable A , then A is added to the set NULL on or before iteration n of the algorithm. The proof is by induction on the length of the derivation of the null string from the variable A .

If $A \xrightarrow{1} \lambda$, then A is added to NULL in step 1. Suppose that all variables whose minimal derivations of the null string have length n or less are added to NULL on or before iteration n . Let A be a variable that derives the null string by a derivation of length $n + 1$. The derivation can be written

$$\begin{aligned} A &\Rightarrow A_1 A_2 \dots A_k \\ &\xrightarrow{n} \lambda. \end{aligned}$$

Each of the variables A_i is nullable with minimal derivations of length n or less. By the inductive hypothesis, each A_i is in NULL prior to iteration $n + 1$. Let $m \leq n$ be the iteration in which all of the A_i 's first appear in NULL. On iteration $m + 1$ the rule

$$A \rightarrow A_1 A_2 \dots A_k$$

causes A to be added to NULL. ■

The language generated by a grammar contains the null string only if it can be derived from the start symbol of the grammar, that is, if the start symbol is nullable. Thus Algorithm 4.2.1 provides a decision procedure for determining whether the null string is in the language of a grammar.

Example 4.2.1

The set of nullable variables of the grammar

$$\begin{aligned} G: S &\rightarrow ACA \\ A &\rightarrow aAa \mid B \mid C \\ B &\rightarrow bB \mid b \\ C &\rightarrow cC \mid \lambda \end{aligned}$$

is constructed using Algorithm 4.2.1. The action of the algorithm is traced by giving the contents of the sets NULL and PREV after each iteration of the repeat-until loop. Iteration zero specifies the composition of NULL prior to entering the loop.

Iteration	NULL	PREV
0	{C}	
1	{A, C}	{C}
2	{S, A, C}	{A, C}
3	{S, A, C}	{S, A, C}

The algorithm halts after three iterations. The nullable variables of G are S , A , and C . Since the start symbol is nullable, the null string is in $L(G)$. \square

A grammar with λ -rules is not noncontracting. To build an equivalent noncontracting grammar, rules must be added to generate the strings whose derivations in the original grammar require the application of λ -rules. There are two distinct roles that a nullable variable B can play in a derivation that is initiated by the application of the rule $A \rightarrow uBv$; it can derive a nonnull terminal string or it can derive the null string. In the latter case, the derivation has the form

$$\begin{aligned} A &\Rightarrow uBv \\ &\stackrel{*}{\Rightarrow} uv \\ &\stackrel{*}{\Rightarrow} w. \end{aligned}$$

The string w can be derived without λ -rules by augmenting the grammar with the rule $A \rightarrow uv$. Lemma 4.1.2 ensures that the addition of this rule does not affect the language of the grammar.

The rule $A \rightarrow BABa$ requires three additional rules to construct derivations without λ -rules. If both of the B 's derive the null string, the rule $A \rightarrow Aa$ can be used in a noncontracting derivation. To account for all possible derivations of the null string from the two instances of the variable B , a noncontracting grammar requires the four rules

$$\begin{aligned} A &\rightarrow BABa \\ A &\rightarrow ABa \\ A &\rightarrow BAa \\ A &\rightarrow Aa \end{aligned}$$

to produce all the strings derivable from the rule $A \rightarrow BABa$. Since the right-hand side of each of these rules is derivable from A , their addition to the rules of the grammar does not alter the language.

The previous technique constructs rules that can be added to a grammar G to derive strings in $L(G)$ without the use of λ -rules. This process is used to construct a grammar without λ -rules that is equivalent to G . If $L(G)$ contains the null string, there is no equivalent noncontracting grammar. All variables occurring in the derivation $S \stackrel{*}{\Rightarrow} \lambda$ must eventually disappear. To handle this special case, the rule $S \rightarrow \lambda$ is allowed in the new grammar, but all other λ -rules are replaced. The derivations in the resulting grammar, with the exception of $S \Rightarrow \lambda$, are noncontracting. A grammar satisfying these conditions is called **essentially noncontracting**.

When constructing equivalent grammars, a subscript is used to indicate the restriction being imposed on the rules. The grammar obtained from G by removing λ -rules is denoted G_L .

Theorem 4.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar.

- i) $L(G_L) = L(G)$
- ii) S_L is nullable
- iii) G_L has no λ -rules

Proof. The proof is similar to the proof of Lemma 4.1.1. The set P_L of rules in G_L is defined as follows:

1. For each

where $A \in V$ and $w \in \Sigma^*$, if $A \Rightarrow w$ in G , then add $A \Rightarrow w$ to P_L .

to P_L .

2. Delete all

Step 1 generates all the strings in $L(G)$ that have at least one occurrence of a nullable variable. Step 2 removes all the λ -rules other than $S \rightarrow \lambda$. The resulting grammar is called G_L .

The opposite direction of the theorem is also established. We show that if G_L is a noncontracting grammar, then there exists a grammar G such that $L(G) = L(G_L)$. Let $w \in \Sigma^+$. Then $w \in L(G_L)$ if and only if $w \in L(G)$.

G. If $n = 1$, the result is trivial.

Assume that the result is true for $n - 1$. We show that it is also true for n . Let G be a grammar with n nullable variables. We can assume that $S \rightarrow \lambda$ is a rule in G . The grammar G' is obtained from G by removing this rule. The grammar G' has $n - 1$ nullable variables. By the induction hypothesis, $L(G') = L(G'_L)$. We show that $L(G) = L(G_L)$.

where $A_i \in V$ and $w \in \Sigma^*$, if $A_i \Rightarrow w$ in G , then add $A_i \Rightarrow w$ to P_L .

C. Since

□

stracting
original
nullable
 $\rightarrow uBv$;
case, the

the rule
language of

s without
used in a
ring from
rules

nd side of
it does not

G to derive
a grammar
equivalent
eventually
immar, but
exception
essentially

restriction
is denoted

Theorem 4.2.3

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. There is an algorithm to construct a context-free grammar $G_L = (V_L, \Sigma, P_L, S_L)$ that satisfies

- i) $L(G_L) = L(G)$.
- ii) S_L is not a recursive variable.
- iii) G_L has no λ -rules other than $S \rightarrow \lambda$ if $\lambda \in L(G)$.

Proof. The start symbol can be made nonrecursive by the technique presented in Lemma 4.1.1. The set of variables V_L is simply V with a new start symbol added, if necessary. The set P_L of rules of G_L is obtained by a two step process.

1. For each rule $A \rightarrow w$ in P , if w can be written

$$w_1 A_1 w_2 A_2 \dots w_k A_k w_{k+1},$$

where A_1, A_2, \dots, A_k are a subset of the occurrences of the nullable variables in w , then add the rule

$$A \rightarrow w_1 w_2 \dots w_k w_{k+1}$$

to P_L .

2. Delete all λ -rules other than $S \rightarrow \lambda$ from P_L .

Step 1 generates rules of P_L from each rule of the original grammar. A rule with n occurrences of nullable variables in the right-hand side produces 2^n rules. Step 2 deletes all λ -rules other than $S_L \rightarrow \lambda$ from P_L . The rules in P_L are either rules of G or derivable using rules of G . Thus, $L(G_L) \subseteq L(G)$.

The opposite inclusion, that every string in $L(G)$ is also in $L(G_L)$, must also be established. We prove this by showing that every nonnull terminal string derivable from a variable of G is also derivable from that variable in G_L . Let $A \xrightarrow{G} w$ be a derivation in G with $w \in \Sigma^+$. We prove that $A \xrightarrow{G_L} w$ by induction on n , the length of the derivation of w in G . If $n = 1$, then $A \rightarrow w$ is a rule in P and, since $w \neq \lambda$, $A \rightarrow w$ is in P_L .

Assume that terminal strings derivable from any variable of G by n or fewer rule applications can be derived from the variable in G_L . Note that this makes no claim concerning the length of the derivation in G_L . Let $A \xrightarrow[G]{n+1} w$ be a derivation of a terminal string. If we explicitly specify the first rule application, the derivation can be written

$$A \Rightarrow w_1 A_1 w_2 A_2 \dots w_k A_k w_{k+1} \xrightarrow[G]{n} w,$$

where $A_i \in V$ and $w_i \in \Sigma^*$. By Lemma 3.1.5, w can be written

$$w = w_1 p_1 w_2 p_2 \dots w_k p_k w_{k+1},$$

where A_i derives p_i in G with a derivation of length n or less. For each $p_i \in \Sigma^+$, the inductive hypothesis ensures the existence of a derivation $A_i \xrightarrow{G_L} p_i$. If $p_j = \lambda$, the variable A_j is nullable in G . Step 1 generates a rule from

$$A \rightarrow w_1 A_1 w_2 A_2 \dots w_k A_k w_{k+1}$$

in which each of the A_j 's that derives the null string is deleted. A derivation of w in G_L can be constructed by first applying this rule and then deriving each $p_i \in \Sigma^+$ using the derivations provided by the inductive hypothesis. ■

Example 4.2.2

Let G be the grammar given in Example 4.2.1. The nullable variables of G are $\{S, A, C\}$. The equivalent essentially noncontracting grammar G_L is given below.

$G: S \rightarrow ACA$	$G_L: S \rightarrow ACA CA AA AC A C \lambda$
$A \rightarrow aAa B C$	$A \rightarrow aAa aa B C$
$B \rightarrow bB b$	$B \rightarrow bB b$
$C \rightarrow cC \lambda$	$C \rightarrow cC c$

The rule $S \rightarrow A$ is obtained from $S \rightarrow ACA$ in two ways: deleting the leading A and C or the final A and C . All λ -rules, other than $S \rightarrow \lambda$, are discarded. □

Although the grammar G_L is equivalent to G , the derivation of a string in these grammars may be quite different. The simplest example is the derivation of the null string. Six rule applications are required to derive the null string from the start symbol of the grammar G in Example 4.2.2, while the λ -rule in G_L generates it immediately. Leftmost derivations of the string aba are given in each of the grammars.

$G: S \Rightarrow ACA$	$G_L: S \Rightarrow A$
$\Rightarrow aAaCA$	$\Rightarrow aAa$
$\Rightarrow aBaCA$	$\Rightarrow aBa$
$\Rightarrow abaCA$	$\Rightarrow aba$
$\Rightarrow abaA$	
$\Rightarrow abaC$	
$\Rightarrow aba$	

The first rule application of the derivation in G_L generates only variables that eventually derive terminals. Thus, all applications of the λ -rule are avoided.

Example 4.2.2

Let G be the

that generates
grammar obtain

The S rule that
Since S is null

4.3 Elimination

The application
it produce additio
called chain ru
is nothing more

The chain rule A
from A . The ex
rules that direct
rule $A \rightarrow w$ for
be replaced by t

Unfortunately, a
could be repeated
develop a techniq

inductive
rule A_j is

w in G_L
using the

$S, A, C\}$.

Example 4.2.3

Let G be the grammar

$$\begin{aligned} G: S &\rightarrow ABC \\ A &\rightarrow aA \mid \lambda \\ B &\rightarrow bB \mid \lambda \\ C &\rightarrow cC \mid \lambda \end{aligned}$$

that generates $a^*b^*c^*$. The nullable variables of G are S , A , B , and C . The equivalent grammar obtained by removing λ rules is

$$\begin{aligned} G_L: S &\rightarrow ABC \mid AB \mid BC \mid AC \mid A \mid B \mid C \mid \lambda \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b \\ C &\rightarrow cC \mid c. \end{aligned}$$

The S rule that initiates a derivation determines which symbols occur in the derived string. Since S is nullable, the rule $S \rightarrow \lambda$ is added to the grammar. \square

A and C or

\square

g in these
null string.
bol of the
. Leftmost

4.3 Elimination of Chain Rules

The application of a rule $A \rightarrow B$ does not increase the length of the derived string, nor does it produce additional terminal symbols; it simply renames a variable. Rules of this form are called **chain rules**. The idea behind the removal of chain rules is realizing that a chain rule is nothing more than a renaming procedure. Consider the rules

$$\begin{aligned} A &\rightarrow aA \mid a \mid B \\ B &\rightarrow bB \mid b \mid C. \end{aligned}$$

The chain rule $A \rightarrow B$ indicates that any string derivable from the variable B is also derivable from A . The extra step, the application of the chain rule, can be eliminated by adding A rules that directly generate the same strings as B . This can be accomplished by adding a rule $A \rightarrow w$ for each rule $B \rightarrow w$ and deleting the chain rule. The chain rule $A \rightarrow B$ can be replaced by three A rules yielding the equivalent rules

$$\begin{aligned} A &\rightarrow aA \mid a \mid bB \mid b \mid C \\ B &\rightarrow bB \mid b \mid C. \end{aligned}$$

Unfortunately, another chain rule was created by this replacement. The preceding procedure could be repeated to remove the new chain rule. Rather than repeating the process, we will develop a technique to remove all chain rules at one time.

eventually

A derivation $A \xrightarrow{*} C$ consisting solely of chain rules is called a **chain**. Algorithm 4.3.1 generates all variables that can be derived by chains from a variable A in an essentially noncontracting grammar. This set is denoted $\text{CHAIN}(A)$. The set NEW contains the variables that were added to $\text{CHAIN}(A)$ on the previous iteration.

Algorithm 4.3.1
Construction of the Set $\text{CHAIN}(A)$

input: essentially noncontracting context-free grammar $G = (V, \Sigma, P, S)$

1. $\text{CHAIN}(A) := \{A\}$
 2. $\text{PREV} := \emptyset$
 3. **repeat**
 - 3.1. $\text{NEW} := \text{CHAIN}(A) - \text{PREV}$
 - 3.2. $\text{PREV} := \text{CHAIN}(A)$
 - 3.3. **for** each variable $B \in \text{NEW}$ **do**
for each rule $B \rightarrow C$ **do**
 $\quad \text{CHAIN}(A) := \text{CHAIN}(A) \cup \{C\}$
 - until** $\text{CHAIN}(A) = \text{PREV}$
-

Algorithm 4.3.1 is fundamentally different from the algorithm that generates the nullable variables. The strategy for finding nullable variables begins by initializing the set with the variables that generate the null string with one rule application. The rules are then applied backward; if the right-hand side of a rule consists entirely of variables in NULL , then the left-hand side is added to the set being built.

The generation of $\text{CHAIN}(A)$ follows a top-down approach. The repeat-until loop iteratively constructs all variables derivable from A using chain rules. Each iteration represents an additional rule application to the previously discovered chains. The proof that Algorithm 4.3.1 generates $\text{CHAIN}(A)$ is left as an exercise.

Lemma 4.3.2

Let $G = (V, \Sigma, P, S)$ be an essentially noncontracting context-free grammar. Algorithm 4.3.1 generates the set of variables derivable from A using only chain rules.

The variables in $\text{CHAIN}(A)$ determine the substitutions that must be made to remove the A chain rules. The grammar obtained by deleting the chain rules from G is denoted G_C .

Theorem 4.3.3

Let $G = (V, \Sigma, P, S)$ be an essentially noncontracting context-free grammar. There is an algorithm to construct a context-free grammar G_C that satisfies

- i) $L(G_C) = L(G)$.
- ii) G_C is essentially noncontracting and has no chain rules.

im 4.3.1
illy non-
variables

Proof. The A rules of G_C are constructed from the set $\text{CHAIN}(A)$ and the rules of G . The rule $A \rightarrow w$ is in P_C if there is a variable B and a string w that satisfy

- i) $B \in \text{CHAIN}(A)$.
- ii) $B \rightarrow w \in P$.
- iii) $w \notin V$.

Condition (iii) ensures that P_C does not contain chain rules. The variables, alphabet, and start symbol of G_C are the same as those of G .

By Lemma 4.1.2, every string derivable in G_C is also derivable in G . Consequently, $L(G_C) \subseteq L(G)$. Now let $w \in L(G)$ and $A \xrightarrow[G]{*} B$ be a maximal sequence of chain rules used in the derivation of w . The derivation of w has the form

$$S \xrightarrow[G]{*} uAv \xrightarrow[G]{*} uBv \xrightarrow[G]{*} upv \xrightarrow[G]{*} w,$$

where $B \rightarrow p$ is a rule, but not a chain rule, in G . The rule $A \rightarrow p$ can be used to replace the sequence of chain rules in the derivation. This technique can be repeated to remove all applications of chain rules, producing a derivation of w in G_C . ■

Example 4.3.1

The grammar G_C is constructed from the grammar G_L in Example 4.2.2. Since G_L is essentially noncontracting, Algorithm 4.3.1 generates the variables derivable using chain rules. The computations construct the sets

$$\begin{aligned}\text{CHAIN}(S) &= \{S, A, C, B\} \\ \text{CHAIN}(A) &= \{A, B, C\} \\ \text{CHAIN}(B) &= \{B\} \\ \text{CHAIN}(C) &= \{C\}.\end{aligned}$$

These sets are used to generate the rules of G_C .

$$\begin{aligned}P_C: S &\rightarrow ACA \mid CA \mid AA \mid AC \mid aAa \mid aa \mid bB \mid b \mid cC \mid c \mid \lambda \\ A &\rightarrow aAa \mid aa \mid bB \mid b \mid cC \mid c \\ B &\rightarrow bB \mid b \\ C &\rightarrow cC \mid c\end{aligned}$$

□

The removal of chain rules increases the number of rules in the grammar but reduces the length of derivations. This is the same trade-off that accompanied the construction of an essentially noncontracting grammar. The restrictions require additional rules to generate the language but simplify the derivations.

Eliminating chain rules from an essentially noncontracting grammar preserves the noncontracting property. Let $A \rightarrow w$ be a rule created by the removal of chain rules. This

implies that there is a rule $B \rightarrow w$ for some variable $B \in \text{CHAIN}(A)$. Since the original grammar was essentially noncontracting, the only λ -rule is $S \rightarrow \lambda$. The start symbol, being nonrecursive, is not a member of $\text{CHAIN}(A)$ for any $A \neq S$. It follows that no additional λ -rules are produced in the construction of P_C .

Each rule in an essentially noncontracting grammar without chain rules has one of the following forms:

- i) $S \rightarrow \lambda$,
- ii) $A \rightarrow a$, or
- iii) $A \rightarrow w$,

where $w \in (V \cup \Sigma)^*$ is of length at least two. The rule $S \rightarrow \lambda$ is used only in the derivation of the null string. The application of any other rule adds a terminal to the derived string or increases the length of the string.

4.4 Useless Symbols

Grammars are designed to generate languages, and variables define the structure of the sentential forms during the string-generation process. Ideally, every variable in a grammar should contribute to the generation of strings of the language. The construction of large grammars, making modifications to existing grammars, or sloppiness may produce variables that do not occur in derivations that generate terminal strings. Consider the grammar

$$\begin{aligned} G: S &\rightarrow AC \mid BS \mid B \\ A &\rightarrow aA \mid aF \\ B &\rightarrow CF \mid b \\ C &\rightarrow cC \mid D \\ D &\rightarrow aD \mid BD \mid C \\ E &\rightarrow aA \mid BSA \\ F &\rightarrow bB \mid b. \end{aligned}$$

What is $L(G)$? Are there variables that cannot possibly occur in the generation of terminal strings, and if so, why? Try to convince yourself that $L(G) = b^+$. To begin the process of identifying and removing useless symbols, we make the following definition.

Definition 4.4.1

Let G be a context-free grammar. A symbol $x \in (V \cup \Sigma)$ is **useful** if there is a derivation

$$S \xrightarrow[G]{*} uxv \xrightarrow[G]{*} w,$$

where $u, v \in (V \cup \Sigma)^*$ and $w \in \Sigma^*$. A symbol that is not useful is said to be **useless**.

A terminal is useful if it occurs in a string in the language of G . For a variable to be useful, two conditions must be satisfied. The variable must occur in a sentential form of the grammar; that is, it must occur in a string derivable from S . Moreover, every symbol occurring in the sentential form must be capable of deriving a terminal string (the null string is considered to be a terminal string). A two-part procedure to eliminate useless variables is presented. Each construction establishes one of the requirements for the variables to be useful.

Algorithm 4.4.2 builds a set TERM consisting of the variables that derive terminal strings. The strategy used in the algorithm is similar to that used to determine the set of nullable variables of a grammar. The proof that Algorithm 4.4.2 generates the desired set follows the strategy employed by the proof of Lemma 4.2.2 and is left as an exercise.

Algorithm 4.4.2
Construction of the Set of Variables That Derive Terminal Strings

input: context-free grammar $G = (V, \Sigma, P, S)$

1. $\text{TERM} := \{A \mid \text{there is a rule } A \rightarrow w \in P \text{ with } w \in \Sigma^*\}$
 2. **repeat**
 - 2.1. $\text{PREV} := \text{TERM}$
 - 2.2. **for each variable** $A \in V$ **do**
 - if** there is an A rule $A \rightarrow w$ and $w \in (\text{PREV} \cup \Sigma)^*$ **then**
 $\text{TERM} := \text{TERM} \cup \{A\}$
 - until** $\text{PREV} = \text{TERM}$
-

Upon termination of the algorithm, TERM contains the variables of G that generate terminal strings. Variables not in TERM are useless; they cannot contribute to the generation of strings in $L(G)$. This observation provides the motivation for the construction of a grammar G_T that is equivalent to G and contains only variables that derive terminal strings.

Theorem 4.4.3

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. There is an algorithm to construct a context-free grammar $G_T = (V_T, \Sigma_T, P_T, S)$ that satisfies

- i) $L(G_T) = L(G)$.
- ii) Every variable in G_T derives a terminal string in G_T .

Proof. P_T is obtained by deleting all rules containing variables of G that do not derive terminal strings, that is, all rules containing variables in $V - \text{TERM}$. The components of G_T are

$$V_T = \text{TERM},$$

$$P_T = \{A \rightarrow w \mid A \rightarrow w \text{ is a rule in } P, A \in \text{TERM}, \text{ and } w \in (\text{TERM} \cup \Sigma)^*\}, \text{ and}$$

$$\Sigma_T = \{a \in \Sigma \mid a \text{ occurs in the right-hand side of a rule in } P_T\}.$$

The alphabet Σ_T consists of all the terminals occurring in the rules in P_T .

We must show that $L(G_T) = L(G)$. Since $P_T \subseteq P$, every derivation in G_T is also a derivation in G and $L(G_T) \subseteq L(G)$. To establish the opposite inclusion, we must show that removing rules that contain variables in $V - \text{TERM}$ has no effect on the set of terminal strings generated. Let $S \xrightarrow[G]{*} w$ be a derivation of a string $w \in L(G)$. This is also a derivation in G_T . If not, a variable from $V - \text{TERM}$ must occur in an intermediate step in the derivation. A derivation from a sentential form containing a variable in $V - \text{TERM}$ cannot produce a terminal string. Consequently, all the rules in the derivation are in P_T and $w \in L(G_T)$. ■

Example 4.4.1

The grammar G_T is constructed for the grammar G introduced at the beginning of this section.

$$\begin{aligned} G: S &\rightarrow AC \mid BS \mid B \\ A &\rightarrow aA \mid aF \\ B &\rightarrow CF \mid b \\ C &\rightarrow cC \mid D \\ D &\rightarrow aD \mid BD \mid C \\ E &\rightarrow aA \mid BSA \\ F &\rightarrow bB \mid b \end{aligned}$$

Algorithm 4.4.2 is used to determine the variables of G that derive terminal strings.

Iteration	TERM	PREV
0	{B, F}	
1	{B, F, A, S}	{B, F}
2	{B, F, A, S, E}	{B, F, A, S}
3	{B, F, A, S, E}	{B, F, A, S, E}

Using the set TERM to build G_T produces

$$\begin{aligned} V_T &= \{S, A, B, E, F\} \\ \Sigma_T &= \{a, b\} \\ P_T: S &\rightarrow BS \mid B \\ A &\rightarrow aA \mid aF \\ B &\rightarrow b \\ E &\rightarrow aA \mid BSA \\ F &\rightarrow bB \mid b. \end{aligned}$$

also a
ow that
terminal
rivation.
ivation.
duce a
t). ■

The indirectly recursive derivation produced by an occurrence of the variables C or D , which can never be exited once entered, is discovered by the algorithm. All rules containing these variables are deleted. \square

The construction of G_T completes the first step in the removal of useless variables. All variables in G_T derive terminal strings. We must now remove the variables that do not occur in sentential forms of the grammar. A set REACH is built that contains all variables derivable from S .

Algorithm 4.4.4

Construction of the Set of Reachable Variables

input: context-free grammar $G = (V, \Sigma, P, S)$

1. REACH := $\{S\}$
 2. PREV := \emptyset
 3. repeat
 - 3.1. NEW := REACH - PREV
 - 3.2. PREV := REACH
 - 3.3. for each variable $A \in \text{NEW}$ do
 - for each rule $A \rightarrow w$ do add all variables in w to REACH
 - until REACH = PREV
-

Algorithm 4.4.4, like Algorithm 4.3.1, uses a top-down approach to construct the desired set of variables. The set REACH is initialized to S . Variables are added to REACH as they are discovered in derivations from S .

Lemma 4.4.5

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. Algorithm 4.4.4 generates the set of variables reachable from S .

Proof. First we show that every variable in REACH is derivable from S . The proof is by induction on the number of iterations of the algorithm.

The set REACH is initialized to S , which is clearly reachable. Assume that all variables in the set REACH after n iterations are reachable from S . Let B be a variable added to REACH in iteration $n + 1$. Then there is a rule $A \rightarrow uBv$ where A is in REACH after n iterations. By induction, there is a derivation $S \xrightarrow{*} xAy$. Extending this derivation with the application of $A \rightarrow uBv$ establishes the reachability of B .

We now prove that every variable reachable from S is eventually added to the set REACH. If $S \xrightarrow{*} uAv$, then A is added to REACH on or before iteration n . The proof is by induction on the length of the derivation from S .

The start symbol, the only variable reachable by a derivation of length zero, is added to REACH at step 1 of the algorithm. Assume that each variable reachable by a derivation of length n or less is inserted into REACH on or before iteration n .

Let $S \xrightarrow{n} xAy \Rightarrow xuBvy$ be a derivation in G where the $(n+1)$ st rule applied is $A \rightarrow uBv$. By the inductive hypothesis, A has been added to REACH by iteration n . B is added to REACH on the succeeding iteration. ■

Theorem 4.4.6

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. There is an algorithm to construct a context-free grammar G_U that satisfies

- i) $L(G_U) = L(G)$.
- ii) G_U has no useless symbols.

Proof. The removal of useless symbols begins by building G_T from G . Algorithm 4.4.4 is used to generate the variables of G_T that are reachable from the start symbol. All rules of G_T that reference variables not reachable from S are deleted to obtain G_U , defined by

$$V_U = \text{REACH},$$

$$P_U = \{A \rightarrow w \mid A \rightarrow w \in P_T, A \in \text{REACH}, \text{ and } w \in (\text{REACH} \cup \Sigma)^*\}, \text{ and}$$

$$\Sigma_U = \{a \in \Sigma \mid a \text{ occurs in the right-hand side of a rule in } P_U\}.$$

To establish the equality of $L(G_U)$ and $L(G_T)$, it is sufficient to show that every string derivable in G_T is also derivable in G_U . Let w be an element of $L(G_T)$. Every variable occurring in the derivation of w is reachable and each rule is in P_U . ■

Example 4.4.2

The grammar G_U is constructed from the grammar G_T in Example 4.4.1. The set of reachable variables of G_T is obtained using Algorithm 4.4.4.

	Iteration	REACH	PREV	NEW
0		{S}		∅
1		{S, B}	{S}	{S}
2		{S, B}	{S, B}	{B}

Removing all references to the variables A , E , and F produces the grammar

$$G_U: S \rightarrow BS \mid B$$

$$B \rightarrow b.$$

The grammar G_U is equivalent to the grammar G given at the beginning of the section. Clearly, the language of these grammars is b^+ . □

Removing useless symbols consists of the two-part process outlined in Theorem 4.4.6. The first step is the removal of variables that do not generate terminal strings. The resulting

applied is
ration n . B

■

construct a

gument

thm 4.4.4 is
All rules of
ned by

, and

every string
ery variable

■

of reachable

f the section.

□

neorem 4.4.6.
The resulting

grammar is then purged of variables that are not derivable from the start symbol. Applying these procedures in reverse order may not remove all the useless symbols, as shown in the next example.

Example 4.4.3

Let G be the grammar

$$\begin{aligned} G: S &\rightarrow a \mid AB \\ A &\rightarrow b. \end{aligned}$$

The necessity of applying the transformations in the specified order is exhibited by applying the processes in both orders and comparing the results.

Remove variables that do not generate terminal strings:

$$\begin{aligned} S &\rightarrow a \\ A &\rightarrow b \end{aligned}$$

Remove unreachable symbols:

$$S \rightarrow a$$

Remove unreachable symbols:

$$\begin{aligned} S &\rightarrow a \mid AB \\ A &\rightarrow b \end{aligned}$$

Remove variables that do not generate terminal strings:

$$\begin{aligned} S &\rightarrow a \\ A &\rightarrow b \end{aligned}$$

The variable A and terminal b are useless, but they remain in the grammar obtained by reversing the order of the transformations. □

The transformation of grammars to normal forms consists of a sequence of algorithmic steps, each of which preserves the previous ones. The removal of useless symbols will not undo any of the restrictions obtained by the construction of G_L or G_C . These transformations only remove rules; they do not alter any other feature of the grammar. However, useless symbols may be created by the process of transforming a grammar to an equivalent non-contracting grammar. This phenomenon is illustrated by the transformations in Exercises 8 and 17.

4.5 Chomsky Normal Form

A normal form is described by a set of conditions that each rule in the grammar must satisfy. The Chomsky normal form places restrictions on the length and the composition of the right-hand side of a rule.

Definition 4.5.1

A context-free grammar $G = (V, \Sigma, P, S)$ is in **Chomsky normal form** if each rule has one of the following forms:

- i) $A \rightarrow BC$,
- ii) $A \rightarrow a$, or
- iii) $S \rightarrow \lambda$,

where $B, C \in V - \{S\}$.

Since the maximal number of symbols on the right-hand side of a rule is two, the derivation tree associated with a derivation in a Chomsky normal form grammar is a binary tree. The application of a rule $A \rightarrow BC$ produces a node with children B and C . All other rule applications produce a node with a single child. The representation of the derivations as binary derivation trees will be used in Chapter 7 to establish repetition properties of strings in context-free languages. In the next section, we will use the ability to transform a grammar G into Chomsky normal form to obtain a decision procedure for membership of a string in $L(G)$.

The conversion of a grammar to Chomsky normal form continues the sequence of modifications presented in the previous sections. We assume that the grammar G to be transformed has a nonrecursive start symbol, no λ -rules other than $S \rightarrow \lambda$, no chain rules, and no useless symbols.

Theorem 4.5.2

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. There is an algorithm to construct a grammar $G' = (V', \Sigma, P', S')$ in Chomsky normal form that is equivalent to G .

Proof. After the preceding transformations, a rule has the form $S \rightarrow \lambda$, $A \rightarrow a$, or $A \rightarrow w$, where $w \in ((V \cup \Sigma) - \{S\})^*$ and $\text{length}(w) > 1$. The set P' of rules of G' is built from the rules of G .

The only rule of G whose right-hand side has length zero is $S \rightarrow \lambda$. Since G does not contain chain rules, the right-hand side of a rule $A \rightarrow w$ is a single terminal whenever the length of w is one. In either case, the rules already satisfy the conditions of Chomsky normal form and are added to P' .

Let $A \rightarrow w$ be a rule with $\text{length}(w)$ greater than one. The string w may contain both variables and terminals. The first step is to remove the terminals from the right-hand side of all such rules. This is accomplished by adding new variables and rules that simply rename each terminal by a variable. For example, the rule

$$A \rightarrow bDcF$$

can be replaced by the three rules

$$A \rightarrow B'DC'F$$

$$B' \rightarrow b$$

$$C' \rightarrow c.$$

After transforming each rule whose right-hand side has length two or more in this manner, the right-hand side of a rule consists of the null string, a terminal, or a string of variables. Rules of the latter form must be broken into a sequence of rules, each of whose right-hand side consists of two variables. The sequential application of these rules should generate the right-hand side of the original rule. Continuing with the previous example, we replace the A rule by the rules

$$\begin{aligned} A &\rightarrow B'T_1 \\ T_1 &\rightarrow DT_2 \\ T_2 &\rightarrow C'F. \end{aligned}$$

The variables T_1 and T_2 are introduced to link the sequence of rules. Rewriting each rule whose right-hand side has length greater than two as a sequence of rules completes the transformation to Chomsky normal form. ■

Example 4.5.1

Let G be the grammar

$$\begin{aligned} S &\rightarrow aABC \mid a \\ A &\rightarrow aA \mid a \\ B &\rightarrow bcB \mid bc \\ C &\rightarrow cC \mid c. \end{aligned}$$

This grammar already satisfies the conditions placed on the start symbol and λ -rules and does not contain chain rules or useless symbols. The equivalent Chomsky normal form grammar is constructed by transforming each rule whose right-hand side has length greater than two.

$$\begin{aligned} G': S &\rightarrow A'T_1 \mid a \\ A' &\rightarrow a \\ T_1 &\rightarrow AT_2 \\ T_2 &\rightarrow BC \\ A &\rightarrow A'A \mid a \\ B &\rightarrow B'T_3 \mid B'C' \\ T_3 &\rightarrow C'B \\ C &\rightarrow C'C \mid c \\ B' &\rightarrow b \\ C' &\rightarrow c \end{aligned}$$

□

Example 4.5.2

The rules

$$X \rightarrow aXb \mid ab$$

generate the strings $\{a^i b^i \mid i \geq 1\}$. Adding a start symbol S , the rule $S \rightarrow X$, and removing chain rules produces the grammar

$$S \rightarrow aXb \mid ab$$

$$X \rightarrow aXb \mid ab.$$

The Chomsky normal form

$$S \rightarrow AT \mid AB$$

$$T \rightarrow XB$$

$$X \rightarrow AT \mid AB$$

$$A \rightarrow a$$

$$B \rightarrow b$$

is obtained by adding the rules $A \rightarrow a$ and $B \rightarrow b$ that provide aliases for the terminals and by reducing the length of the right-hand sides of the S and X rules. \square

4.6 The CYK Algorithm

Given a context-free grammar G and a string u , is u in $L(G)$? This question is called the *membership problem* for context-free grammars. Using the structure of the rules in a Chomsky normal form grammar, J. Cocke, D. Younger, and T. Kasami independently developed an algorithm to answer this question. The CYK algorithm employs a bottom-up approach to determine the derivability of a string.

Let $u = x_1 x_2 \dots x_n$ be a string to be tested for membership and let $x_{i,j}$ denote the substring $x_i \dots x_j$ of u . Note that the substring $x_{i,i}$ is simply x_i , the i th symbol in u . The strategy of the CYK algorithm is

- *Step 1:* For each substring $x_{i,i}$ of u with length one, find the set $X_{i,i}$ of all variables A with a rule $A \rightarrow x_{i,i}$.
- *Step 2:* For each substring $x_{i,i+1}$ of u with length two, find the set $X_{i,i+1}$ of all variables that initiate derivations $A \xrightarrow{*} x_{i,i+1}$.
- *Step 3:* For each substring $x_{i,i+2}$ of u with length three, find the set $X_{i,i+2}$ of all variables that initiate derivations $A \xrightarrow{*} x_{i,i+2}$.
- ⋮
- *Step $n - 1$:* For the substrings $x_{1,n-1}, x_{2,n}$ of u with length $n - 1$, find the sets $X_{1,n-1}$ and $X_{2,n}$ of all variables that initiate derivations $A \xrightarrow{*} x_{1,n-1}$ and $A \xrightarrow{*} x_{2,n}$, respectively.

- *Step n:* For the string $x_{1,n} = u$ of length n , find the set $X_{1,n}$ of all variables that initiate derivations $A \xrightarrow{*} x_{1,n}$.

If the start symbol S is in $X_{1,n}$, then u is in the language of the grammar. The generation of the sets $X_{i,j}$ uses a problem solving technique known as dynamic programming. The important feature of dynamic programming is that all the information needed to compute a set $X_{i,j}$ at step t has already been obtained in steps 1 through $t - 1$.

Let's see why this property is true for derivations using Chomsky normal form grammars. Building the sets in step 1 is straightforward; $A \in X_{i,i}$ if $A \rightarrow x_i$ is a rule of the grammar.

For step 2, a derivation of the substring $x_{i,i+1}$ has the form

$$\begin{aligned} A &\Rightarrow BC \\ &\Rightarrow x_i C \\ &\Rightarrow x_i x_{i+1}. \end{aligned}$$

Since B derives x_i and C derives x_{i+1} , these variables will be in $X_{i,i}$ and $X_{i+1,i+1}$. A variable A is added to $X_{i,i+1}$ when there is a rule $A \rightarrow BC$ with $B \in X_{i,i}$ and $C \in X_{i+1,i+1}$.

Now we consider the generation of the set $X_{i,i+t}$ in step t of the algorithm. We wish to find all variables that derive the substring $x_{i,i+t}$. The first rule application of such a derivation produces two variables, call them B and C . This is followed by derivations beginning with B and C that produce $x_{i,i+t}$. Thus the derivation has the form

$$\begin{aligned} A &\Rightarrow BC \\ &\Rightarrow x_{i,k} C \\ &\Rightarrow x_{i,k} x_{k+1,i+t}, \end{aligned}$$

where B generates $x_{i,k}$ and C generates $x_{k+1,i+t}$ for some k between i and $t - 1$. Consequently, A derives $x_{i,i+t}$ only if there is a rule $A \rightarrow BC$ and a number k between i and $t - 1$ such that $B \in X_{i,k}$ and $C \in X_{k+1,i+t}$. All of the sets that need to be examined in checking this condition are produced prior to step t .

The sets $X_{i,j}$ may be represented as the upper triangular portion of an $n \times n$ matrix.

	1	2	3	...	$n - 1$	n
1	$X_{1,1}$	$X_{1,2}$	$X_{1,3}$...	$X_{1,n-1}$	$X_{1,n}$
2		$X_{2,2}$	$X_{2,3}$...	$X_{2,n-1}$	$X_{2,n}$
3			$X_{3,3}$...	$X_{3,n-1}$	$X_{3,n}$
:				...		:
$n - 1$					$X_{n-1,n-1}$	$X_{n-1,n}$
n						$X_{n,n}$

The CYK algorithm constructs the entries in a diagonal by diagonal manner starting with the main diagonal and culminating in the upper right corner with $X_{1,n}$.

We illustrate the CYK algorithm using the grammar from Example 4.5.2 that generates $\{a^i b^i \mid i \geq 1\}$ and the string $aaabbb$. Table 4.1 traces the steps of the algorithm and the result of the computation is given in the table

	1	2	3	4	5	6
1	{A}	\emptyset	\emptyset	\emptyset	\emptyset	$\{S, X\}$
2		{A}	\emptyset	\emptyset	$\{S, X\}$	$\{T\}$
3			{A}	$\{S, X\}$	$\{T\}$	\emptyset
4				{B}	\emptyset	\emptyset
5					{B}	\emptyset
6						{B}

The sets along the diagonal are obtained from the rules $A \rightarrow a$ and $B \rightarrow b$. Step 2 generates the entries directly above the diagonal. The construction of a set $X_{i,i+1}$ need only consider the substrings $x_{i,i}$ and $x_{i+1,i+1}$. For example, a variable is in $X_{1,2}$ if there are variables in $X_{1,1} = \{A\}$ and $X_{2,2} = \{A\}$ that make up the right-hand side of a rule. Since AA is not the right-hand side of a rule, $X_{1,2} = \emptyset$. The set $X_{3,4}$ is generated from $X_{3,3} = \{A\}$ and $X_{4,4} = \{B\}$. The string AB is the right-hand side of $S \rightarrow AB$ and $X \rightarrow AB$. Consequently, S and X are in $X_{3,4}$.

At step t , there are $t - 1$ separate decompositions of a substring $x_{i,i+1}$ that must be checked. The set $X_{i,j}$, given in the rightmost column of Table 4.1, is the union of variables found examining all $t - 1$ possibilities. For example, computing $X_{3,5}$ needs to consider the two decompositions $x_{3,3}x_{4,5}$ and $x_{3,4}x_{5,5}$ of $x_{3,5}$. The variable T is added to this set since $S \in X_{3,4}$, $B \in X_{5,5}$, and $T \rightarrow SB$ is a rule. The presence of S in the set $X_{1,6}$ indicates that the string $aaabbb$ is in the language of the grammar.

Utilizing the previously outlined steps, the CYK solution to the membership problem is given in Algorithm 4.6.1. The sets along the diagonal are computed in line 2. The variable $step$ indicates the length of the substring being analyzed. In the loop beginning at step 3.1, i indicates the starting position of a substring and k indicates the position of split between the first and second components.

Algorithm 4.6.1 CYK Algorithm

input: context-free grammar $G = (V, \Sigma, P, S)$
string $u = x_1 x_2 \dots x_n \in \Sigma^*$

starting with
at generates
and the result

$\rightarrow b$. Step 2
 $X_{i,i+1}$ need
if there are
ile. Since AA
 $A_3 = \{A\}$ and
Consequently,

that must be
n of variables
o consider the
this set since
indicates that

rship problem
The variable
ng at step 3.1,
split between

1. initialize all $X_{i,j}$ to \emptyset
 2. **for** $i = 1$ to n
 - for each variable A , if there is a rule $A \rightarrow x_i$ then $X_{i,i} := X_{i,i} \cup \{A\}$
 3. **for** $step = 2$ to n
 - 3.1. **for** $i = 1$ to $n - step + 1$
 - 3.1.1. **for** $k = i$ to $i + step - 2$
 - if there are variables $B \in X_{i,k}$, $C \in X_{k+1,i+step-1}$, and a rule $A \rightarrow BC$, then $X_{i,i+step-1} := X_{i,i+step-1} \cup \{A\}$
 4. $u \in L(G)$ if $S \in X_{1,n}$
-

The CYK algorithm, as outlined above, is designed to determine whether a string u is derivable in a Chomsky normal form grammar G . The algorithm can be modified to produce derivations of strings in $L(G)$, that is, to be a parser. This can be accomplished by recording the justifications for the addition of variables into the sets $X_{i,j}$. To demonstrate the approach, we will use the trace of the computation in Table 4.1 to produce the derivation of the string $aaabbb$. The column labeled ‘Sets’ indicates the sets that contain the variables matching the right-hand side of the rule. For example, the variable S is added to $X_{6,6}$ because the occurrence of $A \in X_{1,1}$ and $T \in X_{2,6}$ match the right-hand side of the rule $S \rightarrow AT$. Reversing this construction, the rule $S \rightarrow AT$ is used in the derivation of $aaabbb$.

Derivation	Sets
$S \Rightarrow AT$	$A \in X_{1,1}, T \in X_{2,6}$
$\Rightarrow aT$	$T \in X_{2,6}$
$\Rightarrow aXB$	$X \in X_{2,5}, B \in X_{6,6}$
$\Rightarrow aATB$	$A \in X_{1,1}, T \in X_{2,6}, B \in X_{6,6}$
$\Rightarrow aaTB$	$T \in X_{3,5}, B \in X_{6,6}$
$\Rightarrow aaXBB$	$X \in X_{2,5}, B \in X_{5,5}, B \in X_{6,6}$
$\Rightarrow aaABB$	$A \in X_{1,1}, B \in X_{2,6}, B \in X_{5,5}, B \in X_{6,6}$
$\stackrel{*}{\Rightarrow} aaabbb$	

The applicability of the CYK algorithm as a parser is limited by the computational requirements needed to find a derivation. For an input string of length n , $(n^2 + n)/2$ sets need to be constructed to complete the dynamic programming table. Moreover, each of these sets may require the consideration of multiple decompositions of the associated substring. In Part V of this book we examine grammars and algorithms designed specifically for efficient parsing.

TABLE 4.1 Trace of CYK Algorithm

Step	String $x_{i,j}$	Substrings	$X_{i,k}$	$X_{k+1,j}$	$X_{i,j}$
2	$x_{1,2} = aa$	$x_{1,1}, x_{2,2}$	{A}	{A}	\emptyset
	$x_{2,3} = aa$	$x_{2,2}, x_{3,3}$	{A}	{A}	\emptyset
	$x_{3,4} = ab$	$x_{3,3}, x_{4,4}$	{A}	{B}	{S, X}
	$x_{4,5} = bb$	$x_{4,4}, x_{5,5}$	{B}	{B}	\emptyset
	$x_{5,6} = bb$	$x_{5,5}, x_{6,6}$	{B}	{B}	\emptyset
3	$x_{1,3} = aaa$	$x_{1,1}, x_{2,3}$	{A}	\emptyset	\emptyset
		$x_{1,2}, x_{3,3}$	\emptyset	{A}	\emptyset
	$x_{2,4} = aab$	$x_{2,2}, x_{3,4}$	{A}	{S, X}	\emptyset
		$x_{2,3}, x_{4,4}$	\emptyset	{B}	\emptyset
	$x_{3,5} = abb$	$x_{3,3}, x_{4,5}$	{A}	\emptyset	\emptyset
		$x_{3,4}, x_{5,5}$	{S, X}	{B}	{T}
4	$x_{1,4} = aaab$	$x_{1,1}, x_{2,4}$	{A}	\emptyset	\emptyset
		$x_{1,2}, x_{3,4}$	\emptyset	{S, X}	\emptyset
		$x_{1,3}, x_{4,4}$	\emptyset	{B}	\emptyset
	$x_{2,5} = aabb$	$x_{2,2}, x_{3,5}$	{A}	{T}	{S, X}
		$x_{2,3}, x_{4,5}$	\emptyset	\emptyset	\emptyset
		$x_{2,4}, x_{5,5}$	\emptyset	{B}	\emptyset
5	$x_{3,6} = abbb$	$x_{3,3}, x_{4,6}$	{A}	\emptyset	\emptyset
		$x_{3,4}, x_{5,6}$	{S, X}	\emptyset	\emptyset
		$x_{3,5}, x_{6,6}$	{T}	{B}	\emptyset
	$x_{1,5} = aaabb$	$x_{1,1}, x_{2,5}$	{A}	{S, X}	\emptyset
		$x_{1,2}, x_{3,5}$	\emptyset	{T}	\emptyset
		$x_{1,3}, x_{4,5}$	\emptyset	\emptyset	\emptyset
6	$x_{1,6} = aabbb$	$x_{1,1}, x_{2,6}$	{A}	{T}	{S, X}
		$x_{1,2}, x_{3,6}$	\emptyset	\emptyset	\emptyset
		$x_{1,3}, x_{4,6}$	\emptyset	\emptyset	\emptyset
		$x_{1,4}, x_{5,6}$	\emptyset	\emptyset	\emptyset
		$x_{1,5}, x_{6,6}$	\emptyset	{B}	{T}

4.7 Removal of Direct Left Recursion

In a derivation of an arbitrary context-free grammar, rule applications can generate terminal symbols in any position and in any order in a derivation. For example, derivations in grammar G_1 generate terminals to the right of the variable, while derivations in G_2 generate terminals on both sides.

$$\begin{array}{ll} G_1: S \rightarrow Aa & G_2: S \rightarrow aAb \\ A \rightarrow Aa | b & A \rightarrow aAb | \lambda \end{array}$$

The Greibach normal form adds structure to the generation of the terminals in a derivation. A string is built in a left-to-right manner with one terminal added on each rule application. In a derivation $S \xrightarrow{*} uAv$, where A is the leftmost variable, the string u is called the *terminal prefix* of the sentential form. Our objective is to construct a grammar in which the terminal prefix increases with each rule application.

The grammar G_1 provides an example of rules that do the exact opposite of what is desired. The variable A remains as the leftmost symbol until the derivation terminates with application of the rule $A \rightarrow b$. Consider the derivation of the string $baaa$

$$\begin{aligned} S &\Rightarrow Aa \\ &\Rightarrow Aaa \\ &\Rightarrow Aaaa \\ &\Rightarrow baaa. \end{aligned}$$

Applications of the left-recursive rule $A \rightarrow Aa$ generate a string of a 's but do not increase the length of the terminal prefix. A derivation of this form is called *directly left-recursive*. The prefix grows only when the non-left-recursive rule is applied.

An important component in the transformation to Greibach normal form is the ability to remove left-recursive rules from a grammar. The technique for replacing left-recursive rules is illustrated by the following examples.

- a) $A \rightarrow Aa | b$
- b) $A \rightarrow Aa | Ab | b | c$
- c) $A \rightarrow AB | BA | a$
 $B \rightarrow b | c$

The sets generated by these rules are ba^* , $(b \cup c)(a \cup b)^*$, and $(b \cup c)^*a(b \cup c)^*$, respectively. The left recursion builds a string to the right of the recursive variable. The recursive sequence is terminated by an A rule that is not left-recursive. To build the string in a left-to-right manner, the nonrecursive rule is applied first and the remainder of the string is constructed by right recursion. The following rules generate the same strings as the previous examples without using direct left recursion.

- a) $A \rightarrow bZ | b$
- b) $A \rightarrow bZ | cZ | b | c$
- c) $A \rightarrow BAZ | aZ | BA | a$
 $Z \rightarrow aZ | a$
 $Z \rightarrow aZ | bZ | a | b$
 $Z \rightarrow BZ | B$
 $B \rightarrow b | c$

The rules in (a) generate ba^* with left recursion replaced by right recursion. With these rules, the derivation of $baaa$ increases the length of the terminal prefix with each rule application.

$$\begin{aligned} A &\Rightarrow bZ \\ &\Rightarrow baZ \\ &\Rightarrow baaZ \\ &\Rightarrow baaa \end{aligned}$$

The removal of the direct left recursion requires the addition of a new variable to the grammar. This variable introduces a set of right-recursive rules. Direct right recursion causes the recursive variable to occur as the rightmost symbol in the derived string.

To remove direct left recursion, the A rules are divided into two categories: the left-recursive rules

$$A \rightarrow Au_1 | Au_2 | \dots | Au_j$$

and the rules

$$A \rightarrow v_1 | v_2 | \dots | v_k,$$

in which the first symbol of each v_i is not A . A leftmost derivation from these rules consists of applications of left-recursive rules followed by the application of a rule $A \rightarrow v_i$, which ends the direct left recursion. Using the technique illustrated in the previous examples, we construct new rules that initially generate v_i and then produce the remainder of the string using right recursion.

The A rules

$$A \rightarrow v_1 | \dots | v_k | v_1Z | \dots | v_kZ$$

initially place one of the v_i 's on the left-hand side of the derived string. If the string contains a sequence of u_i 's, they are generated by the Z rules

$$Z \rightarrow u_1Z | \dots | u_jZ | u_1 | \dots | u_j$$

using right recursion.

Example 4.7.1

A set of rules is constructed to generate the same strings as

$$A \rightarrow Aa | Aab | bb | b$$

without using direct left recursion. These rules generate $(b \cup bb)(a \cup ab)^*$. The direct left recursion in derivations using the original rules is terminated by applying $A \rightarrow b$ or $A \rightarrow bb$. To build these strings in a left-to-right manner, we use the A rules

$$A \rightarrow bb | b | bbZ | bZ$$

to generate the leftmost symbols of the string. The Z rules generate $(a \cup ab)^+$ using the right-recursive rules

$$Z \rightarrow aZ | abZ | a | ab. \quad \square$$

Lem
Let C
recur
(V',
Proof
and P
an eq
augme
P usin
T.
A. Th
 u_i 's an

The
while p
and B

exhibit
recursio
indirect

4.8

In the Gr
prefix of
It also er
application

Definition
A context
one of the

- i) $A \rightarrow$
- ii) $A \rightarrow$
- iii) $S \rightarrow$

where $a \in$

se rules,
lication.

le to the
on causes

the left-

s consists
 v_i , which
mples, we
the string

g contains

Lemma 4.7.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar and let $A \in V$ be a directly left-recursive variable in G . There is an algorithm to construct an equivalent grammar $G' = (V', \Sigma, P', S')$ in which A is not directly left-recursive.

Proof. We assume that the start symbol of G is nonrecursive, the only λ -rule is $S \rightarrow \lambda$, and P does not contain the rule $A \rightarrow A$. If this is not the case, G can be transformed to an equivalent grammar satisfying these conditions. The variables of G' are those of G augmented with one additional variable to generate the right-recursive rules. P' is built from P using the technique outlined above.

The new A rules cannot be left-recursive since the first symbol of each of the v_i 's is not A . The Z rules are also not left-recursive. The variable Z does not occur in any one of the u_i 's and the u_i 's are nonnull by the restriction on the A rules of G . ■

This technique can be used repeatedly to remove all occurrences of left-recursive rules while preserving the language of the grammar. However, a derivation using rules $A \rightarrow Bu$ and $B \rightarrow Av$ can generate the sentential forms

$$\begin{aligned} A &\Rightarrow Bu \\ &\Rightarrow Avu \\ &\Rightarrow Buu \\ &\Rightarrow Avuu \\ &\vdots \end{aligned}$$

exhibiting the same lack of growth of the terminal prefix as derivations using direct left recursion. The conversion to Greibach normal form will remove all possible occurrences of indirect left recursion.

4.8 Greibach Normal Form

In the Greibach normal form, the application of every rule adds one symbol to the terminal prefix of the derived string. This ensures that left recursion, direct or indirect, cannot occur. It also ensures that the derivation of a string of length $n > 0$ consists of exactly n rule applications.

Definition 4.8.1

A context-free grammar $G = (V, \Sigma, P, S)$ is in **Greibach normal form** if each rule has one of the following forms:

- i) $A \rightarrow aA_1A_2 \dots A_n$,
- ii) $A \rightarrow a$, or
- iii) $S \rightarrow \lambda$,

where $a \in \Sigma$ and $A_i \in V - \{S\}$ for $i = 1, 2, \dots, n$.

□

The conversion of a Chomsky normal form grammar to Greibach normal form uses two rule transformation techniques: the rule replacement scheme of Lemma 4.1.3 and the transformation that removes left-recursive rules. The procedure begins by ordering the variables of the grammar. The start symbol is assigned the number one; the remaining variables may be numbered in any order. Different numberings change the transformations required to convert the grammar, but any ordering suffices.

The first step of the conversion is to construct a grammar in which every rule has one of the following forms:

- i) $S \rightarrow \lambda$,
- ii) $A \rightarrow aw$, or
- iii) $A \rightarrow Bw$,

where $w \in V^*$ and the number assigned to B in the ordering of the variables is greater than the number of A . The rules are transformed to satisfy condition (iii) according to the order in which the variables are numbered. The conversion of a Chomsky normal form grammar to Greibach normal form is illustrated by tracing the transformation of the rules of the grammar G :

$$\begin{aligned} G: S &\rightarrow AB \mid \lambda \\ A &\rightarrow AB \mid CB \mid a \\ B &\rightarrow AB \mid b \\ C &\rightarrow AC \mid c. \end{aligned}$$

The variables S , A , B , and C are numbered 1, 2, 3, and 4, respectively.

Since the start symbol of a Chomsky normal form grammar is nonrecursive, the S rules already satisfy the three conditions. The process continues by transforming the A rules into a set of rules in which the first symbol on the right-hand side is either a terminal or a variable assigned a number greater than two. The left-recursive rule $A \rightarrow AB$ violates these restrictions. Lemma 4.7.1 can be used to remove the direct left recursion, yielding

$$\begin{aligned} S &\rightarrow AB \mid \lambda \\ A &\rightarrow CBR_1 \mid aR_1 \mid CB \mid a \\ B &\rightarrow AB \mid b \\ C &\rightarrow AC \mid c \\ R_1 &\rightarrow BR_1 \mid B. \end{aligned}$$

Now the B rules must be transformed to the appropriate form. The rule $B \rightarrow AB$ must be replaced since the number of B is three, and A , which occurs as the first symbol on the right-hand side, is two. Lemma 4.1.3 permits the leading A in the right-hand side of the rule $B \rightarrow AB$ to be replaced by the right-hand side of the A rules, producing

rm uses
and the
ring the
maining
tations

has one

greater than
the order
grammar
es of the

ive, the S
ing the A
a terminal
B violates
yielding

→ AB must
nbol on the
e of the rule

$$\begin{aligned}
 S &\rightarrow AB | \lambda \\
 A &\rightarrow CBR_1 | aR_1 | CB | a \\
 B &\rightarrow CBR_1B | aR_1B | CBB | aB | b \\
 C &\rightarrow AC | c \\
 R_1 &\rightarrow BR_1 | B.
 \end{aligned}$$

Applying the replacement techniques of Lemma 4.1.3 to the C rules creates two left-recursive rules.

$$\begin{aligned}
 S &\rightarrow AB | \lambda \\
 A &\rightarrow CBR_1 | aR_1 | CB | a \\
 B &\rightarrow CBR_1B | aR_1B | CBB | aB | b \\
 C &\rightarrow CBR_1C | aR_1C | CBC | aC | c \\
 R_1 &\rightarrow BR_1 | B
 \end{aligned}$$

The left recursion can be removed, introducing the new variable R_2 .

$$\begin{aligned}
 S &\rightarrow AB | \lambda \\
 A &\rightarrow CBR_1 | aR_1 | CB | a \\
 B &\rightarrow CBR_1B | aR_1B | CBB | aB | b \\
 C &\rightarrow aR_1C | aC | c | aR_1CR_2 | aCR_2 | cR_2 \\
 R_1 &\rightarrow BR_1 | B \\
 R_2 &\rightarrow BR_1CR_2 | BCR_2 | BR_1C | BC
 \end{aligned}$$

The original variables now satisfy the condition that the first symbol of the right-hand side of a rule is either a terminal or a variable whose number is greater than the number of the variable on the left-hand side. The variable with the highest number, in this case C , must have a terminal as the first symbol in each rule. The next variable, B , can have only C 's or terminals as the first symbol. A B rule beginning with the variable C can then be replaced by a set of rules, each of which begins with a terminal, using the C rules and Lemma 4.1.3. Making this transformation, we obtain the rules

$$\begin{aligned}
 S &\rightarrow AB | \lambda \\
 A &\rightarrow CBR_1 | aR_1 | CB | a \\
 B &\rightarrow aR_1B | aB | b \\
 &\quad \rightarrow aR_1CB | aCB | cB | aR_1CR_2BR_1B | aCR_2BR_1B | cR_2BR_1B \\
 &\quad \rightarrow aR_1CBB | aCBB | cBB | aR_1CR_2BB | aCR_2BB | cR_2BB \\
 C &\rightarrow aR_1C | aC | c | aR_1CR_2 | aCR_2 | cR_2 \\
 R_1 &\rightarrow BR_1 | B \\
 R_2 &\rightarrow BR_1CR_2 | BCR_2 | BR_1C | BC.
 \end{aligned}$$

The second list of B rules is obtained by substituting for C in the rule $B \rightarrow CBR_1B$ and the third in the rule $B \rightarrow CBB$. The S and A rules must also be rewritten to remove variables from the initial position of the right-hand side of a rule. The substitutions in the A rules use the B and C rules, all of which now begin with a terminal. The A , B , and C rules can then be used to transform the S rules, producing

$$\begin{aligned}
 S &\rightarrow \lambda \\
 &\rightarrow aR_1B \mid aB \\
 &\rightarrow aR_1CBR_1B \mid aCBR_1B \mid cBR_1B \mid aR_1CR_2BR_1B \mid aCR_2BR_1B \mid cR_2BR_1B \\
 &\rightarrow aR_1CBB \mid aCBB \mid cBB \mid aR_1CR_2BB \mid aCR_2BB \mid cR_2BB \\
 A &\rightarrow aR_1 \mid a \\
 &\rightarrow aR_1CBR_1 \mid aCBR_1 \mid cBR_1 \mid aR_1CR_2BR_1 \mid aCR_2BR_1 \mid cR_2BR_1 \\
 &\rightarrow aR_1CB \mid aCB \mid cB \mid aR_1CR_2B \mid aCR_2B \mid cR_2B \\
 B &\rightarrow aR_1B \mid aB \mid b \\
 &\rightarrow aR_1CBR_1B \mid aCBR_1B \mid cBR_1B \mid aR_1CR_2BR_1B \mid aCR_2BR_1B \mid cR_2BR_1B \\
 &\rightarrow aR_1CBB \mid aCBB \mid cBB \mid aR_1CR_2BB \mid aCR_2BB \mid cR_2BB \\
 C &\rightarrow aR_1C \mid aC \mid c \mid aR_1CR_2 \mid aCR_2 \mid cR_2 \\
 R_1 &\rightarrow BR_1 \mid B \\
 R_2 &\rightarrow BR_1CR_2 \mid BCR_2 \mid BR_1C \mid BC.
 \end{aligned}$$

Finally, the substitution process must be applied to each of the variables added in the removal of direct recursion. Rewriting these rules yields

$$\begin{aligned}
 R_1 &\rightarrow aR_1BR_1 \mid aBR_1 \mid bR_1 \\
 &\rightarrow aR_1CBR_1BR_1 \mid aCBR_1BR_1 \mid cBR_1BR_1 \mid aR_1CR_2BR_1BR_1 \mid aCR_2BR_1BR_1 \mid \\
 &\quad cR_2BR_1BR_1 \\
 &\rightarrow aR_1CBBR_1 \mid aCBBR_1 \mid cBBR_1 \mid aR_1CR_2BBR_1 \mid aCR_2BBR_1 \mid cR_2BBR_1 \\
 R_1 &\rightarrow aR_1B \mid aB \mid b \\
 &\rightarrow aR_1CBR_1B \mid aCBR_1B \mid cBR_1B \mid aR_1CR_2BR_1B \mid aCR_2BR_1B \mid cR_2BR_1B \\
 &\rightarrow aR_1CBB \mid aCBB \mid cBB \mid aR_1CR_2BB \mid aCR_2BB \mid cR_2BB \\
 R_2 &\rightarrow aR_1BR_1CR_2 \mid aBR_1CR_2 \mid bR_1CR_2 \\
 &\rightarrow aR_1CBR_1BR_1CR_2 \mid aCBR_1BR_1CR_2 \mid cBR_1BR_1CR_2 \mid aR_1CR_2BR_1BR_1CR_2 \mid \\
 &\quad aCR_2BR_1BR_1CR_2 \mid cR_2BR_1BR_1CR_2 \\
 &\rightarrow aR_1CBBR_1CR_2 \mid aCBBR_1CR_2 \mid cBBR_1CR_2 \mid aR_1CR_2BBR_1CR_2 \mid \\
 &\quad aCR_2BBR_1CR_2 \mid cR_2BBR_1CR_2
 \end{aligned}$$

$_1B$ and the
e variables
A rules use
es can then

$_2BR_1B$

R_2BR_1B

the removal

$3R_1BR_1 |$

$2BBR_1$

$2BR_1B$

$R_1BR_1CR_2 |$

|

$$\begin{aligned}
 R_2 &\rightarrow aR_1BCR_2 \mid aBCR_2 \mid bCR_2 \\
 &\rightarrow aR_1CBR_1BCR_2 \mid aCBR_1BCR_2 \mid cBR_1BCR_2 \mid aR_1CR_2BR_1BCR_2 \mid \\
 &\quad aCR_2BR_1BCR_2 \mid cR_2BR_1BCR_2 \\
 &\rightarrow aR_1CBBR_2 \mid aCBBR_2 \mid cBBCR_2 \mid aR_1CR_2BBCR_2 \mid aCR_2BBCR_2 \mid \\
 &\quad cR_2BBCR_2 \\
 R_2 &\rightarrow aR_1BR_1C \mid aBR_1C \mid bR_1C \\
 &\rightarrow aR_1CBR_1BR_1C \mid aCBR_1BR_1C \mid cBR_1BR_1C \mid aR_1CR_2BR_1BR_1C \mid \\
 &\quad aCR_2BR_1BR_1C \mid cR_2BR_1BR_1C \\
 &\rightarrow aR_1CBBR_1C \mid aCBBR_1C \mid cBBCR_1C \mid aR_1CR_2BBCR_1C \mid aCR_2BBCR_1C \mid \\
 &\quad cR_2BBCR_1C \\
 R_2 &\rightarrow aR_1BC \mid aBC \mid bC \\
 &\rightarrow aR_1CBR_1BC \mid aCBR_1BC \mid cBR_1BC \mid aR_1CR_2BR_1BC \mid aCR_2BR_1BC \mid \\
 &\quad cR_2BR_1BC \\
 &\rightarrow aR_1CBB \mid aCBB \mid cBBC \mid aR_1CR_2BBC \mid aCR_2BBC \mid cR_2BBC.
 \end{aligned}$$

The resulting grammar in Greibach normal form has lost all the simplicity of the original grammar G. Designing a grammar in Greibach normal form is an almost impossible task. The construction of grammars should be done using simpler, intuitive rules. As with all the preceding transformations, the steps necessary to transform an arbitrary context-free grammar to Greibach normal form are algorithmic and can be automatically performed by an appropriately designed computer program. The input to such a program consists of the rules of an arbitrary context-free grammar, and the result is an equivalent Greibach normal form grammar.

It should also be pointed out that useless symbols may be created by the rule replacements using Lemma 4.1.3. The variable A is a useful symbol of G, occurring in the derivation

$$S \Rightarrow AB \Rightarrow aB \Rightarrow ab.$$

In the conversion to Greibach normal form, the substitutions removed all occurrences of A from the right-hand side of rules. The string ab is generated by

$$S \Rightarrow aB \Rightarrow ab$$

in the equivalent Greibach normal form grammar.

Theorem 4.8.2

Let G be a context-free grammar. There is an algorithm to construct an equivalent context-free grammar in Greibach normal form.

Proof. The operations used in the construction of the Greibach normal form have previously been shown to generate equivalent grammars. All that remains is to show that the rules can always be transformed to satisfy the conditions necessary to perform the substitutions. These require that each rule have the form

$$A_k \rightarrow A_j w \text{ with } k < j$$

or

$$A_k \rightarrow aw,$$

where the subscript represents the ordering of the variables.

The proof is by induction on the ordering of the variables. The basis is the start symbol, the variable numbered one. Since S is nonrecursive, this condition trivially holds. Now assume that all variables up to number k satisfy the condition. If there is a rule $A_k \rightarrow A_i w$ with $i < k$, the substitution can be applied to the variable A_i to generate a set of rules, each of which has the form $A_k \rightarrow A_j w'$ where $j > i$. This process can be repeated, $k - i$ times if necessary, to produce a set of rules that are either left-recursive or in the correct form. All directly left-recursive variables can be transformed using the technique of Lemma 4.7.1. ■

Example 4.8.1

The Chomsky and Greibach normal forms are constructed for the grammar

$$\begin{aligned} S &\rightarrow SaB \mid aB \\ B &\rightarrow bB \mid \lambda. \end{aligned}$$

Adding a nonrecursive start symbol S' and removing λ and chain rules yields

$$\begin{aligned} S' &\rightarrow SaB \mid Sa \mid aB \mid a \\ S &\rightarrow SaB \mid Sa \mid aB \mid a \\ B &\rightarrow bB \mid b. \end{aligned}$$

The Chomsky normal form is obtained by transforming the preceding rules. Variables A and C are used as aliases for a and b , respectively, and T represents the string aB .

$$\begin{aligned} S' &\rightarrow ST \mid SA \mid AB \mid a \\ S &\rightarrow ST \mid SA \mid AB \mid a \\ B &\rightarrow CB \mid b \\ T &\rightarrow AB \\ A &\rightarrow a \\ C &\rightarrow b \end{aligned}$$

The variables are ordered by S' , S , B , T , A , and C . Removing the left-recursive S rules produces

$$\begin{aligned} S' &\rightarrow ST \mid SA \mid AB \mid a \\ S &\rightarrow ABZ \mid aZ \mid AB \mid a \\ B &\rightarrow CB \mid b \\ T &\rightarrow AB \\ A &\rightarrow a \\ C &\rightarrow b \\ Z &\rightarrow TZ \mid AZ \mid T \mid A. \end{aligned}$$

These rules satisfy the condition that requires the value of the variable on the left-hand side of a rule to be less than that of a variable in the first position of the right-hand side. Implementing the substitutions beginning with the A and C rules produces the Greibach normal form grammar:

$$\begin{aligned} S' &\rightarrow aBZT \mid aZT \mid aBT \mid aT \mid aBZA \mid aZA \mid aBA \mid aA \mid aB \mid a \\ S &\rightarrow aBZ \mid aZ \mid aB \mid a \\ B &\rightarrow bB \mid b \\ T &\rightarrow aB \\ A &\rightarrow a \\ C &\rightarrow b \\ Z &\rightarrow aBZ \mid aZ \mid aB \mid a. \end{aligned}$$

The leftmost derivation of the string $abaaba$ is given in each of the three equivalent grammars.

G	Chomsky Normal Form	Greibach Normal Form
$S \Rightarrow SaB$	$S' \Rightarrow SA$	$S' \Rightarrow aBZA$
$\Rightarrow SaBaB$	$\Rightarrow STA$	$\Rightarrow abZA$
$\Rightarrow SaBaBaB$	$\Rightarrow SATA$	$\Rightarrow abaZA$
$\Rightarrow aBaBaBaB$	$\Rightarrow ABATA$	$\Rightarrow abaAB$
$\Rightarrow abBaBaBaB$	$\Rightarrow aBATA$	$\Rightarrow abaabA$
$\Rightarrow abaBaBaB$	$\Rightarrow abATA$	$\Rightarrow abaaba$
$\Rightarrow abaaBaB$	$\Rightarrow abaTA$	
$\Rightarrow abaabBaB$	$\Rightarrow abaABA$	
$\Rightarrow abaabab$	$\Rightarrow abaAB$	
$\Rightarrow abaaba$	$\Rightarrow abaAB$	
		$\Rightarrow abaaba$

The derivation in the Chomsky normal form grammar generates six variables. Each of these is transformed to a terminal by a rule of the form $A \rightarrow a$. The Greibach normal form derivation generates a terminal with each rule application. The derivation is completed using only six rule applications. \square

Exercises

For Exercises 1 through 5, construct an equivalent essentially noncontracting grammar G_L with a nonrecursive start symbol. Give a regular expression for the language of each grammar.

1. $G: S \rightarrow aS \mid bS \mid B$

$B \rightarrow bb \mid C \mid \lambda$
 $C \rightarrow cC \mid \lambda$

2. $G: S \rightarrow ABC \mid \lambda$

$A \rightarrow aA \mid a$
 $B \rightarrow bB \mid A$
 $C \rightarrow cC \mid \lambda$

3. $G: S \rightarrow BSA \mid A$

$A \rightarrow aA \mid \lambda$
 $B \rightarrow Bba \mid \lambda$

4. $G: S \rightarrow AB \mid BCS$

$A \rightarrow aA \mid C$
 $B \rightarrow bbB \mid b$
 $C \rightarrow cC \mid \lambda$

5. $G: S \rightarrow ABC \mid aBC$

$A \rightarrow aA \mid BC$
 $B \rightarrow bB \mid \lambda$
 $C \rightarrow cC \mid \lambda$

6. Prove Lemma 4.3.2.

For Exercises 7 through 10, construct an equivalent grammar G_C that does not contain chain rules. Give a regular expression for the language of each grammar. Note that these grammars do not contain λ -rules.

7. $G: S \rightarrow AS \mid A$

$A \rightarrow aA \mid bB \mid C$
 $B \rightarrow bB \mid b$
 $C \rightarrow cC \mid B$

8. $G: S \rightarrow A \mid B \mid C$

$A \rightarrow aa \mid B$

es. Each
h normal
completed

□

grammar
e of each

$$B \rightarrow bb \mid C$$

$$C \rightarrow cc \mid A$$

9. G: $S \rightarrow A \mid C$

$$A \rightarrow aA \mid a \mid B$$

$$B \rightarrow bB \mid b$$

$$C \rightarrow cC \mid c \mid B$$

10. G: $S \rightarrow AB \mid C$

$$A \rightarrow aA \mid B$$

$$B \rightarrow bB \mid C$$

$$C \rightarrow cC \mid a \mid A$$

11. Eliminate the chain rules from the grammar G_L of Exercise 1.

12. Eliminate the chain rules from the grammar G_L of Exercise 4.

13. Prove that Algorithm 4.4.2 generates the set of variables that derive terminal strings.

For Exercises 14 through 16, construct an equivalent grammar without useless symbols. Trace the generation of the sets of TERM and REACH used to construct G_T and G_U . Describe the language generated by the grammar.

14. G: $S \rightarrow AA \mid CD \mid bB$

$$A \rightarrow aA \mid a$$

$$B \rightarrow bB \mid bC$$

$$C \rightarrow cB$$

$$D \rightarrow dD \mid d$$

15. G: $S \rightarrow aA \mid BD$

$$A \rightarrow aA \mid aAB \mid aD$$

$$B \rightarrow aB \mid aC \mid BF$$

$$C \rightarrow Bb \mid aAC \mid E$$

$$D \rightarrow bD \mid bC \mid b$$

$$E \rightarrow aB \mid bC$$

$$F \rightarrow aF \mid aG \mid a$$

$$G \rightarrow a \mid b$$

16. G: $S \rightarrow ACH \mid BB$

$$A \rightarrow aA \mid aF$$

$$B \rightarrow CFH \mid b$$

$$C \rightarrow aC \mid DH$$

$$D \rightarrow aD \mid BD \mid Ca$$

$$F \rightarrow bB \mid b$$

$$H \rightarrow dH \mid d$$

17. Show that all the symbols of the grammar

$$\begin{aligned} G: S &\rightarrow A \mid CB \\ A &\rightarrow C \mid D \\ B &\rightarrow bB \mid b \\ C &\rightarrow cC \mid c \\ D &\rightarrow dD \mid d \end{aligned}$$

are useful. Construct an equivalent grammar G_C by removing the chain rules from G . Show that G_C contains useless symbols.

18. Convert the grammar

$$\begin{aligned} G: S &\rightarrow aA \mid ABa \\ A &\rightarrow AA \mid a \\ B &\rightarrow AbB \mid bb \end{aligned}$$

to Chomsky normal form. G already satisfies the conditions on the start symbol S , λ -rules, useless symbols, and chain rules.

19. Convert the grammar

$$\begin{aligned} G: S &\rightarrow aAbB \mid ABC \mid a \\ A &\rightarrow aA \mid a \\ B &\rightarrow bBcC \mid b \\ C &\rightarrow abc \end{aligned}$$

to Chomsky normal form. G already satisfies the conditions on the start symbol S , λ -rules, useless symbols, and chain rules.

20. Convert the result of Exercise 9 to Chomsky normal form.

21. Convert the result of Exercise 11 to Chomsky normal form.

22. Convert the result of Exercise 12 to Chomsky normal form.

23. Convert the grammar

$$\begin{aligned} G: S &\rightarrow A \mid ABa \mid AbA \\ A &\rightarrow Aa \mid \lambda \\ B &\rightarrow Bb \mid BC \\ C &\rightarrow CB \mid CA \mid bB \end{aligned}$$

to Chomsky normal form.

- * 24. Let G be a grammar in Chomsky normal form.

- a) What is the length of a derivation of a string of length n in $L(G)$?

- b) What is
c) What is

25. Give the up
Chomsky no
 $aabb$.

26. Let G be the

- Give the upp
grammar G a

27. Let G be the

- a) Give a reg
b) Construct

28. Construct a g

Give a leftmo

29. Construct a g

30. Construct a G

- b) What is the maximum depth of a derivation tree for a string of length n in $L(G)$?
 c) What is the minimum depth of a derivation tree for a string of length n in $L(G)$?
 25. Give the upper diagonal matrix produced by the CYK algorithm when run with the Chomsky normal form grammar from Example 4.5.2 and the input strings $abbb$ and $aabbb$.
 26. Let G be the Chomsky normal form grammar

from G .

$$\begin{aligned} S &\rightarrow AX \mid AY \mid a \\ X &\rightarrow AX \mid a \\ Y &\rightarrow BY \mid a \\ A &\rightarrow a \\ B &\rightarrow b. \end{aligned}$$

Give the upper diagonal matrix produced by the CYK algorithm when run with the grammar G and the input strings $baaa$ and $abaaa$.

symbol S ,

27. Let G be the grammar

$$\begin{aligned} G: S &\rightarrow A \mid B \\ A &\rightarrow aaB \mid Aab \mid Aba \\ B &\rightarrow bB \mid Bb \mid aba. \end{aligned}$$

- a) Give a regular expression for $L(G)$.
 b) Construct a grammar G' that contains no left-recursive rules and is equivalent to G .
 28. Construct a grammar G' that contains no left-recursive rules and is equivalent to

symbol S ,

$$\begin{aligned} G: S &\rightarrow A \mid C \\ A &\rightarrow AaB \mid AaC \mid B \mid a \\ B &\rightarrow Bb \mid Cb \\ C &\rightarrow cC \mid c. \end{aligned}$$

Give a leftmost derivation of the string $aaccacb$ in the grammars G and G' .

29. Construct a grammar G' that contains no left-recursive rules and is equivalent to

$$\begin{aligned} G: S &\rightarrow A \mid B \\ A &\rightarrow AAA \mid a \mid B \\ B &\rightarrow BBb \mid b. \end{aligned}$$

30. Construct a Greibach normal form grammar equivalent to

$$\begin{aligned} S &\rightarrow aAb \mid a \\ A &\rightarrow SS \mid b. \end{aligned}$$

31. Convert the Chomsky normal form grammar

$$\begin{aligned}S &\rightarrow BB \\A &\rightarrow AA \mid a \\B &\rightarrow AA \mid BA \mid b\end{aligned}$$

to Greibach normal form. Process the variables according to the order S, A, B .

32. Convert the Chomsky normal form grammar

$$\begin{aligned}S &\rightarrow AB \mid BC \\A &\rightarrow AB \mid a \\B &\rightarrow AA \mid CB \mid b \\C &\rightarrow a \mid b\end{aligned}$$

to Greibach normal form. Process the variables according to the order S, A, B, C .

33. Convert the Chomsky normal form grammar

$$\begin{aligned}S &\rightarrow BA \mid AB \mid \lambda \\A &\rightarrow BB \mid AA \mid a \\B &\rightarrow AA \mid b\end{aligned}$$

to Greibach normal form. Process the variables according to the order S, A, B .

34. Convert the Chomsky normal form grammar

$$\begin{aligned}S &\rightarrow AB \\A &\rightarrow BB \mid CC \\B &\rightarrow AD \mid CA \\C &\rightarrow a \\D &\rightarrow b\end{aligned}$$

to Greibach normal form. Process the variables according to the order S, A, B, C, D .

- * 35. Prove that every context-free language is generated by a grammar in which each of the rules has one of the following forms:

i) $S \rightarrow \lambda$,

ii) $A \rightarrow a$,

iii) $A \rightarrow aB$, or

iv) $A \rightarrow aBC$,

where $A \in V$, $B, C \in V - \{S\}$, and $a \in \Sigma$.

Bibliographic Notes

The constructions for removing λ -rules and chain rules were presented in Bar-Hillel, Perles, and Shamir [1961]. Chomsky normal form was introduced in Chomsky [1959]. The CYK algorithm is named for J. Cocke, D. Younger [1967], and T. Kasami who independently developed this technique for determining derivability. Variations of this algorithm can be used to solve the membership problem for arbitrary context-free grammars without requiring the transformation to Chomsky normal form.

Greibach normal form is from Greibach [1965]. An alternative transformation to Greibach normal form that limits the growth of the number of rules in the resulting grammar can be found in Blum and Koch [1999]. There are several variations on the definition of Greibach normal form. A common formulation requires a terminal symbol in the first position of the string but permits the remainder of the string to contain both variables and terminals. Double Greibach normal form, Engelfriet [1992], requires that both the leftmost and rightmost symbol on the right-hand of rules be terminals.

A grammar whose rules satisfy the conditions of Exercise 35 is said to be in 2-normal form. A proof that 2-normal form grammars generate the entire set of context-free languages can be found in Hopcroft and Ullman [1979] and Harrison [1978]. Additional normal forms for context-free grammars are given in Harrison [1978].

CHAPTER 5

Finite Automata

In this chapter we introduce the family of abstract computing devices known as finite-state machines. The computations of a finite-state machine determine whether a string satisfies a set of conditions or matches a prescribed pattern. Finite-state machines share properties common to many mechanical devices; they process input and generate output. A vending machine takes coins as input and returns food or beverages as output. A combination lock expects a sequence of numbers and opens the lock if the input sequence is correct. The input to a finite-state machine is a string and the result of a computation indicates acceptability of the string. The set of strings that are accepted makes up the language of the machine.

The preceding examples of machines exhibit a property that we take for granted in mechanical computation, determinism. When the appropriate amount of money is inserted into a vending machine, we are upset if nothing is forthcoming. Similarly, we expect the combination to open the lock and all other sequences to fail. Initially, we require finite-state machines to be deterministic. This condition will be relaxed to examine the effects of nondeterminism on the capabilities of finite-state computation.

5.1 A Finite-State Machine

A formal definition of a machine is not concerned with the hardware involved in the operation of the machine, but rather with a description of the internal operations as the machine processes the input. A vending machine may be built with levers, a combination lock with tumblers, and an electronic entry system is controlled by a microchip, but all accept

input and produce an affirmative or negative response. What sort of description encompasses the features of each of these seemingly different types of mechanical computation?

A simple newspaper vending machine, similar to those found on many street corners, is used to illustrate the components of a finite-state machine. The input to the machine consists of nickels, dimes, and quarters. When 30 cents is inserted, the cover of the machine may be opened and a paper removed. If the total of the coins exceeds 30 cents, the machine graciously accepts the overpayment and does not give change.

The newspaper machine on the street corner has no memory, at least not as we usually conceive of memory in a computing machine. However, the machine "knows" that an additional 5 cents will unlatch the cover when 25 cents has previously been inserted. This knowledge is acquired by the machine's altering its internal state whenever input is received and processed.

A machine state represents the status of an ongoing computation. The internal operation of the vending machine can be described by the interactions of the following seven states. The names of the states, given in italics, indicate the progress made toward opening the cover.

- *Needs 30 cents*: The state of the machine before any coins are inserted
- *Needs 25 cents*: The state after a nickel has been input
- *Needs 20 cents*: The state after two nickels or a dime have been input
- *Needs 15 cents*: The state after three nickels or a dime and a nickel have been input
- *Needs 10 cents*: The state after four nickels, a dime and two nickels, or two dimes have been input
- *Needs 5 cents*: The state after a quarter, five nickels, two dimes and a nickel, or one dime and three nickels have been input
- *Needs 0 cents*: The state that represents having at least 30 cents input

The insertion of a coin causes the machine to alter its state. When 30 cents or more is input, the state *needs 0 cents* is entered and the latch is opened. Such a state is called *accepting* since it indicates the correctness of the input.

The design of the machine must represent each of the components symbolically. Rather than a sequence of coins, the input to the abstract machine is a string of symbols. A labeled directed graph known as a **state diagram** is often used to represent the transformations of the internal state of the machine. The nodes of the state diagram are the states described above. The *needs m cents* node is represented simply by m in the state diagram. The state of the machine at the beginning of a computation is designated $\times\circlearrowleft$. The initial state for the newspaper vending machine is the node 30 .

The arcs are labeled n , d , or q , representing the input of a nickel, dime, or quarter. An arc from node x to node y labeled v indicates that processing input v when the machine is in state x causes the machine to enter state y . Figure 5.1 gives the state diagram for the newspaper vending machine. The arc labeled d from node 15 to 5 represents the change of state of the machine when 15 cents has previously been processed and a dime is input. The

cycles of
30 cents le

Input t
during the
diagram. T
by the first
symbol of
node, the n
entire input
in the accep
 $nndn$ is no

5.2 De

The analysi
from the in
referred to a
computation

Definition 5

A determin
is a finite se
as the start s
from $Q \times \Sigma$

We hav
its mechan
present in m
consisting o
tape, a tape a

The stat
 $q_0, q_1, q_2,$

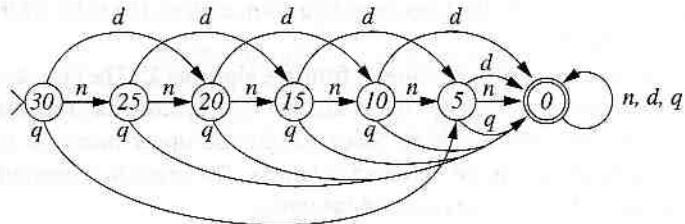


FIGURE 5.1 State diagram of newspaper vending machine.

cycles of length one from node 0 to itself indicate that any input that increases the total past 30 cents leaves the latch unlocked.

Input to the machine consists of strings from $\{n, d, q\}^*$. The sequence of states entered during the processing of an input string can be traced by following the arcs in the state diagram. The machine is in its initial state at the beginning of a computation. The arc labeled by the first input symbol is traversed, specifying the subsequent machine state. The next symbol of the input string is processed by traversing the appropriate arc from the current node, the node reached by traversal of the previous arc. This procedure is repeated until the entire input string has been processed. The string $dndn$ is accepted by the vending machine, while the string $nndn$ is not accepted since the computation terminates in state 5.

5.2 Deterministic Finite Automata

The analysis of the vending machine required separating the fundamentals of the design from the implementational details. The implementation-independent description is often referred to as an *abstract machine*. We now introduce a class of abstract machines whose computations can be used to determine the acceptability of input strings.

Definition 5.2.1

A **deterministic finite automaton** (DFA) is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ a finite set called the *alphabet*, $q_0 \in Q$ a distinguished state known as the *start state*, F a subset of Q called the *final or accepting states*, and δ a total function from $Q \times \Sigma$ to Q known as the *transition function*.

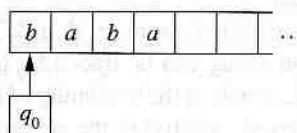
We have referred to a deterministic finite automaton as an abstract machine. To reveal its mechanical nature, the operation of a DFA is described in terms of components that are present in many familiar computing machines. An automaton can be thought of as a machine consisting of five components: a single internal register, a set of values for the register, a tape, a tape reader, and an instruction set.

The states of a DFA represent the internal status of the machine and are often denoted $q_0, q_1, q_2, \dots, q_n$. The register of the machine, also called the finite control, contains

one of the states as its value. At the beginning of a computation, the value of the register is q_0 , the start state of the DFA.

The input is a finite sequence of elements from the alphabet Σ . The tape stores the input until needed by the computation. The tape is divided into squares, each square capable of holding one element from the alphabet. Since there is no upper bound to the length of an input string, the tape must be of unbounded length. The input to a computation of the automaton is placed on an initial segment of the tape.

The tape head reads a single square of the input tape. The body of the machine consists of the tape head and the register. The position of the tape head is indicated by placing the body of the machine under the tape square being scanned. The current state of the automaton is indicated by the value on the register. The initial configuration of a computation with input $baba$ is depicted



A computation of an automaton consists of the execution of a sequence of instructions. The execution of an instruction alters the state of the machine and moves the tape head one square to the right. The instruction set is obtained from the transition function of the DFA. The machine state and the symbol scanned determine the instruction to be executed. The action of a machine in state q_i scanning an a is to reset the state to $\delta(q_i, a)$. Since δ is a total function, there is exactly one instruction specified for every combination of state and input symbol, hence the *deterministic* in deterministic finite automaton.

The objective of a computation of an automaton is to determine the acceptability of the input string. A computation begins with the tape head scanning the leftmost square of the tape and the register containing the state q_0 . The state and symbol are used to select the instruction. The machine then alters its state as prescribed by the instruction, and the tape head moves to the right. The transformation of a machine by the execution of an instruction cycle is exhibited in Figure 5.2. The instruction cycle is repeated until the tape head scans a blank square, at which time the computation terminates. An input string is accepted if the computation terminates in an accepting state; otherwise it is rejected. The computation in Figure 5.2 exhibits the acceptance of the string aba .

Definition 5.2.2

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The language of M , denoted $L(M)$, is the set of strings in Σ^* accepted by M .

A DFA can be considered to be a language acceptor; the language of the machine is simply the set of strings accepted by its computations. The language of the machine in Figure 5.2 is the set of all strings over $\{a, b\}$ that end in a .

A DFA is a read-only machine that processes the input in a left-to-right manner; once an input symbol has been read, it has no further effect on the computation. At any point during the computation, the result depends only on the current state and the unprocessed

input. This con
ordered pair $[q_i, aw]$ in
instruction cycl
 $[q_i, aw] \xrightarrow{M} [q_j,$
execution of one
a function from
M is omitted w

Definition 5.2.3

The function \xrightarrow{M}

for $a \in \Sigma$ and w

The notation
obtained from $[q_i,$

register is

the input
capable of
length of
ion of the

ie consists
lacing the
automaton
with input

structions.
e head one
f the DFA.
cuted. The
ince δ is a
of state and

ptability of
st square of
o select the
nd the tape
instruction
ead scans a
epted if the
putation in

is the set of

machine is
machine in
anner; once
At any point
unprocessed

$$\begin{array}{ll} M: Q = \{q_0, q_1\} & \delta(q_0, a) = q_1 \\ \Sigma = \{a, b\} & \delta(q_0, b) = q_0 \\ F = \{q_1\} & \delta(q_1, a) = q_1 \\ & \delta(q_1, b) = q_0 \end{array}$$

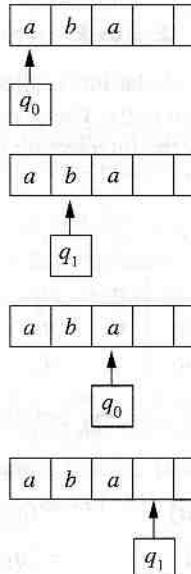


FIGURE 5.2 Computation in a DFA.

input. This combination is called a **machine configuration** and is represented by the ordered pair $[q_i, w]$, where q_i is the current state and $w \in \Sigma^*$ is the unprocessed input. The instruction cycle of a DFA transforms one machine configuration to another. The notation $[q_i, aw] \xrightarrow{M} [q_j, w]$ indicates that configuration $[q_j, w]$ is obtained from $[q_i, aw]$ by the execution of one instruction cycle of the machine M . The symbol \xrightarrow{M} , read “yields,” defines a function from $Q \times \Sigma^+$ to $Q \times \Sigma^*$ that can be used to trace computations of the DFA. The M is omitted when there is no possible ambiguity.

Definition 5.2.3

The function \xrightarrow{M} on $Q \times \Sigma^+$ is defined by

$$[q_i, aw] \xrightarrow{M} [\delta(q_i, a), w]$$

for $a \in \Sigma$ and $w \in \Sigma^*$, where δ is the transition function of the DFA M .

The notation $[q_i, u] \xrightarrow{*} [q_j, v]$ is used to indicate that configuration $[q_j, v]$ can be obtained from $[q_i, u]$ by zero or more transitions.

Example 5.2.1

The DFA M defined below accepts the set of strings over $\{a, b\}$ that contain the substring bb . That is, $L(M) = (a \cup b)^*bb(a \cup b)^*$. The states and alphabet of M are

$$M : Q = \{q_0, q_1, q_2\}$$

$$\Sigma = \{a, b\}$$

$$F = \{q_2\}.$$

The transition function δ is given in a tabular form called the *transition table*. The states are listed vertically and the alphabet horizontally. The action of the automaton in state q_i with input a can be determined by finding the intersection of the row corresponding to q_i and the column corresponding to a .

δ	a	b
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_2	q_2

The computations of M with input strings $abba$ and $abab$ are traced using the function \vdash .

$[q_0, abba]$	$[q_0, abab]$
$\vdash [q_0, bba]$	$\vdash [q_0, bab]$
$\vdash [q_1, ba]$	$\vdash [q_1, ab]$
$\vdash [q_2, a]$	$\vdash [q_0, b]$
$\vdash [q_2, \lambda]$	$\vdash [q_1, \lambda]$
accepts	rejects

The string $abba$ is accepted since the computation halts in state q_2 . \square

Example 5.2.2

The newspaper vending machine from the previous section can be represented by a DFA with the following states, alphabet, and transition function. The start state is the state 30.

$$Q = \{0, 5, 10, 15, 20, 25, 30\}$$

$$\Sigma = \{n, d, q\}$$

$$F = \{0\}$$

δ	n	d	q
0	0	0	0
5	0	0	0
10	5	0	0
15	10	5	0
20	15	10	0
25	20	15	0
30	25	20	5

The language of or more. Can yo

The transition from the alphabet state and a string the domain from

Definition 5.2.4

The extended tra $Q \times \Sigma^*$ to Q . Th

i) Basis: length

length

ii) Recursive st $\delta(\hat{\delta}(q_i, u), a)$, a

The comput evaluation of the function required notation, the lang

5.3 State D

The state diagram states of the mach Figure 5.1 is the intuitive nature of than the sets and t

Definition 5.3.1

The state diagram by the following c

i) The nodes of

ii) The labels on

iii) q_0 is the start

iv) F is the set of

v) There is an ar

vi) For every nod

The language of the vending machine consists of all strings that represent a sum of 30 cents or more. Can you construct a regular expression that defines the language of this machine?

bstring

□

The transition function specifies the action of the machine for a given state and element from the alphabet. This function can be extended to a function $\hat{\delta}$ whose input consists of a state and a string over the alphabet. The function $\hat{\delta}$ is constructed by recursively extending the domain from elements of Σ to strings of arbitrary length.

ates are

q_i with
o q_i and

ction \vdash .

Definition 5.2.4

The **extended transition function**, $\hat{\delta}$, of a DFA with transition function δ is a function from $Q \times \Sigma^*$ to Q . The values of $\hat{\delta}$ are defined by recursion on the length of the input string.

- i) Basis: $\text{length}(w) = 0$. Then $w = \lambda$ and $\hat{\delta}(q_i, \lambda) = q_i$.
 $\text{length}(w) = 1$. Then $w = a$, for some $a \in \Sigma$, and $\hat{\delta}(q_i, a) = \delta(q_i, a)$.
- ii) Recursive step: Let w be a string of length $n > 1$. Then $w = ua$ and $\hat{\delta}(q_i, ua) = \delta(\hat{\delta}(q_i, u), a)$.

The computation of a machine in state q_i with string w halts in state $\hat{\delta}(q_i, w)$. The evaluation of the function $\hat{\delta}(q_0, w)$ simulates the repeated applications of the transition function required to process the string w . A string w is accepted if $\hat{\delta}(q_0, w) \in F$. Using this notation, the language of a DFA M is the set $L(M) = \{w \mid \hat{\delta}(q_0, w) \in F\}$.

5.3 State Diagrams and Examples

The state diagram of a DFA is a labeled directed graph in which the nodes represent the states of the machine and the arcs are obtained from the transition function. The graph in Figure 5.1 is the state diagram for the newspaper vending machine DFA. Because of the intuitive nature of the graphic representation, we will often present the state diagram rather than the sets and transition function that constitute the formal definition of a DFA.

by a DFA
state 30.

□

Definition 5.3.1

The **state diagram** of a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is a labeled directed graph G defined by the following conditions:

- i) The nodes of G are the elements of Q .
- ii) The labels on the arcs of G are elements of Σ .
- iii) q_0 is the start node, which is depicted
- iv) F is the set of accepting nodes; each accepting node is depicted
- v) There is an arc from node q_i to q_j labeled a , if $\delta(q_i, a) = q_j$.
- vi) For every node q_i and symbol $a \in \Sigma$, there is exactly one arc labeled a leaving q_i .

A transition of a DFA is represented by an arc in the state diagram. Tracing the computation of a DFA in the corresponding state diagram constructs a path that begins at node q_0 and "spells" the input string. Let p_w be a path beginning at q_0 that spells w , and let q_w be the terminal node of p_w . Theorem 5.3.2 proves that there is only one such path for every string $w \in \Sigma^*$. Moreover, q_w is the state of the DFA upon completion of the processing of w .

Theorem 5.3.2

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $w \in \Sigma^*$. Then w determines a unique path p_w in the state diagram of M and $\hat{\delta}(q_0, w) = q_w$.

Proof. The proof is by induction on the length of the string. If the length of w is zero, then $\hat{\delta}(q_0, \lambda) = q_0$. The corresponding path is the null path that begins and terminates with q_0 .

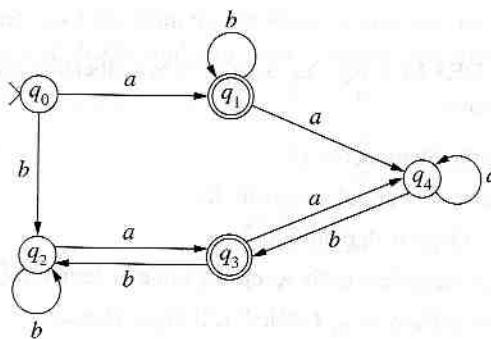
Assume that the result holds for all strings of length n or less. Let $w = ua$ be a string of length $n + 1$. By the inductive hypothesis, there is a unique path p_u that spells u and $\hat{\delta}(q_0, u) = q_u$. The path p_w is constructed by following the arc labeled a from q_u . This is the only path from q_0 that spells w since p_u is the unique path that spells u and there is only one arc leaving q_u labeled a . The terminal state of the path p_w is determined by the transition $\delta(q_u, a)$. From the definition of the extended transition function, $\hat{\delta}(q_0, w) = \delta(\hat{\delta}(q_0, u), a)$. Since $\hat{\delta}(q_0, u) = q_u$, $q_w = \delta(q_u, a) = \delta(\hat{\delta}(q_0, u), a) = \hat{\delta}(q_0, w)$ as desired. ■

The equivalence of computations of a DFA and paths in the state diagram gives us a heuristic method for determining the language of the DFA. The strings accepted in a state q_i are precisely those spelled by paths from q_0 to q_i . We can separate the determination of these paths into two parts:

- First, find regular expressions u_1, \dots, u_n for strings on all paths from q_0 that reach q_i the first time.
- Find regular expressions v_1, \dots, v_m for all ways to leave q_i and return to q_i .

The strings accepted by q_i are $(u_1 \cup \dots \cup u_n)(v_1 \cup \dots \cup v_m)^*$.

Consider the DFA



The lang
the heur

Sta

q_1

q_3

Consequ
additional
ically pro

In th
ability to
we will c
substring
L and one

Example

The state

The st
entered wh
the input is
 $ababb$ and

The string a
terminal stat

Tracing the path that begins at state q_0 and spells w , only one such completion of the

unique path p_w

v is zero, then states with q_0 . Let ua be a string that spells u and ends at q_u . This is true since there is only one transition $\delta(\hat{\delta}(q_0, u), a)$.

Diagram gives us a path in a state termination of

that reach q_i to q_i .

The language of M consists of all strings spelled by paths from q_0 to either q_1 or q_3 . Using the heuristic described previously, the strings on the paths to each of the accepting states are

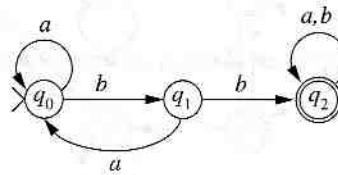
State	Paths to q_i	Simple Cycles from q_i to q_i	Accepted Strings
q_1	a	b	ab^*
q_3	ab^*aa^*b, bb^*a	bb^*a, aa^*b	$(ab^*aa^*b \cup bb^*a)(ab \cup ba)^*$

Consequently, $L(M) = ab^* \cup (ab^*aa^*b \cup bb^*a)(ab \cup ba)^*$. After we have established additional properties of finite-state computation, we will present an algorithm that automatically produces a regular expression for the language of a finite automaton.

In the remainder of this section we examine a number of DFAs to help develop the ability to design automata to check for patterns in strings. The types of conditions that we will consider include the number of occurrences and the relative positions of specified substrings. In addition, we establish the relationship between a DFA that accepts a language L and one that accepts the complement of L .

Example 5.3.1

The state diagram of the DFA in Example 5.2.1 is



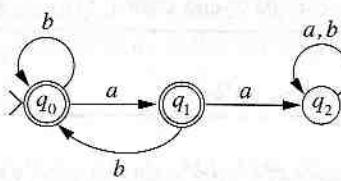
The states are used to record the number of consecutive b 's processed. The state q_2 is entered when a substring bb is encountered. Once the machine enters q_2 , the remainder of the input is processed, leaving the state unchanged. The computation of the DFA with input $ababb$ and the corresponding path in the state diagram are

Computation	Path
$[q_0, ababb]$	$q_0,$
$\vdash [q_0, babb]$	$q_0,$
$\vdash [q_1, abb]$	$q_1,$
$\vdash [q_0, bb]$	$q_0,$
$\vdash [q_1, b]$	$q_1,$
$\vdash [q_2, \lambda]$	q_2

The string $ababb$ is accepted since the halting state of the computation, which is also the terminal state of the path that spells $ababb$, is the accepting state q_2 . \square

Example 5.3.2

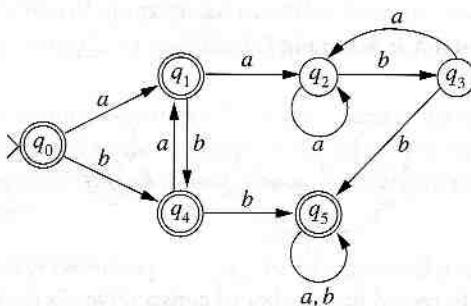
The DFA



accepts $(b \cup ab)^*(a \cup \lambda)$, the set of strings over $\{a, b\}$ that do not contain the substring aa . \square

Example 5.3.3

Strings over $\{a, b\}$ that contain the substring bb or do not contain the substring aa are accepted by the DFA depicted below. This language is the union of the languages of the previous examples.



The state diagrams for machines that accept the strings with substring bb or without substring aa seem simple compared with the machine that accepts the union of those two languages. There does not appear to be an intuitive way to combine the state diagrams of the constituent DFAs to create the desired composite machine.

The next several examples provide a heuristic for designing DFAs. The first step is to produce an interpretation for the states of the DFA. The interpretation of a state describes properties of the string that has been processed when the machine is in the state. The pertinent properties are determined by the conditions required for a string to be accepted.

Example 5.3.4

A successful computation of a DFA that accepts the strings over $\{a, b\}$ containing the substring aaa must process three a 's in a row. Four states are required to record the status of a computation checking for aaa . The interpretation of the states, along with state names, are

Prior to reading
Consequently, this c

Once the states
When computation
entered. On the other
toward aaa and the
be determined for al



On processing aaa , t
the input.

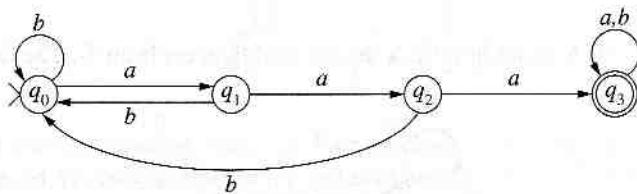
Example 5.3.5

Building a machine
requires checking tw
are required to store t
describes the number
is in the state.

State	Interpretation
q_0 :	No progress toward aaa
q_1 :	Last symbol processed was an a
q_2 :	Last two symbols processed were aa
q_3 :	aaa has been found in the string

Prior to reading the first symbol, no progress has been made toward finding aaa . Consequently, this condition represents the start state.

Once the states are identified, it is frequently easy to determine the proper transitions. When computation in state q_1 processes an a , the last two symbols read are aa and q_2 is entered. On the other hand, if a b is read in q_1 , the resulting string represents no progress toward aaa and the computation enters q_0 . Following a similar strategy, the transitions can be determined for all states producing the DFA



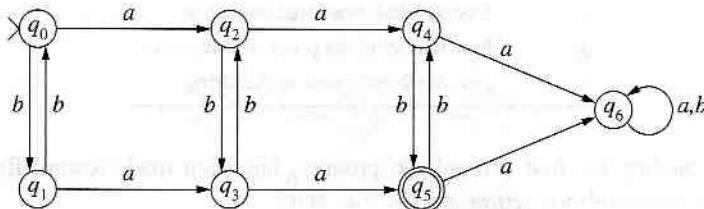
On processing aaa , the computation enters q_3 , reads the remainder of the string, and accepts the input. \square

Example 5.3.5

Building a machine that accepts strings with exactly two a 's and an odd number of b 's requires checking two conditions: the number of a 's and the parity of the b 's. Seven states are required to store the information needed about the string. The interpretation of the states describes the number of a 's read and the parity of the string processed when the computation is in the state.

State	Interpretation
q_0 :	No a 's, even number of b 's
q_1 :	No a 's, odd number of b 's
q_2 :	One a , even number of b 's
q_3 :	One a , odd number of b 's
q_4 :	Two a 's, even number of b 's
q_5 :	Two a 's, odd number of b 's
q_6 :	More than two a 's

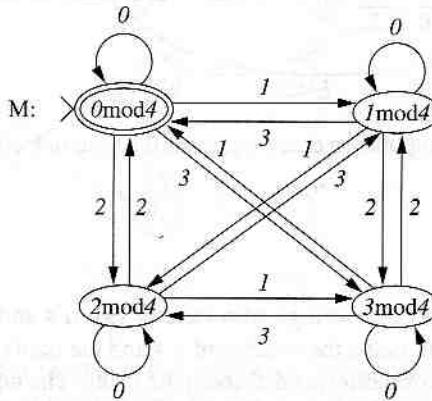
At the beginning of a computation, no a 's and no b 's have been processed and this becomes the condition of the start state. A DFA accepting this language is



The horizontal arcs count the number of a 's in the input string and the vertical pairs of arcs record the parity of the b 's. The accepting state is q_5 , since it represents the condition required of a string in the language. \square

Example 5.3.6

Let $\Sigma = \{0, 1, 2, 3\}$. A string in Σ^* is a sequence of integers from Σ . The DFA



determines whether the sum of integers in an input string is divisible by four. For example, the strings 12302 and 0130 are accepted and 0111 rejected by M . The states represent the value of the sum of the processed input modulo 4. \square

Our definition of DFA allowed only two possible outputs, accept or reject. The definition of output can be extended to have a value associated with each state. The result of a computation is the value associated with the state in which the computation terminates. A machine of this type is called a *Moore machine* after E. F. Moore, who introduced this type of finite-state computation. Associating the value i with the state $i \bmod 4$, the machine in Example 5.3.6 acts as a modulo 4 adder.

The state diagrams for machines in Examples 5.3.1, 5.3.2, and 5.3.3 showed that there is no simple method to obtain a DFA that accepts the union of two languages from DFAs

that accept each
for machines
easily be trans-
the strings rejec-

Example 5.3.7

The DFA M ac-
number of a 's a

At any step of
symbols process-
and odd number
 a 's and odd num-
first component
 b 's that have been
the appropriate t

Example 5.3.8

Let M be the DF
strings over $\{a, b\}$
other words, $L(M)$
versa. A state dia-

the accepting an

becomes

that accept each of the languages. The next two examples show that this is not the case for machines that accept complementary sets of strings. The state diagram for a DFA can easily be transformed into the state diagram for another machine that accepts all, and only, the strings rejected by the original DFA.

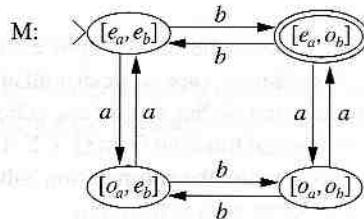
Example 5.3.7

The DFA M accepts the language consisting of all strings over $\{a, b\}$ that contain an even number of a 's and an odd number of b 's.

1 pairs of condition

□

and b is
and
the



At any step of the computation, there are four possibilities for the parities of the input symbols processed: (1) even number of a 's and even number of b 's, (2) even number of a 's and odd number of b 's, (3) odd number of a 's and even number of b 's, (4) odd number of a 's and odd number of b 's. These four states are represented by ordered pairs in which the first component indicates the parity of the a 's and the second component, the parity of the b 's that have been processed. Processing a symbol changes one of the parities, designating the appropriate transition. □

Example 5.3.8

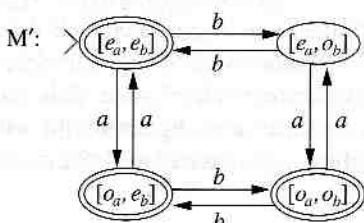
example,
s represent

□

. The defi-
ne result of
terminates.
roduced this
he machine

d that there
from DFAs

Let M be the DFA constructed in Example 5.3.7. A DFA M' is constructed that accepts all strings over $\{a, b\}$ that do not contain an even number of a 's and an odd number of b 's. In other words, $L(M') = \{a, b\}^* - L(M)$. Any string rejected by M is accepted by M' and vice versa. A state diagram for the machine M' can be obtained from that of M by interchanging the accepting and nonaccepting states.



□

The preceding example shows the relationship between DFAs that accept complementary sets of strings. This relationship is formalized by the following result.

Theorem 5.3.3

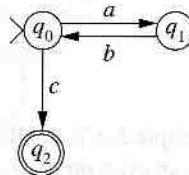
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Then $M' = (Q, \Sigma, \delta, q_0, Q - F)$ is a DFA with $L(M') = \Sigma^* - L(M)$.

Proof. Let $w \in \Sigma^*$ and $\hat{\delta}$ be the extended transition function constructed from δ . For each $w \in L(M)$, $\hat{\delta}(q_0, w) \in F$. Hence, $w \notin L(M')$. Conversely, if $w \notin L(M)$, then $\hat{\delta}(q_0, w) \in Q - F$ and $w \in L(M')$. ■

By definition, a DFA must process the entire input even if the result has already been established. Example 5.3.9 exhibits a type of determinism, sometimes referred to as *incomplete determinism*; each configuration has at most one action specified. The transitions of such a machine are defined by a partial function from $Q \times \Sigma$ to Q . As soon as it is possible to determine that a string is not acceptable, the computation halts. A computation that halts before processing the entire input string rejects the input.

Example 5.3.9

The state diagram below defines an incompletely specified DFA that accepts $(ab)^*c$. A computation terminates unsuccessfully as soon as the input varies from the desired pattern.



The computation with input $abcc$ is rejected since the machine is unable to process the final c from state q_2 . □

Two machines that accept the same language are called *equivalent*. An incompletely specified DFA can easily be transformed into an equivalent DFA. The transformation requires the addition of a nonaccepting “error” state. This state is entered whenever the incompletely specified machine enters a configuration for which no action is indicated. Upon entering the error state, the computation of the DFA reads the remainder of the string and halts.

Example 5.3.10

The DFA

accepts the same language
 q_e is the error state

Example 5.3.11

The incomplete DFA



accepts the language
number of 'a's,
be extended to a
the next chapter
automaton.

5.4 Nondeterministic Finite Automata

We now alter our deterministic automata to give a given machine more power than accepting machines, the nondeterministic acceptors.

plement-

d's not

get close

again will

DFA with

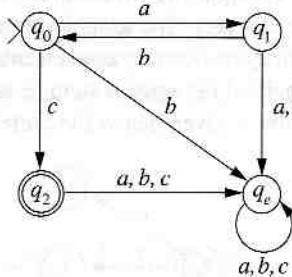
defined

For each
 $(q_0, w) \in$

■

s already
refered to as
transitions
possible
that halts**Example 5.3.10**

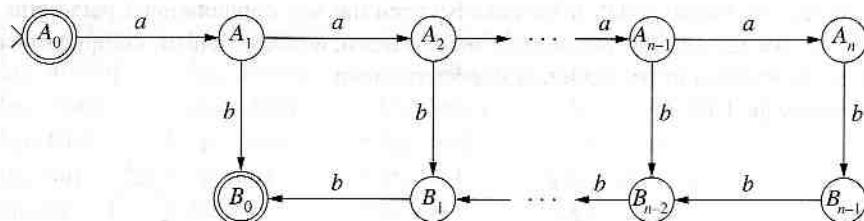
The DFA



accepts the same language as the incompletely specified DFA in Example 5.3.9. The state q_e is the error state that ensures the processing of the entire string. \square

Example 5.3.11

The incompletely specified DFA defined by the state diagram



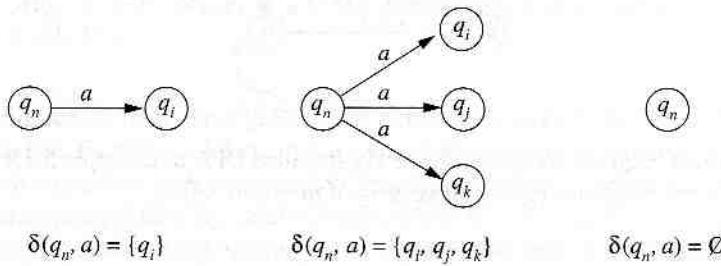
accepts the language $\{a^i b^i \mid i \leq n\}$, for a fixed integer n . The states labeled A_k count the number of a 's, and then the B_k 's ensure an equal number of b 's. This technique cannot be extended to accept $\{a^i b^i \mid i \geq 0\}$ since an infinite number of states would be needed. In the next chapter we will show that the language $\{a^i b^i \mid i \geq 0\}$ is not accepted by any finite automaton. \square

is the final

 \square completely
formation
enever the
indicated.
f the string**5.4 Nondeterministic Finite Automata**

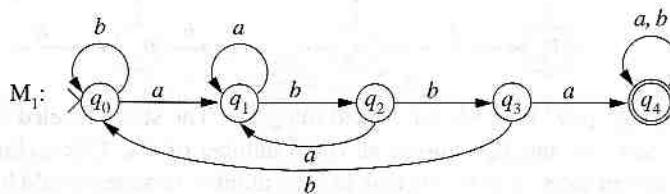
We now alter our definition of machine to allow nondeterministic computations. In a nondeterministic automaton there may be several instructions that can be executed from a given machine configuration. Although this property may seem unnatural for computing machines, the flexibility of nondeterminism often facilitates the design of language acceptors.

A transition in a *nondeterministic finite automaton* (*NFA*) has the same effect as one in a DFA: to change the state of the machine based upon the current state and the symbol being scanned. The transition function must specify all possible states that the machine may enter from a given machine configuration. This is accomplished by having the value of the transition function be a set of states. The graphic representation of state diagrams is used to illustrate the alternatives that can occur in nondeterministic computation. Any finite number of transitions may be specified for a given state q_n and symbol a . The value of the nondeterministic transition function is given below the corresponding diagram.



Because nondeterministic computation differs significantly from its deterministic counterpart, we begin the presentation of nondeterministic machines with an example that demonstrates the fundamental differences between the two computational paradigms. In addition, we use the example to introduce the features of nondeterministic computation and to present an intuitive interpretation of nondeterminism.

Consider the DFA M_1



that accepts $(a \cup b)^*abba(a \cup b)^*$, the strings over $\{a, b\}$ that contain the substring $abba$. The states q_0, q_1, q_2, q_3 record the progress toward obtaining the substring $abba$. The states of the machine are

State	Interpretation
q_0 :	When there is no progress toward $abba$
q_1 :	When the last symbol processed was an a
q_2 :	When the last two symbols processed were ab
q_3 :	When the last three symbols processed were abb

Upon processing a , the string is accepted.

The determinist machine processes the current substring in the string. The machine is in state q_0 at the current configuration.

A nondeterministic machine

M_2 :

There are two possibilities for M_2 to continue. The states q_1, q_2, q_3 to continue.

The first thing to do is to compute the computations for $aabbba$. We will try

$[q_0, aabbba]$
 $\vdash [q_0, abbaa]$
 $\vdash [q_0, bbaa]$
 $\vdash [q_0, baa]$
 $\vdash [q_0, aa]$
 $\vdash [q_0, a]$
 $\vdash [q_0, \lambda]$

What does it mean for a string to be in an accepting state? If the condition is satisfied, the computation is accepted. A string is accepted if it is at least one computation.

- i) processes the entire string
- ii) halts in an accepting state

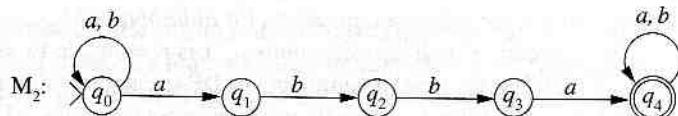
A string is in the language if the machine accepts it; the existence of such a computation is what matters.

is one symbol machine value ams is finite of the

Upon processing $abba$, state q_4 is entered, the remainder of the string is read, and the input is accepted.

The deterministic computation must “back up” in the sequence q_0, q_1, q_2, q_3 when the current substring is discovered not to have the desired form. If a b is scanned when the machine is in state q_3 , then q_0 is entered since the last four symbols processed are $abbb$ and the current configuration represents no progress toward finding $abba$.

A nondeterministic approach to accepting $(a \cup b)^*abba(a \cup b)^*$ is illustrated by the machine



There are two possible transitions when M_2 processes an a in state q_0 . One possibility is for M_2 to continue reading the string in state q_0 . The second option enters the sequence of states q_1, q_2, q_3 to check if the next three symbols complete the substring $abba$.

The first thing to observe is that with a nondeterministic machine, there may be multiple computations for an input string. For example, M_2 has five different computations for string $aabbba$. We will trace the computations using the \vdash notation introduced in Section 5.2.

$[q_0, aabbba]$	$[q_0, aabbba]$	$[q_0, aabbba]$	$[q_0, aabbba]$	$[q_0, aabbba]$
$\vdash [q_0, abbaa]$	$\vdash [q_0, abbaa]$	$\vdash [q_0, abbaa]$	$\vdash [q_0, abbaa]$	$\vdash [q_1, abbaa]$
$\vdash [q_0, bbaa]$	$\vdash [q_0, bbaa]$	$\vdash [q_0, bbaa]$	$\vdash [q_1, bbaa]$	
$\vdash [q_0, baa]$	$\vdash [q_0, baa]$	$\vdash [q_0, baa]$	$\vdash [q_2, baa]$	
$\vdash [q_0, aa]$	$\vdash [q_0, aa]$	$\vdash [q_0, aa]$	$\vdash [q_3, aa]$	
$\vdash [q_0, a]$	$\vdash [q_0, a]$	$\vdash [q_1, a]$	$\vdash [q_4, a]$	
$\vdash [q_0, \lambda]$	$\vdash [q_1, \lambda]$			$\vdash [q_4, \lambda]$

What does it mean for a string to be accepted when there are some computations that halt in an accepting state and others that halt in a rejecting state? The answer lies in the use of the word *check* in the preceding paragraph. An NFA is designed to check whether a condition is satisfied, in this case, whether the input string has a substring $abba$. If one of the computations discovers the presence of the substring, the condition is satisfied and the string is accepted. As with incompletely specified DFAs, it is necessary to read the entire string to receive an affirmative answer. Summing up, a string is accepted by an NFA if there is at least one computation that

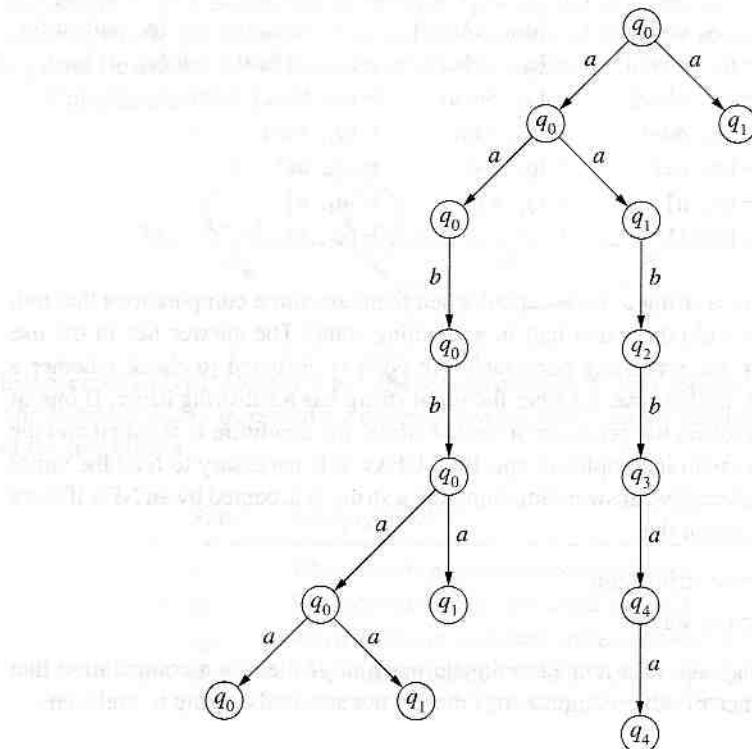
- i) processes the entire string, and
- ii) halts in an accepting state.

A string is in the language of a nondeterministic machine if there is a computation that accepts it; the existence of other computations that do not accept the string is irrelevant.

Nondeterministic machines are frequently designed to employ a “guess and check” strategy. The transition from q_0 to q_1 in M_2 represents the guess that the a being read is the first symbol in the substring $abba$. After the guess, the computation continues to states q_1 , q_2 , and q_3 to check whether the guess is correct. If symbols following the guess are bba , the string is accepted.

If an input string has the substring $abba$, one of the guesses will cause M_2 to enter state q_1 upon reading the initial a in the substring, and this computation accepts the string. Moreover, M_2 enters q_4 only upon processing $abba$. Consequently, the language of M_2 is $(a \cup b)^*abba(a \cup b)^*$. It should be noted that accepting computations are not necessarily unique; there are two distinct accepting computations for $abbabba$ in M_2 .

If this is your first encounter with nondeterminism, it is reasonable to ask about the ability of a machine to perform this type of computation. DFAs can be easily implemented in either software or hardware. What is the analogous implementation for NFAs? We can intuitively imagine nondeterministic computation as a type of multiprocessing. When the computation enters a machine configuration for which there are multiple transitions, a new process is generated for each alternative. With this interpretation, a computation produces a tree of processes running in parallel with the branching generated by the multiple choices in the NFA. The tree corresponding to the computation of $aabbaaa$ is



If one of the branches is accepted and the entire computation terminates without reaching a final state, the string is rejected.

Having introduced the basic idea of nondeterminism with an example, we now turn to state diagrams, and their formal definition. The components of an NFA are identified in the following definitions.

Definition 5.4.1

A nondeterministic finite automaton (NFA) is a quadruple (Q, Σ, δ, q_0) , where Q is a finite set of states, Σ is a finite set of symbols called the *alphabet*, δ is a function from $Q \times \Sigma$ to $\mathcal{P}(Q)$ called the *transition function*, and $q_0 \in Q$ is the *start state*.

Definition 5.4.2

The language of an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is the set of strings $L(M) = \{w \mid \text{there is a computation } q_0 \xrightarrow{*} q \text{ such that } q \in F\}$.

Definition 5.4.3

The state diagram of an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is a directed graph with nodes labeled by states in Q and edges labeled by transitions in δ . The start state q_0 is indicated by a double circle. The final states F are indicated by a single circle.

- i) The nodes of δ are the states of M .
- ii) The labels on the edges of δ are the symbols in Σ .
- iii) q_0 is the start node.
- iv) F is the set of accept states.
- v) There is an arc from q_i to q_j if $(q_i, q_j) \in \delta$.

The relationship between the state diagram and the state transition function is given by omitting condition v) from the definition of the state transition function into its graph representation.

The relationship between the state diagram and the state transition function is given by the following definition. We then consider the relationship between the state transition function and the state diagram.

The following definition is based on the intuitive phrase, “Every computation of a DFA is a computation of an NFA.” The state transition function of a DFA is a function from the states to the symbols in the alphabet, while an NFA’s state transition function is a function from the states to sets of symbols in the alphabet. This may be considered to be a generalization of the DFA state transition function.

The following definition is based on the intuitive phrase, “Every computation of a DFA is a computation of an NFA.”

The following definition is based on the intuitive phrase, “Every computation of a DFA is a computation of an NFA.”

The following definition is based on the intuitive phrase, “Every computation of a DFA is a computation of an NFA.”

The following definition is based on the intuitive phrase, “Every computation of a DFA is a computation of an NFA.”

I check"
ad is the
states q_1 ,
are bba ,

to enter
ie string.
of M_2 is
cessarily

about the
lemented
? We can
When the
ns, a new
produces
e choices

If one of the branches reads the entire string and halts in an accepting state, the input is accepted and the entire computation terminates. The input is rejected only when all branches terminate without accepting the string.

Having introduced the properties of nondeterministic computation in the preceding example, we now present the formal definitions of nondeterministic machines, their state diagrams, and their languages. With the exception of the transition function, the components of an NFA are identical to those of a DFA.

Definition 5.4.1

A **nondeterministic finite automaton** (NFA) is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ a finite set called the *alphabet*, $q_0 \in Q$ a distinguished state known as the *start state*, F a subset of Q called the *final* or *accepting states*, and δ a total function from $Q \times \Sigma$ to $\mathcal{P}(Q)$ known as the *transition function*.

Definition 5.4.2

The language of an NFA M , denoted $L(M)$, is the set of strings accepted by the M . That is, $L(M) = \{w \mid \text{there is a computation } [q_0, w] \xrightarrow{*} [q_i, \lambda] \text{ with } q_i \in F\}$.

Definition 5.4.3

The **state diagram** of an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is a labeled directed graph G defined by the following conditions:

- i) The nodes of G are elements of Q .
- ii) The labels on the arcs of G are elements of Σ .
- iii) q_0 is the start node.
- iv) F is the set of accepting nodes.
- v) There is an arc from node q_i to q_j labeled a , if $q_j \in \delta(q_i, a)$.

The relationship between DFAs and NFAs is clearly exhibited by comparing the properties of the corresponding state diagrams. Definition 5.4.3 is obtained from Definition 5.3.1 by omitting condition (vi), which translates the deterministic property of the DFA transition function into its graphic representation.

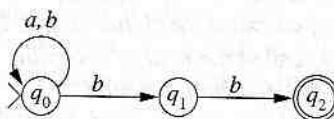
The relationship between DFAs and NFAs can be summarized by the seemingly paradoxical phrase, "Every deterministic finite automaton is nondeterministic." The transition function of a DFA specifies exactly one transition for each combination of state and input symbol, while an NFA allows zero, one, or more transitions. By interpreting the transition function of a DFA as a function from $Q \times \Sigma$ to singleton sets of states, the family of DFAs may be considered to be a subset of the family of NFAs.

The following example describes an NFA in terms of the components in the formal definition. We then construct the corresponding state diagram using the technique outlined in Definition 5.4.3.

Example 5.4.1

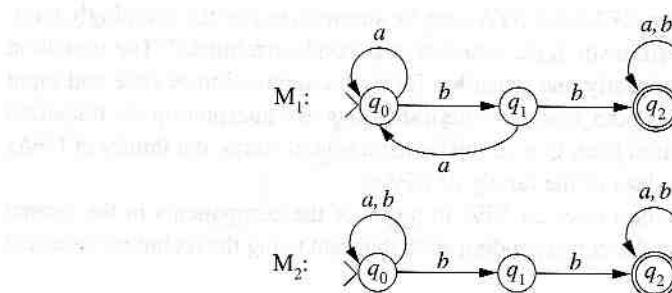
The NFA

$M : Q = \{q_0, q_1, q_2\}$	δ	a	b
$\Sigma = \{a, b\}$	q_0	$\{q_0\}$	$\{q_0, q_1\}$
$F = \{q_2\}$	q_1	\emptyset	$\{q_2\}$
	q_2	\emptyset	\emptyset

with start state q_0 accepts the language $(a \cup b)^*bb$. The state diagram of M isPictorially, it is clear that a string is accepted if, and only if, it ends with the substring bb .As noted previously, an NFA may have multiple computations for an input string. The three computations for the string $ababb$ are

$[q_0, ababb]$	$[q_0, ababb]$	$[q_0, ababb]$
$\vdash [q_0, babb]$	$\vdash [q_0, babb]$	$\vdash [q_0, babb]$
$\vdash [q_0, abb]$	$\vdash [q_1, abb]$	$\vdash [q_0, abb]$
$\vdash [q_0, bb]$		$\vdash [q_0, bb]$
$\vdash [q_0, b]$		$\vdash [q_1, b]$
$\vdash [q_0, \lambda]$		$\vdash [q_2, \lambda]$

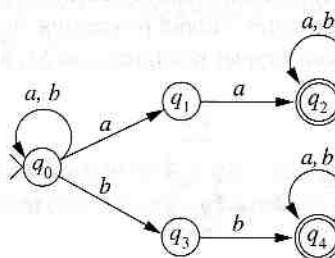
The second computation halts after the execution of three instructions since no action is specified when the machine is in state q_1 scanning an a . The first computation processes the entire input and halts in a rejecting state while the final computation halts in an accepting state. The third computation demonstrates that $ababb$ is in the language of machine M . \square

Example 5.4.2The state diagrams M_1 and M_2 define finite automata that accept $(a \cup b)^*bb(a \cup b)^*$.

M_1 is the DFA from Example 5.3.1. The path exhibiting the acceptance of strings by M_1 enters q_2 when the first substring bb is encountered. M_2 can enter the accepting state upon processing any occurrence of bb . \square

Example 5.4.3

An NFA that accepts strings over $\{a, b\}$ with the substring aa or bb can be constructed by combining a machine that accepts strings with bb (Example 5.4.2) with a similar machine that accepts strings with aa .



A path exhibiting the acceptance of a string reads the input in state q_0 until an occurrence of the substring aa or bb is encountered. At this point, the path branches to either q_1 or q_3 , depending upon the substring. There are three distinct paths that exhibit the acceptance of the string $abaaabb$. \square

string bb .
tring. The

o action is
cesses the
accepting
ine M . \square

The flexibility permitted by the use of nondeterminism does not always simplify the problem of constructing a machine that accepts $L(M_1) \cup L(M_2)$ from the machines M_1 and M_2 . This can be seen by attempting to construct an NFA that accepts the language of the DFA in Example 5.3.3.

5.5 λ -Transitions

The transitions from state to state in both deterministic and nondeterministic automata were initiated by processing an input symbol. The definition of NFA is now relaxed to allow state transitions without requiring input to be processed. A transition of this form is called a λ -transition. The class of nondeterministic machines that utilize λ -transitions is denoted NFA- λ .

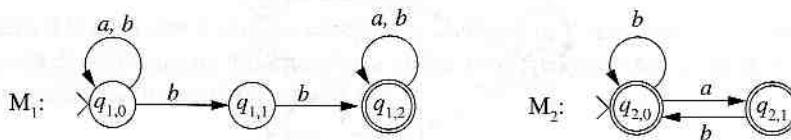
The incorporation of λ -transitions into finite state machines represents another step away from the deterministic computations of a DFA. They do, however, provide a useful tool for the design of machines to accept complex languages.

Definition 5.5.1

A nondeterministic finite automaton with λ -transitions is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q, δ, q_0 , and F are the same as in an NFA. The transition function is a function from $Q \times (\Sigma \cup \{\lambda\})$ to $\mathcal{P}(Q)$.

The definition of halting must be extended to include the possibility that a computation may continue using λ -transitions after the input string has been completely processed. Employing the criteria used for acceptance in an NFA, the input is accepted if there is a computation that processes the entire string and halts in an accepting state. As before, the language of an NFA- λ is denoted $L(M)$. The state diagram for an NFA- λ is constructed according to Definition 5.4.3 with λ -transitions represented by arcs labeled by λ .

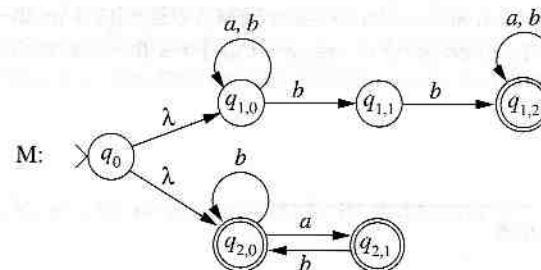
The ability to move between states without processing an input symbol can be used to construct complex machines from simpler machines. Let M_1 and M_2 be the machines



that accept $(a \cup b)^*bb(a \cup b)^*$ and $(b \cup ab)^*(a \cup \lambda)$, respectively. Composite machines are built by appropriately combining the state diagrams of M_1 and M_2 .

Example 5.5.1

The language of the NFA- λ M is $L(M_1) \cup L(M_2)$.



A computation in the composite machine M begins by following a λ -arc to the start state of either M_1 or M_2 . If the path p exhibits the acceptance of a string by machine M_i , then that string is accepted by the path in M consisting of the λ -arc from q_0 to $q_{i,0}$ followed by p in the copy of the machine M_i . Since the initial move in each computation does not process an input symbol, the language of M is $L(M_1) \cup L(M_2)$. Compare the simplicity of the machine obtained by this construction with that of the deterministic state diagram in Example 5.3.3.

□

Example 5.5.2

An NFA- λ that accepts the language of all strings over $\{a, b\}$ is easily constructed by joining the machines M_1 and M_2 .

An input string is accepted by M if it is accepted by M_1 and M_2 . The language of M is the union of the languages of M_1 and M_2 .

Example 5.5.3

We will use λ -transitions to construct a machine that accepts the language $\{a^*b^*\}$. We begin by defining the machines M_1 and M_2 .

To accept the null string, we will add a new start state q_0' and a self-loop transition at q_0' labeled a . The strings a^*b^* are accepted by following a sequence of a 's and b 's.

The construction of the NFA- λ M is similar to the construction of the NFA- λ in Example 5.5.2. The machine M accepts the language $\{a^*b^*\}$ because it is equivalent to the NFA- λ in Example 5.5.2.

Lemma 5.5.2

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA- λ and let $M' = (Q', \Sigma, \delta', q'_0, F')$ be an NFA. Then $L(M) = L(M')$ if and only if M and M' are equivalent.

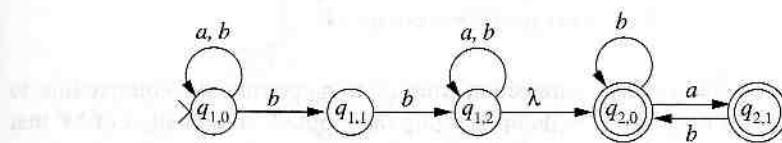
- i) The in-degree of every state in M is equal to the in-degree of the corresponding state in M' .
- ii) The only accepting states in M are the corresponding accepting states in M' .
- iii) The out-degree of every state in M is equal to the out-degree of the corresponding state in M' .

Example 5.5.2

$(Q, \Sigma,$
action is

putation
processed.
here is a
fore, the
constructed
e used to
ines

An NFA- λ that accepts $L(M_1)L(M_2)$, the concatenation of the languages of M_1 and M_2 , is constructed by joining the two machines with a λ -arc.



An input string is accepted only if it consists of a string from $L(M_1)$ concatenated with one from $L(M_2)$. The λ -transition allows the computation to enter M_2 whenever a prefix of the input string is accepted by M_1 . \square

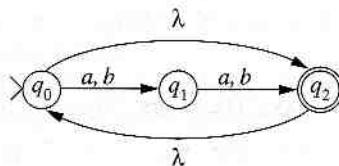
Example 5.5.3

We will use λ -transitions to construct an NFA- λ that accepts all strings of even length over $\{a, b\}$. We begin by building the state diagram of a machine that accepts strings of length two.

chines are



To accept the null string, a λ -arc is added from q_0 to q_2 . Strings of any positive, even length are accepted by following the λ -arc from q_2 to q_0 to repeat the sequence q_0, q_1, q_2 . \square



The constructions presented in Examples 5.5.1, 5.5.2, and 5.5.3 can be generalized to construct machines that accept the union, concatenation, and Kleene star of languages accepted by existing finite-state machines. The first step is to transform the machines into an equivalent NFA- λ whose form is amenable to these constructions.

Lemma 5.5.2

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA- λ . There is an equivalent NFA- λ $M' = (Q' \cup \{q'_0, q'_f\}, \Sigma, \delta', q'_0, \{q'_f\})$ that satisfies the following conditions:

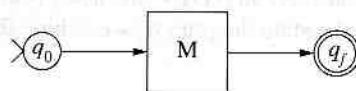
- i) The in-degree of the start state q'_0 is zero.
- ii) The only accepting state of M' is q'_f .
- iii) The out-degree of the accepting state q'_f is zero.

Proof. The transition function of M' is constructed from that of M by adding the λ -transitions

$$\begin{aligned}\delta(q'_0, \lambda) &= \{q_0\} \\ \delta(q_i, \lambda) &= \{q_f\} \text{ for every } q_i \in F\end{aligned}$$

for the new states q'_0 and q_f . The λ -transition from q'_0 to q_0 permits the computation to proceed to the original machine M without affecting the input. A computation of M' that accepts an input string is identical to that of M followed by a λ -transition from the accepting state of M to the accepting state q_f of M' . ■

If a machine satisfies the conditions of Lemma 5.5.2, the sole role of the start state is to initiate a computation, and the computation terminates as soon as q_f is entered. Such a machine can be pictured as



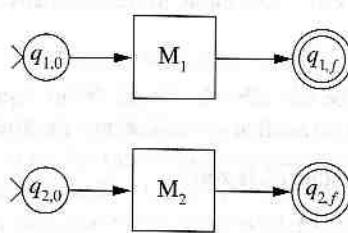
The diagram depicts a machine with three distinct parts: the initial state, the body of the machine, and the final state. This can be likened to a railroad car with couplers on either end. Indeed, the conditions on the start and final state are designed to allow them to act as couplers of finite-state machines.

Theorem 5.5.3

Let M_1 and M_2 be two NFA- λ s. There are NFA- λ s that accept $L(M_1) \cup L(M_2)$, $L(M_1)L(M_2)$, and $L(M_1)^*$.

Proof. We assume, without loss of generality, that M_1 and M_2 satisfy the conditions of Lemma 5.5.2. The machines constructed to accept the languages $L(M_1) \cup L(M_2)$, $L(M_1)L(M_2)$, and $L(M_1)^*$ will also satisfy the conditions of Lemma 5.5.2.

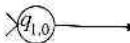
Because of the restrictions on the start and final states, M_1 and M_2 may be depicted



The language $L(M)$

A computation begins at the start state of these machines, then continues through the body of the machine. This constitutes a computation of each machine in parallel. The input string is processed by both machines simultaneously.

Concatenation of languages is depicted by connecting the final state of one machine to the start state of the other. The start state of the concatenated machine is the start state of the first machine, and the final state is the final state of the second machine. The two machines are joined by connecting their final states to the start state of the second machine.



When a prefix of the input string is processed by the first machine, the remainder of the string is processed by the second machine. If the remainder of the string is accepted by the second machine, then the entire string is accepted by the concatenated machine.

A machine that accepts the language $L(M_1) \cup L(M_2)$ is depicted by connecting the start states of M_1 and M_2 to a common final state. The λ -arc from $q_{1,0}$ to $q_{2,0}$ is labeled λ . The λ -arc from $q_{2,0}$ to the final state is labeled λ . The final state is labeled $q_{1,f} \cup q_{2,f}$.



The ability to represent the union of languages is established by connecting the start states of the two machines to a common final state. This allows the two machines to be combined into a single machine that accepts the union of their languages.

lding the λ -

mputation to
on of M' that
the accepting

e start state is
tered. Such a

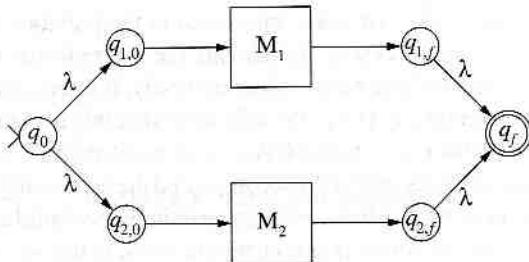
ie body of the
plers on either
them to act as

$(M_1) \cup L(M_2)$,

the conditions
 $(M_1) \cup L(M_2)$,

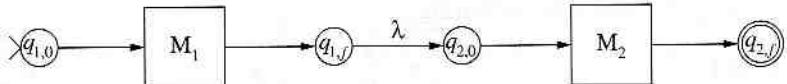
y be depicted

The language $L(M_1) \cup L(M_2)$ is accepted by



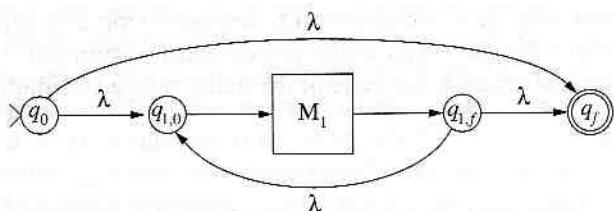
A computation begins by following a λ -arc to M_1 or M_2 . If the string is accepted by either of these machines, the λ -arc can be traversed to reach the accepting state of the composite machine. This construction may be thought of as building a machine that runs M_1 and M_2 in parallel. The input is accepted if either of the machines successfully processes the string.

Concatenation can be obtained by operating the component machines sequentially. The start state of the composite machine is $q_{1,0}$ and the accepting state is $q_{2,f}$. The machines are joined by connecting the final state of M_1 to the start state of M_2 .



When a prefix of the input string is accepted by M_1 , the computation continues with M_2 . If the remainder of the string is accepted by M_2 , the processing terminates in $q_{2,f}$, the accepting state of the composite machine.

A machine that accepts $L(M_1)^*$ must be able to cycle through M_1 any number of times. The λ -arc from $q_{1,f}$ to $q_{1,0}$ permits the necessary cycling. Another λ -arc is added from $q_{1,0}$ to $q_{1,f}$ to accept the null string. These arcs are added to M_1 producing



The ability to repeatedly connect machines of this form will be used in Chapter 6 to establish the equivalence of languages described by regular expressions and accepted by finite-state machines.

5.6 Removing Nondeterminism

Three classes of finite automata have been introduced in the previous sections, each class being a generalization of its predecessor. By relaxing the deterministic restriction, have we created a more powerful class of machines? More precisely, is there a language accepted by an NFA that is not accepted by any DFA? We will show that this is not the case. Moreover, an algorithm is presented that converts an NFA- λ to an equivalent DFA.

The state transitions in DFAs and NFAs accompanied the processing of an input symbol. To relate the transitions in an NFA- λ to the processing of input, we build a modified transition function t , called the *input transition function*, whose value is the set of states that can be entered by processing a single input symbol from a given state. The value of $t(q_1, a)$ for the diagram in Figure 5.3 is the set $\{q_2, q_3, q_5, q_6\}$. State q_4 is omitted since the transition from state q_1 does not process an input symbol.

Intuitively, the definition of the input transition function $t(q_i, a)$ can be broken into three parts. First, the set of states that can be reached from q_i without processing a symbol is constructed. This is followed by processing an a from all the states in that set. Finally, following λ -arcs from the resulting states yields the set $t(q_i, a)$.

The function t is defined in terms of the transition function δ and the paths in the state diagram that spell the null string. A node q_j is said to be in the λ -closure of q_i if there is a path from q_i to q_j that spells the null string.

Definition 5.6.1

The λ -closure of a state q_i , denoted λ -closure(q_i), is defined recursively by

- i) Basis: $q_i \in \lambda$ -closure(q_i).
- ii) Recursive step: Let q_j be an element of λ -closure(q_i). If $q_k \in \delta(q_j, \lambda)$, then $q_k \in \lambda$ -closure(q_i).
- iii) Closure: q_j is in λ -closure(q_i) only if it can be obtained from q_i by a finite number of applications of the recursive step.

The set λ -closure(q_i) can be constructed following the top-down approach used in Algorithm 4.3.1, which determined the chains in a context-free grammar. The input transition function is obtained from the λ -closure of the states and the transition function of the NFA- λ .

Definition 5.6.2

The input transition function t of an NFA- λ M is a function from $Q \times \Sigma$ to $\mathcal{P}(Q)$ defined by

$$t(q_i, a) = \bigcup_{q_j \in \lambda\text{-closure}(q_i)} \lambda\text{-closure}(\delta(q_j, a)),$$

where δ is the transition function of M .

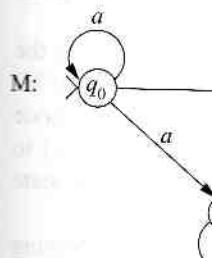
Path

q_1, q_2
q_1, q_2, q_3
q_1, q_4
q_1, q_4, q_5
q_1, q_4, q_5, q_6

The input trans...
That is, it is a func...
input transition fun...

Example 5.6.1

Transition tables a...
of the NFA- λ with

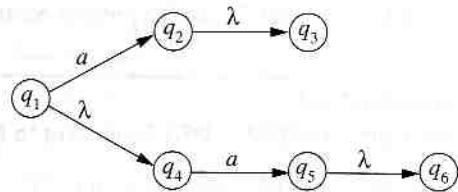


The input trans...
Acceptance in a non...
that processes the e...
in the state diagram
state diagram of a D...

Algorithm 5.6.3
alent to an NFA- λ M
are sets of nodes of M
to the algorithm is st...
is a node in DM, the
processing the symb...
state diagram of DM

String
Path

q_1, q_2	a
q_1, q_2, q_3	a
q_1, q_4	λ
q_1, q_4, q_5	a
q_1, q_4, q_5, q_6	a

FIGURE 5.3 Paths with λ -transitions.

The input transition function has the same form as the transition function of an NFA. That is, it is a function from $Q \times \Sigma$ to sets of states. For an NFA without λ -transitions, the input transition function t is identical to the transition function δ of the automaton.

Example 5.6.1

Transition tables are given for the transition function δ and the input transition function t of the NFA- λ with state diagram M. The language of M is $a^+c^*b^*$.

M:

```

graph LR
    q0((q0)) -- a --> q1((q1))
    q0 -- a --> q2((q2))
    q1 -- λ --> q2
    q2 -- c --> q1
  
```

δ	a	b	c	λ
q_0	$\{q_0, q_1, q_2\}$	\emptyset	\emptyset	\emptyset
q_1	\emptyset	$\{q_1\}$	\emptyset	\emptyset
q_2	\emptyset	\emptyset	$\{q_2\}$	$\{q_1\}$

t	a	b	c
q_0	$\{q_0, q_1, q_2\}$	\emptyset	\emptyset
q_1	\emptyset	$\{q_1\}$	\emptyset
q_2	\emptyset	$\{q_1\}$	$\{q_1, q_2\}$

□

The input transition function of an NFA- λ is used to construct an equivalent DFA. Acceptance in a nondeterministic machine is determined by the existence of a computation that processes the entire string and halts in an accepting state. There may be several paths in the state diagram of an NFA- λ that represent the processing of an input string, while the state diagram of a DFA contains exactly one such path. To remove the nondeterminism, the DFA must simulate the simultaneous exploration of all possible computations in the NFA- λ .

Algorithm 5.6.3 iteratively builds the state diagram of a deterministic machine equivalent to an NFA- λ M. The nodes of the DFA, called DM for *deterministic equivalent of M*, are sets of nodes of M. The start node of DM is the λ -closure of the start node of M. The key to the algorithm is step 2.1.1, which generates the nodes of the deterministic machine. If X is a node in DM, the set Y is constructed that contains all the states that can be entered by processing the symbol a from any state in the set X. This relationship is represented in the state diagram of DM by an arc from X to Y labeled a . The node X is made deterministic by

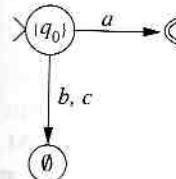
producing an arc from it for every symbol in the alphabet. New nodes generated in step 2.1.1 are added to the set Q' and the process continues until every node in Q' is deterministic.

Algorithm 5.6.3
Construction of DM, a DFA Equivalent to NFA- λ M

input: an NFA- λ M = $(Q, \Sigma, \delta, q_0, F)$

 input transition function t of M

1. initialize Q' to λ -closure(q_0)
 2. repeat
 - 2.1. if there is a node $X \in Q'$ and a symbol $a \in \Sigma$ with no arc leaving X labeled a , then
 - 2.1.1. let $Y = \bigcup_{q_i \in X} t(q_i, a)$
 - 2.1.2. if $Y \notin Q'$, then set $Q' := Q' \cup \{Y\}$
 - 2.1.3. add an arc from X to Y labeled a
 - 2.2. else done := true
 3. until done
 3. the set of accepting states of DM is $F' = \{X \in Q' \mid X \text{ contains an element } q_i \in F\}$
-



(a)

The NFA- λ from Example 5.6.1 is used to illustrate the construction of nodes for the equivalent DFA. The start node of DM is the singleton set containing the start node of M. A transition from q_0 processing an a can terminate in q_0 , q_1 , or q_2 . We construct a node $\{q_0, q_1, q_2\}$ for the DFA and connect it to $\{q_0\}$ by an arc labeled a . The path from $\{q_0\}$ to $\{q_0, q_1, q_2\}$ in DM represents the three possible ways of processing the symbol a from state q_0 in M.

Since DM is to be deterministic, the node $\{q_0\}$ must have arcs labeled b and c leaving it. Arcs from q_0 to \emptyset labeled b and c are added to indicate that there is no action specified by the NFA- λ when the machine is in state q_0 scanning these symbols.

The node $\{q_0\}$ has the deterministic form; there is exactly one arc leaving it for every member of the alphabet. Figure 5.4(a) shows DM at this stage of its construction. Two additional nodes, $\{q_0, q_1, q_2\}$ and \emptyset , have been created. Both of these must be made deterministic.

An arc leaving node $\{q_0, q_1, q_2\}$ terminates in a node consisting of all the states that can be reached by processing the input symbol from the states q_0 , q_1 , or q_2 in M. The input transition function $t(q_i, a)$ specifies the states reachable by processing an a from q_i . The arc from $\{q_0, q_1, q_2\}$ labeled a terminates in the set consisting of the union of the $t(q_0, a)$, $t(q_1, a)$, and $t(q_2, a)$. The set obtained from this union is again $\{q_0, q_1, q_2\}$. An arc from $\{q_0, q_1, q_2\}$ to itself is added to the diagram designating this transition.

The empty set represents an error state for DM. A computation enters \emptyset on reading an a in state Y only if there is no transition for a for any $q_i \in Y$. Once in \emptyset , the computation

processes the remaining symbols in the input string according to the transition diagram by the algorithm.

Figure 5.4(b) shows the state transition diagram of the nondeterministic finite automaton M. The diagram shows the acceptance of the string $aabb$. The computation terminates in state q_1 of M.

The algorithm for constructing the deterministic finite automaton DM involves adding arcs to make the states deterministic. New nodes may be created to handle transitions from non-deterministic states. The nodes are determined by the sets of states they represent. All possible transitions for each state in the new nodes must be constructed. A lower bound on the number of states in the deterministic finite automaton is given by the number of states in the NFA- λ plus the number of states in the sets of states that represent the transitions from non-deterministic states. The equivalence of M and DM is established by showing that they accept the same language.

ep 2.1.1
istic.

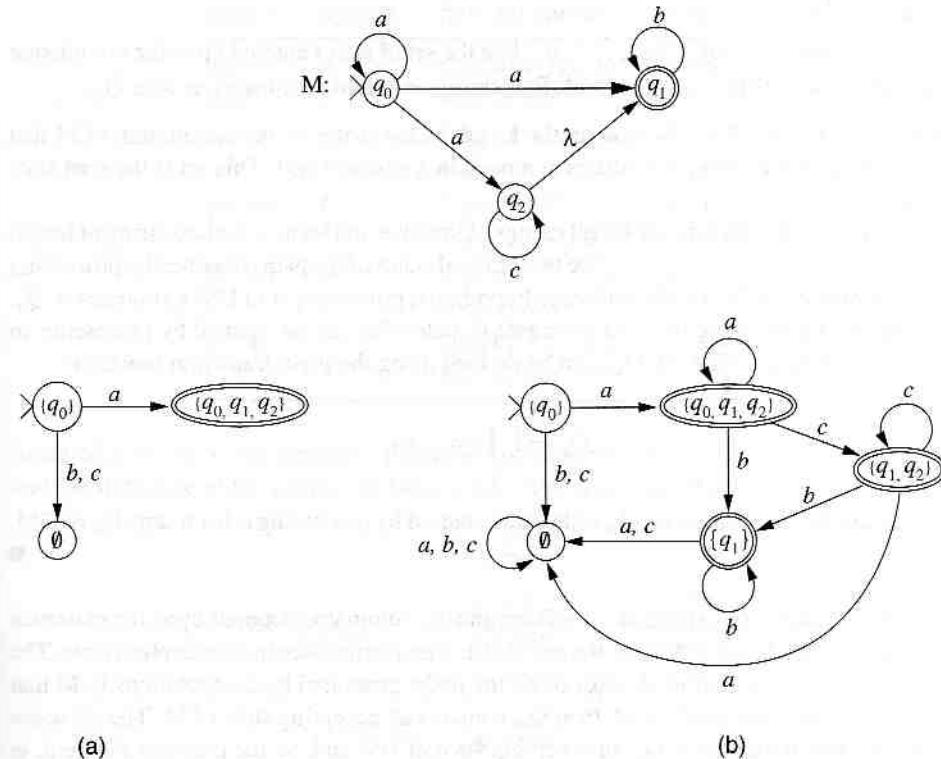


FIGURE 5.4 Construction of equivalent deterministic automaton.

F}

es for the
ode of M.
ct a node
m $\{q_0\}$ to
rom state

c leaving
specified

ing it for
struction.
t be made

states that
n M. The
an a from
ion of the
 $\{q_1, q_2\}$. An

reading an
mputation

processes the remainder of the input and rejects the string. This is indicated in the state diagram by the arc from \emptyset to itself labeled by each alphabet symbol.

Figure 5.4(b) gives the completed deterministic equivalent of the M. Computations of the nondeterministic machine with input aaa can terminate in state q_0 , q_1 , and q_2 . The acceptance of the string is exhibited by the path that terminates in q_1 . Processing aaa in DM terminates in state $\{q_0, q_1, q_2\}$. This state is accepting in DM since it contains the accepting state q_1 of M.

The algorithm for constructing the deterministic state diagram consists of repeatedly adding arcs to make the nodes in the diagram deterministic. As arcs are constructed, new nodes may be created and added to the diagram. The procedure terminates when all the nodes are deterministic. Since each node is a subset of Q, at most $\text{card}(\mathcal{P}(Q))$ nodes can be constructed. Algorithm 5.6.3 always terminates since $\text{card}(\mathcal{P}(Q))\text{card}(\Sigma)$ is an upper bound on the number of iterations of the repeat-until loop. Theorem 5.6.4 establishes the equivalence of M and DM.

Theorem 5.6.4

Let $w \in \Sigma^*$ and $Q_w = \{q_{w_1}, q_{w_2}, \dots, q_{w_j}\}$ be the set of states entered upon the completion of the processing of the string w in M. Processing w in DM terminates in state Q_w .

Proof. The proof is by induction on the length of the string w . A computation of M that processes the empty string terminates at a node in $\lambda\text{-closure}(q_0)$. This set is the start state of DM.

Assume the property holds for all strings of length n and let $w = ua$ be a string of length $n + 1$. Let $Q_u = \{q_{u_1}, q_{u_2}, \dots, q_{u_k}\}$ be the terminal states of the paths obtained by processing the entire string u in M. By the inductive hypothesis, processing u in DM terminates in Q_u . Computations processing ua in M terminate in states that can be reached by processing an a from a state in Q_u . This set, Q_w , can be defined using the input transition function:

$$Q_w = \bigcup_{i=1}^k t(q_{u_i}, a).$$

This completes the proof since Q_w is the state entered by processing a from state Q_u of DM. ■

Since M is an NFA
and the start state

The acceptance of a string in a nondeterministic automaton depends upon the existence of one computation that processes the entire string and terminates in an accepting state. The node Q_w contains the terminal states of all the paths generated by computations in M that process w . If w is accepted by M, then Q_w contains an accepting state of M. The presence of an accepting node makes Q_w an accepting state of DM and, by the previous theorem, w is accepted by DM.

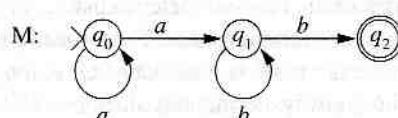
Conversely, let w be a string accepted by DM. Then Q_w contains an accepting state of M. The construction of Q_w guarantees the existence of a computation in M that processes w and terminates in that accepting state. These observations provide the justification for Corollary 5.6.5.

Corollary 5.6.5

The finite automata M and DM are equivalent.

Example 5.6.2

The NFA



accepts the language a^+b^+ . The construction of an equivalent DFA is traced in the following table.

Example 5.6.3

As seen in the preceding example, NFA's are sets of states of which has n states, the DFA

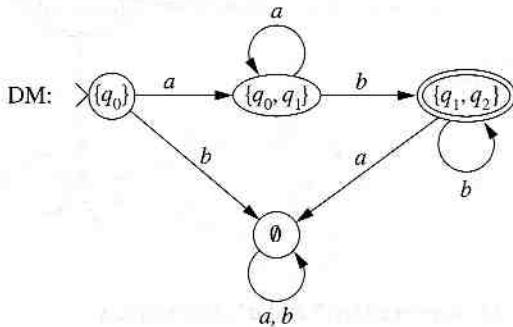
State	Symbol	NFA Transitions	Next State
$\{q_0\}$	a	$\delta(q_0, a) = \{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0\}$	b	$\delta(q_0, b) = \emptyset$	\emptyset
$\{q_0, q_1\}$	a	$\delta(q_0, a) = \{q_0, q_1\}$ $\delta(q_1, a) = \emptyset$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	b	$\delta(q_0, b) = \emptyset$ $\delta(q_1, b) = \{q_1, q_2\}$	$\{q_1, q_2\}$
$\{q_1, q_2\}$	a	$\delta(q_1, a) = \emptyset$ $\delta(q_2, a) = \emptyset$	\emptyset
$\{q_1, q_2\}$	b	$\delta(q_1, b) = \{q_1, q_2\}$ $\delta(q_2, b) = \emptyset$	$\{q_1, q_2\}$

Since M is an NFA, the transition function δ of M serves as the input transition function and the start state of the equivalent DFA is $\{q_0\}$. The resulting DFA is

Q_u of DM.

existence state. The in M that e presence theorem, w

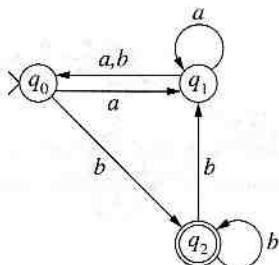
ng state of processes iation for



□

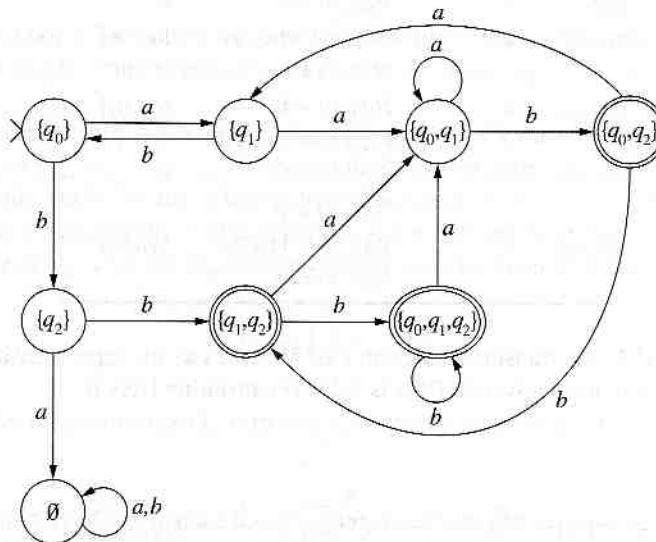
Example 5.6.3

As seen in the preceding examples, the states of the DFA constructed using Algorithm 5.6.3 are sets of states of the original nondeterministic machine. If the nondeterministic machine has n states, the DFA may have 2^n states. The transformation of the NFA



shows that the theoretical upper bound on the number of states may be attained. The start state of DM is $\{q_0\}$ since M does not have λ -transitions.

The input tra



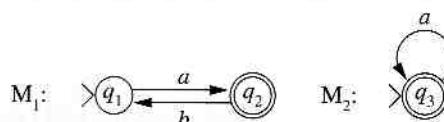
The equivalent DF

□

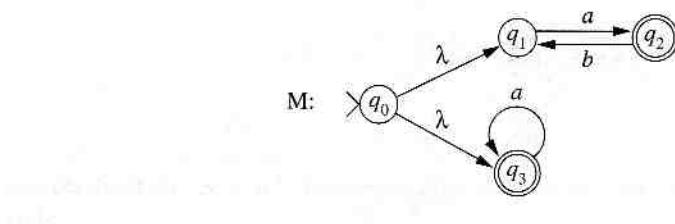
Example 5.6.4

The machines M_1 and M_2 accept $a(ba)^*$ and a^* , respectively.

Algorithm 5.6.
the classes of finite



Using λ -arcs to connect a new start state to the start states of the original machines creates an NFA- λ M that accepts $a(ba)^* \cup a^*$.



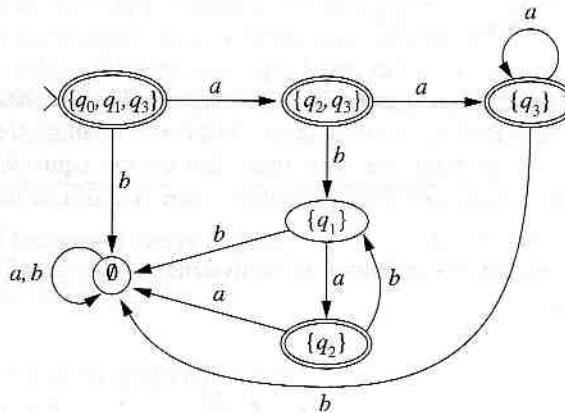
The arrows represent
an NFA- λ . The doubl
deterministic machin

start

The input transition function for M is

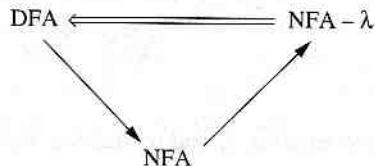
t	a	b
q_0	$\{q_2, q_3\}$	\emptyset
q_1	$\{q_2\}$	\emptyset
q_2	\emptyset	$\{q_1\}$
q_3	$\{q_3\}$	\emptyset

The equivalent DFA obtained from Algorithm 5.6.3 is



creates

Algorithm 5.6.3 completes the following cycle describing the relationships between the classes of finite automata.



The arrows represent inclusion; every DFA can be reformulated as an NFA that is, in turn, an NFA- λ . The double arrow from NFA- λ to DFA indicates the existence of an equivalent deterministic machine.

5.7 DFA Minimization

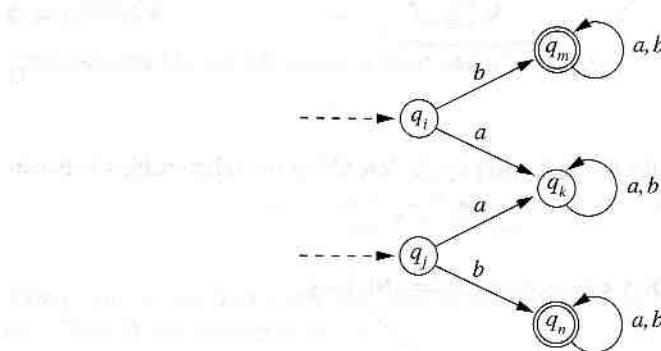
The preceding sections established that the family of languages accepted by DFAs is the same as that accepted by NFAs and NFA- λ s. The flexibility of nondeterminism and λ -transitions aid in the design of machines to accept complex languages. The nondeterministic machine can then be transformed into an equivalent deterministic machine using Algorithm 5.6.3. The resulting DFA, however, may not be the minimal DFA that accepts the language. This section presents a reduction algorithm that produces the minimal state DFA accepting the language L from any DFA that accepts L . To accomplish the reduction, the notion of equivalent states in a DFA is introduced.

Definition 5.7.1

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. States q_i and q_j are equivalent if $\hat{\delta}(q_i, u) \in F$ if, and only if, $\hat{\delta}(q_j, u) \in F$ for every $u \in \Sigma^*$.

Two states that are equivalent are called *indistinguishable*. The binary relation over Q defined by indistinguishability of states is an equivalence relation; that is, the relation is reflexive, symmetric, and transitive. Two states that are not equivalent are said to be *distinguishable*. States q_i and q_j are distinguishable if there is a string u such that $\hat{\delta}(q_i, u) \in F$ and $\hat{\delta}(q_j, u) \notin F$, or vice versa.

The motivation behind this definition of equivalence is illustrated by the following states and transitions:



The unlabeled dotted lines entering q_i and q_j indicate that the method of reaching a state is irrelevant; equivalence depends only upon computations from the state. The states q_i and q_j are equivalent since the computation with any string beginning with b from either state halts in an accepting state and all other computations halt in the nonaccepting state q_k . States q_m and q_n are also equivalent; all computations beginning in these states end in an accepting state.

The intuition behind the transformation is that equivalent states may be merged. Applying this to the preceding example yields

To reduce the size of states must be developed. If q_j , $i < j$, has associated with it, it is determined that the state q_j is not distinguishable. Index $[i, j]$ is in the set of states q_m and q_n .

The algorithm begins with one state q_0 that is accepting and non-distinguishable. It examines each nonaccepting state q_j and makes a call to a recursive routine that determines whether q_j is distinguishable, and if so, marks it as distinguishable, and continues the process until all states are examined.

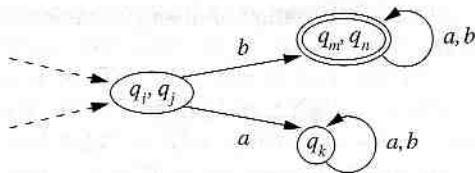
Algorithm 5.7.2 Determination of Equivalence

input: DFA $M = (Q, \Sigma, \delta, q_0, F)$

1. (Initialization)
 - 1.1. $D[i, j] := 0$ for every pair of states $i, j \in Q$
 - 1.2. $S[i, j] := 0$ for every pair of states $i, j \in Q$
2. for every pair $i, j \in Q$ do
 - not an accepting state i
3. for every pair $i, j \in Q$ do
 - 3.1. if there exists $a \in \Sigma$ such that $\delta(i, a) = j$ then
 - $D[m, n] := 1$
 - 3.2. else for each $a \in \Sigma$ do
 - if $m < n$ then
 - $D[m, n] := 1$
 - else if $m > n$ then
 - $D[m, n] := 1$

```

DIST( $i, j$ );
begin
   $D[i, j] := 1$ 
  for all  $[m, n] \in S$  do
    if  $D[m, n] = 1$  then
       $S[i, j] := 1$ 
    end if
  end for
end
  
```



To reduce the size of a DFA M by merging states, a procedure for identifying equivalent states must be developed. In the algorithm to accomplish this, each pair of states q_i and q_j , $i < j$, has associated with it values $D[i, j]$ and $S[i, j]$. $D[i, j]$ is set to 1 when it is determined that the states q_i and q_j are distinguishable. $S[m, n]$ contains a set of indices. Index $[i, j]$ is in the set $S[m, n]$ if the distinguishability of q_i and q_j follows from that of q_m and q_n .

The algorithm begins by marking each pair of states q_i and q_j as distinguishable if one is accepting and the other is rejecting. The remainder of the algorithm systematically examines each nonmarked pair of states. When two states are shown to be distinguishable, a call to a recursive routine $DIST$ sets $D[i, j]$ to 1. The call $DIST(i, j)$ not only marks q_i and q_j as distinguishable, it also marks each pair of states q_m and q_n for which $[m, n] \in S[i, j]$ as distinguishable through a call to $DIST(m, n)$.

Algorithm 5.7.2
Determination of Equivalent States of DFA

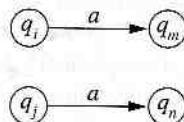
input: DFA $M = (Q, \Sigma, \delta, q_0, F)$

1. (Initialization)
 1. for every pair of states q_i and q_j , $i < j$, do
 - 1.1. $D[i, j] := 0$
 - 1.2. $S[i, j] := \emptyset$
 - end for
 2. for every pair i, j , $i < j$, if one of q_i or q_j is an accepting state and the other is not an accepting state, then set $D[i, j] := 1$
 3. for every pair i, j , $i < j$, with $D[i, j] = 0$, do
 - 3.1. if there exists an $a \in \Sigma$ such that $\delta(q_i, a) = q_m$, $\delta(q_j, a) = q_n$ and $D[m, n] = 1$ or $D[n, m] = 1$, then $DIST(i, j)$
 - 3.2. else for each $a \in \Sigma$, do: Let $\delta(q_i, a) = q_m$ and $\delta(q_j, a) = q_n$
 - if $m < n$ and $[i, j] \neq [m, n]$, then add $[i, j]$ to $S[m, n]$
 - else if $m > n$ and $[i, j] \neq [n, m]$, then add $[i, j]$ to $S[n, m]$
 - end for
- ```

 $DIST(i, j);$
begin
 $D[i, j] := 1$
 for all $[m, n] \in S[i, j]$, $DIST(m, n)$
end

```
-

The motivation behind the identification of distinguishable states is illustrated by the relationships in the diagram



If  $q_m$  and  $q_n$  are already marked as distinguishable when  $q_i$  and  $q_j$  are examined in step 3, then  $D[i, j]$  is set to 1 to indicate the distinguishability of  $q_i$  and  $q_j$ . If the status of  $q_m$  and  $q_n$  is not known when  $q_i$  and  $q_j$  are examined, then a later determination that  $q_m$  and  $q_n$  are distinguishable also provides the answer for  $q_i$  and  $q_j$ . The role of the array  $S$  is to record this information:  $[i, j] \in S[n, m]$  indicates that the distinguishability of  $q_m$  and  $q_n$  is sufficient to establish the distinguishability of  $q_i$  and  $q_j$ . These ideas are formalized in the proof of Theorem 5.7.3.

### Theorem 5.7.3

States  $q_i$  and  $q_j$  are distinguishable if, and only if,  $D[i, j] = 1$  at the termination of Algorithm 5.7.2.

**Proof.** First we show that every pair of states  $q_i$  and  $q_j$  for which  $D[i, j] = 1$  is distinguishable. If  $D[i, j]$  is assigned 1 in the step 2, then  $q_i$  and  $q_j$  are distinguishable by the null string. Step 3.1 marks  $q_i$  and  $q_j$  as distinguishable only if  $\delta(q_i, a) = q_m$  and  $\delta(q_j, a) = q_n$  for some input  $a$  when states  $q_m$  and  $q_n$  have already been determined to be distinguishable by the algorithm. Let  $u$  be a string that exhibits the distinguishability of  $q_m$  and  $q_n$ . Then  $au$  exhibits the distinguishability of  $q_i$  and  $q_j$ .

To complete the proof, it is necessary to show that every pair of distinguishable states is designated as such. The proof is by induction on the length of the shortest string that demonstrates the distinguishability of a pair of states. The basis consists of all pairs of states  $q_i, q_j$  that are distinguishable by a string of length 0. That is, the computations  $\hat{\delta}(q_i, \lambda) = q_i$  and  $\hat{\delta}(q_j, \lambda) = q_j$  distinguish  $q_i$  from  $q_j$ . In this case, exactly one of  $q_i$  or  $q_j$  is accepting and the position  $D[i, j]$  is set to 1 in step 2.

Now assume that every pair of states distinguishable by a string of length  $k$  or less is marked by the algorithm. Let  $q_i$  and  $q_j$  be states for which the shortest distinguishing string  $u$  has length  $k + 1$ . Then  $u$  can be written  $av$  and the computations with input  $u$  have the form  $\hat{\delta}(q_i, u) = \hat{\delta}(q_i, av) = \hat{\delta}(q_m, v) = q_s$  and  $\hat{\delta}(q_j, u) = \hat{\delta}(q_j, av) = \hat{\delta}(q_n, v) = q_t$ . Exactly one of  $q_s$  and  $q_t$  is accepting since the preceding computations distinguish  $q_i$  from  $q_j$ . Clearly, the same computations exhibit the distinguishability of  $q_m$  from  $q_n$  by a string of length  $k$ . By induction, we know that the algorithm will set  $D[m, n]$  to 1.

If  $D[m, n]$  is marked before the states  $q_i$  and  $q_j$  are examined in step 3, then  $D[i, j]$  is set to 1 by the call  $DIST(i, j)$ . If  $q_i$  and  $q_j$  are examined in the loop in step 3.1 and  $D[m, n] \neq 1$  at that time, then  $[i, j]$  is added to the set  $S[m, n]$ . By the inductive hypothesis,  $D[m, n]$  will eventually be set to 1.  $D[i, j]$  will also be set to 1 at this time by a recursive call from  $DIST(m, n)$  since  $[i, j]$  is in  $S[m, n]$ . ■

A new DFA M  
indistinguishability  
indistinguishable s  
The transition fu  
is shown to be well  
 $\hat{\delta}'([q_0], u) = [\hat{\delta}(q_i,$   
states that are unre  
are deleted.

### Example 5.7.1

The minimization p

that accepts the lang

In step 2,  $D[0, 1]$ ,  
 $D[4, 7]$ ,  $D[5, 7]$ , and  
step 3. The table sho

Ind

[0, 7]

[1, 2]

[1, 3]

[1, 4]

[1, 5]

[1, 6]

[2, 3]

by the

A new DFA  $M'$  can be built from the original DFA  $M = (Q, \Sigma, \delta, q_0, F)$  and the indistinguishability relation. The states of  $M'$  are the equivalence classes consisting of indistinguishable states of  $M$ . The start state is  $[q_0]$ , and  $[q_i]$  is a final state if  $q_i \in F$ . The transition function  $\delta'$  of  $M'$  is defined by  $\delta'([q_i], a) = [\delta(q_i, a)]$ . In Exercise 44,  $\delta'$  is shown to be well defined.  $L(M')$  consists of all strings whose computations have the form  $\hat{\delta}'([q_0], u) = [\hat{\delta}(q_i, \lambda)]$  with  $q_i \in F$ . These are precisely the strings accepted by  $M$ . If  $M'$  has states that are unreachable by computations from  $[q_0]$ , these states and all associated arcs are deleted.

step 3,  
of  $q_m$   
 $q_m$  and  
 $S$  is to  
and  $q_n$  is  
in the

ation of

; distin-  
the null  
 $a) = q_n$   
ishable  
 $n$ . Then

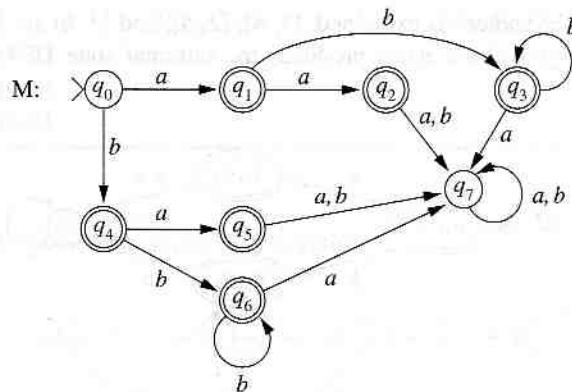
le states  
ing that  
of states  
 $\lambda) = q_i$   
cepting

or less is  
ng string  
the form  
actly one  
Clearly,  
length  $k$ .

in  $D[i, j]$   
o 3.1 and  
pothesis,  
recursive

### Example 5.7.1

The minimization process is exhibited using the DFA  $M$



that accepts the language  $(a \cup b)(a \cup b^*)$ .

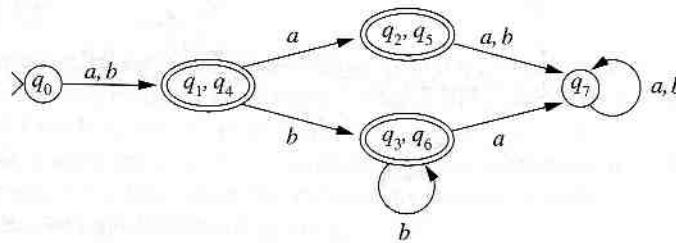
In step 2,  $D[0, 1], D[0, 2], D[0, 3], D[0, 4], D[0, 5], D[0, 6], D[1, 7], D[2, 7], D[3, 7], D[4, 7], D[5, 7]$ , and  $D[6, 7]$  are set to 1. Each index not marked in step 2 is examined in step 3. The table shows the action taken for each such index.

| Index    | Action                                           | Reason               |
|----------|--------------------------------------------------|----------------------|
| $[0, 7]$ | $D[0, 7] = 1$                                    | Distinguished by $a$ |
| $[1, 2]$ | $D[1, 2] = 1$                                    | Distinguished by $a$ |
| $[1, 3]$ | $D[1, 3] = 1$                                    | Distinguished by $a$ |
| $[1, 4]$ | $S[2, 5] = \{[1, 4]\}$<br>$S[3, 6] = \{[1, 4]\}$ |                      |
| $[1, 5]$ | $D[1, 5] = 1$                                    | Distinguished by $a$ |
| $[1, 6]$ | $D[1, 6] = 1$                                    | Distinguished by $a$ |
| $[2, 3]$ | $D[2, 3] = 1$                                    | Distinguished by $b$ |

(Continued)

| Index  | Action        | Reason                                                                     |
|--------|---------------|----------------------------------------------------------------------------|
| [2, 4] | $D[2, 4] = 1$ | Distinguished by $a$                                                       |
| [2, 5] |               | No action since $\delta(q_2, x) = \delta(q_5, x)$ for every $x \in \Sigma$ |
| [2, 6] | $D[2, 6] = 1$ | Distinguished by $b$                                                       |
| [3, 4] | $D[3, 4] = 1$ | Distinguished by $a$                                                       |
| [3, 5] | $D[3, 5] = 1$ | Distinguished by $b$                                                       |
| [3, 6] |               |                                                                            |
| [4, 5] | $D[4, 5] = 1$ | Distinguished by $a$                                                       |
| [4, 6] | $D[4, 6] = 1$ | Distinguished by $a$                                                       |
| [5, 6] | $D[5, 6] = 1$ | Distinguished by $b$                                                       |

After each pair of indices is examined, [1, 4], [2, 5], and [3, 6] are left as equivalent pairs of states. Merging these states produces the minimal state DFA  $M'$  that accepts  $(a \cup b)(a \cup b)^*$ .

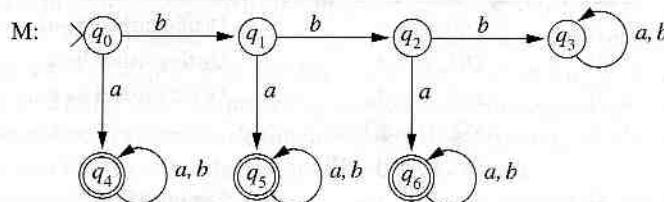


Merging equ

□

### Example 5.7.2

Minimizing the DFA  $M$  illustrates the recursive marking of states by the call to  $DIST$ . The language of  $M$  is  $a(a \cup b)^* \cup ba(a \cup b)^* \cup bba(a \cup b)^*$ .



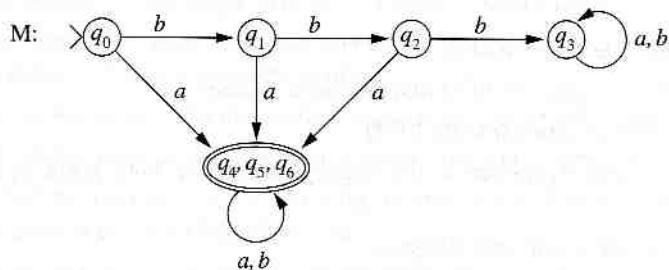
The comparison of accepting states to nonaccepting states assigns 1 to  $D[0, 4]$ ,  $D[0, 5]$ ,  $D[0, 6]$ ,  $D[1, 4]$ ,  $D[1, 5]$ ,  $D[1, 6]$ ,  $D[2, 4]$ ,  $D[2, 5]$ ,  $D[2, 6]$ ,  $D[3, 4]$ ,  $D[3, 5]$ , and  $D[3, 6]$ . Tracing the algorithm produces

The minimization construction of optimal finite automata to 5.6.3 can then be used to be minimal. Algo

For the moment established that it Theorem, which classifies equivalence classes of the machine  $M'$  produced

| Index    | Action                 | Reason               |
|----------|------------------------|----------------------|
| [0, 1]   | $S[4, 5] = \{[0, 1]\}$ |                      |
| $\Sigma$ | $S[1, 2] = \{[0, 1]\}$ |                      |
| [0, 2]   | $S[4, 6] = \{[0, 2]\}$ |                      |
|          | $S[1, 3] = \{[0, 2]\}$ |                      |
| [0, 3]   | $D[0, 3] = 1$          | Distinguished by $a$ |
| [1, 2]   | $S[5, 6] = \{[1, 2]\}$ |                      |
|          | $S[2, 3] = \{[1, 2]\}$ |                      |
| [1, 3]   | $D[1, 3] = 1$          | Distinguished by $a$ |
|          | $D[0, 2] = 1$          | Call to $DIST(1, 3)$ |
| [2, 3]   | $D[2, 3] = 1$          | Distinguished by $a$ |
|          | $D[1, 2] = 1$          | Call to $DIST(1, 2)$ |
|          | $D[0, 1] = 1$          | Call to $DIST(0, 1)$ |
| [4, 5]   |                        |                      |
| [4, 6]   |                        |                      |
| [5, 6]   |                        |                      |

Merging equivalent states  $q_4$ ,  $q_5$ , and  $q_6$  yields



ST. The

$D[0, 5]$ ,  
 $D[3, 6]$ .

The minimization algorithm completes the sequence of algorithms required for the construction of optimal DFAs. Nondeterminism and  $\lambda$ -transitions provide tools for designing finite automata to match complicated patterns or to accept complex languages. Algorithm 5.6.3 can then be used to transform the nondeterministic machine into a DFA, which may not be minimal. Algorithm 5.7.2 completes the process by producing the minimal state DFA.

For the moment, we have presented an algorithm for DFA reduction but have not established that it produces the minimal DFA. In Section 6.7 we prove the Myhill-Nerode Theorem, which characterizes the language accepted by a finite automaton in terms of equivalence classes of strings. This characterization will then be used to prove that the machine  $M'$  produced by Algorithm 5.7.2 is the unique minimal state DFA that accepts  $L$ .

**Exercises**

1. Let  $M$  be the deterministic finite automaton defined by

|                         |          |       |       |
|-------------------------|----------|-------|-------|
| $Q = \{q_0, q_1, q_2\}$ | $\delta$ | $a$   | $b$   |
| $\Sigma = \{a, b\}$     | $q_0$    | $q_0$ | $q_1$ |
| $F = \{q_2\}$           | $q_1$    | $q_2$ | $q_1$ |
|                         | $q_2$    | $q_2$ | $q_0$ |

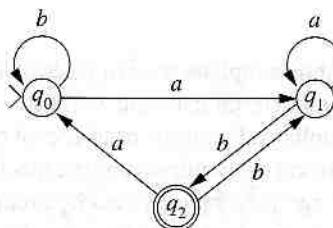
- a) Give the state diagram of  $M$ .
- b) Trace the computations of  $M$  that process the strings  $abaa$ ,  $bbbabb$ ,  $bababa$ , and  $bbbaaa$ .
- c) Which of the strings from part (b) are accepted by  $M$ ?
- d) Give a regular expression for  $L(M)$ .

2. Let  $M$  be the deterministic finite automaton

|                         |          |       |       |
|-------------------------|----------|-------|-------|
| $Q = \{q_0, q_1, q_2\}$ | $\delta$ | $a$   | $b$   |
| $\Sigma = \{a, b\}$     | $q_0$    | $q_1$ | $q_0$ |
| $F = \{q_0\}$           | $q_1$    | $q_1$ | $q_2$ |
|                         | $q_2$    | $q_1$ | $q_0$ |

- a) Give the state diagram of  $M$ .
- b) Trace the computation of  $M$  that processes  $babaab$ .
- c) Give a regular expression for  $L(M)$ .
- d) Give a regular expression for the language accepted if both  $q_0$  and  $q_1$  are accepting states.

3. Let  $M$  be the DFA with state diagram



- a) Construct the transition table of  $M$ .
- b) Which of the strings  $baba$ ,  $baab$ ,  $abab$ ,  $abaaab$  are accepted by  $M$ ?
- c) Give a regular expression for  $L(M)$ .

- \* 4. The recursive definition of  $\hat{\delta}$  may be replaced by the formula that  $\hat{\delta} = \hat{\delta}'$ .

For Exercises 5 through 10,

- 5. The set of strings that precede the character  $c$ .
- 6. The set of strings of length  $n$ .
- 7. The set of strings of length  $n$  that end in  $cc$ .
- 8. The set of strings of length  $n$  that contain one  $c$ .
- 9. The set of strings of length  $n$  that do not contain  $c$ .
- 10. The set of strings of length  $n$  that end in  $aa$ . Note that  $a$  is a string, not a character.
- 11. The set of strings of length  $n$  that immediately follow  $c$ .
- 12. The set of strings of length  $n$  that are twice the number of  $c$ 's.
- 13. The set of strings of length  $n$  that are three times the number of  $c$ 's.
- 14. The set of strings of length  $n$  that are four times the number of  $c$ 's.
- 15. The set of strings of length  $n$  that are five times the number of  $c$ 's.
- 16. The set of strings of length  $n$  that end in  $aa$ . Note that  $a$  is a string, not a character.
- 17. The set of strings of length  $n$  that contain exactly two  $c$ 's.
- 18. The set of strings of length  $n$  that contain exactly three  $c$ 's.
- 19. The set of strings of length  $n$  that contain exactly twice the number of  $a$ 's as  $c$ 's.
- 20. The set of strings of length  $n$  that contain exactly three  $a$ 's. Note that every string of length  $n$  contains at least one  $a$ .
- \* 21. The set of strings of length  $n$  that contain exactly one  $c$ .
- 22. For each of the following languages,

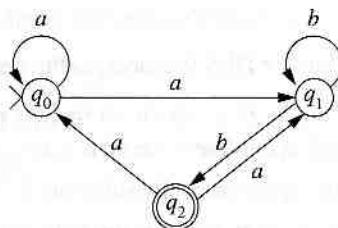
  - a)  $(ab)^*ba$
  - b)  $(ab)^*(ba)^*$
  - c)  $aa(a \cup b)^*$
  - d)  $((aa)^+bb)^*$
  - e)  $(ab^*a)^*$

- \* 4. The recursive step in the definition of the extended transition function (Definition 5.2.4) may be replaced by  $\hat{\delta}'(q_i, au) = \hat{\delta}'(\delta(q_i, a), u)$ , for all  $u \in \Sigma^*$ ,  $a \in \Sigma$ , and  $q_i \in Q$ . Prove that  $\hat{\delta} = \hat{\delta}'$ .

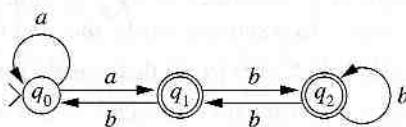
For Exercises 5 through 21, build a DFA that accepts the described language.

5. The set of strings over  $\{a, b, c\}$  in which all the  $a$ 's precede the  $b$ 's, which in turn precede the  $c$ 's. It is possible that there are no  $a$ 's,  $b$ 's, or  $c$ 's.
6. The set of strings over  $\{a, b\}$  in which the substring  $aa$  occurs at least twice.
7. The set of strings over  $\{a, b\}$  that do not begin with the substring  $aaa$ .
8. The set of strings over  $\{a, b\}$  that do not contain the substring  $aaa$ .
9. The set of strings over  $\{a, b, c\}$  that begin with  $a$ , contain exactly two  $b$ 's, and end with  $cc$ .
10. The set of strings over  $\{a, b, c\}$  in which every  $b$  is immediately followed by at least one  $c$ .
11. The set of strings over  $\{a, b\}$  in which the number of  $a$ 's is divisible by three.
12. The set of strings over  $\{a, b\}$  in which every  $a$  is either immediately preceded or immediately followed by  $b$ , for example,  $baab$ ,  $aba$ , and  $b$ .
13. The set of strings of odd length over  $\{a, b\}$  that contain the substring  $bb$ .
14. The set of strings over  $\{a, b\}$  that have odd length or end with  $aaa$ .
15. The set of strings of even length over  $\{a, b, c\}$  that contain exactly one  $a$ .
16. The set of strings over  $\{a, b\}$  that have an odd number of occurrences of the substring  $aa$ . Note that  $aaa$  has two occurrences of  $aa$ .
17. The set of strings over  $\{a, b\}$  that contain an even number of substrings  $ba$ .
18. The set of strings over  $\{1, 2, 3\}$  the sum of whose elements is divisible by six.
19. The set of strings over  $\{a, b, c\}$  in which the number of  $a$ 's plus the number of  $b$ 's plus twice the number of  $c$ 's is divisible by six.
20. The set of strings over  $\{a, b\}$  in which every substring of length four has at least one  $b$ . Note that every substring with length less than four is in this language.
- \* 21. The set of strings over  $\{a, b, c\}$  in which every substring of length four has exactly one  $b$ .
22. For each of the following languages, give the state diagram of a DFA that accepts the languages.
  - a)  $(ab)^*ba$
  - b)  $(ab)^*(ba)^*$
  - c)  $aa(a \cup b)^+bb$
  - d)  $((aa)^+bb)^*$
  - e)  $(ab^*a)^*$

23. Let  $M$  be the nondeterministic finite automaton



- a) Construct the transition table of  $M$ .
  - b) Trace all computations of the string  $aaabb$  in  $M$ .
  - c) Is  $aaabb$  in  $L(M)$ ?
  - d) Give a regular expression for  $L(M)$ .
24. Let  $M$  be the nondeterministic finite automaton



- a) Construct the transition table of  $M$ .
  - b) Trace all computations of the string  $aabb$  in  $M$ .
  - c) Is  $aabb$  in  $L(M)$ ?
  - d) Give a regular expression for  $L(M)$ .
  - e) Construct a DFA that accepts  $L(M)$ .
  - f) Give a regular expression for the language accepted if both  $q_0$  and  $q_1$  are accepting states.
25. For each of the following languages, give the state diagram of an NFA that accepts the language.
- a)  $(a \cup ab \cup aab)^*$
  - b)  $(ab)^* \cup a^*$
  - c)  $(abc)^*a^*$
  - d)  $(ba \cup bb)^* \cup (ab \cup aa)^*$
  - e)  $(ab^+a)^+$
26. Give a recursive definition of the extended transition function  $\hat{\delta}$  of an NFA- $\lambda$ . The value  $\hat{\delta}(q_i, w)$  is the set of states that can be reached by computations that begin at node  $q_i$  and completely process the string  $w$ .

For Exercises 27 to 36, let  $M$  be the NFA given in Exercise 23. Assume that  $M$  accepts a language. Remember that a string is accepted if it leads to an accepting state whenever it is appended to a computation.

- 27. The set of strings accepted by  $M$ .
- 28. The set of strings rejected by  $M$ .
- \* 29. The set of strings accepted by  $M$  that end in  $a$ .
- 30. The set of strings accepted by  $M$  that end in  $b$ . For example,  $aaba$  is accepted, while  $aaabb$  is not.
- 31. The set of strings accepted by  $M$  that contain exactly one  $a$ .
- 32. The set of strings accepted by  $M$  that contain exactly two  $a$ 's, with the same symbol appearing in both positions.
- 33. The set of strings accepted by  $M$  that contain exactly two  $a$ 's.
- 34. The set of strings accepted by  $M$  that contain exactly two symbols of the same type.
- 35. Construct the state diagram of an NFA that accepts the substring  $ab$  of the language accepted by  $M$ . That is, the substrings of the language accepted by  $M$  that contain the substring  $ab$ .
- 36. Let  $M$  be the NFA given in Exercise 23. Compute  $\hat{\delta}(q_0, \lambda)$ .
- a) Compute  $\lambda$ -closure( $q_0$ ).
- b) Give the input string  $abab$ .
- c) Use Algorithm 5.1 to compute  $\hat{\delta}(q_0, abab)$ .
- d) Give a regular expression for  $L(M)$ .

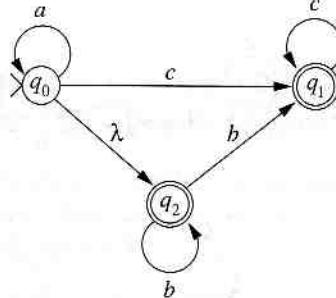
For Exercises 27 through 34, give the state diagram of an NFA that accepts the given language. Remember that an NFA may be deterministic, but you should use nondeterminism whenever it is appropriate.

27. The set of strings over  $\{a, b\}$  that contain either  $aa$  and  $bb$  as substrings.
28. The set of strings over  $\{a, b\}$  that contain both or neither  $aa$  and  $bb$  as substrings.
- \* 29. The set of strings over  $\{a, b\}$  whose third-to-the-last symbol is  $b$ .
30. The set of strings over  $\{a, b\}$  whose third and third-to-last symbols are both  $b$ . For example,  $aababaaa$ ,  $abbbbbbbb$ , and  $abba$  are in the language.
31. The set of strings over  $\{a, b\}$  in which every  $a$  is followed by  $b$  or  $ab$ .
32. The set of strings over  $\{a, b\}$  that have a substring of length four that begins and ends with the same symbol.
33. The set of strings over  $\{a, b\}$  that contain substrings  $aaa$  and  $bbb$ .
34. The set of strings over  $\{a, b, c\}$  that have a substring of length three containing each of the symbols exactly once.
35. Construct the state diagram of a DFA that accepts the strings over  $\{a, b\}$  ending with the substring  $abba$ . Give the state diagram of an NFA with six arcs that accepts the same language.
36. Let  $M$  be the NFA- $\lambda$

are accepting

that accepts the

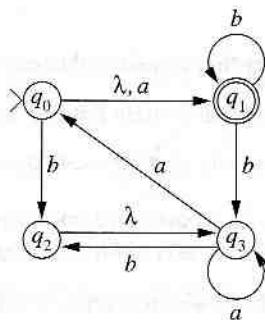
$A-\lambda$ . The value  
begin at node  $q_i$



- a) Compute  $\lambda$ -closure( $q_i$ ) for  $i = 0, 1, 2$ .
- b) Give the input transition function  $t$  for  $M$ .
- c) Use Algorithm 5.6.3 to construct a state diagram of a DFA that is equivalent to  $M$ .
- d) Give a regular expression for  $L(M)$ .

37. Let  $M$  be the NFA- $\lambda$

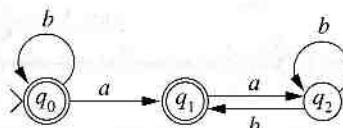
c)



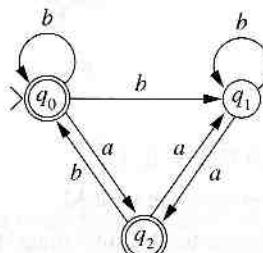
d)

- a) Compute  $\lambda$ -closure( $q_i$ ) for  $i = 0, 1, 2, 3$ .
  - b) Give the input transition function  $t$  for  $M$ .
  - c) Use Algorithm 5.6.3 to construct a state diagram of a DFA that is equivalent to  $M$ .
  - d) Give a regular expression for  $L(M)$ .
38. Use Algorithm 5.6.3 to construct the state diagram of a DFA equivalent to the NFA in Example 5.5.2.
39. Use Algorithm 5.6.3 to construct the state diagram of a DFA equivalent to the NFA in Exercise 17.
40. For each of the following NFAs, use Algorithm 5.6.3 to construct the state diagram of an equivalent DFA.

a)



b)



41. Build an NFA  $M_1$  with transitions to obtain the function of  $M$ . Use  $L(M)$ .

42. Build an NFA  $M_1$  with transitions to obtain the function of  $M$ . Use  $L(M)$ .

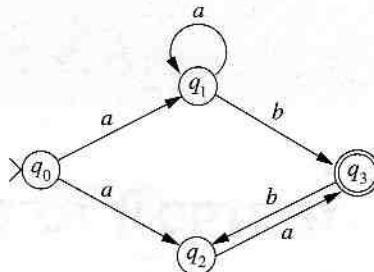
43. Assume that  $q_i$  and  $\hat{\delta}(q_i, u) = q_m$  and  $\hat{\delta}$

\* 44. Show that the transition states is well defined.  $\delta'([q_i], a) = \delta'([q_j], a)$

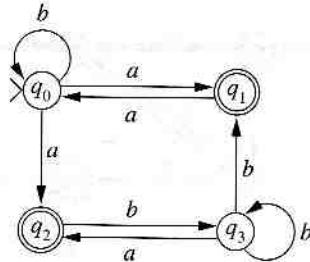
45. For each DFA:

- i) Trace the actions and the values of  $D$ .
- ii) Give the equivalence classes.
- iii) Give the state diagram.

c)

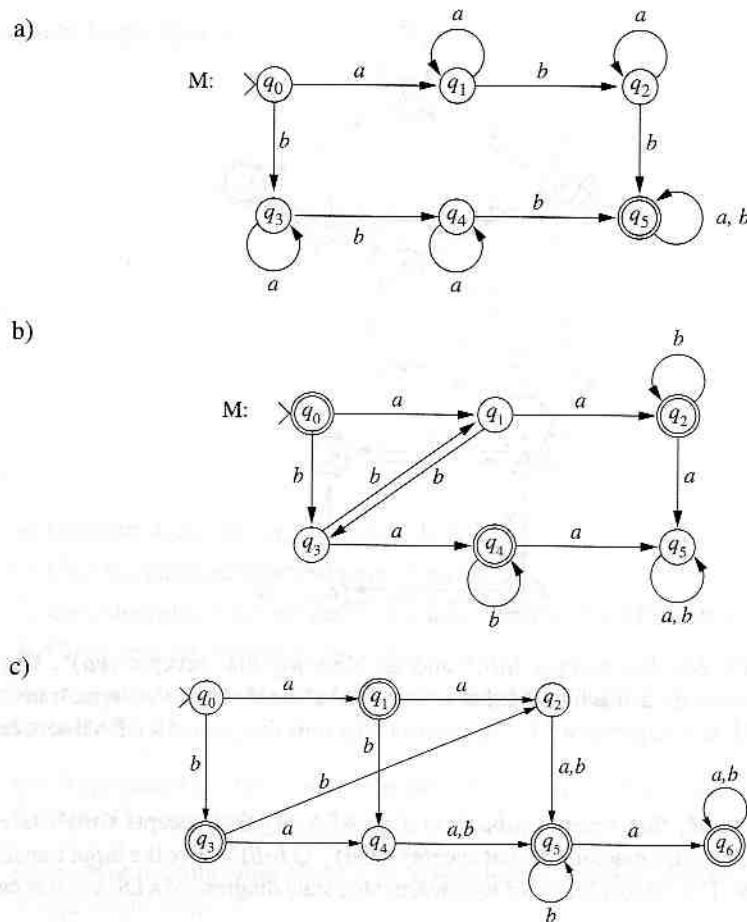


d)



- nt to M.
- NFA in
- NFA in
- agram of
41. Build an NFA  $M_1$  that accepts  $(ab)^*$  and an NFA  $M_2$  that accepts  $(ba)^*$ . Use  $\lambda$ -transitions to obtain a machine  $M$  that accepts  $(ab)^*(ba)^*$ . Give the input transition function of  $M$ . Use Algorithm 5.6.3 to construct the state diagram of a DFA that accepts  $L(M)$ .
42. Build an NFA  $M_1$  that accepts  $(aba)^+$  and an NFA  $M_2$  that accepts  $(ab)^*$ . Use  $\lambda$ -transitions to obtain a machine  $M$  that accepts  $(aba)^+ \cup (ab)^*$ . Give the input transition function of  $M$ . Use Algorithm 5.6.3 to construct the state diagram of a DFA that accepts  $L(M)$ .
43. Assume that  $q_i$  and  $q_j$  are equivalent states of a DFA  $M$  (as in Definition 5.7.1) and  $\hat{\delta}(q_i, u) = q_m$  and  $\hat{\delta}(q_j, u) = q_n$  for a string  $u \in \Sigma^*$ . Prove that  $q_m$  and  $q_n$  are equivalent.
- \* 44. Show that the transition function  $\delta'$  obtained in the process of merging equivalent states is well defined. That is, show that if  $q_i$  and  $q_j$  are states with  $[q_i] = [q_j]$ , then  $\delta'([q_i], a) = \delta'([q_j], a)$  for every  $a \in \Sigma$ .
45. For each DFA:
- Trace the actions of Algorithm 5.7.2 to determine the equivalent states of  $M$ . Give the values of  $D[i, j]$  and  $S[i, j]$  computed by the algorithm.
  - Give the equivalence classes of states.
  - Give the state diagram of the minimal state DFA that accepts  $L(M)$ .

## Proper Language



Grammars were introduced and regular expressions between these three approaches to automata as languages.

---

## Bibliographic Notes

Alternative interpretations of the result of finite-state computations were studied in Mealy [1955] and Moore [1956]. Transitions in Mealy machines are accompanied by the generation of output. A two-way automaton allows the tape head to move in both directions. A proof that two-way and one-way automata accept the same languages can be found in Rabin and Scott [1959] and Sheperdson [1959]. Nondeterministic finite automata were introduced by Rabin and Scott [1959]. The algorithm for minimizing the number of states in a DFA was presented in Nerode [1958]. The algorithm of Hopcroft [1971] increases the efficiency of the minimization technique.

The theory and applications of finite automata are developed in greater depth in the books by Minsky [1967]; Salomaa [1973]; Denning, Dennis, and Qualitz [1978]; and Bavel [1983].

## 6.1 Finite-State

In this section we show that Regular sets are built from the alphabet by the following steps or blocks rather than sets.

## CHAPTER 6

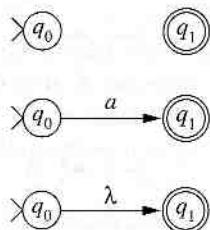
# Properties of Regular Languages

Grammars were introduced as language generators, finite automata as language acceptors, and regular expressions as pattern descriptors. This chapter develops the relationship between these three approaches to language definition and explores the limitations of finite automata as language acceptors.

### 6.1 Finite-State Acceptance of Regular Languages

In this section we show that an NFA- $\lambda$  can be constructed to accept any regular language. Regular sets are built recursively from  $\emptyset$ ,  $\{\lambda\}$ , and singleton sets containing elements from the alphabet by applications of union, concatenation, and the Kleene star operation (Definition 2.3.2). The construction of an NFA- $\lambda$  that accepts a regular set can be obtained following the steps of its recursive generation, but using state diagrams as the building blocks rather than sets.

State diagrams for machines that accept  $\emptyset$ ,  $\{\lambda\}$ , and singleton sets  $\{a\}$  are



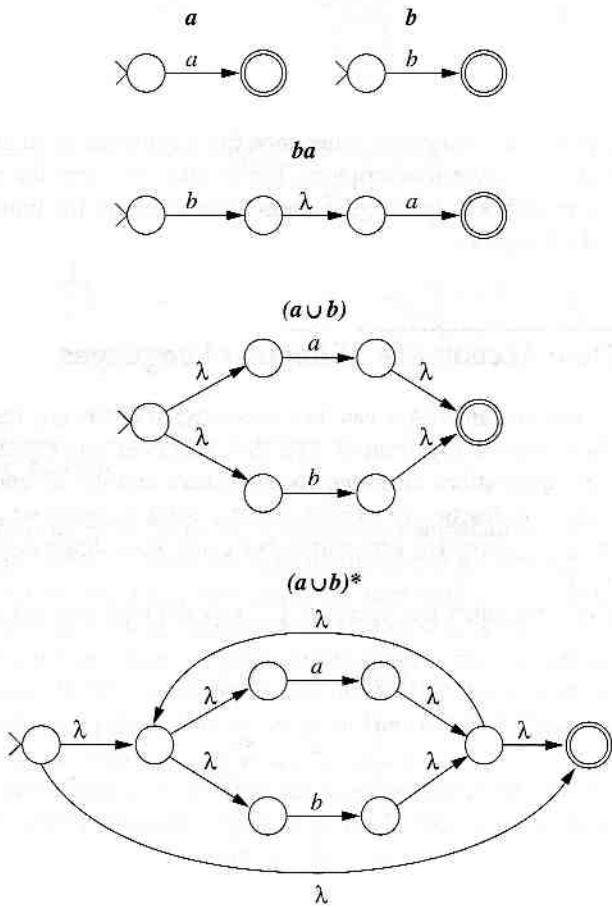
Note that each of these machines satisfies the restrictions described in Lemma 5.5.2. That is, the machines contain a single accepting state and there are no arcs entering the start state or leaving the accepting state.

As shown in Theorem 5.5.3,  $\lambda$ -transitions can be used to combine machines of this form to produce machines that accept more complex languages. Using repeated applications of these techniques, the construction of the regular expression from the basis elements can be mimicked by the corresponding machine operations. This process is illustrated in the following example.

---

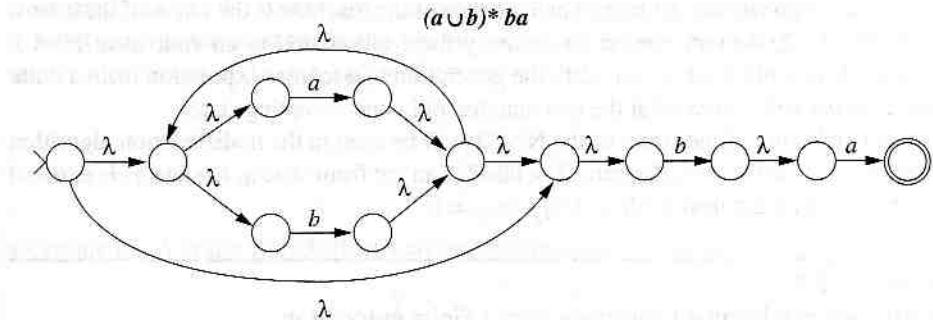
**Example 6.1.1**

An NFA- $\lambda$  that accepts  $(a \cup b)^*ba$  is constructed following the steps in the recursive definition of the regular expression. The language accepted by each intermediate machine is indicated by the regular expression above the state diagram.



a 5.5.2. That  
the start state

s of this form  
lications of  
lements can  
strated in the



the recursive  
diate machine

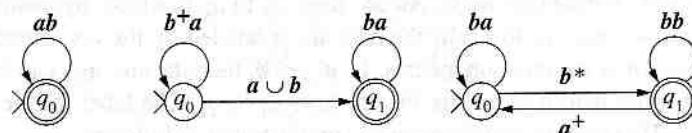
## 6.2 Expression Graphs

The construction in the previous section demonstrates that every regular language is recognized by a finite automaton. We will now show that every language accepted by a finite automaton is regular by constructing a regular expression for the language of the machine. To accomplish this, we extend the notion of a state diagram.

### Definition 6.2.1

An **expression graph** is a labeled directed graph in which the arcs are labeled by regular expressions. An expression graph, like a state diagram, contains a distinguished start node and a set of accepting nodes.

The state diagram of a finite automaton with alphabet  $\Sigma$  is a special case of an expression graph; the labels consist of  $\lambda$  and expressions corresponding to the elements of  $\Sigma$ . Paths in expression graphs generate regular expressions. The language of an expression graph is the union of the regular expressions along paths from the start node to an accepting node. For example, the expression graphs



accept the languages  $(ab)^*$ ,  $(b^+a)^*(a \cup b)(ba)^*$ , and  $(ba)^*b^*(bb \cup (a^+(ba)^*b^*))^*$ , respectively.

Because of the simplicity of the graphs, the expressions for the languages accepted by the previous examples were obvious. A procedure is developed to reduce an arbitrary expression graph to an expression graph containing at most two nodes. The reduction is accomplished by repeatedly removing nodes from the graph in a manner that preserves the language of the graph.

The state diagram of a finite automaton may have any number of accepting states. Each of these states exhibits the acceptance of a set of strings, the strings whose processing

successfully terminates in the state. The language of the machine is the union of these sets. By Lemma 5.5.2, we can convert an arbitrary finite automaton to an equivalent NFA- $\lambda$  with a single accepting set. To simplify the generation of a regular expression from a finite automaton, we will assume that the machine has only one accepting state.

The numbering of the states of the NFA- $\lambda$  will be used in the node deletion algorithm to identify paths in the state diagram. The label of an arc from state  $q_i$  to state  $q_j$  is denoted  $w_{i,j}$ . If there is no arc from node  $q_i$  to  $q_j$ ,  $w_{i,j} = \emptyset$ .

---

**Algorithm 6.2.2**
**Construction of a Regular Expression from a Finite Automaton**

input: state diagram  $G$  of a finite automaton with one accepting state

Let  $q_0$  be the start state and  $q_t$  the accepting state of  $G$ .

**1. repeat**

    1.1. choose a node  $q_i$  that is neither  $q_0$  nor  $q_t$

    1.2. delete the node  $q_i$  from  $G$  according to the following procedure:

        1.2.1   **for every**  $j, k$  not equal to  $i$  (this includes  $j = k$ ) **do**

            i) **if**  $w_{j,i} \neq \emptyset$ ,  $w_{i,k} \neq \emptyset$  and  $w_{i,i} = \emptyset$ , **then add an arc**  
                **from node  $j$  to node  $k$  labeled  $w_{j,i}w_{i,k}$**

            ii) **if**  $w_{j,i} \neq \emptyset$ ,  $w_{i,k} \neq \emptyset$  and  $w_{i,i} \neq \emptyset$ , **then add an arc from**  
                **node  $q_j$  to node  $q_k$  labeled  $w_{j,i}(w_{i,i})^*w_{i,k}$**

            iii) **if nodes  $q_j$  and  $q_k$  have arcs labeled  $w_1, w_2, \dots, w_s$**   
                **connecting them, then replace the arcs by a single**  
                **arc labeled  $w_1 \cup w_2 \cup \dots \cup w_s$**

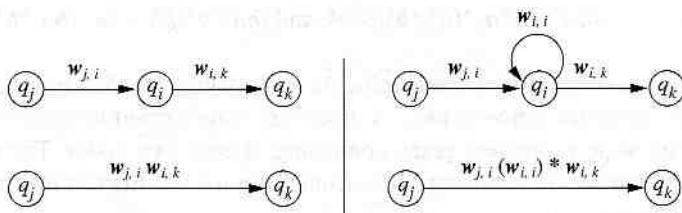
        1.2.2   **remove the node  $q_i$  and all arcs incident to it in  $G$**

**until** the only nodes in  $G$  are  $q_0$  and  $q_t$

**2. determine the expression accepted by  $G$** 


---

The deletion of node  $q_i$  is accomplished by finding all paths  $q_j, q_i, q_k$  of length two that have  $q_i$  as the intermediate node. An arc from  $q_j$  to  $q_k$  is added, bypassing the node  $q_i$ . If there is no arc from  $q_i$  to itself, the new arc is labeled by the concatenation of the expressions on each of the component arcs. If  $w_{i,i} \neq \emptyset$ , then the arc  $w_{i,i}$  can be traversed any number of times before following the arc from  $q_i$  to  $q_k$ . The label for the new arc is  $w_{j,i}(w_{i,i})^*w_{i,k}$ . These graph transformations are illustrated as follows:



Step 2 in the algorithm may appear to be begging the question; the objective of the entire algorithm is to determine the expression accepted by  $G$ . After the node deletion process is

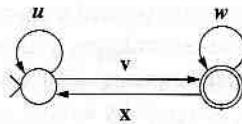
these sets.  
nt NFA- $\lambda$   
om a finite

algorithm  
is denoted

completed, the regular expression can easily be obtained from the resulting graph. The reduced graph has at most two nodes, the start node and the accepting node. If these are the same node, the reduced graph has the form



accepting  $u^*$ . A graph with distinct start and accepting nodes reduces to

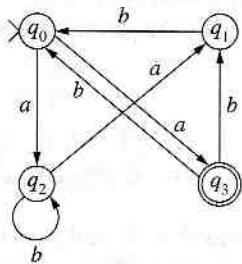


and accepts the expression  $u^*v(w \cup xu^*v)^*$ . This expression may be simplified if any of the arcs in the graph are labeled  $\emptyset$ .

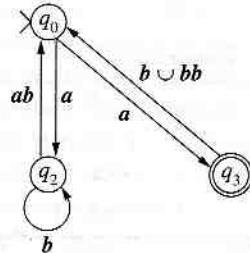
Algorithm 6.2.2 can also be used to construct the language of a finite state machine with multiple accepting states. For each accepting state, we can produce an expression for the strings accepted by that state. The language of the machine is simply the union of the regular expressions obtained for each accepting state.

### Example 6.2.1

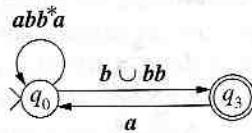
The reduction technique of Algorithm 6.2.2 is used to generate a regular expression for the language of the NFA with state diagram



Deleting node  $q_1$  yields



The deletion of  $q_1$  produced a second path from  $q_3$  to  $q_0$ , which is indicated by the union in the expression on the arc from  $q_3$  to  $q_0$ . Removing  $q_2$  produces



with associated language  $(abb^*a)^*(b \cup bb)(a(abb^*a)^*(b \cup bb))^*$ .  $\square$

The results of the previous two sections yield a characterization of regular languages originally established by Kleene. The construction outlined in Section 6.1 can be used to build an NFA- $\lambda$  to accept any regular language. Conversely, Algorithm 6.2.2 produces a regular expression for the language accepted by a finite automaton. Using the equivalence of deterministic and nondeterministic machines, Kleene's Theorem can be expressed in terms of languages accepted by deterministic finite automata.

### Theorem 6.2.3 (Kleene)

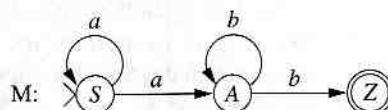
A language  $L$  is accepted by a DFA with alphabet  $\Sigma$  if, and only if,  $L$  is a regular language over  $\Sigma$ .

## 6.3 Regular Grammars and Finite Automata

A context-free grammar is called regular (Section 3.3) if each rule is of the form  $A \rightarrow aB$ ,  $A \rightarrow a$ , or  $A \rightarrow \lambda$ . A string derivable in a regular grammar contains at most one variable which, if present, occurs as the rightmost symbol. A derivation is terminated by the application of a rule of the form  $A \rightarrow a$  or  $A \rightarrow \lambda$ .

The language  $a^+b^+$  is generated by the grammar  $G$  and accepted by the NFA  $M$

$$\begin{aligned} G: S &\rightarrow aS \mid aA \\ A &\rightarrow bA \mid b \end{aligned}$$



where the states of  $M$  have been named  $S$ ,  $A$ , and  $Z$  to simplify the comparison of computation and generation. The computation of  $M$  that accepts  $aabb$  is given along with the derivation that generates the string in  $G$ .

| Derivation         | Computation                 | String Processed |
|--------------------|-----------------------------|------------------|
| $S \Rightarrow aS$ | $[S, aabb] \vdash [S, abb]$ | $a$              |
| $\Rightarrow aaA$  | $\vdash [A, bb]$            | $aa$             |
| $\Rightarrow aabA$ | $\vdash [A, b]$             | $aab$            |
| $\Rightarrow aabb$ | $\vdash [Z, \lambda]$       | $aabb$           |

A computation in an automaton begins with the input string, sequentially processes the leftmost symbol, and halts when the entire string has been analyzed. Generation, on the other hand, begins with the start symbol of the grammar and adds terminal symbols to the prefix of the derived sentential form. The derivation terminates with the application of a  $\lambda$ -rule or a rule whose right-hand side is a single terminal.

The example illustrates the correspondence between generating a terminal string with a regular grammar and processing the string by a computation of an automaton. The state of the automaton is identical to the variable in the derived string. A computation terminates when the entire string has been processed, and the result is designated by the final state. The accepting state  $Z$ , which does not correspond to a variable in the grammar, is added to  $M$  to represent the completion of the derivation of  $G$ .

The state diagram of an NFA  $M$  can be constructed directly from the rules of a grammar  $G$ . The states of the automaton consist of the variables of the grammar and, possibly, an additional accepting state. In the previous example, transitions  $\delta(S, a) = S$ ,  $\delta(S, a) = A$ , and  $\delta(A, b) = A$  of  $M$  correspond to the rules  $S \rightarrow aS$ ,  $S \rightarrow aA$ , and  $A \rightarrow bA$  of  $G$ . The left-hand side of the rule represents the current state of the machine. The terminal on the right-hand side is the input symbol. The state corresponding to the variable on the right-hand side of the rule is entered as a result of the transition.

Since the rule terminating a derivation does not add a variable to the string, the consequences of an application of a  $\lambda$ -rule or a rule of the form  $A \rightarrow a$  must be incorporated into the construction of the corresponding automaton.

### Theorem 6.3.1

Let  $G = (V, \Sigma, P, S)$  be a regular grammar. Define the NFA  $M = (Q, \Sigma, \delta, S, F)$  as follows:

- i)  $Q = \begin{cases} V \cup \{Z\} & \text{where } Z \notin V, \text{ if } P \text{ contains a rule } A \rightarrow a \\ V & \text{otherwise.} \end{cases}$
- ii)  $\delta(A, a) = B$  whenever  $A \rightarrow aB \in P$   
 $\delta(A, a) = Z$  whenever  $A \rightarrow a \in P$ .
- iii)  $F = \begin{cases} \{A \mid A \rightarrow \lambda \in P\} \cup \{Z\} & \text{if } Z \in Q \\ \{A \mid A \rightarrow \lambda \in P\} & \text{otherwise.} \end{cases}$

Then  $L(M) = L(G)$ .

**Proof.** The construction of the machine transitions from the rules of the grammar allows every derivation of  $G$  to be traced by a computation in  $M$ . The derivation of a terminal string has the form  $S \Rightarrow \lambda$ ,  $S \xrightarrow{*} wC \Rightarrow wa$ , or  $S \xrightarrow{*} wC \Rightarrow w$  where the derivation  $S \xrightarrow{*} wC$  consists of the application of rules of the form  $A \rightarrow aB$ . Induction can be used to establish the existence of a computation in  $M$  that processes the string  $w$  and terminates in state  $C$  whenever  $wC$  is a sentential form of  $G$  (Exercise 6).

First we show that every string generated by  $G$  is accepted by  $M$ . If  $L(G)$  contains the null string, then  $S$  is an accepting state of  $M$  and  $\lambda \in L(M)$ . The derivation of a nonnull string is terminated by the application of a rule  $C \rightarrow a$  or  $C \rightarrow \lambda$ . In a derivation of the form  $S \xrightarrow{*} wC \Rightarrow wa$ , the final rule application corresponds to the transition  $\delta(C, a) = Z$ ,

causing the machine to halt in the accepting state  $Z$ . A derivation of the form  $S \xrightarrow{*} wC \Rightarrow w$  is terminated by the application of a  $\lambda$ -rule. Since  $C \rightarrow \lambda$  is a rule of  $G$ , the state  $C$  is accepting in  $M$ . The acceptance of  $w$  in  $M$  is exhibited by the computation that corresponds to the derivation  $S \xrightarrow{*} wC$ .

Conversely, we must show that  $L(M) \subseteq L(G)$ . Let  $w = ua$  be a string accepted by  $M$ . A computation accepting  $w$  has the form

$$[S, w] \xleftarrow{*} [B, \lambda], \quad \text{where } B \neq Z,$$

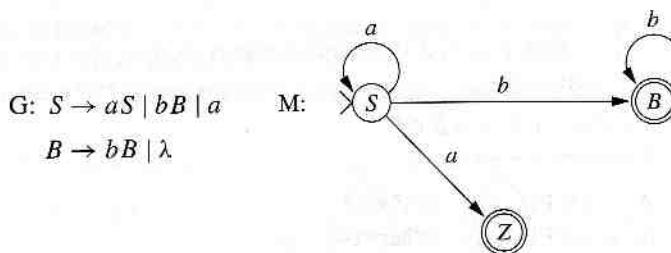
or

$$[S, w] \xleftarrow{*} [A, a] \vdash [Z, \lambda].$$

In the former case,  $B$  is the left-hand side of a  $\lambda$ -rule of  $G$ . The string  $wB$  can be derived by applying the rules that correspond to transitions in the computation. The generation of  $w$  is completed by the application of the  $\lambda$ -rule. Similarly, a derivation of  $uA$  can be constructed from the rules corresponding to the transitions in the computation  $[S, w] \xleftarrow{*} [A, a]$ . The string  $w$  is obtained by terminating this derivation with the rule  $A \rightarrow a$ . Thus every string accepted by  $M$  is in the language of  $G$ . ■

### Example 6.3.1

The grammar  $G$  generates and the NFA  $M$  accepts the language  $a^*(a \cup b^+)$ .



The preceding transformation can be reversed to construct a regular grammar from an NFA. The transition  $\delta(A, a) = B$  produces the rule  $A \rightarrow aB$ . Since every transition results in a new machine state, no rules of the form  $A \rightarrow a$  are produced. The rules obtained from the transitions generate derivations of the form  $S \xrightarrow{*} wC$  that mimic computations in the automaton. Rules must be added to terminate the derivations. When  $C$  is an accepting state, a computation that terminates in state  $C$  exhibits the acceptance of  $w$ . Completing the derivation  $S \xrightarrow{*} wC$  with the application of a rule  $C \rightarrow \lambda$  generates  $w$  in  $G$ . The grammar is completed by adding  $\lambda$ -rules for all accepting states of the automaton. This informal argument justifies Theorem 6.3.2. The formal proof is left as an exercise.

$wC \Rightarrow w$   
state  $C$  is  
corresponds  
to string  $w$ .  
The automaton  
accepts by  $M$ .

is derived by  
construction of  $w$  is  
constructed  
from  $[A, a]$ . The  
every string

### Theorem 6.3.2

Let  $M = (Q, \Sigma, \delta, q_0, F)$  be an NFA. Define a regular grammar  $G = (V, \Sigma, P, q_0)$  as follows:

- i)  $V = Q$ ,
- ii)  $q_i \rightarrow aq_j \in P$  whenever  $\delta(q_i, a) = q_j$ ,
- iii)  $q_i \rightarrow \lambda \in P$  if  $q_i \in F$ .

Then  $L(G) = L(M)$ .

The constructions outlined in Theorems 6.3.1 and 6.3.2 can be applied sequentially to shift from automaton to grammar and back again. Beginning with an NFA  $M$ , the sequence of transformations would have the form

$$M \longrightarrow G \longrightarrow M'$$

Since  $G$  contains only rules of the form  $A \rightarrow aB$  or  $A \rightarrow \lambda$ , the NFA  $M'$  is identical to  $M$ .

A regular grammar  $G$  can be converted to an NFA that, in turn, can be reconverted into a grammar  $G'$ :

$$G \longrightarrow M \longrightarrow G'$$

The grammar  $G'$  that results from these conversions can be obtained directly from  $G$  by adding a single new variable, call it  $Z$ , to the grammar and the rule  $Z \rightarrow \lambda$ . All rules  $A \rightarrow a$  are then replaced by  $A \rightarrow aZ$ .

### Example 6.3.2

The regular grammar  $G'$  that accepts  $L(M)$  is constructed from the automaton  $M$  from Example 6.3.1.

$$G': S \rightarrow aS \mid bB \mid aZ$$

$$B \rightarrow bB \mid \lambda$$

$$Z \rightarrow \lambda$$

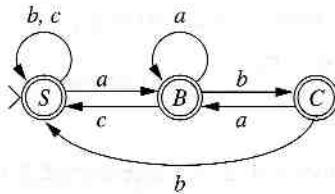
The transitions provide the  $S$  rules and the first  $B$  rule. The  $\lambda$ -rules are added since  $B$  and  $Z$  are accepting states.  $\square$

The two conversions allow us to conclude that the languages generated by regular grammars are precisely those accepted by finite automata. It follows from Theorems 6.2.3 and 6.3.1 that the language generated by a regular grammar is a regular set. The conversion from automaton to regular grammar guarantees that every regular set is generated by some regular grammar. This yields the characterization of regular languages promised in Section 3.3: the languages generated by regular grammars.

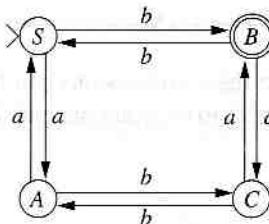
**Example 6.3.3**

The language of the regular grammar from Example 3.2.12 is the set of strings over  $\{a, b, c\}$  that do not contain the substring  $abc$ . Theorem 6.3.1 is used to construct an NFA that accepts this language.

$$\begin{aligned} S &\rightarrow bS \mid cS \mid aB \mid \lambda \\ B &\rightarrow aB \mid cS \mid bC \mid \lambda \\ C &\rightarrow aB \mid bS \mid \lambda \end{aligned}$$

**Example 6.3.4**

A regular grammar with alphabet  $\{a, b\}$  that generates strings with an even number of  $a$ 's and an odd number of  $b$ 's can be constructed from the DFA in Example 5.3.5. This machine is reproduced below with the states  $[e_a, e_b]$ ,  $[o_a, e_b]$ ,  $[e_a, o_b]$ , and  $[o_a, o_b]$  renamed  $S$ ,  $A$ ,  $B$ , and  $C$ , respectively.



The associated grammar is

$$\begin{aligned} S &\rightarrow aA \mid bB \\ A &\rightarrow aS \mid bC \\ B &\rightarrow bS \mid aC \mid \lambda \\ C &\rightarrow aB \mid bA. \end{aligned}$$

## 6.4 Closure Properties of Regular Languages

Regular languages have been defined, generated, and accepted. A language over an alphabet  $\Sigma$  is regular if it is

- i) a regular set (expression) over  $\Sigma$ ,
- ii) accepted by a DFA, NFA, or NFA- $\lambda$ , or
- iii) generated by a regular grammar.

A family of languages is *closed* under an operation if the application of the operation to members of the family produces a member of the family. Each of the equivalent formulations of regularity will be used to demonstrate closure properties of the family of regular languages.

The recursive definition of regular sets establishes closure for the unary operation Kleene star and the binary operations union and concatenation. This was also proved in Theorem 5.5.3 using acceptance by finite-state machines.

#### **Theorem 6.4.1**

Let  $L_1$  and  $L_2$  be two regular languages. The languages  $L_1 \cup L_2$ ,  $L_1L_2$ , and  $L_1^*$  are regular languages.

The regular languages are also closed under complementation. If  $L$  is regular over the alphabet  $\Sigma$ , then so is  $\bar{L} = \Sigma^* - L$ , the set containing all strings in  $\Sigma^*$  that are not in  $L$ . Theorem 5.3.3 used the properties of DFAs to construct a machine that accepts  $\bar{L}$  from one that accepts  $L$ . Complementation and union combine to establish the closure of regular languages under intersection.

#### **Theorem 6.4.2**

Let  $L$  be a regular language over  $\Sigma$ . The language  $\bar{L}$  is regular.

#### **Theorem 6.4.3**

Let  $L_1$  and  $L_2$  be regular languages over  $\Sigma$ . The language  $L_1 \cap L_2$  is regular.

**Proof.** By DeMorgan's Law

$$L_1 \cap L_2 = \overline{(L_1 \cup L_2)}.$$

The right-hand side of the equality is regular since it is built from  $L_1$  and  $L_2$  using union and complementation. ■

Closure properties provide additional tools for establishing the regularity of languages. The operations of complementation and intersection, as well as union, concatenation, and Kleene star, preserve regularity when combining regular languages.

---

#### **Example 6.4.1**

Let  $L$  be the language over  $\{a, b\}$  consisting of all strings that contain the substring  $aa$  but do not contain  $bb$ . The regular languages  $L_1 = (a \cup b)^*aa(a \cup b)^*$  and  $L_2 = (a \cup b)^*bb(a \cup b)^*$  consist of strings containing substrings  $aa$  and  $bb$ , respectively. Hence,  $\bar{L} = L_1 \cap \bar{L}_2$  is regular. □

**Example 6.4.2**

Let  $L$  be any regular language over  $\{a, b\}$ . The language

$$L_1 = \{u \mid u \in L \text{ and } u \text{ has exactly one } a\}$$

is regular. The regular expression  $b^*ab^*$  describes the set of strings with exactly one  $a$ . The language  $L_1 = L \cap b^*ab^*$  is regular since it is the intersection of regular languages.  $\square$

The next example exhibits the robustness of the family of regular languages. Adding or removing a small number, in fact any finite number, of strings cannot turn a regular language into a nonregular language.

**Example 6.4.3**

Let  $L_1$  be a regular language over an alphabet  $\Sigma$  and let  $L_2 \subseteq \Sigma^*$  be any finite set of strings. Then  $L_1 \cup L_2$  and  $L_1 - L_2$  are both regular. The critical observation is that any finite language is regular. Why? The regularity of  $L_1 \cup L_2$  and  $L_1 - L_2$  then follows from the closure of the regular languages under union and set difference (Exercise 8).  $\square$

**Example 6.4.4**

The set  $SUF(L) = \{v \mid uv \in L\}$  consists of all suffixes of strings of the language  $L$ . For example, if  $aabb \in L$ , then  $\lambda$ ,  $b$ ,  $bb$ ,  $abb$ , and  $aabb$  are in  $SUF(L)$ . We will show that if  $L$  is regular, then so is  $SUF(L)$ . Since  $L$  is regular, we know that it is defined by a regular expression, accepted by a finite automaton, and generated by a regular grammar. We may use any of these categorizations of regularity to show that  $SUF(L)$  is regular.

Using the grammatical characterization, we know that  $L$  is generated by a regular grammar  $G = (V, \Sigma, P, S)$ . We may assume that  $G$  has no useless symbols. If it did, we would use the algorithm from Section 4.4 to remove them while preserving the language.

A suffix of  $v$  of  $G$  is produced by a derivation of the form

$$S \xrightarrow{*} uA \xrightarrow{*} uv.$$

Intuitively, we would like to add a rule  $S \rightarrow A$  to  $G$  to directly generate the suffix

$$S \Rightarrow A \xrightarrow{*} v.$$

Unfortunately, the resulting grammar would not be regular. To fix that problem, we will use grammar transformations from Chapter 4.

We begin by defining a new grammar  $G' = (V', \Sigma, P', S')$  by

$$V' = V \cup \{S'\}$$

$$P' = P \cup \{S' \rightarrow A \mid A \in V\}.$$

A derivation in  $G'$  uses only one rule not in  $G$ . Any string in  $L$  is produced by a derivation of the form

$$S' \Rightarrow S \xrightarrow{*} w,$$

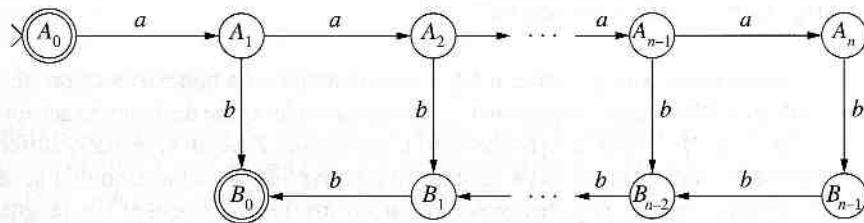
while the remaining suffixes are generated by

$$S' \Rightarrow A \xrightarrow{*} w.$$

Consequently,  $L(G) = \text{SUF}(L)$ . We can obtain an equivalent regular grammar by removing  $\lambda$ -rules and chain rules from  $G'$ .  $\square$

## 6.5 A Nonregular Language

The incompletely specified DFA



accepts the language  $\{a^i b^i \mid i \leq n\}$ . The states  $A_i$  count the number of leading  $a$ 's in the input string. Upon processing the first  $b$ , the machine enters the sequence of states labeled  $B_i$ . The accepting state  $B_0$  is entered when an equal number of  $b$ 's are processed. This strategy cannot be extended to accept the language  $L = \{a^i b^i \mid i \geq 0\}$  since it would require infinitely many states. However, there may be other strategies and machines that accept  $L$  that only require finitely many states. We will show that this is not the case, that  $L$  is not accepted by any DFA and therefore is not a regular language.

The proof of the nonregularity of the language  $L = \{a^i b^i \mid i \geq 0\}$  is by contradiction. We assume that there is a DFA that accepts  $L$  and show that it must have states that record the number of  $a$ 's in the same manner as the states  $A_1, A_2, \dots$  in the preceding diagram. It follows that the machine must have infinitely many states, which contradicts the requirement that a DFA has only finitely many states. The contradiction allows us to conclude that no DFA can accept  $L$ .

We begin with the assumption that  $L$  is accepted by some DFA, call it  $M$ . The extended transition function  $\hat{\delta}$  is used to show that the automaton  $M$  must have an infinite number of states. Let  $A_i$  be the state of the machine entered upon processing the string  $a^i$ ; that is,  $\hat{\delta}(q_0, a^i) = A_i$ . For all  $i, j \geq 0$  with  $i \neq j$ ,  $a^i b^i \in L$  and  $a^j b^j \notin L$ . Hence,  $\hat{\delta}(q_0, a^i b^i) \neq \hat{\delta}(q_0, a^j b^j)$  since the former is an accepting state and the latter rejecting. Now

$$\hat{\delta}(q_0, a^i b^i) = \hat{\delta}(\hat{\delta}(q_0, a^i), b^i) = \hat{\delta}(A_i, b^i) \in L$$

and

$$\hat{\delta}(q_0, a^j b^i) = \hat{\delta}(\hat{\delta}(q_0, a^j), b^i) = \hat{\delta}(A_j, b^i) \notin L.$$

Consequently,  $\hat{\delta}(A_i, b^i) \neq \hat{\delta}(A_j, b^i)$ . In a deterministic machine, two computations that begin in the same state and process the same string must end in the same state. Since the computations  $\hat{\delta}(A_i, b^i)$  and  $\hat{\delta}(A_j, b^i)$  process the same string but terminate in different states, we conclude that  $A_i \neq A_j$ .

We have shown that states  $A_i$  and  $A_j$  are distinct for all values of  $i \neq j$ . Any deterministic finite-state machine that accepts  $L$  must contain an infinite sequence of states corresponding to  $A_0, A_1, A_2, \dots$ . This violates the restriction that limits a DFA to a finite number of states. Consequently, there is no DFA that accepts  $L$ , or equivalently,  $L$  is not regular. The preceding argument justifies Theorem 6.5.1.

### Theorem 6.5.1

The language  $\{a^i b^i \mid i \geq 0\}$  is not regular.

The argument establishing Theorem 6.5.1 is an example of a nonexistence proof. We have shown that no DFA can be constructed, no matter how clever the designer, to accept the language  $\{a^i b^i \mid i \geq 0\}$ . Proofs of existence and nonexistence have an essentially different flavor. A language can be shown to be regular by constructing an automaton that accepts it. A proof of nonregularity requires proving that no machine can accept the language. Theorem 6.5.1 can be generalized to establish the nonregularity of a number of languages.

### Corollary 6.5.2 (to the proof of Theorem 6.5.1)

Let  $L$  be a language over  $\Sigma$ . If there are sequences of distinct strings  $u_i \in \Sigma^*$  and  $v_i \in \Sigma^*$ ,  $i \geq 0$ , with  $u_i v_i \in L$  and  $u_i v_j \notin L$  for  $i \neq j$ , then  $L$  is not a regular language.

The proof is identical to that of Theorem 6.5.1, with  $u_i$  replacing  $a^i$  and  $v_i$  replacing  $b^i$ .

---

### Example 6.5.1

The set  $L$  of palindromes over  $\{a, b\}$  is not regular. By Corollary 6.5.2, it is sufficient to discover two sequences of strings  $u_i$  and  $v_i$  that satisfy  $u_i v_i \in L$  and  $u_i v_j \notin L$  for all  $i \neq j$ . The strings

$$u_i = a^i b$$

$$v_i = a^i$$

fulfill these requirements. □

**Example 6.5.2**

Grammars were introduced as a formal structure for defining the syntax of languages. Corollary 6.5.2 can be used to show that regular grammars are not a sufficiently powerful tool to define programming languages containing arithmetic or Boolean expressions in infix form. The grammar AE

$$\begin{aligned} \text{AE: } S &\rightarrow A \\ A &\rightarrow T \mid A + T \\ T &\rightarrow b \mid (A) \end{aligned}$$

generates additive expressions using  $+$ , parentheses, and the operand  $b$ . For example,  $(b)$ ,  $b + (b)$ , and  $((b))$  are in  $L(\text{AE})$ .

Infix notation permits—in fact, requires—the nesting of parentheses. The derivation

$$\begin{aligned} S &\Rightarrow T \\ &\Rightarrow (A) \\ &\Rightarrow (T) \\ &\Rightarrow (b) \end{aligned}$$

exhibits the generation of the string  $(b)$  using the rules of AE. Repeated applications of the sequence of rules  $T \Rightarrow (A) \Rightarrow (T)$  before terminating the derivation with the application of the rule  $T \rightarrow b$  generates the strings  $((b))$ ,  $((((b))))$ , . . . . The strings  $(^i b)$  and  $(^i)$  satisfy the requirements of the sequences  $u_i$  and  $v_i$  of Corollary 6.5.2. Thus the language defined by the grammar AE is not regular. A similar argument can be used to show that programming languages such as C, C++, and Java, among others, are not regular.  $\square$

Just as the closure properties of regular languages can be used to establish regularity, they can also be used to demonstrate the nonregularity of languages.

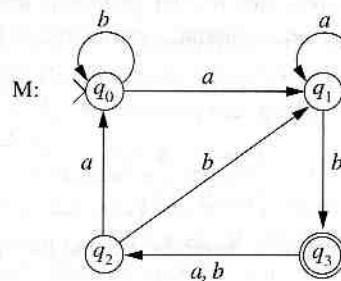
**Example 6.5.3**

The language  $L = \{a^i b^j \mid i, j \geq 0 \text{ and } i \neq j\}$  is not regular. If  $L$  is regular then, by Theorems 6.4.2 and 6.4.3, so is  $\bar{L} \cap a^* b^*$ . But  $\bar{L} \cap a^* b^* = \{a^i b^i \mid i \geq 0\}$ , which we know is not regular.  $\square$

## 6.6 The Pumping Lemma for Regular Languages

The existence of nonregular languages was established in the previous section by demonstrating the impossibility of constructing a DFA to accept the language. In this section a more general criterion for establishing nonregularity is developed. The main result, the pumping lemma for regular languages, requires strings in a regular language to admit decompositions satisfying certain repetition properties.

Pumping a string refers to constructing new strings by repeating (pumping) substrings in the original string. Acceptance in the state diagram of the DFA



illustrates pumping strings. Consider the string  $z = ababbaaab$  in  $L(M)$ . This string can be decomposed into substrings  $u$ ,  $v$ , and  $w$  where  $u = a$ ,  $v = bab$ ,  $w = baaab$ , and  $z = uvw$ . The strings  $a(bab)^i baaab$  are obtained by pumping the substring  $bab$  in  $ababbaaab$ .

As usual, processing  $z$  in the DFA  $M$  corresponds to generating a path in the state diagram of  $M$ . The decomposition of  $z$  into  $u$ ,  $v$ , and  $w$  breaks the path in the state diagram into three subpaths. The subpaths generated by the computation of substrings  $u = a$  and  $w = baaab$  are  $q_0$ ,  $q_1$  and  $q_1$ ,  $q_3$ ,  $q_2$ ,  $q_0$ ,  $q_1$ ,  $q_3$ . Processing the second component of the decomposition generates the cycle  $q_1$ ,  $q_3$ ,  $q_2$ ,  $q_1$ . The pumped strings  $uv^iw$  are also accepted by the DFA since the repetition of the substring  $v$  simply adds additional trips around the cycle  $q_1$ ,  $q_3$ ,  $q_2$ ,  $q_1$  before the processing of  $w$  terminates the computation in state  $q_3$ .

The pumping lemma requires the existence of such a decomposition for all sufficiently long strings in the language of a DFA. Two lemmas are presented establishing conditions guaranteeing the existence of cycles in paths in the state diagram of a DFA. The proofs utilize a simple counting argument known as the *pigeonhole principle*. This principle is based on the observation that given a number of boxes and a greater number of items to be distributed among them, at least one of the boxes must receive more than one item.

### **Lemma 6.6.1**

Let  $G$  be the state diagram of a DFA with  $k$  states. Any path of length  $k$  in  $G$  contains a cycle.

**Proof.** A path of length  $k$  contains  $k + 1$  nodes. Since there are only  $k$  nodes in  $G$ , there must be a node, call it  $q_i$ , that occurs in at least two positions in the path. The subpath from the first occurrence of  $q_i$  to the second produces the desired cycle. ■

Paths with length greater than  $k$  can be divided into an initial subpath of length  $k$  and the remainder of the path. Lemma 6.6.1 guarantees the existence of a cycle in the initial subpath. The preceding remarks are formalized in Corollary 6.6.2.

substrings

### Corollary 6.6.2

Let  $G$  be the state diagram of a DFA with  $k$  states and let  $p$  be a path of length  $k$  or more. The path  $p$  can be decomposed into subpaths  $q$ ,  $r$ , and  $s$  where  $p = qrs$ , the length of  $qr$  is less than or equal to  $k$ , and  $r$  is a cycle.

### Theorem 6.6.3 (Pumping Lemma for Regular Languages)

Let  $L$  be a regular language that is accepted by a DFA  $M$  with  $k$  states. Let  $z$  be any string in  $L$  with  $\text{length}(z) \geq k$ . Then  $z$  can be written  $uvw$  with  $\text{length}(uv) \leq k$ ,  $\text{length}(v) > 0$ , and  $uv^iw \in L$  for all  $i \geq 0$ .

**Proof.** Let  $z \in L$  be a string with length  $n \geq k$ . Processing  $z$  in  $M$  generates a path of length  $n$  in the state diagram of  $M$ . By Corollary 6.6.2, this path can be broken into subpaths  $q$ ,  $r$ , and  $s$ , where  $r$  is a cycle in the state diagram. The decomposition of  $z$  into  $u$ ,  $v$ , and  $w$  consists of the strings spelled by the paths  $q$ ,  $r$ , and  $s$ . ■

The paths corresponding to the strings  $uv^iw$  begin and end at the same nodes as the computation for  $uvw$ . The sole difference is the number of trips around the cycle  $r$ . Consequently, if  $uvw$  is accepted by  $M$ , then so is  $uv^iw$ .

Properties of the particular DFA that accepts the language  $L$  are not specifically mentioned in the proof of the pumping lemma. The argument holds for all such DFAs, including the DFA with the minimal number of states. The statement of the theorem could be strengthened to specify  $k$  as the number of states in the minimal DFA accepting  $L$ .

The pumping lemma is a powerful tool for proving that languages are not regular. Every string of length  $k$  or more in a regular language, where  $k$  is the value specified by the pumping lemma, must have an appropriate decomposition. To show that a language is not regular, it suffices to find one string that does not satisfy the conditions of the pumping lemma. The use of the pumping lemma to establish nonregularity is illustrated in the following examples. The technique consists of choosing a string  $z$  in  $L$  and showing that there is no decomposition  $uvw$  of  $z$  for which  $uv^iw$  is in  $L$  for all  $i \geq 0$ .

The first two examples show that computations of a finite state machine are not sufficiently powerful to determine whether a number is a perfect square or a prime.

string can be  
nd  $z = uvw$ .  
 $bbaaab$ .

in the state  
state diagram  
gs  $u = a$  and  
component of  
 $v^i w$  are also  
ditional trips  
mputation in

ll sufficiently  
ng conditions  
A. The proofs  
is principle is  
of items to be  
e item.

$1 G$  contains a

des in  $G$ , there  
e subpath from

of length  $k$  and  
le in the initial

### Example 6.6.1

Let  $L = \{z \in \{a\}^* \mid \text{length}(z) \text{ is a perfect square}\}$ . Assume that  $L$  is regular. This implies that  $L$  is accepted by some DFA. Let  $k$  be the number of states of the DFA. By the pumping lemma, every string  $z \in L$  of length  $k$  or more can be decomposed into substrings  $u$ ,  $v$ , and  $w$  such that  $\text{length}(uv) \leq k$ ,  $v \neq \lambda$ , and  $uv^iw \in L$  for all  $i \geq 0$ .

Consider the string  $z = a^{k^2}$  of length  $k^2$ . Since  $z$  is in  $L$  and its length is greater than  $k$ ,  $z$  can be written  $z = uvw$  where the  $u$ ,  $v$ , and  $w$  satisfy the conditions of the pumping lemma.

In particular,  $0 < \text{length}(v) \leq k$ . This observation can be used to place an upper bound on the length of  $uv^2w$ :

$$\begin{aligned}\text{length}(uv^2w) &= \text{length}(uvw) + \text{length}(v) \\ &= k^2 + \text{length}(v) \\ &\leq k^2 + k \\ &< k^2 + 2k + 1 \\ &= (k + 1)^2.\end{aligned}$$

The length of  $uv^2w$  is greater than  $k^2$  and less than  $(k + 1)^2$  and therefore is not a perfect square. Thus the string  $uv^2w$  obtained by pumping  $v$  once is not in  $L$ . We have shown that there is no decomposition of  $z$  that satisfies the conditions of the pumping lemma. The assumption that  $L$  is regular leads to a contradiction, establishing the nonregularity of  $L$ .  $\square$

### Example 6.6.2

To show that the language  $L = \{a^i \mid i \text{ is prime}\}$  is not regular, we assume that there is a DFA with some number  $k$  states that accepts it. Let  $n$  be a prime greater than  $k$ . The pumping lemma implies that  $a^n$  can be decomposed into substrings  $uvw$ ,  $v \neq \lambda$ , such that  $uv^iw$  is in  $L$  for all  $i \geq 0$ . Assume that such a decomposition exists.

If  $uv^{n+1}w \in L$ , then its length must be prime. But

$$\begin{aligned}\text{length}(uv^{n+1}w) &= \text{length}(uvv^n w) \\ &= \text{length}(uvw) + \text{length}(v^n) \\ &= n + n(\text{length}(v)) \\ &= n(1 + \text{length}(v)).\end{aligned}$$

Since its length is not prime,  $uv^{n+1}w$  is not in  $L$ . Thus there is no division of  $a^n$  into  $uvw$  that satisfies the pumping lemma and we conclude that  $L$  is not regular.  $\square$

In the preceding examples, the constraints on the length of the strings were sufficient to prove that the languages were not regular. Often the numeric relationships among the elements of a string are used to show that there is no substring that satisfies the conditions of the pumping lemma. We will now present another argument, this time using the pumping lemma, that demonstrates the nonregularity of  $\{a^i b^i \mid i \geq 0\}$ .

### Example 6.6.3

To show that  $L = \{a^i b^i \mid i \geq 0\}$  is not regular, we must find a string in  $L$  of appropriate length that has no pumpable substring. Assume that  $L$  is regular and let  $k$  be the number specified

$\exists$  bound on

by the pumping lemma. Let  $z$  be the string  $a^k b^k$ . Any decomposition of  $uvw$  of  $z$  satisfying the conditions of the pumping lemma must have the form

$$a^i \quad a^j \quad a^{k-i-j} b^k,$$

where  $i + j \leq k$  and  $j > 0$ . Pumping any substring of this form produces  $uv^2w = a^i a^j a^j a^{k-i-j} b^k = a^k a^j b^k$ , which is not in L. Since  $z \in L$  has no decomposition that satisfies the conditions of the pumping lemma, we conclude that L is not regular.  $\square$

not a perfect have shown lemma. The regularity of  $L$ ,

### Example 6.6.4

The language  $L = \{a^i b^m c^n \mid 0 < i, 0 < m < n\}$  is not regular. Assume that  $L$  is accepted by a DFA with  $k$  states. Then, by the pumping lemma, every string  $z \in L$  with length  $k$  or more can be written  $z = uvw$ , with  $\text{length}(uv) \leq k$ ,  $\text{length}(v) > 0$ , and  $uv^i w \in L$  for all  $i \geq 0$ .

Consider the string  $z = ab^k c^{k+1}$ , which is in  $L$ . We must show that there is no suitable decomposition of  $z$ . Any decomposition of  $z$  must have one of two forms, and the cases are examined separately.

Case 1: A decomposition in which  $a \notin v$  has the form

$$ab^i \quad b^j \quad b^{k-i-j}c^{k+1}$$

where  $i + j \leq k - 1$  and  $j > 0$ . Pumping  $v$  produces  $uv^2w = ab^i b^j b^j b^{k-i-j} c^{k+1} = ab^k b^j c^{k+1}$ , which is not in L.

Case 2: A decomposition of  $z$  in which  $a \in v$  has the form

$$\begin{array}{ccc} u & v & w \\ \lambda & ab^i & b^{k-i}c^{k+1} \end{array}$$

where  $i \leq k - 1$ . Pumping  $v$  zero times produces  $uv^0w = b^{k-i}c^{k+1}$ , which is not in  $L$  since it does not contain an  $a$ .

Since  $ab^k c^{k+1}$  has no decomposition with a “pumpable” substring, L is not regular.  $\square$

of  $a^n$  into  $uvw$

were sufficient tips among the the conditions being the pumping

proper length  
number specified

### Theorem 6.6.4

Let  $M$  be a DFA with  $k$  states.

- i)  $L(M)$  is not empty if, and only if,  $M$  accepts a string  $z$  with  $\text{length}(z) < k$ .
  - ii)  $L(M)$  has an infinite number of members if, and only if,  $M$  accepts a string  $z$  where  $k \leq \text{length}(z) < 2k$ .

**Proof.**

i)  $L(M)$  is clearly not empty if a string of length less than  $k$  is accepted by  $M$ .

Now let  $M$  be a machine whose language is not empty and let  $z$  be the smallest string in  $L(M)$ . Assume that the length of  $z$  is greater than  $k - 1$ . By the pumping lemma,  $z$  can be written  $uvw$  where  $uv^iw \in L$ . In particular,  $uv^0w = uw$  is a string smaller than  $z$  in  $L$ . This contradicts the assumption of the minimality of the length of  $z$ . Therefore,  $\text{length}(z) < k$ .

ii) If  $M$  accepts a string  $z$  with  $k \leq \text{length}(z) < 2k$ , then  $z$  can be written  $uvw$  where  $u, v$ , and  $w$  satisfy the conditions of the pumping lemma. This implies that the strings  $uv^iw$  are in  $L$  for all  $i \geq 0$ .

Assume that  $L(M)$  is infinite. We must show that there is a string whose length is between  $k$  and  $2k - 1$  in  $L(M)$ . Since there are only finitely many strings over a finite alphabet with length less than  $k$ ,  $L(M)$  must contain strings of length greater than  $k - 1$ . Choose a string  $z \in L(M)$  whose length is as small as possible but greater than  $k - 1$ . If  $k \leq \text{length}(z) < 2k$ , there is nothing left to show. Assume that  $\text{length}(z) \geq 2k$ . By the pumping lemma,  $z = uvw$ ,  $\text{length}(v) \leq k$ , and  $uv^0w = uw \in L(M)$ . But this is a contradiction since  $uw$  is a string whose length is greater than  $k - 1$  but strictly smaller than the length of  $z$ . ■

The preceding result establishes a decision procedure for determining the cardinality of the language of a DFA. If  $k$  is the number of states and  $j$  the size of the alphabet of the automaton, there are  $(j^k - 1)/(j - 1)$  strings having length less than  $k$ . By Theorem 6.6.4, testing each of these determines whether the language is empty. Testing all strings with length between  $k$  and  $2k - 1$  resolves the question of finite or infinite. This, of course, is an extremely inefficient procedure. Nevertheless, it is effective, yielding the following corollary.

**Corollary 6.6.5**

Let  $M$  be a DFA. There is an algorithm that determines whether  $L(M)$  is empty, finite, or infinite.

The closure properties of regular language can be combined with Corollary 6.6.5 to develop a decision procedure that determines whether two DFAs accept the same language.

**Corollary 6.6.6**

Let  $M_1$  and  $M_2$  be two DFAs. There is a decision procedure to determine whether  $M_1$  and  $M_2$  are equivalent.

**Proof.** Let  $L_1$  and  $L_2$  be the languages accepted by  $M_1$  and  $M_2$ . By Theorems 6.4.1, 6.4.2, and 6.4.3, the language

$$L = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$$

is regular.  $L$  is empty if, and only if,  $L_1$  and  $L_2$  are identical. By Corollary 6.6.5, there is a decision procedure to determine whether  $L$  is empty, or equivalently, whether  $M_1$  and  $M_2$  accept the same language. ■

## 6.7 The Myhill-Nerode Theorem

Kleene's Theorem established the relationship between regular languages and finite automata. In this section regularity is characterized by the existence of an equivalence relation on the strings of the language. This characterization provides a method for obtaining the minimal state DFA that accepts a regular language and provides the justification for the DFA minimization presented in Algorithm 5.7.2.

### Definition 6.7.1

Let  $L$  be a language over  $\Sigma$ . Strings  $u, v \in \Sigma^*$  are indistinguishable in  $L$  if, for every  $w \in \Sigma^*$ , either  $uw$  and  $vw$  are both in  $L$  or neither  $uw$  nor  $vw$  is in  $L$ .

Using membership in  $L$  as the criterion for differentiating strings,  $u$  and  $v$  are distinguishable if there is some string  $w$  whose concatenation with  $u$  and  $v$  produces strings with different membership values in  $L$ . That is,  $w$  distinguishes  $u$  and  $v$  if one of  $uw$  and  $vw$  is in  $L$  and the other is not.

Indistinguishability in a language  $L$  defines a binary relation  $\equiv_L$  on  $\Sigma^*$ ;  $u \equiv_L v$  if  $u$  and  $v$  are indistinguishable. It is easy to see that  $\equiv_L$  is reflexive, symmetric, and transitive. These observations provide the basis for Lemma 6.7.2.

### Lemma 6.7.2

For any language  $L$ , the relation  $\equiv_L$  is an equivalence relation.

### Example 6.7.1

Let  $L$  be the regular language  $a(a \cup b)(bb)^*$ . Strings  $aa$  and  $ab$  are indistinguishable since, for any  $w$ ,  $aa w$  and  $ab w$  are either both in  $L$  or both not in  $L$ . The former arises when  $w$  consists of an even number of  $b$ 's and the latter for any other string. The pair of strings  $b$  and  $ba$  are also indistinguishable in  $L$  since  $bw$  and  $baw$  are not in  $L$  for any string  $w$ . Strings  $a$  and  $ab$  are distinguishable in  $L$  since concatenating  $bb$  to  $a$  produces  $abb \notin L$  and to  $ab$  produces  $abbb \in L$ .

The equivalence classes of  $\equiv_L$  are

| Representative Element | Equivalence Class                                                                        |
|------------------------|------------------------------------------------------------------------------------------|
| $[\lambda]_{\equiv_L}$ | $\lambda$                                                                                |
| $[b]_{\equiv_L}$       | $b(a \cup b)^* \cup a(a \cup b)(bb)^*a(a \cup b)^* \cup a(a \cup b)(bb)^*ba(a \cup b)^*$ |
| $[a]_{\equiv_L}$       | $a$                                                                                      |
| $[aa]_{\equiv_L}$      | $a(a \cup b)(bb)^*$                                                                      |
| $[aab]_{\equiv_L}$     | $a(a \cup b)b(bb)^*$                                                                     |

**Example 6.7.2**

Let  $L$  be the language  $\{a^i b^i \mid i \geq 0\}$ . The strings  $a^i$  and  $a^j$ , where  $i \neq j$ , are distinguishable in  $L$ . Concatenating  $b^i$  produces  $a^i b^i \in L$  and  $a^j b^i \notin L$ . Thus each string  $a^i$ ,  $i = 0, 1, \dots$ , is in a different equivalence class. This example shows that the indistinguishability relation  $\equiv_L$  may generate infinitely many equivalence classes.  $\square$

The equivalence relation  $\equiv_L$  defines indistinguishability on the basis of membership in the language  $L$ . We now define the indistinguishability of strings on the basis of computations of a DFA.

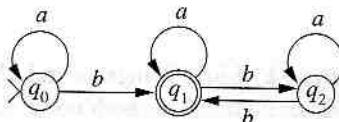
**Definition 6.7.3**

Let  $M = (Q, \Sigma, \delta, q_0, F)$  be a DFA that accepts  $L$ . Strings  $u, v \in \Sigma^*$  are indistinguishable by  $M$  if  $\hat{\delta}(q_0, u) = \hat{\delta}(q_0, v)$ .

Strings  $u$  and  $v$  are indistinguishable by  $M$  if the computation of  $M$  with input  $u$  halts in the same state as the computation with  $v$ . It is easy to see that indistinguishability defined in this manner is also an equivalence relation over  $\Sigma^*$ . Each state  $q_i$  of  $M$  that is reachable by computations of  $M$  has an associated equivalence class: the set of all strings whose computations halt in  $q_i$ . Thus the number of equivalence classes of a DFA  $M$  is at most the number of states of  $M$ . Indistinguishability by a machine  $M$  will be denoted  $\equiv_M$ .

**Example 6.7.3**

Let  $M$  be the DFA



that accepts the language  $a^*ba^*(ba^*ba^*)^*$ , the set of strings with an odd number of  $b$ 's. The equivalence classes of  $\Sigma^*$  defined by the relation  $\equiv_M$  are

| State | Associated Equivalence Class |
|-------|------------------------------|
| $q_0$ | $a^*$                        |
| $q_1$ | $a^*ba^*(ba^*ba^*)^*$        |
| $q_2$ | $a^*ba^*ba^*(ba^*ba^*)^*$    |

Indistinguishability relations can be used to provide additional characterizations of regularity. These characterizations use the *right-invariance* of the indistinguishability equivalence relations. An equivalence relation  $\equiv$  over  $\Sigma^*$  is said to be right-invariant if  $u \equiv v$  implies  $uw \equiv vw$  for every  $w \in \Sigma^*$ . Both  $\equiv_L$  and  $\equiv_M$  are right-invariant.

**Theorem 6.7.4 (Myhill-Nerode)**

The following are equivalent:

- i)  $L$  is regular over  $\Sigma$ .
- ii) There is a right-invariant equivalence relation  $\equiv$  on  $\Sigma^*$  with finitely many equivalence classes such that  $L$  is the union of a subset of the equivalence classes of  $\equiv$ .
- iii)  $\equiv_L$  has finitely many equivalence classes.

**Proof.**

Condition (i) implies condition (ii): Since  $L$  is regular, it is accepted by some DFA  $M = (Q, \Sigma, \delta, q_0, F)$ . We will show that  $\equiv_M$  satisfies the conditions of statement (ii). As previously noted,  $\equiv_M$  has at most as many equivalence classes as  $M$  has states. Consequently, the number of equivalence classes of  $\equiv_M$  is finite. Right-invariance follows from the determinism of the computations of  $M$ , which ensures that  $\hat{\delta}(q_0, uw) = \hat{\delta}(q_0, vw)$  whenever  $\hat{\delta}(q_0, u) = \hat{\delta}(q_0, v)$ .

It remains to show that  $L$  is the union of some of the equivalence classes of  $\equiv_M$ . For each state  $q_i$  of  $M$ , there is an equivalence class consisting of the strings whose computations halt in  $q_i$ . The language  $L$  is the union of the equivalence classes associated with the accepting states of  $M$ .

Condition (ii) implies condition (iii): Let  $\equiv$  be an equivalence relation that satisfies (ii). We begin by showing that every  $\equiv$  equivalence class  $[u]_\equiv$  is a subset of the  $\equiv_L$  equivalence class  $[u]_{\equiv_L}$ .

Let  $u$  and  $v$  be any two strings from  $[u]_\equiv$ ; that is,  $u \equiv v$ . By right-invariance,  $uw \equiv vw$  for any  $w \in \Sigma^*$ . Thus  $uw$  and  $vw$  are in the same  $\equiv$  equivalence class. Since  $L$  is the union of some set of equivalence classes of  $\equiv$ , every string in a particular  $\equiv$  equivalence class has the same membership value in  $L$ . Consequently,  $uw$  and  $vw$  are either both in  $L$  or both not in  $L$ . It follows that  $u$  and  $v$  are in the same equivalence class of  $\equiv_L$ .

Since  $[u]_\equiv \subseteq [u]_{\equiv_L}$  for every string  $u \in \Sigma^*$ , there is at least one  $\equiv$  equivalence class in each of the  $\equiv_L$  equivalence classes. It follows that the number of equivalence classes of  $\equiv_L$  is no greater than the number of equivalence classes of  $\equiv$ , which is finite.

Condition (iii) implies condition (i): To prove that  $L$  is regular when  $\equiv_L$  has only a finite number of equivalence classes, we will build a DFA  $M_L$  that accepts  $L$ . The alphabet of  $M_L$  consists of the symbols in  $L$  and the states are the equivalence classes of  $\equiv_L$ . The start state is the equivalence class containing  $\lambda$ . An equivalence class is an accepting state if it contains an element  $u \in L$ . All that remains is to define the transition function and show that the language of  $M_L$  is  $L$ .

For a symbol  $a \in \Sigma$ , we define  $\delta([u]_{\equiv_L}, a) = [ua]_{\equiv_L}$ . By this definition, the result of a transition from state  $[u]_{\equiv_L}$  with symbol  $a$  is the equivalence class  $[ua]_{\equiv_L}$ . We must show that the definition of the transition is independent of the choice of a particular element from the equivalence class  $[u]_{\equiv_L}$ .

Let  $u$  and  $v$  be two strings in  $L$  that are  $\equiv_L$  equivalent. For the transition function  $\delta$  to be well defined,  $[ua]_{\equiv_L}$  must be the same equivalence class as  $[va]_{\equiv_L}$ , or equivalently,

$ua \equiv_L va$ . To establish this, we need to show that for any string  $x \in \Sigma^*$ ,  $uax$  and  $vax$  are either both in  $L$  or both not in  $L$ . By the definition of  $\equiv_L$ ,  $uw$  and  $vw$  are both in  $L$  or both not in  $L$  for any  $w \in \Sigma^*$ . Letting  $w = ax$  gives the desired result.

All that remains is to show that  $L(M_L) = L$ . For any string  $u$ ,  $\hat{\delta}([\lambda]_{\equiv_L}, u) = [u]_{\equiv_L}$ . If  $u$  is in  $L$ , the computation  $\hat{\delta}([\lambda]_{\equiv_L}, u)$  halts in the accepting state  $[u]_{\equiv_L}$ . Exercise 25 shows that either all of the elements in an equivalence  $[u]_{\equiv_L}$  are in  $L$  or none of the elements are in  $L$ . Thus if  $u \notin L$ , then  $[u]_{\equiv_L}$  is not an accepting state. It follows that a string  $u$  is accepted by  $M_L$  if, and only if,  $u \in L$ .

Note that the equivalence classes of  $\equiv_L$  are precisely those of  $\equiv_{M_L}$ , the indistinguishability relation over  $\Sigma^*$  generated by the machine  $M_L$ . ■

#### Example 6.7.4

The DFA  $M$  from Example 5.7.1 accepts the language  $(a \cup b)(a \cup b^*)$ . The eight equivalence classes of the relation  $\equiv_M$  with the associated states of  $M$  are

| State | Equivalence Class | State | Equivalence Class                                                      |
|-------|-------------------|-------|------------------------------------------------------------------------|
| $q_0$ | $\lambda$         | $q_4$ | $b$                                                                    |
| $q_1$ | $a$               | $q_5$ | $ba$                                                                   |
| $q_2$ | $aa$              | $q_6$ | $bb^+$                                                                 |
| $q_3$ | $ab^+$            | $q_7$ | $(aa(a \cup b) \cup ab^+ a \cup ba(a \cup b) \cup bb^+ a)(a \cup b)^*$ |

The equivalence relation  $\equiv_L$  identifies strings  $u$  and  $v$  as indistinguishable if for any  $w$ , either both  $uw$  and  $vw$  are in  $L$  or both are not in  $L$ . The  $\equiv_L$  equivalence classes of the language  $(a \cup b)(a \cup b^*)$  are

| $\equiv_L$ Equivalence Classes |                                                                        |
|--------------------------------|------------------------------------------------------------------------|
| $[\lambda]_{\equiv_L}$         | $\lambda$                                                              |
| $[a]_{\equiv_L}$               | $a \cup b$                                                             |
| $[aa]_{\equiv_L}$              | $aa \cup ba$                                                           |
| $[ab]_{\equiv_L}$              | $ab^+ \cup bb^+$                                                       |
| $[aba]_{\equiv_L}$             | $(aa(a \cup b) \cup ab^+ a \cup ba(a \cup b) \cup bb^+ a)(a \cup b)^*$ |

where the string inside the brackets is a representative element of the class. It is easy to see that the strings within an equivalence class are indistinguishable and that strings from different classes are distinguishable.

If we denote the  $\equiv_M$  equivalence class of strings whose computations halt in state  $q_i$  by  $cl_M(q_i)$ , the relationship between the equivalence classes of  $\equiv_L$  and  $\equiv_M$  is

and  $vax$  are  
in  $L$  or both

$\iota = [u]_{\equiv_L}$ . If  
Exercise 25 shows  
elements are  
 $u$  is accepted

indistinguish-

eight equiva-

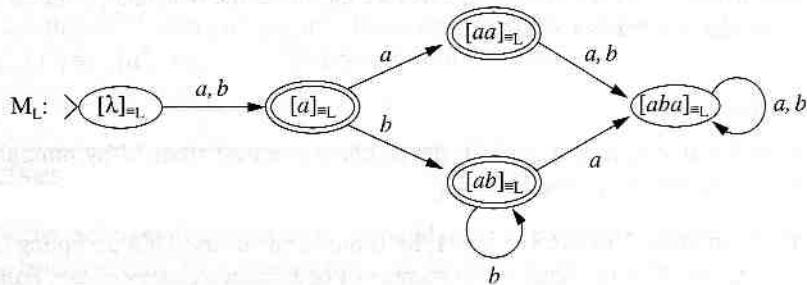
$b^+a)(a \cup b)^*$

able if for any  
e classes of the

ass. It is easy to  
that strings from  
is halt in state  $q_i$   
 $M$  is

$$\begin{aligned} [\lambda]_{\equiv_L} &= cl_M(q_0) \\ [a]_{\equiv_L} &= cl_M(q_1) \cup cl_M(q_4) \\ [aa]_{\equiv_L} &= cl_M(q_2) \cup cl_M(q_5) \\ [ab]_{\equiv_L} &= cl_M(q_3) \cup cl_M(q_6) \\ [aba]_{\equiv_L} &= cl_M(q_7). \end{aligned}$$

Using the technique outlined in the Myhill-Nerode Theorem, we can construct a DFA  $M_L$  accepting  $L$  from the equivalence classes of  $\equiv_L$ . The DFA obtained by this construction is



which is identical to the DFA  $M'$  in Example 5.7.1 obtained using the minimization technique presented in Section 5.7.  $\square$

Theorem 6.7.5 shows that the DFA  $M_L$  obtained from the  $\equiv_L$  equivalence classes is the minimal state DFA that accepts  $L$ .

### Theorem 6.7.5

Let  $L$  be a regular language and  $\equiv_L$  the indistinguishability relation defined by  $L$ . The minimal state DFA accepting  $L$  is the machine  $M_L$  defined from the equivalence classes of  $\equiv_L$  as specified in Theorem 6.7.4.

**Proof.** Let  $M = (Q, \Sigma, \delta, q_0, F)$  be any DFA that accepts  $L$  and let  $\equiv_M$  be the equivalence relation generated by  $M$ . By the Myhill-Nerode Theorem, each equivalence class of  $\equiv_M$  is a subset of an equivalence class of  $\equiv_L$ . Since the equivalence classes of both  $\equiv_M$  and  $\equiv_L$  partition  $\Sigma^*$ ,  $\equiv_M$  must have at least as many equivalence classes as  $\equiv_L$ . Combining the preceding observation with the construction of  $M_L$  from the equivalence classes of  $\equiv_L$ , we see that

$$\begin{aligned} &\text{the number of states of } M \\ &\geq \text{the number of equivalence classes of } \equiv_M \\ &\geq \text{the number of equivalence classes of } \equiv_L \\ &= \text{the number of states of } M_L. \end{aligned}$$

Thus a DFA  $M$  that accepts  $L$  may not have fewer states than  $M_L$ , and we conclude that  $M_L$  is the minimal state DFA that accepts  $L$ . ■

The statement of Theorem 6.7.5 asserts that the  $M_L$  is *the* minimal state DFA that accepts  $L$ . Exercise 31 establishes that all minimal state DFAs accepting  $L$  are identical to  $M_L$ , except possibly for the names assigned to the states.

Theorems 6.7.4 and 6.7.5 establish the existence of a unique minimal state DFA  $M_L$  that accepts a language  $L$ . The minimal state machine can be constructed from the equivalence classes of the relation  $\equiv_L$ . Unfortunately, to this point we have not provided a straightforward method to obtain these equivalence classes. Theorem 6.7.6 shows that the machine whose states are the  $\equiv_L$  equivalence classes is the machine produced by the minimization algorithm in Section 5.7.

### Theorem 6.7.6

Let  $M$  be a DFA that accepts  $L$  and  $M'$  the machine obtained from  $M$  by minimization construction in Section 5.7. Then  $M' = M_L$ .

**Proof.** By Theorem 6.7.5 and Exercise 31,  $M'$  is the minimal state DFA accepting  $L$  if the number of states of  $M'$  is the same as the number of equivalence classes of  $\equiv_L$ . Following Definition 6.7.3, there is an equivalence relation  $\equiv_{M'}$  that associates a set of strings with each state of  $M'$ . The equivalence class of  $\equiv_{M'}$  associated with state  $[q_i]$  is

$$cl_{M'}([q_i]) = \{u \mid \hat{\delta}'([q_0], u) = [q_i]\} = \bigcup_{q_j \in [q_i]} \{u \mid \hat{\delta}(q_0, u) = q_j\},$$

where  $\hat{\delta}'$  and  $\hat{\delta}$  are the extended transition functions of  $M'$  and  $M$ , respectively. By the Myhill-Nerode Theorem,  $cl_{M'}([q_i])$  is a subset of an equivalence class of  $\equiv_{M_L}$ .

Assume that the number of states of  $M'$  is greater than the number of equivalence classes of  $\equiv_L$ . Then there are two states  $[q_i]$  and  $[q_j]$  of  $M'$  such that  $cl_{M'}([q_i])$  and  $cl_{M'}([q_j])$  are both subsets of the same equivalence class of  $\equiv_L$ . This implies that there are strings  $u$  and  $v$  such that  $\hat{\delta}(q_0, u) = q_i$ ,  $\hat{\delta}(q_0, v) = q_j$ , and  $u \equiv_L v$ .

Since  $[q_i]$  and  $[q_j]$  are distinct states in  $M'$ , there is a string  $w$  that distinguishes these states. That is, either  $\hat{\delta}(q_i, w)$  is accepting and  $\hat{\delta}(q_j, w)$  is nonaccepting or vice versa. It follows that  $uw$  and  $vw$  have different membership values in  $L$ . This is a contradiction since  $u \equiv_L v$  implies that  $uw$  and  $vw$  have the same membership value in  $L$  for all strings  $w$ . Consequently, the assumption that the number of states of  $M'$  is greater than the number of equivalence classes of  $\equiv_L$  must be false. ■

The characterization of regularity in the Myhill-Nerode Theorem gives another method for establishing the nonregularity of a language. A language  $L$  is not regular if the equivalence relation  $\equiv_L$  has infinitely many equivalence classes.

de that  $M_L$

DFA that  
identical to

state DFA  
d from the  
t provided  
shows that  
uced by the

minimization

ting  $L$  if the  
Following  
strings with

ively. By the

lence classes  
 $\mathcal{E}_{M'}([q_j])$  are  
strings  $u$  and

guishes these  
vice versa. It  
contradiction  
for all strings  
an the number

another method  
if the equiva-

### Example 6.7.5

In Example 6.7.2, it was shown that the language  $\{a^i b^i \mid i \geq 0\}$  has infinitely many  $\equiv_L$  equivalence classes and therefore is not regular.  $\square$

### Example 6.7.6

The Myhill-Nerode Theorem will be used to show that the language  $L = \{a^{2^i} \mid i \geq 0\}$  is not regular. To accomplish this, we show that  $a^{2^i}$  and  $a^{2^j}$  are distinguishable by the  $\equiv_L$  equivalence relation whenever  $i < j$ . Concatenating  $a^{2^i}$  with each of these strings produces  $a^{2^i} a^{2^j} = a^{2^{i+1}} \in L$  and  $a^{2^j} a^{2^i} \notin L$ . The latter string is not in  $L$  since it has length greater than  $2^j$  but less than  $2^{j+1}$ . Thus,  $a^{2^i} \not\equiv_L a^{2^j}$ . These strings produce an infinite sequence  $[a^0] \equiv_L [a^1] \equiv_L [a^2] \equiv_L [a^4] \equiv_L \dots$  of distinct equivalence classes of  $L$ .  $\square$

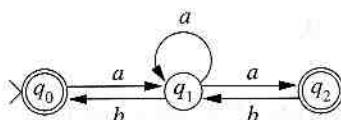
## Exercises

1. Use the technique from Section 6.2 to build the state diagram of an NFA- $\lambda$  that accepts the language  $(ab)^*ba$ . Compare this with the DFA constructed in Exercise 5.22(a).
2. For each of the state diagrams in Exercise 5.40, use Algorithm 6.2.2 to construct a regular expression for the language accepted by the automaton.
3. The language of the DFA  $M$  in Example 5.3.4 consists of all strings over  $\{a, b\}$  with an even number of  $a$ 's and an odd number of  $b$ 's. Use Algorithm 6.2.2 to construct a regular expression for  $L(M)$ . Exercise 2.38 requested a nonalgorithmic construction of a regular expression for this language, which, as you now see, is a formidable task.
4. Let  $G$  be the grammar

$$\begin{aligned} G: S &\rightarrow aS \mid bA \mid a \\ A &\rightarrow aS \mid bA \mid b. \end{aligned}$$

- a) Use Theorem 6.3.1 to build an NFA  $M$  that accepts  $L(G)$ .
- b) Using the result of part (a), build a DFA  $M'$  that accepts  $L(G)$ .
- c) Construct a regular grammar from  $M$  that generates  $L(M)$ .
- d) Construct a regular grammar from  $M'$  that generates  $L(M')$ .
- e) Give a regular expression for  $L(G)$ .

5. Let  $M$  be the NFA



- a) Construct a regular grammar from M that generates L(M).  
 b) Give a regular expression for L(M).
- \*6. Let G be a regular grammar and M the NFA obtained from G according to Theorem 6.3.1. Prove that if  $S \xrightarrow{*} wC$ , then there is a computation  $[S, w] \xrightarrow{*} [C, \lambda]$  in M.
7. Let L be a regular language over  $\{a, b, c\}$ . Show that each of the following sets is regular.
- $\{w \mid w \in L \text{ and } w \text{ ends with } aa\}$
  - $\{w \mid w \in L \text{ or } w \text{ contains an } a\}$
  - $\{w \mid w \notin L \text{ and } w \text{ does not contain an } a\}$
  - $\{uv \mid u \in L \text{ and } v \notin L\}$
8. Prove that the family of regular languages is closed under the operation of set difference.
9. Prove that the family of regular languages is not closed under intersection with context-free languages. That is, if L is regular and  $L_1$  context-free,  $L \cap L_1$  need not be regular.
10. Is the family of regular languages closed under infinite unions? That is, if  $L_0, L_1, L_2, \dots$  are regular, is  $\bigcup_{i=0}^{\infty} L_i$  necessarily regular? If so, prove it. If not, give a counterexample.
- \*11. Let L be a regular language. Show that the following languages are regular.
- The set  $P = \{u \mid uv \in L\}$  of prefixes of strings in L.
  - The set  $L^R = \{w^R \mid w \in L\}$  of reversals of strings in L.
  - The set  $E = \{uv \mid v \in L\}$  of strings that have a suffix in L.
  - The set  $SUB = \{v \mid uvw \in L\}$  of strings that are substrings of a string in L.
- \*12. Let L be a regular language containing only strings of even length. Let  $L'$  be the language  $\{u \mid uv \in L \text{ and } \text{length}(u) = \text{length}(v)\}$ .  $L'$  is the set of all strings that contain the first half of strings from L. Prove that  $L'$  is regular.
13. Use Corollary 6.5.2 to show that each of the following sets is not regular.
- The set of strings over  $\{a, b\}$  with the same number of a's and b's.
  - The set of palindromes of even length over  $\{a, b\}$ .
  - The set of strings over  $\{(), ()\}$  in which the parentheses are paired, for example,  $\lambda, (), ()(), ((())()$ .
  - The language  $\{a^i(ab)^j(ca)^{2i} \mid i, j > 0\}$ .
14. Use the pumping lemma to show that each of the following sets is not regular.
- The set of palindromes over  $\{a, b\}$
  - $\{a^n b^m \mid n < m\}$
  - $\{a^i b^j c^{2j} \mid i \geq 0, j \geq 0\}$
  - $\{ww \mid w \in \{a, b\}^*\}$
  - The set of initial sequences of the infinite string

*abaabaaaabaaaab . . . ba<sup>n</sup>ba<sup>n+1</sup>b . . .*

- f) The set of strings over  $\{a, b\}$  in which the number of  $a$ 's is a perfect cube.
- 15. Prove that the set of nonpalindromes over  $\{a, b\}$  is not a regular language.
- 16. Let  $L$  be a regular language and let  $L_1 = \{uu \mid u \in L\}$  be the language  $L$  “doubled.” Is  $L_1$  necessarily regular? Prove your answer.
- 17. Let  $L_1$  be a nonregular language and  $L_2$  an arbitrary finite language.
  - a) Prove that  $L_1 \cup L_2$  is nonregular.
  - b) Prove that  $L_1 - L_2$  is nonregular.
  - c) Show that the conclusions of parts (a) and (b) are not true if  $L_2$  is not assumed to be finite.
- 18. Give examples of languages  $L_1$  and  $L_2$  over  $\{a, b\}$  that satisfy the following descriptions.
  - a)  $L_1$  is regular,  $L_2$  is nonregular, and  $L_1 \cup L_2$  is regular.
  - b)  $L_1$  is regular,  $L_2$  is nonregular, and  $L_1 \cup L_2$  is nonregular.
  - c)  $L_1$  is regular,  $L_2$  is nonregular, and  $L_1 \cap L_2$  is regular.
  - d)  $L_1$  is nonregular,  $L_2$  is nonregular, and  $L_1 \cup L_2$  is regular.
  - e)  $L_1$  is nonregular and  $L_1^*$  is regular.
- \*19. Let  $\Sigma_1$  and  $\Sigma_2$  be two alphabets. A **string homomorphism** is a total function  $h$  from  $\Sigma_1^*$  to  $\Sigma_2^*$  that preserves concatenation. That is,  $h$  satisfies
  - i)  $h(\lambda) = \lambda$
  - ii)  $h(uv) = h(u)h(v)$ .
  - a) Let  $L_1 \subseteq \Sigma_1^*$  be a regular language. Show that the set  $\{h(w) \mid w \in L_1\}$  is regular over  $\Sigma_2$ . This set is called the *homomorphic image* of  $L_1$  under  $h$ .
  - b) Let  $L_2 \subseteq \Sigma_2^*$  be a regular language. Show that the set  $\{w \in \Sigma_1^* \mid h(w) \in L_2\}$  is regular. This set is called the *inverse image* of  $L_2$  under  $h$ .
- 20. A context-free grammar  $G = (V, \Sigma, P, S)$  is called **right-linear** if each rule is of the form
  - i)  $A \rightarrow u$ , or
  - ii)  $A \rightarrow uB$ ,
 where  $A, B \in V$ , and  $u \in \Sigma^*$ . Use the techniques from Section 6.3 to show that the right-linear grammars generate precisely the regular sets.
- \*21. A context-free grammar  $G = (V, \Sigma, P, S)$  is called **left-regular** if each rule is of the form
  - i)  $A \rightarrow \lambda$ ,
  - ii)  $A \rightarrow a$ , or
  - iii)  $A \rightarrow Ba$ ,
 where  $A, B \in V$ , and  $a \in \Sigma$ .

- a) Design an algorithm to construct an NFA that accepts the language of a left-regular grammar.
- b) Show that the left-regular grammars generate precisely the regular sets.
22. A context-free grammar  $G = (V, \Sigma, P, S)$  is called **left-linear** if each rule is of the form
- $A \rightarrow u$ , or
  - $A \rightarrow Bu$ ,
- where  $A, B \in V$ , and  $u \in \Sigma^*$ . Show that the left-linear grammars generate precisely the regular sets.
23. Give a regular language  $L$  such that  $\equiv_L$  has exactly three equivalence classes.
24. Give the  $\equiv_L$  equivalence classes of the language  $a^+b^+$ .
25. Let  $[u]_{\equiv_L}$  be a  $\equiv_L$  equivalence class of a language  $L$ . Show that if  $[u]_{\equiv_L}$  contains one string  $v \in L$ , then every string in  $[u]_{\equiv_L}$  is in  $L$ .
26. Prove that  $\equiv_L$  is right-invariant for any regular language  $L$ . That is, if  $u \equiv_L v$ , then  $ux \equiv_L vx$  for any  $x \in \Sigma^*$ , where  $\Sigma$  is the alphabet of the language  $L$ .
27. Use the Myhill-Nerode Theorem to prove that the language  $\{a^i \mid i \text{ is a perfect square}\}$  is not regular.
28. Let  $u \in [ab]_{\equiv_M}$  and  $v \in [aba]_{\equiv_M}$  be strings from the equivalence classes of  $(a \cup b)(a \cup b^*)$  defined in Example 6.7.4. Show that  $u$  and  $v$  are distinguishable.
29. Give the equivalence classes defined by the relation  $\equiv_M$  for the DFA in Example 5.3.1.
30. Give the equivalence classes defined by the relation  $\equiv_M$  for the DFA in Example 5.3.3.
- \* 31. Let  $M_L$  be the minimal state DFA that accepts a language  $L$  defined in Theorems 6.7.4 and 6.7.5. Let  $M$  be another DFA that accepts  $L$  with the same number of states as  $M_L$ . Prove that  $M_L$  and  $M$  are identical except (possibly) for the names assigned to the states. Two such DFAs are said to be *isomorphic*.

---

### Bibliographic Notes

The equivalence of regular sets and languages accepted by finite automata was established by Kleene [1956]. The proof given in Section 6.2 is modeled after that of McNaughton and Yamada [1960]. Chomsky and Miller [1958] established the equivalence of the languages generated by regular grammars and accepted by finite automata. Closure under homomorphisms (Exercise 19) is from Ginsburg and Rose [1963b]. The closure of regular sets under reversal was noted by Rabin and Scott [1959]. Additional closure results for regular sets can be found in Bar-Hillel, Perles, and Shamir [1961], Ginsburg and Rose [1963b], and Ginsburg [1966]. The pumping lemma for regular languages is from Bar-Hillel, Perles, and Shamir [1961]. The relationship between the number of equivalence classes of a language and regularity was established in Myhill [1957] and Nerode [1958].

Regu  
and a  
auto  
mach  
the p  
comb  
accep  
A  
existe  
provid

7.1

Theore  
autom  
any fin  
autom  
of auto  
with th  
A  
machin  
of the s

of a left-regular

sets.

ch rule is of the

erate precisely  
lasses.

$\equiv_L$  contains one  
if  $u \equiv_L v$ , then  
perfect square }

ice classes of  
guishable.

Example 5.3.1.

Example 5.3.3.

Theorems 6.7.4

per of states as  
assigned to the

was established  
Naughton and  
the languages  
der homomor-  
ular sets under  
or regular sets  
[1963b], and  
lel, Perles, and  
of a language

## CHAPTER 7

# Pushdown Automata and Context-Free Languages

Regular languages have been characterized as the languages generated by regular grammars and accepted by finite automata. This chapter presents a class of machines, the pushdown automata, that accepts the context-free languages. A pushdown automaton is a finite-state machine augmented with an external stack memory. The addition of a stack provides the pushdown automaton with a last-in, first-out memory management capability. The combination of stack and states overcomes the memory limitations that prevented the acceptance of the language  $\{a^i b^i \mid i \geq 0\}$  by a deterministic finite automaton.

As with regular languages, a pumping lemma for context-free languages ensures the existence of repeatable substrings in strings of a context-free language. The pumping lemma provides a technique for showing that many easily definable languages are not context-free.

### 7.1 Pushdown Automata

Theorem 6.5.1 established that the language  $\{a^i b^i \mid i \geq 0\}$  is not accepted by any finite automaton. To accept this language, a machine needs the ability to record the processing of any finite number of  $a$ 's. The restriction of having finitely many states does not allow the automaton to "remember" the number of leading  $a$ 's in an arbitrary input string. A new type of automaton is constructed that augments the state-input transitions of a finite automaton with the ability to utilize unlimited memory.

A pushdown stack, or simply a stack, is added to a finite automaton to construct a new machine known as a pushdown automaton (PDA). Stack operations affect only the top item of the stack; a pop removes the top element from the stack and a push places an element

on the stack top. Definition 7.1.1 formalizes the concept of a pushdown automaton. The components  $Q$ ,  $\Sigma$ ,  $q_0$ , and  $F$  of a PDA are the same as in a finite automaton.

### Definition 7.1.1

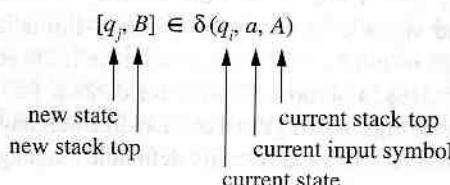
A **pushdown automaton** is a sextuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$ , where  $Q$  is a finite set of states,  $\Sigma$  a finite set called the *input alphabet*,  $\Gamma$  a finite set called the *stack alphabet*,  $q_0$  the start state,  $F \subseteq Q$  a set of final states, and  $\delta$  a transition function from  $Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\})$  to subsets of  $Q \times (\Gamma \cup \{\lambda\})$ .

A PDA has two alphabets: an input alphabet  $\Sigma$  from which the input strings are built and a stack alphabet  $\Gamma$  whose elements are stored on the stack. The stack is represented as a string of stack elements; the element on the top of the stack is the leftmost symbol in the string. We will use capital letters to represent stack elements and Greek letters to represent strings of stack elements. The notation  $A\alpha$  represents a stack with  $A$  as the top element. An empty stack is denoted  $\lambda$ . The computation of a PDA begins with the machine in state  $q_0$ , the input on the tape, and the stack empty.

A PDA consults the current state, input symbol, and the symbol on the top of the stack to determine the machine transition. The transition function  $\delta$  lists all possible transitions for a given state, symbol, and stack top combination. The value of the transition function

$$\delta(q_i, a, A) = \{[q_j, B], [q_k, C]\}$$

indicates that two transitions are possible when the automaton is in state  $q_i$  scanning an  $a$  with  $A$  on the top of the stack. The transition



causes the machine to

- i) change the state from  $q_i$  to  $q_j$ ,
- ii) process the symbol  $a$  (advance the tape head),
- iii) remove  $A$  from the top of the stack (pop the stack), and
- iv) push  $B$  onto the stack.

Since multiple transitions may be specified for a machine configuration, PDAs are non-deterministic machines.

A pushdown automaton can also be depicted by a state diagram. The labels on the arcs indicate both the input and the stack operation. The transition  $\delta(q_i, a, A) = \{[q_j, B]\}$  is depicted by

The symbol / indicates that  $B$ .

The domain of  $\delta$  specifies that  $\lambda$  may occur in the input, stack, or state. The range of  $\delta$  specifies that the stack top after the transition; the applicable transitions do not contain  $\lambda$ .

When  $\lambda$  occurs in the input, the transition is applied regardless of the current state. The stack may be empty if the machine is in state  $q_i$  to enter  $q_j$  and add  $B$  to the stack.

The symbol  $\lambda$  in the range of  $\delta(q_i, a, A)$ . The example will now look at state  $q_i$ .

If the input  $a$  is  $\lambda$ , the transition (i) pops a stack symbol.

i)  $[q_i, \lambda] \in \delta(q_i, a, A)$

ii)  $[q_i, A] \in \delta(q_i, a, A)$

iii)  $[q_j, \lambda] \in \delta(q_i, a, A)$

If the action specified in the transition is pushed onto the stack. The applicability is determined by consulting the stack top and the input symbol. If the action does not affect the stack top, it is applicable.

maton. The

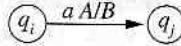
set of states,  
 $q_0$  the start  
 $\times (\Gamma \cup \{\lambda\})$

ngs are built  
presented as  
ymbol in the  
to represent  
lement. An  
e in state  $q_0$ ,

o of the stack  
le transitions  
on function

canning an  $a$

As are nonde-  
erministic  
els on the arcs  
 $= \{[q_j, B]\}$  is



The symbol / indicates replacement:  $A/B$  indicates that  $A$  is replaced on the top of the stack by  $B$ .

The domain of the transition function is  $Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\})$ , which indicates that  $\lambda$  may occur in either the input or stack top positions of a transition. A  $\lambda$  argument specifies that the value of the component should be neither consulted nor acted upon by the transition; the applicability of the transition is completely determined by the positions that do not contain  $\lambda$ .

When  $\lambda$  occurs as an argument in the stack position of the transition function, the transition is applicable whenever the current state and input symbol match those in the transition regardless of the status of the stack. The stack top may contain any symbol or the stack may be empty. The transition  $[q_j, B] \in \delta(q_i, a, \lambda)$  is applicable whenever a machine is in state  $q_i$  scanning an  $a$ ; the application of the transition will cause the machine to enter  $q_j$  and add  $B$  to the top of the stack.

The symbol  $\lambda$  may also occur in the new stack position of a transition,  $[q_j, \lambda] \in \delta(q_i, a, A)$ . The execution such a transition does not push a symbol onto the stack. We will now look at several examples of the effect of  $\lambda$  in PDA transitions.

If the input position is  $\lambda$ , the transition does not process an input symbol. Thus, transition (i) pops and (ii) pushes the stack symbol  $A$  without altering the state or the input.

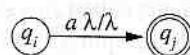
i)  $[q_i, \lambda] \in \delta(q_i, \lambda, A)$



ii)  $[q_i, A] \in \delta(q_i, \lambda, \lambda)$



iii)  $[q_j, \lambda] \in \delta(q_i, a, \lambda)$



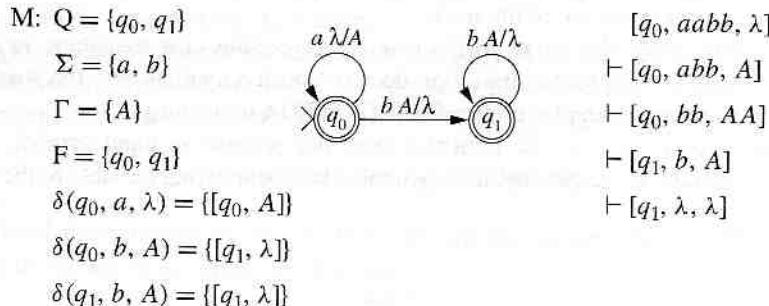
If the action specified by a transition has  $\lambda$  in the new stack top position,  $[q_j, \lambda]$ , no symbol is pushed onto the stack. Transition (iii) is the PDA equivalent of a finite automaton transition. The applicability is determined only by the state and input symbol; the transition does not consult nor does it alter the stack.

A PDA configuration is represented by the triple  $[q_i, w, \alpha]$ , where  $q_i$  is the machine state,  $w$  the unprocessed input, and  $\alpha$  the stack. The notation

$$[q_i, w, \alpha] \xrightarrow{M} [q_j, v, \beta]$$

indicates that configuration  $[q_j, v, \beta]$  can be obtained from  $[q_i, w, \alpha]$  by a single transition of the PDA  $M$ . As before,  $\xrightarrow{M}^*$  represents the result of a sequence of transitions. When there is no possibility of confusion, the subscript  $M$  is omitted. A computation of a PDA is a sequence of transitions beginning with the machine in the initial state with an empty stack.

We are now ready to construct a PDA  $M$  to accept the language  $\{a^i b^i \mid i \geq 0\}$ . The computation begins with the input string  $w$  and an empty stack. Processing input symbol  $a$  causes  $A$  to be pushed onto the stack. Processing  $b$  pops the stack, matching the number of  $b$ 's to the number of  $a$ 's. The computation generated by the input string  $aabb$  illustrates the actions of  $M$ .



The computation of  $M$  with input  $a^i b^i$  processes the entire string and halts in an accepting state with an empty stack. These conditions become our criteria for acceptance.

### Definition 7.1.2

Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a PDA. A string  $w \in \Sigma^*$  is accepted by  $M$  if there is a computation

$$[q_0, w, \lambda] \xrightarrow{*} [q_i, \lambda, \lambda]$$

where  $q_i \in F$ . The language of  $M$ , denoted  $L(M)$ , is the set of strings accepted by  $M$ .

A computation that accepts a string is called *successful*. A computation that processes the entire input string and halts in a nonaccepting configuration is said to be *unsuccessful*. Because of the nondeterministic nature of the transition function, there may be computations that cannot complete the processing of the input string. Computations of this form are also considered unsuccessful.

Acceptance by a PDA follows the standard pattern for nondeterministic machines; one computation that processes the entire string and halts in a final state is sufficient for the

string to  
not affect

### Example

The PDA  
string  $w$  a

$M:$

A suc  
is encoun  
 $w^R$ . The co  
the stack. T

A PDA  
combination  
and  $[q_k, D]$   
satisfied:

- i)  $u = v$  and
- ii)  $u = v$  and
- iii)  $A = B$  and
- iv)  $u = \lambda$  or

Compatible t  
ministic if it d  
and the mach

the machine

single transition  
s. When there  
of a PDA is  
with an empty

$a^i b^i \mid i \geq 0\}$ . The  
input symbol  
ng the number  
abb illustrates

,  $\lambda]$

$A]$

$[A]$

and halts in an  
for acceptance.

y M if there is a

epted by M.

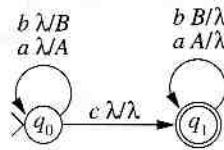
on that processes  
be unsuccessful.  
/ be computations  
this form are also  
tic machines; one  
sufficient for the

string to be in the language. The existence of additional unsuccessful computations does not affect the acceptance of the string.

### Example 7.1.1

The PDA M accepts the language  $\{wcw^R \mid w \in \{a, b\}^*\}$ . The stack is used to record the string  $w$  as it is processed. Stack symbols  $A$  and  $B$  represent input  $a$  and  $b$ , respectively.

$$\begin{array}{ll} M: Q = \{q_0, q_1\} & \delta(q_0, a, \lambda) = \{[q_0, A]\} \\ \Sigma = \{a, b, c\} & \delta(q_0, b, \lambda) = \{[q_0, B]\} \\ \Gamma = \{A, B\} & \delta(q_0, c, \lambda) = \{[q_1, \lambda]\} \\ F = \{q_1\} & \delta(q_1, a, A) = \{[q_1, \lambda]\} \\ & \delta(q_1, b, B) = \{[q_1, \lambda]\} \end{array}$$



A successful computation records the string  $w$  on the stack as it is processed. Once the  $c$  is encountered, the accepting state  $q_1$  is entered and the stack contains a string representing  $w^R$ . The computation is completed by matching the remaining input with the elements on the stack. The computation of M with input  $abcba$  is

$$\begin{aligned} & [q_0, abcba, \lambda] \\ \leftarrow & [q_0, bcba, A] \\ \leftarrow & [q_0, cba, BA] \\ \leftarrow & [q_1, ba, BA] \\ \leftarrow & [q_1, a, A] \\ \leftarrow & [q_1, \lambda, \lambda] \end{aligned}$$

□

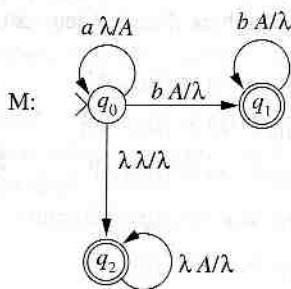
A PDA is *deterministic* if there is at most one transition that is applicable for each combination of state, input symbol, and stack top. Two transitions  $[q_j, C] \in \delta(q_i, u, A)$  and  $[q_k, D] \in \delta(q_i, v, B)$  are called *compatible* if any of the following conditions are satisfied:

- i)  $u = v$  and  $A = B$ .
- ii)  $u = v$  and  $A = \lambda$  or  $B = \lambda$ .
- iii)  $A = B$  and  $u = \lambda$  or  $v = \lambda$ .
- iv)  $u = \lambda$  or  $v = \lambda$  and  $A = \lambda$  or  $B = \lambda$ .

Compatible transitions can be applied to the same machine configurations. A PDA is deterministic if it does not contain distinct compatible transitions. Both the PDA in Example 7.1.1 and the machine constructed to accept  $\{a^i b^i \mid i \geq 0\}$  are deterministic.

**Example 7.1.2**

The language  $L = \{a^i \mid i \geq 0\} \cup \{a^i b^i \mid i \geq 0\}$  contains strings consisting solely of  $a$ 's or an equal number of  $a$ 's and  $b$ 's. The stack of the PDA  $M$  that accepts  $L$  maintains a record of the number of  $a$ 's processed until a  $b$  is encountered or the input string is completely processed.

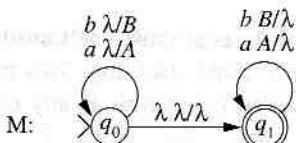


When scanning an  $a$  in state  $q_0$ , there are two transitions that are applicable. A string of the form  $a^i b^i$ ,  $i > 0$ , is accepted by a computation that remains in states  $q_0$  and  $q_1$ . If a transition to state  $q_2$  follows the processing of the final  $a$  in a string  $a^i$ , the stack is emptied and the input is accepted. Reaching  $q_2$  in any other manner results in an unsuccessful computation, since no input is processed after  $q_2$  is entered.

The  $\lambda$ -transition allows  $M$  to enter  $q_2$  any time it is in  $q_0$ . This transition introduces nondeterminism into the computations of  $M$ . The accepting computation of a string  $a^i$  processes the entire string in  $q_0$ , transitions to  $q_2$ , empties the stack, and accepts.  $\square$

**Example 7.1.3**

The even-length palindromes over  $\{a, b\}$  are accepted by the PDA



That is,  $L(M) = \{ww^R \mid w \in \{a, b\}^*\}$ . A successful computation remains in state  $q_0$  while processing the string  $w$  and enters state  $q_1$  upon reading the first symbol in  $w^R$ . Unlike the strings in Example 7.1.1, the strings in  $L$  do not contain a middle marker that induces the change from state  $q_0$  to  $q_1$ . Nondeterminism allows the machine to guess when the middle of the string has been reached. Transitions to  $q_1$  that do not occur immediately after processing the last element of  $w$  result in unsuccessful computations.  $\square$

In Chapter 5 we showed that deterministic and nondeterministic finite automata accepted the same family of languages. Nondeterminism was a useful design feature but did

not increase the power of the automata.

There is no simple way to prove that Example 7.1.3 is correct. Intuitively, the machine is nondeterministic because it can move from  $q_0$  to  $q_1$  reading any symbol of the input. The resulting computation depends on the stack elements pushed at each step.

Consider the computation

When an  $a$  or  $b$  is read, the stack symbol  $B$  must be pushed. After reading the first symbol, the stack symbol is being processed. To begin the machine must push a  $B$  onto the stack configuration and

The languages accepted by PDAs and are called context-free languages and are important in computer science, which is important in the study of languages that can be generated by a grammar. The definition and derivation of context-free languages are based on the concept of a pushdown automaton (PDA).

## 7.2 Variations

Pushdown automata (PDAs) are a type of nondeterministic finite automaton (NFA) that have a stack to store information. In this section we will learn how to construct PDAs for accepted languages.

Along with choosing the stack symbol to be pushed or popped, the stack symbol can be manipulated. This manipulation is called atomic if each stack operation involves exactly one symbol. An atomic PDA has a stack that is manipulated in atomic steps.

- $[q_j, \lambda] \in \delta(q_i)$
- $[q_j, \lambda] \in \delta(q_i)$
- $[q_j, A] \in \delta(q_i)$

not increase the ability of the machine to accept languages. This is not the case for pushdown automata.

There is no deterministic PDA that accepts the language  $L = \{ww^R \mid w \in \{a, b\}^*\}$  from Example 7.1.3. This can be seen intuitively by considering the properties needed by a PDA to accept  $L$ . Since the computation of a PDA processes the input in a left-to-right manner, the machine is not able to determine when the first half of the input string has been read. For the nondeterministic machine  $M$  in Example 7.1.3, this poses no problem. The transition from  $q_0$  to  $q_1$  represents a nondeterministic guess that the symbol being scanned is the first symbol of the second copy of  $w$ . For a string in  $L$ , one of the guesses will be correct and the resulting computation accepts the input by matching the second half of the string with the stack elements.

Consider the possible actions of a deterministic PDA processing the input strings

$aabbbaa$  and  $aabbbbbaa$ .

When an  $a$  or  $b$  is read in the first half of a string, the corresponding stack symbol  $A$  or  $B$  must be pushed onto the stack to be compared with the second half of the input. After reading the first three symbols, the stack is  $BAA$ . Regardless of which of the two strings is being processed, the next symbol is a  $b$ . To accept  $aabbbaa$ , it is necessary to pop the stack to begin the matching of  $aab$  with  $baa$ . However, to accept the  $aabbbbbaa$  the machine must push a  $B$  onto the stack. A deterministic machine can have only one option for this configuration and consequently one of these two strings will not be accepted.

The languages accepted by deterministic pushdown automata include all regular languages and are a proper subset of the context-free languages. This family of languages, which is important for programming language definition and parsing, consists of the languages that can be generated by  $LR(k)$  grammars. The use of  $LR(k)$  grammars for language definition and deterministic parsing will be examined in Chapter 19.

## 7.2 Variations on the PDA Theme

Pushdown automata are often defined in a manner that differs slightly from Definition 7.1.1. In this section we examine several alterations to our definition that preserve the set of accepted languages.

Along with changing the state, a transition in a PDA is accompanied by three actions: popping the stack, pushing a stack element, and processing an input symbol. A PDA is called **atomic** if each transition causes only one of the three actions to occur. Transitions in an atomic PDA have the form

- i)  $[q_j, \lambda] \in \delta(q_i, a, \lambda)$ ,
- ii)  $[q_j, \lambda] \in \delta(q_i, \lambda, A)$ , or
- iii)  $[q_j, A] \in \delta(q_i, \lambda, \lambda)$ .

Clearly, every atomic PDA is a PDA in the sense of Definition 7.1.1. Theorem 7.2.1 shows that the languages accepted by atomic PDAs are the same as those accepted by PDAs. Moreover, it outlines a method to construct an equivalent atomic PDA from an arbitrary PDA.

### Theorem 7.2.1

Let  $M$  be a PDA. Then there is an atomic PDA  $M'$  with  $L(M') = L(M)$ .

**Proof.** To construct  $M'$ , the nonatomic transitions of  $M$  are replaced by a sequence of atomic transitions. Let  $[q_j, B] \in \delta(q_i, a, A)$  be a transition of  $M$ . The atomic equivalent requires two new states,  $p_1$  and  $p_2$ , and the transitions

$$\begin{aligned}[p_1, \lambda] &\in \delta(q_i, a, \lambda) \\ \delta(p_1, \lambda, A) &= \{[p_2, \lambda]\} \\ \delta(p_2, \lambda, \lambda) &= \{[q_j, B]\}\end{aligned}$$

to accomplish the same result as the nonatomic single transition.

In a similar manner, a transition that consists of changing the state and performing two additional actions can be replaced with a sequence of two atomic transitions. Replacing all nonatomic transitions with a sequence of atomic transitions produces an equivalent atomic PDA. ■

An extended transition is an operation on a PDA that pushes a string of elements, rather than just a single element, onto the stack. The transition  $[q_j, BCD] \in \delta(q_i, a, A)$  pushes  $BCD$  onto the stack with  $B$  becoming the new stack top. A PDA containing extended transitions is called an **extended PDA**. The apparent generalization does not increase the set of languages accepted by pushdown automata. Each extended PDA can be converted into an equivalent PDA in the sense of Definition 7.1.1.

To construct a PDA from an extended PDA, extended transitions are transformed into a sequence of transitions each of which pushes a single stack element. To achieve the result of an extended transition that pushes  $k$  elements requires  $k - 1$  additional states. The sequence of transitions

$$\begin{aligned}[p_1, D] &\in \delta(q_i, a, A) \\ \delta(p_1, \lambda, \lambda) &= \{[p_2, C]\} \\ \delta(p_2, \lambda, \lambda) &= \{[q_j, B]\}\end{aligned}$$

pushes the string  $BCD$  onto the stack and leaves the machine in state  $q_j$ . The sequential execution of these three transitions produces the same result as the single extended transition  $[q_j, BCD] \in \delta(q_i, a, A)$ . The preceding argument can be generalized to yield Theorem 7.2.2.

### Theorem 7.2.2

Let  $M$  be an extended PDA. Then there is a PDA  $M'$  such that  $L(M') = L(M)$ .

### Example 7.2.1

Let  $L = \{a^i b^{2i} \mid i \geq 0\}$  be the language of Example 7.1.1. The states of  $M$  are  $q_0, q_1, q_2$ . The transitions of  $M$  are

| PDA                                                  |
|------------------------------------------------------|
| $Q = \{q_0, q_1, q_2\}$                              |
| $\delta(q_0, a, \lambda) = \{[q_1, \lambda]\}$       |
| $\delta(q_2, \lambda, \lambda) = \{[q_0, \lambda]\}$ |
| $\delta(q_0, b, \lambda) = \{[q_2, \lambda]\}$       |
| $\delta(q_1, b, A) = \{[q_0, B]\}$                   |

As might be expected, the PDA has fewer transitions than the standard PDA. It also has fewer states, and the number of states required to accept the language is reduced from three to two. ■

By Definition 7.2.1, the entire string  $a^i b^{2i}$  is accepted by the PDA. The acceptance is referred to as the final state of the PDA in terms of the final state of the standard PDA that recognizes the language. ■

A string  $w$  is accepted by a PDA if there is a sequence of transitions starting in state  $q_i$  and terminating in a final state  $q_f$  with acceptance by final state  $q_f$ . ■

### Lemma 7.2.3

Let  $L$  be a language accepted by a PDA  $M$ . Let  $q_f$  be a final state of  $M$ . Then

**Proof.** A PDA  $M'$  is constructed with states  $Q' = Q \cup \{q_f\}$  and transitions  $\delta'$  such that  $q_f$  is a final state. The transitions of  $M'$  are identical to one of the transitions of  $M$  except that the final state is  $q_f$ . The transition function  $\delta'$  is defined as follows:

em 7.2.1  
by PDAs.  
arbitrary

quence of  
equivalent

rrning two  
placing all  
ent atomic  
■

ents, rather  
A) pushes  
g extended  
increase the  
e converted

formed into a  
the result of  
he sequence

The sequen-  
gle extended  
ized to yield

### Example 7.2.1

Let  $L = \{a^i b^{2i} \mid i \geq 1\}$ . A standard PDA, an atomic PDA, and an extended PDA are constructed to accept L. The input alphabet  $\{a, b\}$ , stack alphabet  $\{A\}$ , and accepting state  $q_1$  are the same for each automaton. The states and transitions are

| PDA                                            | Atomic PDA                                     | Extended PDA                              |
|------------------------------------------------|------------------------------------------------|-------------------------------------------|
| $Q = \{q_0, q_1, q_2\}$                        | $Q = \{q_0, q_1, q_2, q_3, q_4\}$              | $Q = \{q_0, q_1\}$                        |
| $\delta(q_0, a, \lambda) = \{[q_2, A]\}$       | $\delta(q_0, a, \lambda) = \{[q_3, \lambda]\}$ | $\delta(q_0, a, \lambda) = \{[q_0, AA]\}$ |
| $\delta(q_2, \lambda, \lambda) = \{[q_0, A]\}$ | $\delta(q_3, \lambda, \lambda) = \{[q_2, A]\}$ | $\delta(q_0, b, A) = \{[q_1, \lambda]\}$  |
| $\delta(q_0, b, A) = \{[q_1, \lambda]\}$       | $\delta(q_2, \lambda, \lambda) = \{[q_0, A]\}$ | $\delta(q_1, b, A) = \{[q_1, \lambda]\}$  |
| $\delta(q_1, b, A) = \{[q_1, \lambda]\}$       | $\delta(q_0, b, \lambda) = \{[q_4, \lambda]\}$ |                                           |
|                                                | $\delta(q_4, \lambda, A) = \{[q_1, \lambda]\}$ |                                           |
|                                                | $\delta(q_1, b, \lambda) = \{[q_4, \lambda]\}$ |                                           |

As might be expected, the atomic PDA requires more transitions and the extended PDA fewer transitions than the equivalent standard PDA. The stack symbol A is used to count the number of matching b's required to accept the string. The extended transition  $\delta(q_0, a, \lambda) = \{[q_0, AA]\}$  pushes both counters on the stack with a single transition. The standard PDA requires two transitions and the atomic PDA three to accomplish the same result.  $\square$

By Definition 7.1.2, an input string is accepted if there is a computation that processes the entire string and terminates in an accepting state with an empty stack. This type of acceptance is referred to as acceptance by *final state and empty stack*. Defining acceptance in terms of the final state or the configuration of the stack alone does not change the set of languages recognized by pushdown automaton.

A string  $w$  is accepted by *final state* if there is a computation  $[q_0, w, \lambda] \xrightarrow{*} [q_i, \lambda, \alpha]$ , where  $q_i$  is an accepting state and  $\alpha \in \Gamma^*$ , that is, a computation that processes the input and terminates in an accepting state. The contents of the stack at termination are irrelevant with acceptance by final state. A language accepted by final state is denoted  $L_F$ .

### Lemma 7.2.3

Let  $L$  be a language accepted by a PDA  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  with acceptance defined by final state. Then there is a PDA that accepts  $L$  by final state and empty stack.

**Proof.** A PDA  $M' = (Q \cup \{q_f\}, \Sigma, \Gamma, \delta', q_0, \{q_f\})$  is constructed from  $M$  by adding a state  $q_f$  and transitions for  $q_f$ . Intuitively, a computation in  $M'$  that accepts a string should be identical to one in  $M$  except for the addition of transitions that empty the stack. The transition function  $\delta'$  is constructed by augmenting  $\delta$  with the transitions

$$\delta'(q_i, \lambda, \lambda) = \{[q_f, \lambda]\} \quad \text{for all } q_i \in F$$

$$\delta'(q_f, \lambda, A) = \{[q_f, \lambda]\} \quad \text{for all } A \in \Gamma.$$

Let  $[q_0, w, \lambda] \xrightarrow{M} [q_i, \lambda, \alpha]$  be a computation of  $M$  accepting  $w$  by final state. In  $M'$ , this computation is completed by entering the accepting state  $q_f$  and emptying the stack

$$\begin{aligned}[q_0, w, \lambda] \\ \xrightarrow{M} & [q_i, \lambda, \alpha] \\ \xrightarrow{M'} & [q_f, \lambda, \alpha] \\ \xrightarrow{M'} & [q_f, \lambda, \lambda]\end{aligned}$$

showing that  $w$  is accepted in  $M'$ .

We must also guarantee that the new transitions do not cause  $M'$  to accept strings that are not in  $L(M)$ . The sole accepting state of  $M'$  is  $q_f$ , which can be entered only on a transition from any accepting state of  $M$ . Since the transitions for  $q_f$  do not process input, entering  $q_f$  with unprocessed input results in an unsuccessful computation. Consequently, a string  $w$  is accepted by  $M'$  only if there is computation in  $M$  that processes all of  $w$  and halts in an accepting state of  $M$ . That is,  $w \in L(M')$  only when  $w \in L(M)$  as desired. ■

A string  $w$  is said to be accepted by *empty stack* if there is a computation  $[q_0, w, \lambda] \xrightarrow{*} [q_i, \lambda, \lambda]$ . No restriction is placed on the halting state  $q_i$ . When acceptance is defined by empty stack, it is necessary to require at least one transition to permit the acceptance of languages that do not contain the null string. The language accepted by empty stack is denoted  $L_E(M)$ .

#### Lemma 7.2.4

Let  $L$  be a language accepted by a PDA  $M = (Q, \Sigma, \Gamma, \delta, q_0)$  with acceptance defined by empty stack. Then there is a PDA that accepts  $L$  by final state and empty stack.

**Proof.** Let  $M' = (Q \cup \{q'_0\}, \Sigma, \Gamma, \delta', q'_0, Q)$ , where  $\delta'(q_i, x, A) = \delta(q_i, x, A)$  and  $\delta'(q'_0, x, A) = \delta(q_0, x, A)$  for every  $q_i \in Q$ ,  $x \in \Sigma \cup \{\lambda\}$ , and  $A \in \Gamma \cup \{\lambda\}$ . Every state of the original machine  $M$  is an accepting state of  $M'$ .

The computations of  $M$  and  $M'$  are identical except that those of  $M$  begin in state  $q_0$  and  $M'$  in state  $q'_0$ . A computation of length one or more in  $M'$  that halts with an empty stack also halts in a final state. Since  $q'_0$  is not accepting, the null string is accepted by  $M'$  only if it is accepted by  $M$ . Thus,  $L(M') = L_E(M)$ . ■

Lemmas 7.2.3 and 7.2.4 show that a language accepted by either final state or empty stack alone is also accepted by final state and empty stack. Exercises 8 and 9 establish that any language accepted by final state and empty stack is accepted by a pushdown automaton using the less restrictive forms of acceptance. These observations yield the following theorem.

#### Theorem 7.2.5

The following three conditions are equivalent:

- i) The language  $L$  is accepted by some PDA.
- ii) There is a PDA  $M_1$  with  $L_F(M_1) = L$ .
- iii) There is a PDA  $M_2$  with  $L_E(M_2) = L$ .

We have co  
acceptance criter  
assume that ther  
marker can be re  
machine to recog  
the role of a bott

#### Example 7.2.2

The pushdown a

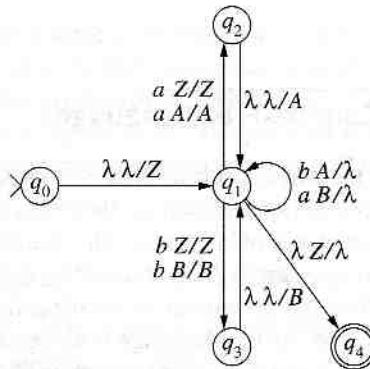
accepts strings tha  
of a bottom marke  
the computation.

The stack rec  
stack will contain  
number of  $B$ 's on  
have been process  
have been process

We have considered alternatives to the standard PDA model obtained by changing the acceptance criteria and the form of the transitions. Another common modification is to assume that there is a distinguished element that marks the bottom of the stack. A bottom marker can be read but not popped from the stack. Reading the bottom marker allows the machine to recognize an empty stack and act accordingly. The following example illustrates the role of a bottom marker and shows how it can be simulated in a standard PDA.

### Example 7.2.2

The pushdown automaton  $M$  defined by the transitions



accepts strings that have the same number of  $a$ 's and  $b$ 's. The stack symbol  $Z$  plays the role of a bottom marker; it is placed on the stack with the first transition and remains throughout the computation.

The stack records the difference in the number of  $a$ 's and  $b$ 's that have been read. The stack will contain  $n$   $A$ 's if the automaton has processed  $n$  more  $a$ 's than  $b$ 's. Similarly, the number of  $B$ 's on the stack indicates the number of  $b$ 's in excess of the number of  $a$ 's that have been processed. The bottom marker  $Z$  is read when the same number of  $a$ 's and  $b$ 's have been processed. The computation

$$\begin{aligned}
 & [q_0, abba, \lambda] \\
 \leftarrow & [q_1, abba, Z] \\
 \leftarrow & [q_2, bba, Z] \\
 \leftarrow & [q_1, bba, AZ] \\
 \leftarrow & [q_1, ba, Z] \\
 \leftarrow & [q_3, a, Z] \\
 \leftarrow & [q_1, a, BZ] \\
 \leftarrow & [q_1, \lambda, Z] \\
 \leftarrow & [q_4, \lambda, \lambda]
 \end{aligned}$$

exhibits the acceptance of *abba*. When an *a* is read with an *A* or *Z* on the top of the stack, an *A* is added to the stack by the transitions to  $q_2$  and back to  $q_1$ . If the stack top is a *B*, the stack is popped in  $q_1$  since reading the *a* decreases the difference between the number of *b*'s and *a*'s that have been processed. A similar strategy is employed when a *b* is read.

The lone accepting state of the automaton is  $q_4$ . If the input string has the same number of *a*'s and *b*'s, the transition to  $q_4$  pops the *Z* and terminates the computation.  $\square$

The variations of pushdown automata that accept the same family of languages illustrate the robustness of acceptance using a stack memory. In the next section we show that the languages accepted by pushdown automata are precisely those generated by context-free grammars.

### 7.3 Acceptance of Context-Free Languages

In Chapter 6 we showed that the languages generated by regular grammars were precisely those accepted by DFAs. In this section we continue the relationship between grammatical generation and mechanical acceptance of languages. The characterization of pushdown automata as acceptors of context-free languages is obtained by establishing a correspondence between computations of a PDA and derivations in a context-free grammar.

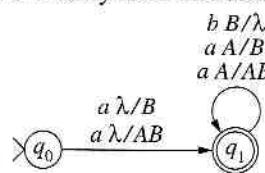
First we prove that every context-free language is accepted by an extended PDA. To accomplish this, the rules of the grammar are used to generate the transitions of an equivalent PDA. Let  $L$  be a context-free language and  $G$  a grammar in Greibach normal form with  $L(G) = L$ . The rules of  $G$ , except for  $S \rightarrow \lambda$ , have the form  $A \rightarrow aA_1A_2 \dots A_n$ . In a leftmost derivation, the variables  $A_i$  must be processed in a left-to-right manner. Pushing  $A_1A_2 \dots A_n$  onto the stack stores the variables in the order required by the derivation. The PDA has two states: a start state  $q_0$  and an accepting state  $q_1$ . An  $S$  rule of the form  $S \rightarrow aA_1A_2 \dots A_n$  generates a transition that processes the terminal symbol *a*, pushes the variables  $A_1A_2 \dots A_n$  onto the stack, and enters state  $q_1$ . The remainder of the computation uses the input symbol and the stack top to determine the appropriate transition.

The Greibach normal form grammar  $G$  that accepts  $\{a^i b^i \mid i > 0\}$  is used to illustrate the construction of an equivalent PDA.

$$\begin{aligned} G: S &\rightarrow aAB \mid aB \\ A &\rightarrow aAB \mid aB \\ B &\rightarrow b \end{aligned}$$

The transition function of the equivalent PDA is defined directly from the rules of  $G$ .

$$\begin{aligned} \delta(q_0, a, \lambda) &= \{[q_1, AB], [q_1, B]\} \\ \delta(q_1, a, A) &= \{[q_1, AB], [q_1, B]\} \\ \delta(q_1, b, B) &= \{[q_1, \lambda]\} \end{aligned}$$



The computation derivations in the C

The derivation of variables. Proces stack of the PDA c of a PDA equivalent to show that every c

#### Theorem 7.3.1

Let  $L$  be a context-fre

**Proof.** Let  $G = (V, T, P, S)$  be a grammar in Greibach normal form with  $L(G) = L$ . We will prove that the extended PDA

and transitions

$\delta(q_0, a, \lambda)$

$\delta(q_1, a, A)$

$\delta(q_1, b, B)$

accepts  $L$ .

We first show that the PDA

in  $M$ . The proof is by induction on the length of the derivation between derivations

the stack,  
is a  $B$ , the  
number of  
s read.  
me number  
 $\square$

es illustrate  
ow that the  
context-free

re precisely  
grammatical  
ishdown au-  
respondence

led PDA. To  
in equivalent  
al form with  
 $\dots A_n$ . In a  
mer. Pushing  
e derivation.  
e of the form  
 $a$ , pushes the  
computation  
n.

d to illustrate

les of G.

$/\lambda$   
 $/B$   
 $/AB$

$t_1$

The computation obtained by processing  $aaabbb$  exhibits the correspondence between derivations in the Greibach normal form grammar and computations in the associated PDA.

|                      |                                                  |
|----------------------|--------------------------------------------------|
| $S \Rightarrow aAB$  | $[q_0, aaabbb, \lambda] \vdash [q_1, aabbb, AB]$ |
| $\Rightarrow aaABB$  | $\vdash [q_1, abbb, ABB]$                        |
| $\Rightarrow aaaBBB$ | $\vdash [q_1, bbb, BBB]$                         |
| $\Rightarrow aaabBB$ | $\vdash [q_1, bb, BB]$                           |
| $\Rightarrow aaabbB$ | $\vdash [q_1, b, B]$                             |
| $\Rightarrow aaabbb$ | $\vdash [q_1, \lambda, \lambda]$                 |

The derivation generates a string consisting of a prefix of terminals followed by a suffix of variables. Processing an input symbol corresponds to its generation in the derivation. The stack of the PDA contains the variables in the derived string. This strategy for the generation of a PDA equivalent to a Greibach normal form grammar is formalized in Theorem 7.3.1 to show that every context-free language is accepted by a PDA.

### Theorem 7.3.1

Let  $L$  be a context-free language. Then there is a PDA that accepts  $L$ .

**Proof.** Let  $G = (V, \Sigma, P, S)$  be a grammar in Greibach normal form that generates  $L$ . The extended PDA  $M$  with start state  $q_0$  defined by

$$\begin{aligned} Q_M &= \{q_0, q_1\} \\ \Sigma_M &= \Sigma \\ \Gamma_M &= V - \{S\} \\ F_M &= \{q_1\} \end{aligned}$$

and transitions

$$\begin{aligned} \delta(q_0, a, \lambda) &= \{[q_1, w] \mid S \rightarrow aw \in P\} \\ \delta(q_1, a, A) &= \{[q_1, w] \mid A \rightarrow aw \in P \text{ and } A \in V - \{S\}\} \\ \delta(q_0, \lambda, \lambda) &= \{[q_1, \lambda]\} \text{ if } S \rightarrow \lambda \in P \end{aligned}$$

accepts  $L$ .

We first show that  $L \subseteq L(M)$ . Let  $S \xrightarrow{*} uw$  be a derivation with  $u \in \Sigma^+$  and  $w \in V^*$ . We will prove that there is a computation

$$[q_0, u, \lambda] \vdash [q_1, \lambda, w]$$

in  $M$ . The proof is by induction on the length of the derivation and utilizes the correspondence between derivations in  $G$  and computations of  $M$ .

The basis consists of derivations  $S \Rightarrow aw$  of length one. The transition generated by the rule  $S \rightarrow aw$  yields the desired computation. Assume that for all strings  $uw$  generated by derivations  $S \xrightarrow{n} uw$  there is a computation

$$[q_0, u, \lambda] \vdash [q_1, \lambda, w]$$

in M.

Now let  $S \xrightarrow{n+1} uw$  be a derivation with  $u = va \in \Sigma^+$  and  $w \in V^*$ . This derivation can be written

$$S \xrightarrow{n} vAw_2 \Rightarrow uw,$$

where  $w = w_1w_2$  and  $A \rightarrow aw_1$  is a rule in P. The inductive hypothesis and the transition  $[q_1, w_1] \in \delta(q_1, a, A)$  combine to produce the computation

$$[q_0, va, \lambda] \vdash [q_1, a, Aw_2]$$

$$\vdash [q_1, \lambda, w_1w_2].$$

For every string  $u$  in L of positive length, the acceptance of  $u$  is exhibited by the computation in M corresponding to the derivation  $S \xrightarrow{*} u$ . If  $\lambda \in L$ , then  $S \rightarrow \lambda$  is a rule of G and the computation  $[q_0, \lambda, \lambda] \vdash [q_1, \lambda, \lambda]$  accepts the null string.

The opposite inclusion,  $L(M) \subseteq L$ , is established by showing that for every computation  $[q_0, u, \lambda] \vdash [q_1, \lambda, w]$  there is a corresponding derivation  $S \xrightarrow{*} uw$  in G. The proof is by induction on the number of transitions in a computation and is left as an exercise. ■

To complete the characterization of context-free languages as precisely those accepted by pushdown automata, we must show that every language accepted by a PDA is context-free. The rules of a context-free grammar are constructed from the transitions of the automaton so that the application of a rule corresponds to a transition in the computation in the PDA. To simplify the proof, we divide the presentation into four stages:

1. The addition of transitions to the PDA so that each string in the language is accepted by a computation in which every transition both pops and pushes the stack;
2. The construction of the rules of a grammar from the modified PDA;
3. The presentation of an example that illustrates the correspondence between computations of the PDA and derivations of the grammar;
4. Finally, the formal proof that the language of the grammar and the PDA are the same.

The first two steps are constructive—adding transitions and building rules. The final step is accomplished by Lemmas 7.3.3 and 7.3.4, which show that the rules generate exactly the strings accepted by the PDA. We start with an arbitrary PDA M and show that  $L(M)$  is context-free. The proof begins by modifying M so that the transitions can be converted to rules.

Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a PDA. An extended PDA  $M'$  with transition function  $\delta'$  is obtained from M by augmenting  $\delta$  with the transitions

- i) If  $[q_j, \lambda] \in \delta(q_i, \cdot)$
- ii) If  $[q_j, B] \in \delta(q_i, \cdot)$

The interpretation of element from the stack symbol on the top of transition can also be

A grammar  $G =$   
of  $G$  is the input alpha  
the form  $(q_i, A, q_j)$  w  
represents a computat  
the stack. The rules o

1.  $S \rightarrow (q_0, \lambda, q_j)$
2. Each transition  $[q_i, A, q_j]$

3. Each transition  $[q_i, \lambda, q_j]$

4. For each state  $q_k$  e

A derivation begin  
that begins in state  $q_0$   
words, a successful co  
machine. Rules of type  
these transitions increa  
introduce an additional

Rules of type 4 are  
computation from a stat

### Example 7.3.1

A grammar  $G$  is constru

generated by  
generated

ivation can

e transition

ited by the  
•  $\lambda$  is a rule

ry computa-  
i. The proof  
exercise. ■

ose accepted  
 $A$  is context-  
tions of the  
mputation in

e is accepted  
k;

en computa-

are the same.

The final step  
erately exactly  
that  $L(M)$  is  
converted to

ition function

- i) If  $[q_j, \lambda] \in \delta(q_i, u, \lambda)$ , then  $[q_j, A] \in \delta'(q_i, u, A)$  for every  $A \in \Gamma$ .
- ii) If  $[q_j, B] \in \delta(q_i, u, \lambda)$ , then  $[q_j, BA] \in \delta'(q_i, u, A)$  for every  $A \in \Gamma$ .

The interpretation of these transitions is that a transition of  $M$  that does not remove an element from the stack can be considered to initially pop the stack and later replace the same symbol on the top of the stack. Any string accepted by a computation that utilizes a new transition can also be obtained by applying the original transition; hence,  $L(M) = L(M')$ .

A grammar  $G = (V, \Sigma, P, S)$  is constructed from the transitions of  $M'$ . The alphabet of  $G$  is the input alphabet of  $M'$ . The variables of  $G$  consist of a start symbol  $S$  and objects of the form  $\langle q_i, A, q_j \rangle$  where the  $q$ 's are states of  $M'$  and  $A \in \Gamma \cup \{\lambda\}$ . The variable  $\langle q_i, A, q_j \rangle$  represents a computation that begins in state  $q_i$ , ends in  $q_j$ , and removes the symbol  $A$  from the stack. The rules of  $G$  are constructed as follows:

1.  $S \rightarrow \langle q_0, \lambda, q_j \rangle$  for each  $q_j \in F$ .
2. Each transition  $[q_j, B] \in \delta'(q_i, x, A)$ , where  $A \in \Gamma \cup \{\lambda\}$ , generates the set of rules

$$\{ \langle q_i, A, q_k \rangle \rightarrow x \langle q_j, B, q_k \rangle \mid q_k \in Q \}.$$

3. Each transition  $[q_j, BA] \in \delta'(q_i, x, A)$ , where  $A \in \Gamma$ , generates the set of rules

$$\{ \langle q_i, A, q_k \rangle \rightarrow x \langle q_j, B, q_n \rangle \langle q_n, A, q_k \rangle \mid q_k, q_n \in Q \}.$$

4. For each state  $q_k \in Q$ ,

$$\langle q_k, \lambda, q_k \rangle \rightarrow \lambda.$$

A derivation begins with a rule of type 1 whose right-hand side represents a computation that begins in state  $q_0$ , ends in a final state, and terminates with an empty stack, in other words, a successful computation in  $M'$ . Rules of types 2 and 3 trace the action of the machine. Rules of type 3 correspond to the extended transitions of  $M'$ . In a computation, these transitions increase the size of the stack. The effect of the corresponding rule is to introduce an additional variable into the derivation.

Rules of type 4 are used to terminate derivations. The rule  $\langle q_k, \lambda, q_k \rangle \rightarrow \lambda$  represents a computation from a state  $q_k$  to itself that does not alter the stack, that is, the null computation.

### Example 7.3.1

A grammar  $G$  is constructed from the PDA  $M$ . The language of  $M$  is the set  $\{a^n cb^n \mid n \geq 0\}$ .

$$\begin{array}{ll} M: Q = \{q_0, q_1\} & \delta(q_0, a, \lambda) = \{[q_0, A]\} \\ \Sigma = \{a, b, c\} & \delta(q_0, c, \lambda) = \{[q_1, \lambda]\} \\ \Gamma = \{A\} & \delta(q_1, b, A) = \{[q_1, \lambda]\} \\ F = \{q_1\} & \end{array}$$

The transitions  $\delta'(q_0, a, A) = \{[q_0, AA]\}$  and  $\delta'(q_0, c, A) = \{[q_1, A]\}$  are added to M to construct  $M'$ . The rules of the equivalent grammar G and the transition from which they were constructed are

| Transition                                     | Rule                                                                                                                                                                                                                                                                                                                                                                                                                     |
|------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\delta(q_0, a, \lambda) = \{[q_0, A]\}$       | $S \rightarrow \langle q_0, \lambda, q_1 \rangle$<br>$\langle q_0, \lambda, q_0 \rangle \rightarrow a \langle q_0, A, q_0 \rangle$<br>$\langle q_0, \lambda, q_1 \rangle \rightarrow a \langle q_0, A, q_1 \rangle$                                                                                                                                                                                                      |
| $\delta(q_0, a, A) = \{[q_0, AA]\}$            | $\langle q_0, A, q_0 \rangle \rightarrow a \langle q_0, A, q_0 \rangle \langle q_0, A, q_0 \rangle$<br>$\langle q_0, A, q_1 \rangle \rightarrow a \langle q_0, A, q_0 \rangle \langle q_0, A, q_1 \rangle$<br>$\langle q_0, A, q_0 \rangle \rightarrow a \langle q_0, A, q_1 \rangle \langle q_1, A, q_0 \rangle$<br>$\langle q_0, A, q_1 \rangle \rightarrow a \langle q_0, A, q_1 \rangle \langle q_1, A, q_1 \rangle$ |
| $\delta(q_0, c, \lambda) = \{[q_1, \lambda]\}$ | $\langle q_0, \lambda, q_0 \rangle \rightarrow c \langle q_1, \lambda, q_0 \rangle$<br>$\langle q_0, \lambda, q_1 \rangle \rightarrow c \langle q_1, \lambda, q_1 \rangle$                                                                                                                                                                                                                                               |
| $\delta(q_0, c, A) = \{[q_1, A]\}$             | $\langle q_0, A, q_0 \rangle \rightarrow c \langle q_1, A, q_0 \rangle$<br>$\langle q_0, A, q_1 \rangle \rightarrow c \langle q_1, A, q_1 \rangle$                                                                                                                                                                                                                                                                       |
| $\delta(q_1, b, A) = \{[q_1, \lambda]\}$       | $\langle q_1, A, q_0 \rangle \rightarrow b \langle q_1, \lambda, q_0 \rangle$<br>$\langle q_1, A, q_1 \rangle \rightarrow b \langle q_1, \lambda, q_1 \rangle$<br>$\langle q_0, \lambda, q_0 \rangle \rightarrow \lambda$<br>$\langle q_1, \lambda, q_1 \rangle \rightarrow \lambda$                                                                                                                                     |

□

The relationship between computations in a PDA and derivations in the associated grammar are demonstrated using the grammar and PDA of Example 7.3.1. The derivation begins with the application of an S rule; the remaining steps correspond to the processing of an input symbol in  $M'$ . The first component of the leftmost variable contains the state of the computation. The third component of the rightmost variable contains the accepting state in which the computation will terminate. The stack can be obtained by concatenating the second components of the variables.

|                                  |                                                                                                                                    |
|----------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| $[q_0, aacbb, \lambda]$          | $S \Rightarrow \langle q_0, \lambda, q_1 \rangle$                                                                                  |
| $\vdash [q_0, acbb, A]$          | $\Rightarrow a \langle q_0, A, q_1 \rangle$                                                                                        |
| $\vdash [q_0, cbb, AA]$          | $\Rightarrow aa \langle q_0, A, q_1 \rangle \langle q_1, A, q_1 \rangle$                                                           |
| $\vdash [q_1, bb, AA]$           | $\Rightarrow aac \langle q_1, A, q_1 \rangle \langle q_1, A, q_1 \rangle$                                                          |
| $\vdash [q_1, b, A]$             | $\Rightarrow aacb \langle q_1, \lambda, q_1 \rangle \langle q_1, A, q_1 \rangle$<br>$\Rightarrow aacb \langle q_1, A, q_1 \rangle$ |
| $\vdash [q_1, \lambda, \lambda]$ | $\Rightarrow aacbb \langle q_1, \lambda, q_1 \rangle$<br>$\Rightarrow aacbb$                                                       |

where  $\langle q_i, A, q_j \rangle$  computation  $[q_k, \dots]$

The rule  $\langle q_i, u, A \rangle$ . Com hypothesis yields

are added to M from which they

The variable  $\langle q_0, \lambda, q_1 \rangle$ , obtained by the application of the  $S$  rule, indicates that a computation from state  $q_0$  to state  $q_1$  that does not alter the stack is required. The result of subsequent rule application signals the need for a computation from  $q_0$  to  $q_1$  that removes an  $A$  from the top of the stack. The fourth rule application demonstrates the necessity for augmenting the transitions of  $M$  when  $\delta$  contains transitions that do not remove a symbol from the stack. The application of the rule  $\langle q_0, A, q_1 \rangle \rightarrow c \langle q_1, A, q_1 \rangle$  represents a computation that processes  $c$  without removing the  $A$  from the top of the stack.

We are now ready to prove that a language accepted by a PDA is context-free. This result combines with Theorem 7.3.1 to establish the equivalence of string generation using context-free rules and string acceptance by pushdown automata.

- a)
- b)
- c)
- d)

### Theorem 7.3.2

Let  $M$  be a PDA. Then there is a context-free grammar  $G$  with  $L(G) = L(M)$ .

The grammar  $G$  is constructed as outlined from the extended PDA  $M'$  that is equivalent to  $M$ . We must show that there is a derivation  $S \xrightarrow{*} w$  if, and only if,  $[q_0, w, \lambda] \vdash [q_j, \lambda, \lambda]$  for some  $q_j \in F$ . This follows from Lemmas 7.3.3 and 7.3.4, which establish the correspondence of derivations in  $G$  to computations in  $M'$ .

### Lemma 7.3.3

If  $\langle q_i, A, q_j \rangle \xrightarrow{*} w$  where  $w \in \Sigma^*$  and  $A \in \Gamma \cup \{\lambda\}$ , then  $[q_i, w, A] \vdash [q_j, \lambda, \lambda]$ .

**Proof.** The proof is by induction on the length of derivations of terminal strings from variables of the form  $\langle q_i, A, q_j \rangle$ . The basis consists of derivations of strings consisting of a single rule application. The null string is the only terminal string derivable with one rule application. The derivation has the form  $\langle q_i, \lambda, q_i \rangle \xrightarrow{*} \lambda$  utilizing a rule of type 4. The null computation in state  $q_i$  yields  $[q_i, \lambda, \lambda] \vdash [q_i, \lambda, \lambda]$  as desired.

Assume that there is a computation  $[q_i, v, A] \vdash [q_j, \lambda, \lambda]$  whenever  $\langle q_i, A, q_j \rangle \xrightarrow{*} v$ . Let  $w$  be a terminal string derivable from  $\langle q_i, A, q_j \rangle$  by a derivation of length  $n + 1$ . The first step of the derivation consists of the application of a rule of type 2 or 3. A derivation initiated by a rule of type 2 can be written

$$\begin{aligned} \langle q_i, A, q_j \rangle &\xrightarrow{*} u \langle q_k, B, q_j \rangle \\ &\xrightarrow{n} uv = w, \end{aligned}$$

where  $\langle q_i, A, q_j \rangle \xrightarrow{*} u \langle q_k, B, q_j \rangle$  is a rule of  $G$ . By the inductive hypothesis, there is a computation  $[q_k, v, B] \vdash [q_j, \lambda, \lambda]$  corresponding to the derivation  $\langle q_k, B, q_j \rangle \xrightarrow{n} v$ .

The rule  $\langle q_i, A, q_j \rangle \xrightarrow{*} u \langle q_k, B, q_j \rangle$  in  $G$  is generated by a transition  $[q_k, B] \in \delta(q_i, u, A)$ . Combining this transition with the computation established by the inductive hypothesis yields

$$\begin{aligned} [q_i, uv, A] &\vdash [q_k, v, B] \\ &\vdash [q_j, \lambda, \lambda]. \end{aligned}$$

If the first step of the derivation is a rule of type 3, the derivation can be written

$$\langle q_i, A, q_j \rangle \Rightarrow u \langle q_k, B, q_m \rangle \langle q_m, A, q_j \rangle \\ \stackrel{n}{\Rightarrow} w.$$

The corresponding computation is constructed from the transition  $[q_k, BA] \in \delta(q_i, u, A)$  and two invocations of the inductive hypothesis.

### **Lemma 7.3.4**

If  $[q_i, w, A] \vdash [q_j, \lambda, \lambda]$  where  $A \in \Gamma \cup \{\lambda\}$ , then there is a derivation  $\langle q_i, A, q_j \rangle \Rightarrow^* w$ .

**Proof.** The null computation from configuration  $[q_i, \lambda, \lambda]$  is the only computation of M that uses no transitions. The corresponding derivation consists of a single application of the rule  $(q_i, \lambda, q_i) \rightarrow \lambda$ .

Assume that every computation  $[q_i, v, A] \Vdash [q_j, \lambda, \lambda]$  has a corresponding derivation  $\langle q_i, A, q_j \rangle \stackrel{*}{\Rightarrow} v$  in  $G$ . Consider a computation of length  $n + 1$ . A computation of the prescribed form beginning with a nonextended transition can be written

$$\vdash [q_k, v, B]$$

where  $w = uv$  and  $[q_k, B] \in \delta(q_i, u, A)$ . By the inductive hypothesis, there is a derivation  $\langle q_k, B, q_j \rangle \xrightarrow{*} v$ . The first transition generates the rule  $\langle q_i, A, q_j \rangle \rightarrow u \langle q_k, B, q_j \rangle$  in G. Hence a derivation of  $w$  from  $\langle q_i, A, q_j \rangle$  can be obtained by

$$\langle q_i, A, q_j \rangle \Rightarrow u \langle q_k, B, q_j \rangle \\ \stackrel{*}{\Rightarrow} uu$$

A computation in  $M'$  beginning with an extended transition  $[q_j, BA] \in \delta(q_i, u, A)$  has the form

$[q_i, w, A]$   
 $\vdash [q_k, v, BA]$   
 $\vdash^* [q_m, y, A]$   
 $\vdash^* [q_j, \lambda, \lambda],$

where  $w = uv$  and  $v = xy$ . The rule  $\langle q_i, A, q_j \rangle \rightarrow u \langle q_k, B, q_m \rangle \langle q_l, A, q_j \rangle$  is generated by the first transition of the computation. By the inductive hypothesis,  $G$  contains derivations

$$\langle q_k, B, q_m \rangle \xrightarrow{*} x$$

$$\langle q_m, A, q_i \rangle \xrightarrow{*} y.$$

Combining these observations, we have

*Proof of Theorem*

Conversely, if  $\langle q_0, \lambda, q_j \rangle \xrightarrow{*} w$  in  $\mathcal{A}$ , then there is a computation path that accepts  $w$ . Let  $\langle q_0, \lambda, q_j \rangle \xrightarrow{*} w$  in  $\mathcal{A}$ . Initiating the previous computation at  $q_0$  leads to  $w$ .

## 7.4 The Pump

The pumping lemma for strings in a regular language states that with the resulting string, it is possible to simultaneously repeat some part of it with a Chomsky normal form lemma.

There are two main types of derivation trees based on the number  $k$  such that

1. the derivation of  
 $A \xrightarrow{*} vAx$ , with
  2. the strings  $v$  and  
 remaining in the

The relationship between the first milestone, and the relationship between form grammars is obt

**Lemma 7.4.1**

Let  $G$  be a context-free grammar. A string  $w \in \Sigma^*$  with derivation

**Proof.** The proof is by induction on the length of strings. Since  $G$  is in  $C$ , it generates at least one string. This is the base case of the induction.

written

Combining these derivations with the preceding rule produces a derivation of  $w$  from  $\langle q_i, A, q_j \rangle$ . ■

$\in \delta(q_i, u, A)$

,  $q_j \rangle \xrightarrow{*} w$ .

putation of M  
lication of the

ing derivation  
itation of the

is a derivation  
,  $B, q_j \rangle$  in G.

$\delta(q_i, u, A)$  has

**Proof of Theorem 7.3.2.** Let  $w$  be any string in  $L(G)$  with derivation  $S \Rightarrow \langle q_0, \lambda, q_j \rangle \xrightarrow{*} w$ . By Lemma 7.3.3, there is a computation  $[q_0, w, \lambda] \xrightarrow{M} [q_j, \lambda, \lambda]$  exhibiting the acceptance of  $w$  by  $M'$ .

Conversely, if  $w \in L(M) = L(M')$ , then there is a computation  $[q_0, w, \lambda] \xrightarrow{M} [q_j, \lambda, \lambda]$  that accepts  $w$ . Lemma 7.3.4 establishes the existence of a corresponding derivation  $\langle q_0, \lambda, q_j \rangle \xrightarrow{*} w$  in  $G$ . Since  $q_j$  is an accepting state,  $G$  contains a rule  $S \rightarrow \langle q_0, \lambda, q_j \rangle$ . Initiating the previous derivation with this rule generates  $w$  in the grammar  $G$ . ■

## 7.4 The Pumping Lemma for Context-Free Languages

The pumping lemma for regular languages, Theorem 6.6.3, showed that sufficiently long strings in a regular language have a substring that can be repeated any number of times with the resulting string remaining in the language. In this section we establish a pumping lemma for context-free languages. For context-free languages, however, pumping refers to simultaneously repeating two substrings. The ability to generate any context-free language with a Chomsky normal form grammar provides the structure needed to prove the pumping lemma.

There are two milestones in the proof of the pumping lemma. Using the properties of derivation trees built using the rules of Chomsky normal form grammars, we obtain a number  $k$  such that

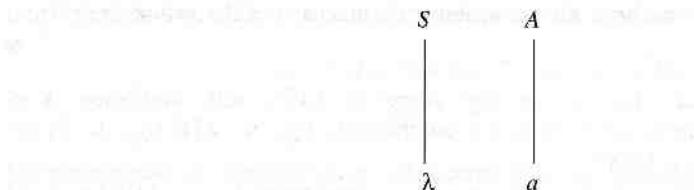
1. the derivation of any string of length  $k$  or more must have a recursive subderivation  $A \xrightarrow{*} vAx$ , with  $v, x \in \Sigma^*$ , and
2. the strings  $v$  and  $x$  can be simultaneously pumped in  $z$  with the resulting string remaining in the language.

The relationship between the number of leaves and depth of a binary tree is used to achieve the first milestone, and the repetition of the recursive subderivation establishes the latter. The relationship between string length and depth of a derivation tree for Chomsky normal form grammars is obtained in Lemma 7.4.1 and restated in Corollary 7.4.2.

### Lemma 7.4.1

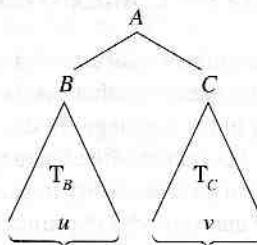
Let  $G$  be a context-free grammar in Chomsky normal form and  $A \xrightarrow{*} w$  a derivation of  $w \in \Sigma^*$  with derivation tree  $T$ . If the depth of  $T$  is  $n$ , then  $length(w) \leq 2^{n-1}$ .

**Proof.** The proof is by induction on the depth of the derivation trees that generate terminal strings. Since  $G$  is in Chomsky normal form, a derivation tree of depth 1 that represents the generation of a terminal string must have one of the following two forms.



In either case, the length of the derived string is less than or equal to  $2^0 = 1$  as required.

Assume that the property holds for all derivation trees of depth  $n$  or less. Let  $A \xrightarrow{*} w$  be a derivation with tree  $T$  of depth  $n + 1$ . Since the grammar is in Chomsky normal form, the derivation can be written  $A \Rightarrow BC \xrightarrow{*} uv$  where  $B \xrightarrow{*} u$ ,  $C \xrightarrow{*} v$ , and  $w = uv$ . The derivation tree of  $A \xrightarrow{*} w$  is constructed from  $T_B$  and  $T_C$ , the derivation trees of  $B \xrightarrow{*} u$  and  $C \xrightarrow{*} v$ .



The trees  $T_B$  and  $T_C$  both have depth  $n$  or less. By the inductive hypothesis,  $\text{length}(u) < 2^{n-1}$  and  $\text{length}(v) < 2^{n-1}$ . Therefore,  $\text{length}(w) = \text{length}(uv) < 2^n$ . ■

### **Corollary 7.4.2**

Let  $G = (V, \Sigma, P, S)$  be a context-free grammar in Chomsky normal form and  $S \xrightarrow{*} w$  a derivation of  $w \in L(G)$ . If  $\text{length}(w) \geq 2^n$ , then the derivation tree has depth at least  $n + 1$ .

### Theorem 7.4.3 (Pumping Lemma for Context-Free Languages)

Let  $L$  be a context-free language. There is a number  $k$ , depending on  $L$ , such that any string  $z \in L$  with  $\text{length}(z) > k$  can be written  $z = uvwxy$  where

- i)  $\text{length}(vwx) \leq k$
  - ii)  $\text{length}(v) + \text{length}(x) > 0$
  - iii)  $uv^iwx^i y \in L$ , for  $i \geq 0$ .

**Proof.** Let  $G = (V, \Sigma, P, S)$  be a Chomsky normal form grammar that generates  $L$  and let  $k = 2^n$  with  $n = \text{card}(V)$ . We show that all strings in  $L$  with length  $k$  or greater can be decomposed to satisfy the conditions of the pumping lemma. Let  $z \in L(G)$  be such a string and let  $S \xrightarrow{*} z$  be a derivation in  $G$ . By Corollary 7.4.2, there is a path of length at least  $n + 1 = \text{card}(V) + 1$  in the derivation tree of  $S \xrightarrow{*} z$ .

Let  $p$  be a path of maximal length from the root  $S$  to a leaf of the derivation tree. Then  $p$  must contain at least  $n + 2$  nodes, all of which are labeled by variables except the

leaf node, which some variable  $A$  appear more than one occurrence in  $p$ .

Translating  
tion of  $z$  can be done

where  $z = uvwx_1$  is a variable  $A$ . The string  $z$  before applying  $A$  is  $uv^iwx^iy \in L(G)$ .

We now show decomposition. The second occurs *B*, the derivation of

The string  $t$  is now in normal form grammar nonnull. If the second shows that  $v$  must

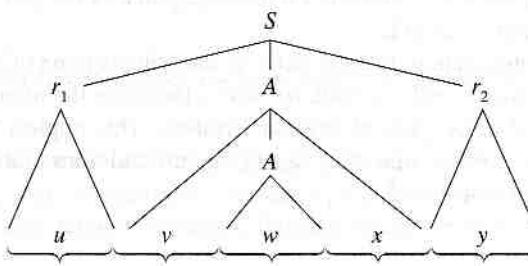
The subpath begins at most  $n + 2$ . The depth of  $\pi$  is at least  $n + 1$ . It follows from the fact that the length of  $\pi$  is  $k \equiv 2^n$  or less.

Like its counter-part, demonstrating that a sufficiently long string is able to show that a language that satisfies the recursive

leaf node, which is labeled by a terminal symbol. The pigeonhole principle guarantees that some variable  $A$  must occur twice in the final  $n + 2$  nodes of this path. Although  $A$  may appear more than twice in the path, we will be concerned only with its last and next to last occurrence in  $p$ .

Translating the properties of a path in the derivation tree to subderivations, the derivation of  $z$  can be depicted

required.  
Let  $A \xrightarrow{*} w$  be  
in normal form,  
 $v = uv$ . The  
cases of  $B \xrightarrow{*} u$



where  $z = uvwxy$ . The derivation  $S \xrightarrow{*} r_1 A r_2$  produces the next to last occurrence of the variable  $A$ . The subderivation  $A \xrightarrow{*} vAx$  may be omitted or repeated any number of times before applying  $A \xrightarrow{*} w$  to halt the recursion. The resulting derivations generate the strings  $uv^iwx^iy \in L(G) = L$ .

We now show that conditions (i) and (ii) in the pumping lemma are satisfied by this decomposition. The subderivation  $A \xrightarrow{*} vAx$  must begin with a rule of the form  $A \rightarrow BC$ . The second occurrence of the variable  $A$  is derived from either  $B$  or  $C$ . If it is derived from  $B$ , the derivation can be written

$$\begin{aligned} A &\Rightarrow BC \\ &\xrightarrow{*} vAsC \\ &\xrightarrow{*} vAst \\ &= vAx. \end{aligned}$$

> hypothesis,  
 $\leq 2^n$ . ■

and  $S \xrightarrow{*} w$  at  
at least  $n + 1$ .

that any string

generates  $L$  and  
greater can be  
such a string  
length at least  
derivation tree.  
bles except the

The string  $t$  is nonnull since it is obtained by a derivation from a variable in a Chomsky normal form grammar that is not the start symbol of the grammar. It follows that  $x$  is also nonnull. If the second occurrence of  $A$  is derived from the variable  $C$ , a similar argument shows that  $v$  must be nonnull.

The subpath between the final two occurrences of  $A$  in the path  $p$  must be of length at most  $n + 2$ . The derivation tree generated by the derivation  $A \xrightarrow{*} vwx$  has depth of at most  $n + 1$ . It follows from Lemma 7.4.1 that the string  $vwx$  obtained from this derivation has length  $k = 2^n$  or less. ■

Like its counterpart for regular languages, the pumping lemma provides a tool for demonstrating that languages are not context-free. By the pumping lemma, every sufficiently long string in a context-free grammar must have pumpable substrings. Thus we can show that a language is not context-free by finding a string that has no decomposition  $uvwxy$  that satisfies the requirement of Theorem 7.4.3.

**Example 7.4.1**

The language  $L = \{a^i b^j c^l \mid i, j, l \geq 0\}$  is not context-free. Assume  $L$  is context-free. By Theorem 7.4.1, the string  $z = a^k b^k c^k$ , where  $k$  is the number specified by the pumping lemma, can be decomposed into substrings  $uvwxy$  that satisfy the repetition properties. Consider the possibilities for the substrings  $v$  and  $x$ . If either of these contains more than one type of terminal symbol, then  $uv^2wx^2y$  contains a  $b$  preceding an  $a$  or a  $c$  preceding a  $b$ . In either case, the resulting string is not in  $L$ .

By the previous observation,  $v$  and  $x$  must be substrings of one of  $a^k$ ,  $b^k$ , or  $c^k$ . Since at most one of the strings  $v$  and  $x$  is null,  $uv^2wx^2y$  increases the number of at least one, maybe two, but not all three types of terminal symbols. This implies that  $uv^2wx^2y \notin L$ . Thus there is no decomposition of  $a^k b^k c^k$  satisfying the conditions of the pumping lemma; consequently,  $L$  is not context-free.  $\square$

**Example 7.4.2**

The language  $L = \{a^i b^j a^i b^j \mid i, j \geq 0\}$  is not context-free. Let  $k$  be the number specified by the pumping lemma and  $z = a^k b^k a^k b^k$ . Assume there is a decomposition  $uvwxy$  of  $z$  that satisfies the conditions of the pumping lemma. Condition (ii) requires the length of  $vwx$  to be at most  $k$ . This implies that  $vwx$  is a string containing only one type of terminal or the concatenation of two such strings. That is,

- i)  $vwx \in a^*$  or  $vwx \in b^*$ , or
- ii)  $vwx \in a^*b^*$  or  $vwx \in b^*a^*$ .

By an argument similar to that in Example 7.4.1, the substrings  $v$  and  $x$  must contain only one type of terminal. Pumping  $v$  and  $x$  increases the number of  $a$ 's or  $b$ 's in only one of the substrings in  $z$ . Since there is no decomposition of  $z$  satisfying the conditions of the pumping lemma, we conclude that  $L$  is not context-free.  $\square$

**Example 7.4.3**

The language  $L = \{w \in a^* \mid \text{length}(w) \text{ is prime}\}$  is not context-free. Assume  $L$  is context-free and  $n$  a prime greater than  $k$ , the constant of Theorem 7.4.3. The string  $a^n$  must have a decomposition  $uvwxy$  that satisfies the conditions of the pumping lemma. Let  $m = \text{length}(u) + \text{length}(w) + \text{length}(y)$ . The length of any string  $uv^iwx^i y$  is  $m + i(n - m)$ .

In particular,  $\text{length}(uv^{n+1}wx^{n+1}y) = m + (n + 1)(n - m) = n(n - m + 1)$ . Both of the terms in the preceding product are natural numbers greater than 1. Consequently, the length of  $uv^{n+1}wx^{n+1}y$  is not prime and the string is not in  $L$ . Thus,  $L$  is not context-free.  $\square$

**7.5 Closure**

The flexibility of the pumping lemma allows us to prove closure properties for the set of context-free languages. This is another tool for proving that a language is not context-free. We will use the pumping lemma to prove closure properties for context-free languages.

**Theorem 7.5.1**

The family of context-free languages is closed under union and Kleene star.

**Proof.** Let  $L_1$  and  $L_2$  be context-free languages. Let  $G_1 = (V_1, \Sigma_1, P_1, S_1)$  and  $G_2 = (V_2, \Sigma_2, P_2, S_2)$  be their respective grammars. Since  $L_1 \cup L_2$  is the union of  $L_1$  and  $L_2$ , it is disjoint. Since we can construct a grammar for the union of two disjoint context-free languages,  $L_1 \cup L_2$  is context-free.

A context-free grammar is closed under Kleene star. This follows from the closure property of regular languages under Kleene star.

**Union:** Define  $G = (V, \Sigma, P, S)$  where  $V = V_1 \cup V_2$ ,  $\Sigma = \Sigma_1 \cup \Sigma_2$ ,  $P = P_1 \cup P_2$ , and  $S = S_1 \cup S_2$ . Then  $L(G) = L_1 \cup L_2$ . On the other hand, if  $L_1 \cup L_2$  is context-free, then there exists a grammar  $G$  such that  $L(G) = L_1 \cup L_2$ . The derivation of  $w$  in  $L(G)$  must be a sequence of derivations in  $L_1$  and  $L_2$ . The derivation of  $w$  in  $L_1$  must be a sequence of derivations in  $L_1$  and  $w$  must be in  $L_1$ . The derivation of  $w$  in  $L_2$  must be a sequence of derivations in  $L_2$  and  $w$  must be in  $L_2$ . The derivation of  $w$  in  $L(G)$  must be a sequence of derivations in  $L_1$  and  $L_2$  and  $w$  must be in  $L_1 \cup L_2$ . This proves that  $L_1 \cup L_2$  is context-free.

**Concatenation:** Define  $G = (V, \Sigma, P, S)$  where  $V = V_1 \cup V_2$ ,  $\Sigma = \Sigma_1 \cup \Sigma_2$ ,  $P = P_1 \cup P_2$ , and  $S = S_1 \cup S_2$ . Then  $L(G) = L_1 L_2$ . On the other hand, if  $L_1 L_2$  is context-free, then there exists a grammar  $G$  such that  $L(G) = L_1 L_2$ . The derivation of  $w$  in  $L(G)$  must be a sequence of derivations in  $L_1$  and  $L_2$ . The derivation of  $w$  in  $L_1$  must be a sequence of derivations in  $L_1$  and  $w$  must be in  $L_1$ . The derivation of  $w$  in  $L_2$  must be a sequence of derivations in  $L_2$  and  $w$  must be in  $L_2$ . The derivation of  $w$  in  $L(G)$  must be a sequence of derivations in  $L_1$  and  $L_2$  and  $w$  must be in  $L_1 L_2$ . This proves that  $L_1 L_2$  is context-free.

**Kleene star:** Define  $G = (V, \Sigma, P, S)$  where  $V = V_1 \cup V_2$ ,  $\Sigma = \Sigma_1 \cup \Sigma_2$ ,  $P = P_1 \cup P_2$ , and  $S = S_1 \cup S_2$ . Then  $L(G) = L_1^*$ . On the other hand, if  $L_1^*$  is context-free, then there exists a grammar  $G$  such that  $L(G) = L_1^*$ . The derivation of  $w$  in  $L(G)$  must be a sequence of derivations in  $L_1$ . The derivation of  $w$  in  $L_1$  must be a sequence of derivations in  $L_1$  and  $w$  must be in  $L_1$ . The derivation of  $w$  in  $L(G)$  must be a sequence of derivations in  $L_1$  and  $w$  must be in  $L_1^*$ . This proves that  $L_1^*$  is context-free.

**Theorem 7.5.1** *The family of context-free languages is closed under union and Kleene star.*

**Theorem 7.5.2**

The set of context-free languages is closed under intersection.

**Proof.**

**Intersection:** Let  $L_1$  and  $L_2$  be context-free languages. Define  $G = (V, \Sigma, P, S)$  where  $V = V_1 \cup V_2$ ,  $\Sigma = \Sigma_1 \cup \Sigma_2$ ,  $P = P_1 \cup P_2$ , and  $S = S_1 \cup S_2$ . Then  $L(G) = L_1 \cap L_2$ . On the other hand, if  $L_1 \cap L_2$  is context-free, then there exists a grammar  $G$  such that  $L(G) = L_1 \cap L_2$ . The derivation of  $w$  in  $L(G)$  must be a sequence of derivations in  $L_1$  and  $L_2$ . The derivation of  $w$  in  $L_1$  must be a sequence of derivations in  $L_1$  and  $w$  must be in  $L_1$ . The derivation of  $w$  in  $L_2$  must be a sequence of derivations in  $L_2$  and  $w$  must be in  $L_2$ . The derivation of  $w$  in  $L(G)$  must be a sequence of derivations in  $L_1$  and  $L_2$  and  $w$  must be in  $L_1 \cap L_2$ . This proves that  $L_1 \cap L_2$  is context-free.

## 7.5 Closure Properties of Context-Free Languages

By The pumping lemma, consider one type of  $a^k b$ . In either

, or  $c^k$ . Since at least one,  $v^2 w x^2 y \notin L$ .  
pumping lemma;

□

specified by  $wxy$  of  $z$  that  $gth$  of  $vwx$  to  $x$  terminal or the

t contain only  
n only one of  
ditions of the

□

The flexibility of the rules of context-free grammars is used to establish closure results for the set of context-free languages. Operations that preserve context-free languages provide another tool for proving that languages are context-free. These operations, combined with the pumping lemma, can also be used to show that certain languages are not context-free.

### Theorem 7.5.1

The family of context-free languages is closed under the operations union, concatenation, and Kleene star.

**Proof.** Let  $L_1$  and  $L_2$  be context-free languages generated by  $G_1 = (V_1, \Sigma_1, P_1, S_1)$  and  $G_2 = (V_2, \Sigma_2, P_2, S_2)$ , respectively. The sets  $V_1$  and  $V_2$  of variables are assumed to be disjoint. Since we may rename variables, this assumption imposes no restriction on the grammars.

A context-free grammar will be constructed from  $G_1$  and  $G_2$  that establishes the desired closure property.

**Union:** Define  $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 | S_2\}, S)$ . A string  $w$  is in  $L(G)$  if, and only if, there is a derivation  $S \Rightarrow S_i \xrightarrow{G_i} w$  for  $i = 1$  or 2. Thus  $w$  is in  $L_1$  or  $L_2$ . On the other hand, any derivation  $S_i \xrightarrow{G_i} w$  can be initialized with the rule  $S \rightarrow S_i$  to generate  $w$  in  $G$ .

**Concatenation:** Define  $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}, S)$ . The start symbol initiates derivations in both  $G_1$  and  $G_2$ . A leftmost derivation of a terminal string in  $G$  has the form  $S \Rightarrow S_1 S_2 \xrightarrow{G_1} u S_2 \xrightarrow{G_2} uv$ , where  $u \in L_1$  and  $v \in L_2$ . The derivation of  $u$  uses only rules from  $P_1$  and  $v$  rules from  $P_2$ . Hence  $L(G) \subseteq L_1 L_2$ . The opposite inclusion is established by observing that every string  $w$  in  $L_1 L_2$  can be written  $uv$  with  $u \in L_1$  and  $v \in L_2$ . The derivations  $S_1 \xrightarrow{G_1} u$  and  $S_2 \xrightarrow{G_2} v$ , along with the  $S$  rule of  $G$ , generate  $w$  in  $G$ .

**Kleene star:** Define  $G = (V_1, \Sigma_1, P_1 \cup \{S \rightarrow S_1 S \mid \lambda\}, S)$ . The  $S$  rule of  $G$  generates any number of copies of  $S_1$ . Each of these, in turn, initiates the derivation of a string in  $L_1$ . The concatenation of any number of strings from  $L_1$  yields  $L_1^*$ . ■

Theorem 7.5.1 presented positive closure results for the set of context-free languages. A simple example is given to show that the context-free languages are not closed under intersection. Finally, we combine the closure properties of union and intersection to obtain a similar negative result for complementation.

### Theorem 7.5.2

The set of context-free languages is not closed under intersection or complementation.

#### Proof.

**Intersection:** Let  $L_1 = \{a^i b^j c^j \mid i, j \geq 0\}$  and  $L_2 = \{a^j b^i c^i \mid i, j \geq 0\}$ .  $L_1$  and  $L_2$  are both context-free since they are generated by  $G_1$  and  $G_2$ , respectively.

$$\begin{array}{ll} G_1: S \rightarrow BC & G_2: S \rightarrow AB \\ B \rightarrow aBb \mid \lambda & A \rightarrow aA \mid \lambda \\ C \rightarrow cC \mid \lambda & B \rightarrow bBc \mid \lambda \end{array}$$

The intersection of  $L_1$  and  $L_2$  is the set  $\{a^i b^i c^i \mid i \geq 0\}$ , which is not context-free by Example 7.4.1.

*Complementation:* Let  $L_1$  and  $L_2$  be any two context-free languages. If the context-free languages are closed under complementation, then by Theorem 7.5.1, the language

$$L = \overline{\overline{L_1} \cup \overline{L_2}}$$

is context-free. By DeMorgan's Law,  $L = L_1 \cap L_2$ . This implies that the context-free languages are closed under intersection, contradicting the result of part (i). ■

Exercise 9 of Chapter 6 showed that the intersection of a regular and context-free language need not be regular. The correspondence between languages and pushdown automata is used to establish a positive closure property for the intersection of regular and context-free languages.

Let  $R$  be a regular language accepted by a DFA  $N$  and  $L$  a context-free language accepted by PDA  $M$ . We show that  $R \cap L$  is context-free by constructing a single PDA that simulates the operation of both  $N$  and  $M$ . The states of this composite machine are ordered pairs consisting of a state from  $M$  and one from  $N$ .

### Theorem 7.5.3

Let  $R$  be a regular language and  $L$  a context-free language. Then the language  $R \cap L$  is context-free.

**Proof.** Let  $N = (Q_N, \Sigma_N, \delta_N, q_0, F_N)$  be a DFA that accepts  $R$  and let  $M = (Q_M, \Sigma_M, \Gamma, \delta_M, p_0, F_M)$  be a PDA that accepts  $L$ . The machines  $N$  and  $M$  are combined to construct a PDA

$$M' = (Q_M \times Q_N, \Sigma_M \cup \Sigma_N, \Gamma, \delta, [p_0, q_0], F_M \times F_N)$$

that accepts  $R \cap L$ . The transition function of  $M'$  is defined to "run the machines  $M$  and  $N$  in parallel." The first component of the ordered pair traces the sequence of states entered by the machine  $M$  and the second component by  $N$ . The transition function of  $M'$  is defined by

- i)  $\delta([p, q], a, A) = \{[p', q'], B] \mid [p', B] \in \delta_M(p, a, A) \text{ and } \delta_N(q, a) = q'\}$
- ii)  $\delta([p, q], \lambda, A) = \{[p', q], B] \mid [p', B] \in \delta_M(p, \lambda, A)\}.$

Every transition of a DFA processes an input symbol, whereas a PDA may contain transitions that do not process input. The transitions introduced by condition (ii) simulate the action of a PDA transition that does not process an input symbol.

A string  $w$  is accepted by  $M'$  if there is a computation

$$[[p_0, q_0], w, \lambda]^* [[p_i, q_j], \lambda, \lambda],$$

where  $p_i$  and  $q_j$  are final states of  $M$  and  $N$ , respectively.

The inclusion  
of  $L_1$  and  $L_2$

whenever

are computation  
PDA  $M$ .

The basis of  
 $p_0, u = w$ , and  
the original struc-

Assume th-

be computation

where either  $v$   
[[ $p_i, q_j$ ],  $u$ ,  $\alpha$ ].

Case 1:  $v = u$ . I  
an input symbol  
 $\delta_M(p_k, \lambda, A)$ . T-

is obtained from

Case 2:  $v = au$ . T

where the final s  
for input symbol

The inclusion  $L(N) \cap L(M) \subseteq L(M')$  is established by showing that there is a computation

$$[[p_0, q_0], w, \lambda] \xrightarrow{M} [[p_i, q_j], u, \alpha]$$

context-free by

whenever

context-free  
age

context-free

xt-free lan-  
n automata  
ontext-free

e language  
single PDA  
achine are

ge  $R \cap L$  is

$Q_M, \Sigma_M, \Gamma,$   
construct a

nes M and N  
es entered by  
is defined by

'}

contain trans-  
) simulate the

$$[p_0, w, \lambda] \xrightarrow{M} [p_i, u, \alpha] \quad \text{and} \quad [q_0, w] \xrightarrow{N} [q_j, u]$$

are computations in M and N. The proof is by induction on the number of transitions in the PDA M.

The basis consists of the null computation in M. This computation terminates with  $p_i = p_0$ ,  $u = w$ , and M containing an empty stack. The only computation in N that terminates with the original string is the null computation; thus,  $q_j = q_0$ . The corresponding computation in the composite machine is the null computation in  $M'$ .

Assume the result holds for all computations of M having length  $n$ . Let

$$[p_0, w, \lambda] \xrightarrow{M} [p_i, u, \alpha] \quad \text{and} \quad [q_0, w] \xrightarrow{N} [q_j, u]$$

be computations in the PDA and DFA, respectively. The computation in M can be written

$$\begin{aligned} & [p_0, w, \lambda] \\ & \xleftarrow{M} [p_k, v, \beta] \\ & \xleftarrow{M} [p_i, u, \alpha], \end{aligned}$$

where either  $v = u$  or  $v = au$ . To show that there is a computation  $[[p_0, q_0], w, \lambda] \xrightarrow{M'} [[p_i, q_j], u, \alpha]$ , we consider each of the possibilities for  $v$  separately.

**Case 1:**  $v = u$ . In this case, the final transition of the computation in M does not process an input symbol. The computation in M is completed by a transition of the form  $[p_i, B] \in \delta_M(p_k, \lambda, A)$ . This transition generates  $[[p_i, q_j], B] \in \delta([p_k, q_j], \lambda, A)$  in  $M'$ . The computation

$$\begin{aligned} & [[p_0, q_0], w, \lambda] \xrightarrow{M} [[p_k, q_j], v, \beta] \\ & \xrightarrow{M'} [[p_i, q_j], v, \alpha] \end{aligned}$$

is obtained from the inductive hypothesis and the preceding transition of  $M'$ .

**Case 2:**  $v = au$ . The computation in N that reduces  $w$  to  $u$  can be written

$$\begin{aligned} & [q_0, w] \\ & \xrightarrow{N} [q_m, v] \\ & \xrightarrow{N} [q_j, u], \end{aligned}$$

where the final step utilizes a transition  $\delta_N(q_m, a) = q_j$ . The DFA and PDA transitions for input symbol  $a$  combine to generate the transition  $[[p_i, q_j], B] \in \delta([p_k, q_m], a, A)$  in

$M'$ . Applying this transition to the result of the computation established by the inductive hypothesis produces

$$\begin{aligned} [[p_0, q_0], w, \lambda] &\xrightarrow{p_0} [[p_k, q_m], v, \beta] \\ &\xrightarrow{M'} [[p_i, q_j], u, \alpha]. \end{aligned}$$

The opposite inclusion,  $L(M') \subseteq L(N) \cap L(M)$ , is proved using induction on the length of computations in  $M'$ . The proof is left as an exercise. ■

Theorem 7.5.2 used DeMorgan's Law to show that the family of context-free languages is not closed under complementation. The next example gives a grammar that explicitly demonstrates this property.

### Example 7.5.1

The language  $L = \{ww \mid w \in \{a, b\}^*\}$  is not context-free, but  $\bar{L}$  is. First we show that  $L$  is not context-free using a proof by contradiction. Assume  $L$  is context-free. Then, by Theorem 7.5.3,

$$L \cap a^*b^*a^*b^* = \{a^i b^j a^i b^j \mid i, j \geq 0\}$$

is context-free. However, this language was shown not to be context-free in Example 7.4.2, contradicting our assumption.

To show that  $\bar{L}$  is context-free, we construct two context-free grammars  $G_1$  and  $G_2$  with  $L(G_1) \cup L(G_2) = \bar{L}$ .

$$\begin{array}{ll} G_1: S \rightarrow aA \mid bA \mid a \mid b & G_2: S \rightarrow AB \mid BA \\ A \rightarrow aS \mid bS & A \rightarrow ZAZ \mid a \\ & B \rightarrow ZBZ \mid b \\ & Z \rightarrow a \mid b \end{array}$$

The grammar  $G_1$  generates the strings of odd length over  $\{a, b\}$ , all of which are in  $\bar{L}$ .  $G_2$  generates the set of even length string in  $\bar{L}$ . Such a string may be written  $u_1xv_1u_2yv_2$ , where  $x, y \in \Sigma$  and  $x \neq y$ ;  $u_1, u_2, v_1, v_2 \in \Sigma^*$  with  $\text{length}(u_1) = \text{length}(u_2)$  and  $\text{length}(v_1) = \text{length}(v_2)$ . That is,  $x$  and  $y$  are different symbols that occur in the same position in the substrings that make up the first half and the second half of  $u_1xv_1u_2yv_2$ . Since the  $u$ 's and  $v$ 's are arbitrary strings in  $\Sigma^*$ , this characterization can be rewritten  $u_1xpqv_2$ , where  $\text{length}(p) = \text{length}(u_1)$  and  $\text{length}(q) = \text{length}(v_2)$ . The recursive variables of  $G_2$  generate precisely this set of strings. □

### Exercises

1. Let  $M$  be the PDA
- a) Describe the  
b) Give the state  
c) Trace all com  
d) Show that  $aa$
2. Let  $M$  be the PD
- a) Give the tra  
b) Trace all com  
c) Show that  $aa$
3. Construct PDAs
- a)  $\{a^i b^j \mid 0 \leq i \leq j\}$   
b)  $\{a^i c^j b^i \mid i, j \geq 0\}$   
c)  $\{a^i b^j c^k \mid i + j = k\}$   
d)  $\{w \mid w \in \{a, b\}^*\}$   
e)  $\{a^i b^i \mid i \geq 0\}$   
f)  $\{a^i b^j c^k \mid i = j \neq k\}$   
g)  $\{a^i b^j \mid i \neq j\}$   
h)  $\{a^i b^j \mid 0 \leq i \leq j\}$   
i)  $\{a^{i+j} b^i c^j \mid i, j \geq 0\}$   
j) The set of palin
4. Construct a PDA

---

**Exercises**

1. Let M be the PDA defined by

$$\begin{array}{ll}
 Q = \{q_0, q_1, q_2\} & \delta(q_0, a, \lambda) = \{[q_0, A]\} \\
 \Sigma = \{a, b\} & \delta(q_0, \lambda, \lambda) = \{[q_1, \lambda]\} \\
 \Gamma = \{A\} & \delta(q_0, b, A) = \{[q_2, \lambda]\} \\
 F = \{q_1, q_2\} & \delta(q_1, \lambda, A) = \{[q_1, \lambda]\} \\
 & \delta(q_2, b, A) = \{[q_2, \lambda]\} \\
 & \delta(q_2, \lambda, A) = \{[q_2, \lambda]\}.
 \end{array}$$

- a) Describe the language accepted by M.
  - b) Give the state diagram of M.
  - c) Trace all computations of the strings  $abb$ ,  $abb$ ,  $aba$  in M.
  - d) Show that  $aabb$ ,  $aaab \in L(M)$ .
2. Let M be the PDA in Example 7.1.3.
- a) Give the transition table of M.
  - b) Trace all computations of the strings  $ab$ ,  $abb$ ,  $abbb$  in M.
  - c) Show that  $aaaa$ ,  $baab \in L(M)$ .
  - d) Show that  $aaa$ ,  $ab \notin L(M)$ .
3. Construct PDAs that accept each of the following languages.
- a)  $\{a^i b^j \mid 0 \leq i \leq j\}$
  - b)  $\{a^i c^j b^i \mid i, j \geq 0\}$
  - c)  $\{a^i b^j c^k \mid i + k = j\}$
  - d)  $\{w \mid w \in \{a, b\}^*\text{ and }w\text{ has twice as many }a\text{'s as }b\text{'s}\}$
  - e)  $\{a^i b^i \mid i \geq 0\} \cup a^* \cup b^*$
  - f)  $\{a^i b^j c^k \mid i = j \text{ or } j = k\}$
  - g)  $\{a^i b^j \mid i \neq j\}$
  - h)  $\{a^i b^j \mid 0 \leq i \leq j \leq 2i\}$
  - i)  $\{a^{i+j} b^i c^j \mid i, j > 0\}$
  - j) The set of palindromes over  $\{a, b\}$
4. Construct a PDA with only two stack elements that accepts the language

$$\{wdw^R \mid w \in \{a, b, c\}^*\}.$$

5. Give the state diagram of a PDA  $M$  that accepts  $\{a^{2i}b^{i+j} \mid 0 \leq j \leq i\}$  with acceptance by empty stack. Explain the role of the stack symbols in the computation of  $M$ . Trace the computations of  $M$  with input  $aabb$  and  $aaaabb$ .

6. The machine  $M$



accepts the language  $L = \{a^i b^i \mid i > 0\}$  by final state and empty stack.

- a) Give the state diagram of a PDA that accepts  $L$  by empty stack.
- b) Give the state diagram of a PDA that accepts  $L$  by final state.
- 7. Let  $L$  be the language  $\{w \in \{a, b\}^* \mid w \text{ has a prefix containing more } b\text{'s than } a\text{'s}\}$ . For example,  $baa, abba, abbaaa \in L$ , but  $aab, aabbab \notin L$ .
  - a) Construct a PDA that accepts  $L$  by final state.
  - b) Construct a PDA that accepts  $L$  by empty stack.
- 8. Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a PDA that accepts  $L$  by final state and empty stack. Prove that there is a PDA that accepts  $L$  by final state alone.
- 9. Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a PDA that accepts  $L$  by final state and empty stack. Prove that there is a PDA that accepts  $L$  by empty stack alone.
- 10. Let  $L = \{a^{2i}b^i \mid i \geq 0\}$ .
  - a) Construct a PDA  $M_1$  with  $L(M_1) = L$ .
  - b) Construct an atomic PDA  $M_2$  with  $L(M_2) = L$ .
  - c) Construct an extended PDA  $M_3$  with  $L(M_3) = L$  that has fewer transitions than  $M_1$ .
  - d) Trace the computation that accepts the string  $aab$  in each of the automata constructed in parts (a), (b), and (c).
- 11. Let  $L = \{a^{2i}b^{3i} \mid i \geq 0\}$ .
  - a) Construct a PDA  $M_1$  with  $L(M_1) = L$ .
  - b) Construct an atomic PDA  $M_2$  with  $L(M_2) = L$ .
  - c) Construct an extended PDA  $M_3$  with  $L(M_3) = L$  that has fewer transitions than  $M_1$ .
  - d) Trace the computation that accepts the string  $aabbb$  in each of the automata constructed in parts (a), (b), and (c).
- 12. Use the technique of Theorem 7.3.1 to construct a PDA that accepts the language of the Greibach normal form grammar

$$\begin{aligned} S &\rightarrow aABA \mid aBB \\ A &\rightarrow bA \mid b \\ B &\rightarrow cB \mid c \end{aligned}$$

- 13. Let  $G$  be a grammar. Prove that  $L(G)$  is closed under union. This completes the proof of Theorem 7.3.1.
- 14. Let  $M$  be the PDA
- a) Give the state diagram of a PDA that accepts  $L$  by final state.
- b) Give a sequence of strings that generates  $L$ .
- c) Using the PDA, trace the computation that generates  $w$ .
- d) Trace the computation that accepts the string  $w$ .
- e) Give the state diagram of a PDA that accepts  $L$  by empty stack.
- 15. Let  $M$  be the PDA
- a) Trace the computation that accepts the string  $w$ .
- b) Use the technique of Theorem 7.3.1 to construct a PDA that accepts  $L$  by empty stack.
- c) Give the state diagram of a PDA that accepts  $L$  by final state.
- \* 16. Theorem 7.3.1. If  $L$  is a context-free language accepted by a PDA, then there is an extended PDA that accepts  $L$  by final state.
- 17. Use the pumping lemma for context-free languages to prove that  $\{a^k b^k c^k \mid k \geq 0\}$  is not context-free.
- a)  $\{a^k \mid k \text{ is a multiple of } 3\}$
- b)  $\{a^i b^j c^i d^j \mid i, j \geq 0\}$
- c)  $\{a^i b^{2i} a^i \mid i \geq 0\}$
- d)  $\{a^i b^j c^k \mid 0 \leq i, j, k \leq n\}$
- e)  $\{ww^Rw \mid w \in \{a, b\}^*\}$
- f) The set of strings that are palindromes over the alphabet  $\{a, b\}$ .
- 18. a) Prove that  $L(G)$  is closed under union.
- b) Prove that  $L(G)$  is closed under intersection.
- c) Prove that  $L(G)$  is closed under complement.

13. Let  $G$  be a grammar in Greibach normal form and  $M$  the PDA constructed from  $G$ . Prove that if  $[q_0, u, \lambda] \vdash [q_1, \lambda, w]$  in  $M$ , then there is a derivation  $S \xrightarrow{*} uw$  in  $G$ . This completes the proof of Theorem 7.3.1.

14. Let  $M$  be the PDA

$$\begin{array}{ll} Q = \{q_0, q_1, q_2\} & \delta(q_0, a, \lambda) = \{[q_0, A]\} \\ \Sigma = \{a, b\} & \delta(q_0, b, A) = \{[q_1, \lambda]\} \\ \Gamma = \{A\} & \delta(q_1, b, \lambda) = \{[q_2, \lambda]\} \\ F = \{q_2\} & \delta(q_2, b, A) = \{[q_1, \lambda]\}. \end{array}$$

- a) Give the state diagram of  $M$ .
- b) Give a set-theoretic definition of  $L(M)$ .
- c) Using the technique from Theorem 7.3.2, build a context-free grammar  $G$  that generates  $L(M)$ .
- d) Trace the computation of  $aabb$  in  $M$ .
- e) Give the derivation of  $aabb$  in  $G$ .

15. Let  $M$  be the PDA in Example 7.1.1.

- a) Trace the computation in  $M$  that accepts  $bbcb$ .
- b) Use the technique from Theorem 7.3.2 to construct a grammar  $G$  that accepts  $L(M)$ .
- c) Give the derivation of  $bbcb$  in  $G$ .

- \*16. Theorem 7.3.2 presented a technique for constructing a grammar that generates the language accepted by an extended PDA. The transitions of the PDA pushed at most two variables onto the stack. Generalize this construction to build grammars from arbitrary extended PDAs. Prove that the resulting grammar generates the language of the PDA.

17. Use the pumping lemma to prove that each of the following languages is not context-free.

- a)  $\{a^k \mid k \text{ is a perfect square}\}$
- b)  $\{a^i b^j c^i d^j \mid i, j \geq 0\}$
- c)  $\{a^i b^{2i} a^i \mid i \geq 0\}$
- d)  $\{a^i b^j c^k \mid 0 < i < j < k < zi\}$
- e)  $\{ww^Rw \mid w \in \{a, b\}^*\}$
- f) The set of finite-length prefixes of the infinite string

$abaabaaaabaaaab \dots ba^nba^{n+1}b \dots$

18. a) Prove that the language  $L_1 = \{a^i b^{2i} c^j \mid i, j \geq 0\}$  is context-free.  
 b) Prove that the language  $L_2 = \{a^j b^i c^{2i} \mid i, j \geq 0\}$  is context-free.  
 c) Prove that  $L_1 \cap L_2$  is not context-free.

19. a) Prove that the language  $L_1 = \{a^i b^i c^j d^j \mid i, j \geq 0\}$  is context-free.  
 b) Prove that the language  $L_2 = \{a^j b^i c^i d^k \mid i, j, k \geq 0\}$  is context-free.  
 c) Prove that  $L_1 \cap L_2$  is not context-free.
20. Let  $L$  be the language consisting of all strings over  $\{a, b\}$  with the same number of  $a$ 's and  $b$ 's. Show that the pumping lemma is satisfied for  $L$ . That is, show that every string  $z$  of length  $k$  or more has a decomposition that satisfies the conditions of the pumping lemma.
21. Let  $M$  be a PDA. Prove that there is a decision procedure to determine whether  
 a)  $L(M)$  is empty.  
 b)  $L(M)$  is finite.  
 c)  $L(M)$  is infinite.
- \* 22. A grammar  $G = (V, \Sigma, P, S)$  is called **linear** if every rule has the form

$$\begin{aligned} A &\rightarrow u \\ A &\rightarrow uBv \end{aligned}$$

where  $u, v \in \Sigma^*$  and  $A, B \in V$ . A language is called linear if it is generated by a linear grammar. Prove the following pumping lemma for linear languages.

Let  $L$  be a linear language. Then there is a constant  $k$  such that for all  $z \in L$  with  $\text{length}(z) \geq k$ ,  $z$  can be written  $z = uvwxy$  with

- i)  $\text{length}(uvx) \leq k$ ,
- ii)  $\text{length}(vx) > 0$ , and
- iii)  $uv^iwx^i y \in L$ , for  $i \geq 0$ .

23. a) Construct a DFA  $N$  that accepts all strings in  $\{a, b\}^*$  with an odd number of  $a$ 's.  
 b) Construct a PDA  $M$  that accepts  $\{a^{3i}b^i \mid i \geq 0\}$ .  
 c) Use the technique from Theorem 7.5.3 to construct a PDA  $M'$  that accepts  $L(N) \cap L(M)$ .  
 d) Trace the computations that accept  $aaab$  in  $N$ ,  $M$ , and  $M'$ .
24. Let  $G = (V, \Sigma, P, S)$  be a context-free grammar. Define an extended PDA  $M$  as follows:

$$\begin{aligned} Q &= \{q_0\} & \delta(q_0, \lambda, \lambda) &= \{[q_0, S]\} \\ \Sigma &= \Sigma_G & \delta(q_0, \lambda, A) &= \{[q_0, w] \mid A \rightarrow w \in P\} \\ \Gamma &= \Sigma_G \cup V & \delta(q_0, a, a) &= \{[q_0, \lambda] \mid a \in \Sigma\}. \\ F &= \{q_0\} \end{aligned}$$

Prove that  $L(M) = L(G)$ .

25. Complete the proof of Theorem 7.5.3.

26. Prove that  
 \* 27. Let  $L$  be a language generated by a PDA. Prove that  $L$  with  $a$  removed is still context-free. Prove that every context-free language may be generated by a PDA.
- \* 28. The notion of context-freeness is based on the following three properties:  
 a) Prove that the language  $\{a^n b^n \mid n \geq 0\}$  is not context-free.  
 b) Use the pumping lemma to prove that the language  $\{a^n b^n c^n \mid n \geq 0\}$  is not context-free.  
 c) Give an example of a context-free language that may be generated by a PDA.

29. Let  $h : \Sigma^* \rightarrow \{w \mid h(w) \text{ is closed under } \cup\}$ .  
 30. Use closure properties of context-free languages to prove that  
 a)  $\{a^i b^j c^i \mid i, j \geq 0\}$  is context-free.  
 b)  $\{a^i b^{2i} c^3 \mid i \geq 0\}$  is context-free.  
 c)  $\{(ab)^i \mid i \geq 0\}$  is context-free.

## Bibliography

Pushdown automata were studied by many authors. Some of the languages between context-free and context-sensitive were studied by Evey [1963], and some others were presented in Section 7.6. A solution to the pumping problem was given by Chomsky [1960]. A strong form of the pumping lemma was given by Chomsky and Miller [1961]. A strong form of the pumping lemma was given by Chomsky and Miller [1966].

- er of  $a$ 's  
ry string  
pumping  
er  
by a linear  
 $z \in L$  with  
er of  $a$ 's.  
hat accepts  
PDA M as
26. Prove that the set of context-free languages is closed under reversal.
  - \* 27. Let  $L$  be a context-free language over  $\Sigma$  and  $a \in \Sigma$ . Define  $er_a(L)$  to be the set obtained by removing all occurrences of  $a$  from strings of  $L$ . The language  $er_a(L)$  is the language  $L$  with  $a$  erased. For example, if  $abab, babc, aa \in L$ , then  $bb, bcb$ , and  $\lambda \in er_a(L)$ . Prove that  $er_a(L)$  is context-free. Hint: Convert the grammar that generates  $L$  to one that generates  $er_a(L)$ .
  - \* 28. The notion of a string homomorphism was introduced in Exercise 6.19. Let  $L$  be a context-free language over  $\Sigma$  and let  $h : \Sigma^* \rightarrow \Sigma^*$  be a homomorphism.
    - a) Prove that  $h(L) = \{h(w) \mid w \in L\}$  is context-free, that is, that the context-free languages are closed under homomorphisms.
    - b) Use the result of part (a) to show that  $er_a(L)$  is context-free.
    - c) Give an example to show that the homomorphic image of a noncontext-free language may be context-free.
  29. Let  $h : \Sigma^* \rightarrow \Sigma^*$  be a homomorphism and  $L$  a context-free language over  $\Sigma$ . Prove that  $\{w \mid h(w) \in L\}$  is context-free. In other words, the family of context-free languages is closed under inverse homomorphic images.
  30. Use closure under homomorphic images and inverse images to show that the following languages are not context-free.
    - a)  $\{a^i b^j c^i d^j \mid i, j \geq 0\}$
    - b)  $\{a^i b^{2i} c^{3i} \mid i \geq 0\}$
    - c)  $\{(ab)^i (bc)^i (ca)^i \mid i \geq 0\}$

---

### Bibliographic Notes

Pushdown automata were introduced in Oettinger [1961]. Deterministic pushdown automata were studied in Fischer [1963] and Schutzenberger [1963] and their acceptance of the languages generated by  $LR(k)$  grammars is from Knuth [1965]. The relationship between context-free languages and pushdown automata was discovered by Chomsky [1962], Evey [1963], and Schutzenberger [1963]. The closure properties for context-free languages presented in Section 7.5 are from Bar-Hillel, Perles, and Shamir [1961] and Scheinberg [1960]. A solution to Exercises 28 and 29 can be found in Ginsburg and Rose [1963b].

The pumping lemma for context-free languages is from Bar-Hillel, Perles, and Shamir [1961]. A stronger version of the pumping lemma is given in Ogden [1968]. Parikh's Theorem [1966] provides another tool for establishing that languages are not context-free.

## PART III

# Computability

We now begin our exploration of the capabilities and limitations of algorithmic computation. The term *effective procedure* is used to describe processes that we intuitively understand as computable. An effective procedure consists of a finite set of instructions and a specification, based on the input, of the order of execution of the instructions. The execution of an instruction is mechanical; it requires no cleverness or ingenuity on the part of the machine or person doing the computation. A computation produced by an effective procedure executes a finite number of instructions and terminates. The preceding properties can be summarized as follows: An effective procedure is a deterministic discrete process that halts for all possible inputs.

In 1936 British mathematician Alan Turing designed a family of abstract machines for performing effective computation. The Turing machine represents the culmination of a series of increasingly powerful abstract computing devices that include finite and pushdown automata. As with a finite automaton, the applicable Turing machine instruction is determined by the state of the machine and the symbol being read. A Turing machine may read its input multiple times and an instruction may write information to memory. The ability to perform multiple reads and writes increases the computational power of the Turing machine and provides a theoretical prototype for the modern computer.

The Church-Turing Thesis, proposed by logician Alonzo Church in 1936, asserts that any effective computation in any algorithmic system can be accomplished using a Turing machine. The Church-Turing Thesis should not be considered as providing a definition of algorithmic computation—this would be an extremely limiting viewpoint. Many systems have been designed to perform effective computations. Moreover, who can predict the formalisms and techniques that will be developed in the future? The Church-Turing Thesis does not claim that these other systems do not perform algorithmic computation. It does, however, assert that a computation performed in any such system can be accomplished by a suitably designed Turing machine. Perhaps the strongest evidence supporting the Church-Turing Thesis is that after 70 years, no counterexamples have been discovered. The formulation of this thesis and its implications for computability are discussed in Chapter 11.

The correspondence between the generation of languages by grammars and their recognition by machines extends to the languages of Turing machines. If Turing machines represent the ultimate in string recognition machines, it seems reasonable to expect the associated family of grammars to be the most general string transformation systems. This is indeed the case; the grammars that correspond to Turing machines are called unrestricted grammars because there are no restrictions on the form or the applicability of their rules. To establish the correspondence between recognition by a Turing machine and generation by an unrestricted grammar, we show that a computation of a Turing machine can be simulated by a derivation in an unrestricted grammar.

With the acceptance of the Church-Turing Thesis, the extent of algorithmic problem solving can be identified with the capabilities of Turing machine computations. Consequently, to prove a problem to be unsolvable, it suffices to show that there is no Turing machine solution to the problem. Using this approach, we show that the Halting Problem for Turing machines is undecidable. That is, there is no algorithm that can determine, for an arbitrary Turing machine  $M$  and string  $w$ , whether  $M$  will halt when run with  $w$ . We will then use problem reduction to establish undecidability of additional questions about the results of Turing machine computations, of the existence of derivations using the rules of a grammar, and of properties of context-free languages.

The Turing developer for the study with a model design and operations, unlike a computation

The Ch that any eff variations o chapters ind

### 8.1 The

A Turing ma The tape he the input as a finite autom

their recognizability. The machines represented by the associated regular grammars are indeed the regular languages. To establish this result, we can proceed by an unformalized argument.

imic problem reductions. Consequently, there is no Turing-Computability Problem. We can determine, for example, whether a string  $w$  is in a language  $L$  by an unformalized argument involving the rules

## CHAPTER 8

# Turing Machines

The Turing machine, introduced by Alan Turing in 1936, represents another step in the development of finite-state computing machines. Turing machines were originally proposed for the study of effective computation and exhibit many of the features commonly associated with a modern computer. This is no accident; the Turing machine provided a model for the design and development of the stored-program computer. Utilizing a sequence of elementary operations, a Turing machine may access and alter any memory position. A Turing machine, unlike a computer, has no limitation on the amount of time or memory available for a computation.

The Church-Turing Thesis, which will be discussed in detail in Chapter 11, asserts that any effective procedure can be realized by a suitably designed Turing machine. The variations of Turing machine architectures and applications presented in the next two chapters indicate the robustness and the versatility of Turing machine computation.

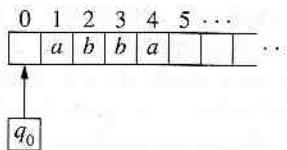
### 3.1 The Standard Turing Machine

A Turing machine is a finite-state machine in which a transition prints a symbol on the tape. The tape head may move in either direction, allowing the machine to read and manipulate the input as many times as desired. The structure of a Turing machine is similar to that of a finite automaton, with the transition function incorporating these additional features.

**Definition 8.1.1**

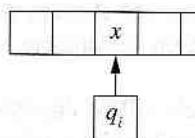
A **Turing machine** is a quintuple  $M = (Q, \Sigma, \Gamma, \delta, q_0)$  where  $Q$  is a finite set of states,  $\Gamma$  is a finite set called the *tape alphabet*,  $\Gamma$  contains a special symbol  $B$  that represents a blank,  $\Sigma$  is a subset of  $\Gamma - \{B\}$  called the *input alphabet*,  $\delta$  is a partial function from  $Q \times \Gamma$  to  $Q \times \Gamma \times \{L, R\}$  called the *transition function*, and  $q_0 \in Q$  is a distinguished state called the *start state*.

The tape of a Turing machine has a left boundary and extends indefinitely to the right. Tape positions are numbered by the natural numbers, with the leftmost position numbered zero. Each tape position contains one element from the tape alphabet.

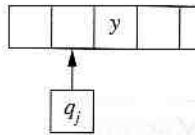


A computation begins with the machine in state  $q_0$  and the tape head scanning the leftmost position. The input, a string from  $\Sigma^*$ , is written on the tape beginning at position one. Position zero and the remainder of the tape are blank. The diagram shows the initial configuration of a Turing machine with input *abba*. The tape alphabet provides additional symbols that may be used during a computation.

A transition consists of three actions: changing the state, writing a symbol on the square scanned by the tape head, and moving the tape head. The direction of the movement is specified by the final component of the transition. An *L* indicates a move of one tape position to the left and *R* one position to the right. The machine configuration



and transition  $\delta(q_i, x) = [q_j, y, L]$  combine to produce the new configuration



The transition changed the state from  $q_i$  to  $q_j$ , replaced the tape symbol  $x$  with  $y$ , and moved the tape head one square to the left. The ability of the machine to move in both directions and process blanks introduces the possibility of a computation continuing indefinitely.

A computer defined. A tran of the tape. We say that a com

The Turin one transition deterministic T standard Turin to manipulate will use Turing

**Example 8.1.1**

The tabular rep alphabet  $\{a, b\}$

The transition f transitions in s transitions in  $q_2$

A Turing n  $\delta(q_i, x) = [q_j, . state diagram$

represents the Tu

A machine head. At any step the tape is nonbla of the  $B$  are bla hand boundary to

set of states,  
represents a  
from  $Q \times \Gamma$   
state called

y to the right.  
on numbered

g the leftmost  
position one.  
ows the initial  
ides additional

l on the square  
e movement is  
the tape position

ion

ith  $y$ , and moved  
both directions  
indefinitely.

A computation halts when it encounters a state, symbol pair for which no transition is defined. A transition from tape position zero may specify a move to the left of the boundary of the tape. When this occurs, the computation is said to *terminate abnormally*. When we say that a computation halts, we mean that it terminates in a normal fashion.

The Turing machine presented in Definition 8.1.1 is deterministic, that is, at most one transition is specified for every combination of state and tape symbol. The one-tape deterministic Turing machine, with initial conditions as described above, is referred to as the **standard Turing machine**. The first two examples demonstrate the use of Turing machines to manipulate strings. After developing a facility with Turing machine computations, we will use Turing machines to accept languages and to compute functions.

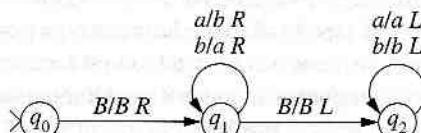
### Example 8.1.1

The tabular representation of the transition function of a standard Turing machine with input alphabet  $\{a, b\}$  is given in the table below.

| $\delta$ | $B$         | $a$         | $b$         |
|----------|-------------|-------------|-------------|
| $q_0$    | $q_1, B, R$ |             |             |
| $q_1$    | $q_2, B, L$ | $q_1, a, R$ | $q_1, a, R$ |
| $q_2$    |             | $q_2, a, L$ | $q_2, b, L$ |

The transition from state  $q_0$  moves the tape head to position one to read the input. The transitions in state  $q_1$  read the input string and interchange the symbols  $a$  and  $b$ . The transitions in  $q_2$  return the machine to the initial position.

A Turing machine can be graphically represented by a state diagram. The transition  $\delta(q_i, x) = [q_j, y, d]$ ,  $d \in \{L, R\}$  is depicted by an arc from  $q_i$  to  $q_j$  labeled  $x/y d$ . The state diagram



represents the Turing machine defined in the preceding transition table. □

A machine configuration consists of the state, the tape, and the position of the tape head. At any step in a computation of a standard Turing machine, only a finite segment of the tape is nonblank. A configuration is denoted  $uq_i vB$ , where all tape positions to the right of the  $B$  are blank and  $uv$  is the string spelled by the symbols on the tape from the left-hand boundary to the  $B$ . Blanks may occur in the string  $uv$ ; the only requirement is that the

entire nonblank portion of the tape be included in  $uv$ . The notation  $uq_i v B$  indicates that the machine is in state  $q_i$  scanning the first symbol of  $v$  and the entire tape to the right of  $uvB$  is blank.

This representation of machine configurations can be used to trace the computations of a Turing machine. The notation  $uq_i v B \xrightarrow{M} xq_j y B$  indicates that the configuration  $xq_j y B$  is obtained from  $uq_i v B$  by a single transition of  $M$ . Following the standard conventions,  $uq_i v B \xrightarrow{* M} xq_j y B$  signifies that  $xq_j y B$  can be obtained from  $uq_i v B$  by a finite number, possibly zero, of transitions. The reference to the machine is omitted when there is no possible ambiguity.

The Turing machine in Example 8.1.1 interchanges the  $a$ 's and  $b$ 's in the input string. Tracing the computation generated by the input string  $abab$  yields

$$\begin{aligned} & q_0 BababB \\ \xrightarrow{} & Bq_1 ababB \\ \xrightarrow{} & Bbq_1 babB \\ \xrightarrow{} & Bbaq_1 abB \\ \xrightarrow{} & Bbabq_1 bB \\ \xrightarrow{} & Bbabaq_1 B \\ \xrightarrow{} & Bbabq_2 aB \\ \xrightarrow{} & Bbaq_2 baB \\ \xrightarrow{} & Bbq_2 ababB \\ \xrightarrow{} & Bq_2 babaB \\ \xrightarrow{} & q_2 BbabaB. \end{aligned}$$

COPY: X

The computation symbol in the is copied. The first  $q_1$ . The cycle  $q$  constructed. Since the entire string  $a$ 's and  $b$ 's and return

## 8.2 Turing

Turing machine computation machine state and previous section. The computation to the result of a compute function the computation

In this section computation according state of the computer automata to accept need not read the final states is a s

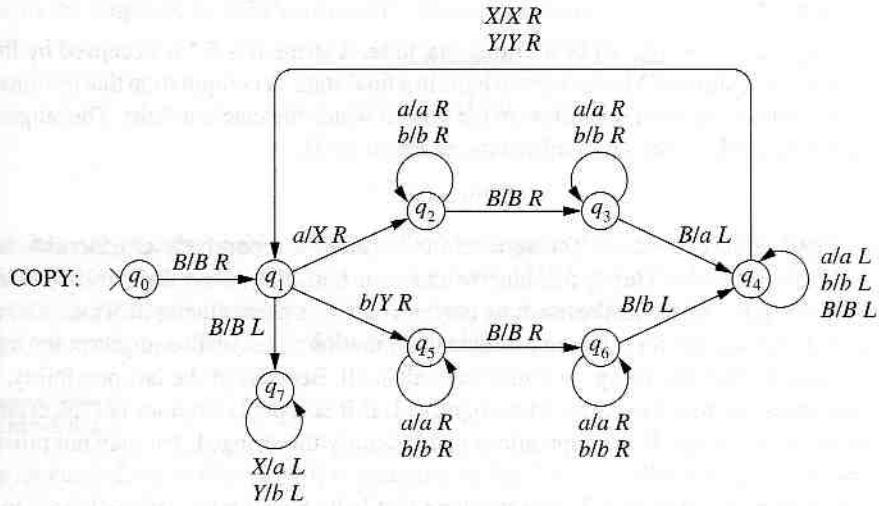
The Turing machine from Example 8.1.1 made two passes through the input string. Moving left to right, the first pass interchanged the  $a$ 's and  $b$ 's. The second pass, going right to left, simply returned the tape head to the leftmost tape position. The next example shows how Turing machine transitions can be used to make a copy of a string. The ability to copy data is an important component in many algorithmic processes. When copies are needed, the strategy employed by this machine can be modified to suit the type of data considered in the particular problem.

### Example 8.1.2

The Turing machine COPY with input alphabet  $\{a, b\}$  produces a copy of the input string. That is, a computation that begins with the tape having the form  $BuB$  terminates with tape  $BuBuB$ .

tes that the  
ght of  $uvB$

mputations  
ion  $xq_jyB$   
onventions,  
te number,  
there is no  
input string.



The computation copies the input string one symbol at a time beginning with the leftmost symbol in the input. Tape symbols  $X$  and  $Y$  record the portion of the input that has been copied. The first unmarked symbol in the string specifies the arc to be taken from state  $q_1$ . The cycle  $q_1, q_2, q_3, q_4, q_1$  replaces an  $a$  with  $X$  and adds an  $a$  to the string being constructed. Similarly, the lower branch copies a  $b$  using  $Y$  to mark the input string. After the entire string has been copied, the transitions in state  $q_7$  change the  $X$ 's and  $Y$ 's to  $a$ 's and  $b$ 's and return the tape head to the initial position.  $\square$

## 8.2 Turing Machines as Language Acceptors

input string.  
l pass, going  
ext example  
g. The ability  
en copies are  
type of data

and encoding  
algorithms.  
e input string.  
ates with tape

Turing machines have been introduced as a paradigm for effective computation. A Turing machine computation consists of a sequence of elementary operations determined from the machine state and the symbol being read by the tape head. The machines constructed in the previous section were designed to illustrate the features of Turing machine computations. The computations read and manipulated the symbols on the tape; no interpretation was given to the result of a computation. Turing machines can be designed to accept languages and to compute functions. The result of a computation can be defined in terms of the state in which the computation terminates or the configuration of the tape at the end of the computation.

In this section we consider the use of Turing machines as language acceptors; a computation accepts or rejects the input string. Initially, acceptance is defined by the final state of the computation. This is similar to the technique used by finite-state and pushdown automata to accept strings. Unlike finite-state and pushdown automata, a Turing machine need not read the entire input string to accept the string. A Turing machine augmented with final states is a sextuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$ , where  $F \subseteq Q$  is the set of final states.

**Definition 8.2.1**

Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a Turing machine. A string  $u \in \Sigma^*$  is **accepted by final state** if the computation of  $M$  with input  $u$  halts in a final state. A computation that terminates abnormally rejects the input regardless of the state in which the machine halts. The language of  $M$ , denoted  $L(M)$ , is the set of all strings accepted by  $M$ .

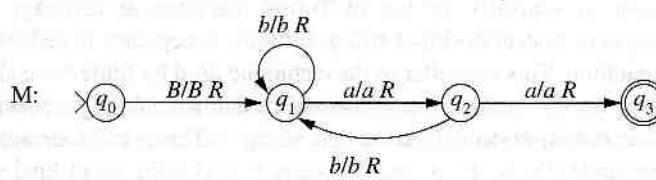
A language accepted by a Turing machine is called a **recursively enumerable language**. The ability of a Turing machine to move in both directions and process blanks introduces the possibility that the machine may not halt for a particular input. Thus there are three possible outcomes for a Turing machine computation: It may halt and accept the input string; halt and reject the string; or it may not halt at all. Because of the last possibility, we will sometimes say that a machine  $M$  *recognizes*  $L$  if it accepts  $L$  but does not necessarily halt for all input strings. The computations of  $M$  identify the strings  $L$  but may not provide answers for strings not in  $L$ .

A language accepted by a Turing machine that halts for all input strings is said to be **recursive**. Membership in a recursive language is decidable; the computations of a Turing machine that halts for all inputs provide a procedure for determining whether a string is in the language. A Turing machine of this type is sometimes said to *decide* the language. Being recursive is a property of a language, not of a Turing machine that accepts it. There are multiple Turing machines that accept a particular language; some may halt for all input, whereas others may not. The existence of one Turing machine that halts for all inputs is sufficient to show that the membership in the language is decidable and the language is recursive.

In Chapter 12 we will show that there are languages that are recognized by a Turing machine but are not decided by any Turing machine. It follows that the set of recursive languages is a proper subset of the recursively enumerable languages. The terms *recursive* and *recursively enumerable* have their origins in the functional interpretation of Turing computability that will be presented in Chapter 13.

**Example 8.2.1**

The Turing machine  $M$



accepts the language

examines only the first two symbols of the input string  $(a \cup b)^*aa(a \cup b)^*$ . If the computation halts upon reading the symbol  $a$ , the computation is successful.

**Example 8.2.2**

The language  $L = \{aa\}$  is accepted by the following Turing machine.

The tape symbols  $X, Y, Z$  are used to represent the blank tape. The computation successfully terminates when the machine halts in state  $q_3$  with the appropriate tape symbols.

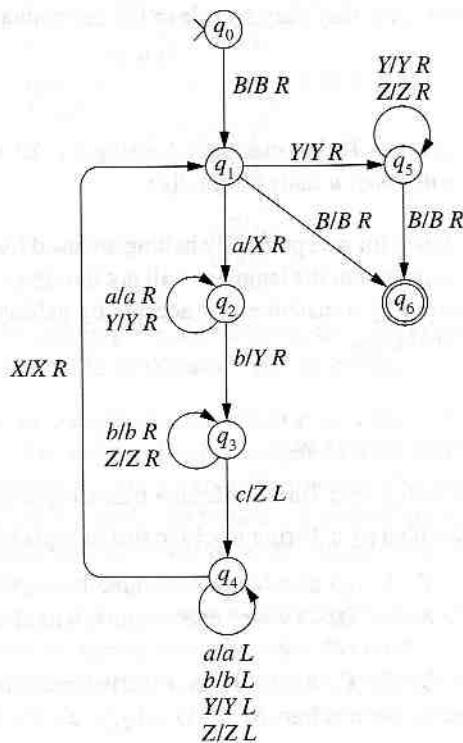
accepts the language  $(a \cup b)^*aa(a \cup b)^*$ . The computation

$$\begin{aligned} q_0 & BaabbB \\ \vdash & Bq_1aabbB \\ \vdash & Baq_2abbB \\ \vdash & Baaq_3bbB \end{aligned}$$

examines only the first half of the input before accepting the string  $aabb$ . The language  $(a \cup b)^*aa(a \cup b)^*$  is recursive; the computations of  $M$  halt for every input string. A successful computation terminates when a substring  $aa$  is encountered. All other computations halt upon reading the first blank following the input.  $\square$

### Example 8.2.2

The language  $L = \{a^i b^i c^i \mid i \geq 0\}$  is accepted by the Turing machine



The tape symbols  $X$ ,  $Y$ , and  $Z$  mark the  $a$ 's,  $b$ 's, and  $c$ 's as they are matched. A computation successfully terminates when all the symbols in the input string have been transformed to the appropriate tape symbol. The transition from  $q_1$  to  $q_6$  accepts the null string.

The Turing machine  $M$  shows that  $L$  is recursive. The computations for strings in  $L$  halt in  $q_6$ . For strings not in  $L$ , the computations halt in a nonaccepting state as soon as it is discovered that the input string does not match the pattern  $a^i b^j c^l$ . For example, the computation with input  $bca$  halts in  $q_1$  and with input  $abb$  in  $q_3$ .  $\square$

### 8.3 Alternative Acceptance Criteria

Using Definition 8.2.1, the acceptance of a string by a Turing machine is determined by the state of the machine when the computation halts. Alternative approaches to defining acceptance are presented in this section.

The first alternative is acceptance by halting. In a Turing machine that is designed to accept by halting, an input string is accepted if the computation initiated with the string halts. Computations for which the machine terminates abnormally reject the string. When acceptance is defined by halting, the machine is defined by the quintuple  $(Q, \Sigma, \Gamma, \delta, q_0)$ . The final states are omitted since they play no role in the determination of the language of the machine.

#### Definition 8.3.1

Let  $M = (Q, \Sigma, \Gamma, \delta, q_0)$  be a Turing machine. A string  $u \in \Sigma^*$  is **accepted by halting** if the computation of  $M$  with input  $u$  halts (normally).

Turing machines designed for acceptance by halting are used for language recognition. The computation for any input not in the language will not terminate. Theorem 8.3.2 shows that any language recognized by a machine that accepts by halting is also accepted by a machine that accepts by final state.

#### Theorem 8.3.2

The following statements are equivalent:

- i) The language  $L$  is accepted by a Turing machine that accepts by final state.
- ii) The language  $L$  is accepted by a Turing machine that accepts by halting.

**Proof.** Let  $M = (Q, \Sigma, \Gamma, \delta, q_0)$  be a Turing machine that accepts  $L$  by halting. The machine  $M' = (Q, \Sigma, \Gamma, \delta, q_0, F)$ , in which every state is a final state, accepts  $L$  by final state.

Conversely, let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a Turing machine that accepts the language  $L$  by final state. Define the machine  $M' = (Q \cup \{q_e\}, \Sigma, \Gamma, \delta', q_0)$  that accepts by halting as follows:

- i) If  $\delta(q_i, x)$  is defined, then  $\delta'(q_i, x) = \delta(q_i, x)$ .
- ii) For each state  $q_i \in Q - F$ , if  $\delta(q_i, x)$  is undefined, then  $\delta'(q_i, x) = [q_e, x, R]$ .
- iii) For each  $x \in \Gamma$ ,  $\delta'(q_e, x) = [q_e, x, R]$ .

Computations in  $M$  may be unsuccessful. The states entered by  $M'$  are those in  $L(M') = L$ .

#### Example 8.3.1

The Turing machine  $M$  in Example 8.2.1 accepts by halting. The machine  $M'$  in Example 8.3.1 accepts by final state.

The machine  $M'$  accepts  $L$  by final state. It also accepts  $a/a$  by final state.

In Exercise 8.3.1, you are asked to prove that any language accepted by a Turing machine that uses final states can also be accepted by a Turing machine that accepts by halting.

Unless otherwise specified, the alternative acceptance criteria in this section apply to all machines in this chapter.

The alternative acceptance criteria in this section apply to all machines in this chapter.

#### 8.4 More on Languages

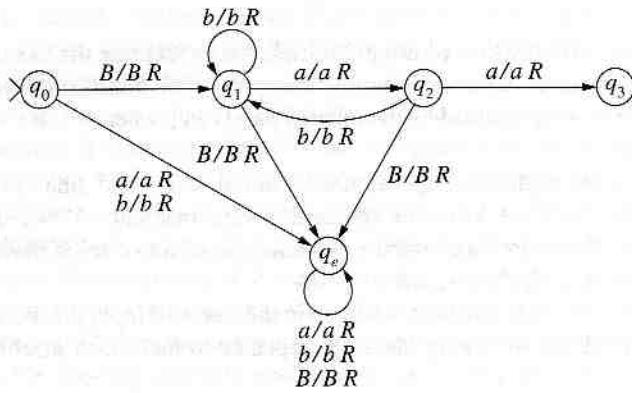
The remainder of this chapter is devoted to the study of more complex languages and machines.

ings in L  
; soon as  
nple, the  
□

Computations that accept strings in M and M' are identical. An unsuccessful computation in M may halt in a rejecting state, terminate abnormally, or fail to terminate. When an unsuccessful computation in M halts, the computation in M' enters the state  $q_e$ . Upon entering  $q_e$ , the machine moves indefinitely to the right. The only computations that halt in M' are those that are generated by computations of M that halt in an accepting state. Thus  $L(M') = L(M)$ . ■

### Example 8.3.1

The Turing machine from Example 8.2.1 is altered to accept  $(a \cup b)^*aa(a \cup b)^*$  by halting. The machine below is constructed as specified by Theorem 8.3.2. A computation enters  $q_e$  when the entire input string has been read and no aa has been encountered.



The machine obtained by deleting the arcs from  $q_0$  to  $q_e$  and those from  $q_e$  to  $q_e$  labeled  $a/a R$  and  $b/b R$  also accepts  $(a \cup b)^*aa(a \cup b)^*$  by halting. □

In Exercise 7 a type of acceptance, referred to as *acceptance by entering*, is introduced that uses final states but does not require the accepting computations to terminate. A string is accepted if the computation ever enters a final state; after entering a final state, the remainder of the computation is irrelevant to the acceptance of the string. As with acceptance by halting, any Turing machine designed to accept by entering can be transformed into a machine that accepts the same language by final state.

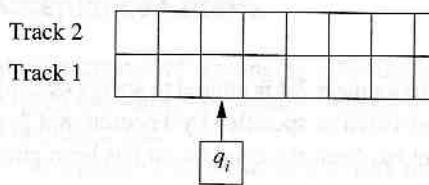
Unless noted otherwise, Turing machines will accept by final state as in Definition 8.2.1. The alternative definitions are equivalent in the sense that machines designed in this manner accept the same family of languages as those accepted by standard Turing machines.

## 8.4 Multitrack Machines

The remainder of the chapter is dedicated to examining variations of the standard Turing machine model. Each of the variations appears to increase the capability of the machine.

We prove that the languages accepted by these generalized machines are precisely those accepted by the standard Turing machines. Additional variations will be presented in the exercises.

A multitrack tape is one in which the tape is divided into tracks. A tape position in an  $n$ -track tape contains  $n$  symbols from the tape alphabet. The diagram depicts a two-track tape with the tape head scanning the second position.



The machine reads an entire tape position. Multiple tracks increase the amount of information that can be considered when determining the appropriate transition. A tape position in a two-track machine is represented by the ordered pair  $[x, y]$ , where  $x$  is the symbol in track 1 and  $y$  is in track 2.

The states, input alphabet, tape alphabet, initial state, and final states of a two-track machine are the same as in the standard Turing machine. A two-track transition reads and rewrites the entire tape position. A transition of a two-track machine is written  $\delta(q_i, [x, y]) = [q_j, [z, w], d]$ , where  $d \in \{L, R\}$ .

The input to a two-track machine is placed in the standard input position in track 1. All the positions in track 2 are initially blank. Acceptance in multitrack machines is by final state.

We will show that the languages accepted by two-track machines are precisely the recursively enumerable languages. The argument easily generalizes to  $n$ -track machines.

#### Theorem 8.4.1

A language  $L$  is accepted by a two-track Turing machine if, and only if, it is accepted by a standard Turing machine.

**Proof.** Clearly, if  $L$  is accepted by a standard Turing machine, it is accepted by a two-track machine. The equivalent two-track machine simply ignores the presence of the second track.

Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a two-track machine. A one-track machine will be constructed in which a single tape square contains the same information as a tape position in the two-track tape. The representation of a two-track tape position as an ordered pair indicates how this can be accomplished. The tape alphabet of the equivalent one-track machine  $M'$  consists of ordered pairs of tape elements of  $M$ . The input to the two-track machine consists of ordered pairs whose second component is blank. The input symbol  $a$  of  $M$  is identified with the ordered pair  $[a, B]$  of  $M'$ . The one-track machine

$$M' = (Q, \Sigma \times \{B\}, \Gamma \times \Gamma, \delta', q_0, F)$$

with transition func

accepts  $L(M)$ .

#### 8.5 Two-Way

A Turing machine v  
tape extends indefini  
input can be placed a  
The tape head is init  
The advantage of a tw  
crossing the left bound

A machine with  
machine by placing a  
way tape. The symb  
the standard machine  
equivalent machine w  
tape head position. T  
to that of the one-wa  
terminates abnormall  
tape boundary, the tw  
that terminates the co

The standard Tur

$$M: \quad \begin{array}{c} \times \\ q_0 \end{array} \quad \overbrace{\quad \quad \quad}^{B/B} \quad R$$

will be used to demons  
two-way machine. All  
encountered, the tape  
completely determined  
the tape head attempts  
the bounds of the tape a  
 $\{a, b\}$  in which the first

A machine  $M'$  with  
states  $q_s$ ,  $q_t$ , and  $q_e$ . Th  
to the left of the initial  
 $L(M)$ . After writing the

isely those  
nted in the  
sition in an  
a two-track

t of informa-  
e position in  
mbol in track

es of a two-  
ck transition  
ine is written

n track 1. All  
es is by final

precisely the  
k machines.

accepted by a

by a two-track  
e second track.  
achine will be  
a tape position  
n ordered pair  
alent one-track  
o the two-track  
input symbol  $a$   
e

with transition function

$$\delta'(q_i, [x, y]) = \delta(q_i, [x, y])$$

accepts  $L(M)$ . ■

## 8.5 Two-Way Tape Machines

A Turing machine with a two-way tape is identical to the standard model except that the tape extends indefinitely in both directions. Since a two-way tape has no left boundary, the input can be placed anywhere on the tape. All other tape positions are assumed to be blank. The tape head is initially positioned on the blank to the immediate left of the input string. The advantage of a two-way tape is that the Turing machine designer need not worry about crossing the left boundary of the tape.

A machine with a two-way tape can be constructed to simulate the actions of a standard machine by placing a special symbol on the tape to represent the left boundary of the one-way tape. The symbol #, which is assumed not to be an element of the tape alphabet of the standard machine, is used to simulate the boundary of the tape. A computation in the equivalent machine with two-way tape begins by writing # to the immediate left of the initial tape head position. The remainder of a computation in the two-way machine is identical to that of the one-way machine except when the computation of the one-way machine terminates abnormally. When the one-way computation attempts to move to the left of the tape boundary, the two-way machine reads the symbol # and enters a nonaccepting state that terminates the computation.

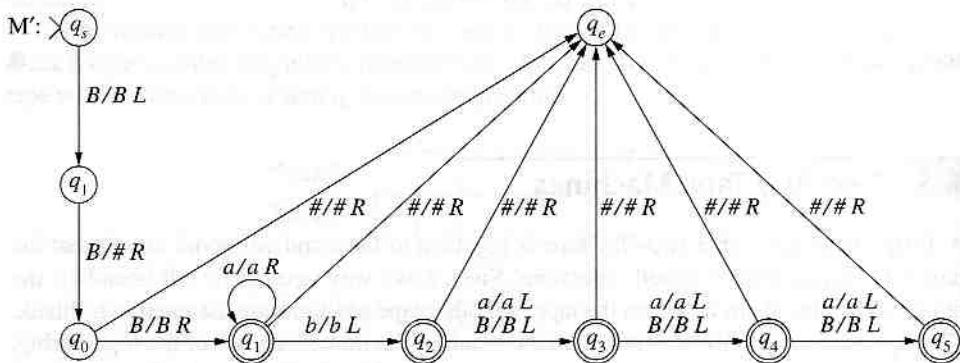
The standard Turing machine  $M$



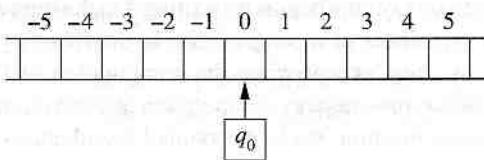
will be used to demonstrate the conversion of a machine with a one-way tape to an equivalent two-way machine. All the states of  $M$  other than  $q_0$  are accepting. When the first  $b$  is encountered, the tape head moves four positions to the left, if possible. Acceptance is completely determined by the boundary of the tape. A string is rejected by  $M$  whenever the tape head attempts to cross the left-hand boundary. All computations that remain within the bounds of the tape accept the input. Thus the language of  $M$  consists of all strings over  $\{a, b\}$  in which the first  $b$ , if present, is preceded by at least three  $a$ 's.

A machine  $M'$  with a two-way tape can be obtained from  $M$  by the addition of three states  $q_s$ ,  $q_t$ , and  $q_e$ . The transitions from states  $q_s$  and  $q_t$  insert the simulated endmarker to the left of the initial position of the tape head of  $M'$ , the two-way machine that accepts  $L(M)$ . After writing the simulated boundary, the computation enters a copy of the one-way

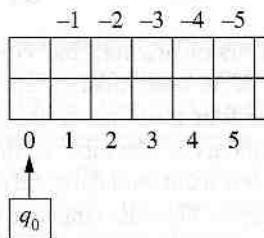
machine  $M$ . The error state  $q_e$  is entered in  $M'$  when a computation in  $M$  attempts to move to the left of the tape boundary.



We will now show that a language accepted by a machine with a two-way tape is accepted by a standard Turing machine. The argument utilizes Theorem 8.4.1, which establishes the interdefinability of two-track and standard machines. The tape positions of the two-way tape can be numbered by the complete set of integers. The initial position of the tape head is numbered zero, and the input begins at position one.



Imagine taking the two-way infinite tape and folding it so that position  $-i$  sits directly above position  $i$ . Adding an unnumbered tape square over position zero produces a two-track tape. The symbol in tape position  $i$  of the two-way tape is stored in the corresponding position of the one-way, two-track tape. A computation on a two-way infinite tape can be simulated on this one-way, two-track tape.



Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a Turing machine with a two-way tape. Using the correspondence between a two-way tape and a two-track tape, we construct a Turing machine  $M'$  with a two-track, one-way tape to accept  $L(M)$ . A transition of  $M$  is specified by the state and the symbol scanned.  $M'$ , scanning a two-track tape, reads two symbols at each

tape position  
which of the  
The co

The initial s  
the upper tra

A trans  
simulation c  
is used to in  
U to D in th

1.  $\delta'([q_s, L])$
2. For every  $i \in \mathbb{Z}$
3. For every  $i \in \mathbb{Z}$  ever  $\delta(q_i, U)$
4. For every  $i \in \mathbb{Z}$  ever  $\delta(q_i, D)$
5.  $\delta'([q_i, L])$  of  $M$ .
6.  $\delta'([q_i, L])$  of  $M$ .
7.  $\delta'([q_i, U])$  of  $M$ .
8.  $\delta'([q_i, D])$  of  $M$ .

A transit  
begins and e  
represented b  
only the upper  
to the left of

The rema  
way tape. Re  
by the tape sy  
by schema 5

The prec  
two-way Tur

s to move

tape position. Symbols  $U$  (up) and  $D$  (down) are included in the states of  $M'$  to designate which of the two tracks should be used to determine the transition.

The components of  $M'$  are constructed from those of  $M$  and the symbols  $U$  and  $D$ :

$$Q' = (Q \cup \{q_s, q_t\}) \times \{U, D\}$$

$$\Sigma' = \Sigma$$

$$\Gamma' = \Gamma \cup \{\#\}$$

$$F' = \{[q_i, U], [q_i, D] \mid q_i \in F\}.$$

The initial state of  $M'$  is a pair  $[q_s, D]$ . The transition from this state writes the marker  $\#$  on the upper track in the leftmost tape position.

A transition from  $[q_t, D]$  returns the tape head to its original position to begin the simulation of a computation of  $M$ . During the remainder of a computation, the  $\#$  on track 2 is used to indicate when the tape head is reading position zero and to trigger changes from  $U$  to  $D$  in the state. The transitions of  $M'$  are defined as follows:

1.  $\delta'([q_s, D], [B, B]) = [[q_t, D], [B, \#], R]$ .
2. For every  $x \in \Gamma$ ,  $\delta'([q_t, D], [x, B]) = [[q_0, D], [x, B], L]$ .
3. For every  $z \in \Gamma - \{\#\}$  and  $d \in \{L, R\}$ ,  $\delta'([q_i, D], [x, z]) = [[q_j, D], [y, z], d]$  whenever  $\delta(q_i, x) = [q_j, y, d]$  is a transition of  $M$ .
4. For every  $x \in \Gamma - \{\#\}$  and  $d \in \{L, R\}$ ,  $\delta'([q_i, U], [z, x]) = [[q_j, U], [z, y], d']$  whenever  $\delta(q_i, x) = [q_j, y, d]$  is a transition of  $M$ , where  $d'$  is the opposite direction of  $d$ .
5.  $\delta'([q_i, D], [x, \#]) = [[q_j, U], [y, \#], R]$  whenever  $\delta(q_i, x) = [q_j, y, L]$  is a transition of  $M$ .
6.  $\delta'([q_i, D], [x, \#]) = [[q_j, D], [y, \#], R]$  whenever  $\delta(q_i, x) = [q_j, y, R]$  is a transition of  $M$ .
7.  $\delta'([q_i, U], [x, \#]) = [[q_j, D], [y, \#], R]$  whenever  $\delta(q_i, x) = [q_j, y, R]$  is a transition of  $M$ .
8.  $\delta'([q_i, U], [x, \#]) = [[q_j, U], [y, \#], R]$  whenever  $\delta(q_i, x) = [q_j, y, L]$  is a transition of  $M$ .

A transition generated by schema 3 simulates a transition of  $M$  in which the tape head begins and ends in positions labeled with nonnegative values. In the simulation, this is represented by writing on the lower track of the tape. Transitions defined in schema 4 use only the upper track of the two-track tape. These correspond to transitions of  $M$  that occur to the left of position zero on the two-way infinite tape.

The remaining transitions simulate the transitions of  $M$  from position zero on the two-way tape. Regardless of the  $U$  or  $D$  in the state, transitions from position zero are determined by the tape symbol on track 1. When the track is specified by  $D$ , the transition is defined by schema 5 or 6. Transitions defined in 7 and 8 are applied when the state is  $[q_i, U]$ .

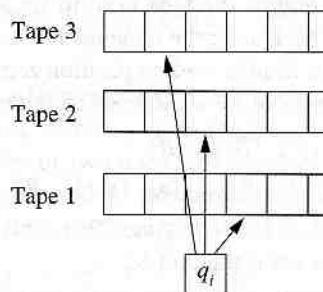
The preceding informal arguments outline the proof of the equivalence of one-way and two-way Turing machines.

### Theorem 8.5.1

A language L is accepted by a Turing machine with a two-way tape if, and only if, it is accepted by a standard Turing machine.

## 8.6 Multitape Machines

A  $k$ -tape machine has  $k$  tapes and  $k$  independent tape heads. The states and alphabets of a multitape machine are the same as in a standard Turing machine. The machine reads the tapes simultaneously but has only one state. This is depicted by connecting each of the independent tape heads to a single control indicating the current state.



A transition is determined by the state and the symbols scanned by each of the tape heads. A transition in a multitape machine may

- i) change the state,
  - ii) write a symbol on each of the tapes,
  - iii) independently reposition each of the tape heads.

The repositioning consists of moving the tape head one square to the left or one square to the right or leaving it at its current position. A transition of a two-tape machine scanning  $x_1$  on tape 1 and  $x_2$  on tape 2 is written  $\delta(q_i, x_1, x_2) = [q_j; y_1, d_1; y_2, d_2]$ , where  $x_i, y_i \in \Gamma$  and  $d_i \in \{L, R, S\}$ . This transition causes the machine to write  $y_i$  on tape  $i$ . The symbol  $d_i$  specifies the direction of the movement of tape head  $i$ :  $L$  signifies a move to the left,  $R$  a move to the right, and  $S$  means the head remains stationary. Any tape head attempting to move to the left of the boundary of its tape terminates the computation abnormally.

The input to a multtape machine is placed in the standard position on tape 1. All the other tapes are assumed to be blank. The tape heads originally scan the leftmost position of each tape. A multtape machine can be represented by a state diagram in which the label on an arc specifies the action for each tape. For example, the transition  $\delta(q_i, x_1, x_2) = [q_j; y_1, d_1; y_2, d_2]$  will be represented by an arc from  $q_i$  to  $q_j$  labeled  $[x_1/y_1 \ d_1, x_2/y_2 \ d_2]$ .

Two advantages of multtape machines are the ability to copy data between tapes and to compare strings on different tapes. Both of these features will be demonstrated in the following example.

### Example 8.6.1

### The machine

accepts the language leading  $a$ 's to tape  $q_2$  to compare them to precede and follow for strings without  $a$  for strings with  $a$  in  $q_2$ . Since every string  $[a^i b a^i \mid i \geq 0]$  appears

A standard consequence, every recent result shows that the complexity of a five-track machine can be simulated by a  $k$ -tape machine, and vice versa, by multitrack and

### Theorem 8.6.1

A language L is a standard Turing machine.

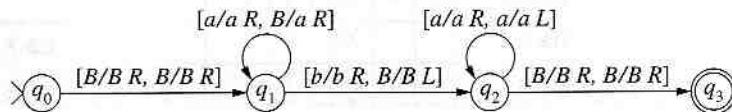
Let  $M = (Q,$   
heads of a multit

The single tape  $H_4$  is a five-track machine which maintains the information in  $H_4$  have a single node. The five-track machine.

**Example 8.6.1**

if, it is

The machine



bets of a  
eads the  
ch of the

accepts the language  $\{a^i b a^i \mid i \geq 0\}$ . A computation with input string  $a^i b a^i$  copies the leading  $a$ 's to tape 2 in state  $q_1$ . When the  $b$  is read on tape 1, the computation enters state  $q_2$  to compare the  $a$ 's on tape 2 with the  $a$ 's after the  $b$  on tape 1. If the same number of  $a$ 's precede and follow the  $b$ , the computation halts in  $q_3$  and accepts the input. The computations for strings without a  $b$  halt in  $q_1$  and strings with more than one  $b$  in  $q_2$ . The computations for strings with one  $b$  and an unequal number of leading and trailing  $a$ 's also halt in  $q_2$ . Since every computation halts,  $M$  provides a decision procedure for membership in  $\{a^i b a^i \mid i \geq 0\}$  and consequently the language is recursive.  $\square$

f the tape

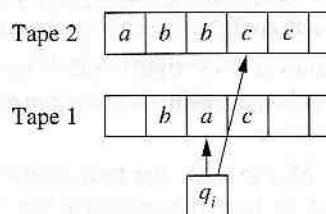
A standard Turing machine is a multitape Turing machine with a single tape. Consequently, every recursively enumerable language is accepted by a multitape machine. We will show that the computations of a two-tape machine can be simulated by computations of a five-track machine. The argument can be generalized to show that any language accepted by a  $k$ -tape machine is accepted by a  $2k + 1$ -track machine. The equivalence of acceptance by multitrack and standard machines then allows us to conclude the following.

**Theorem 8.6.1**

A language  $L$  is accepted by a multitape Turing machine if, and only if, it is accepted by a standard Turing machine.

square to  
scanning  
 $x_i, y_i \in \Gamma$   
symbol  $d_i$   
left,  $R$  a  
empting to  
ally.  
1. All the  
st position  
which the  
 $(x_1, x_2) =$   
 $x_2/y_2 d_2$ .  
n tapes and  
rated in the

Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a two-tape machine. During a computation, the tape heads of a multitape machine are independently positioned on the two tapes.



The single tape head of a multitrack machine reads all the tracks of a fixed position. The five-track machine  $M'$  is constructed to simulate the computations of  $M$ . Tracks 1 and 3 maintain the information stored on tapes 1 and 2 of the two-tape machine. Tracks 2 and 4 have a single nonblank square indicating the position of the tape heads of the multitape machine.

|         |   |   |   |   |   |
|---------|---|---|---|---|---|
| Track 5 | # |   |   |   |   |
| Track 4 |   |   | X |   |   |
| Track 3 | a | b | b | c | c |
| Track 2 |   |   | X |   |   |
| Track 1 |   | b | a | c |   |

The initial action of the simulation in the multitrack machine is to write # in the leftmost position of track 5 and X in the leftmost positions of tracks 2 and 4. The remainder of the computation of the multitrack machine consists of a sequence of actions that simulate the transitions of the two-tape machine.

A transition of the two-tape machine is determined by the two symbols being scanned and the machine state. The simulation in the five-track machine records the symbols marked by each of the X's. The states are 8-tuples of the form  $[s, q_i, x_1, x_2, y_1, y_2, d_1, d_2]$ , where  $q_i \in Q$ ;  $x_i, y_i \in \Sigma \cup \{U\}$ ; and  $d_i \in \{L, R, S, U\}$ . The element  $s$  represents the status of the simulation of the transition of M. The symbol  $U$ , added to the tape alphabet and the set of directions, indicates that this item is unknown.

Let  $\delta(q_i, x_1, x_2) = [q_j; y_1, d_1; y_2, d_2]$  be the applicable two-tape transition of M. M' begins the simulation of the transition in the state  $[f1, q_i, U, U, U, U, U, U]$ . The following five actions simulate the transition of M in the multitrack machine.

1.  $f1$  (find first symbol): M' moves to the right until it reads the X on track 2. State  $[f1, q_i, x_1, U, U, U, U, U]$  is entered, where  $x_1$  is the symbol in track 1 under the X. After recording the symbol on track 1 in the state, M' returns to the initial position. The # on track 5 is used to reposition the tape head.
2.  $f2$  (find second symbol): The same sequence of actions records the symbol beneath the X on track 4. M' enters state  $[f2, q_i, x_1, x_2, U, U, U, U]$ , where  $x_2$  is the symbol in track 3 under the X. The tape head is then returned to the initial position.
3. M' enters the state  $[p1, q_j, x_1, x_2, y_1, y_2, d_1, d_2]$ , where the values  $q_j, y_1, y_2, d_1$ , and  $d_2$  are obtained from the transition  $\delta(q_i, x_1, x_2)$ . This state contains the information needed to simulate the transition of the M.
4.  $p1$  (print first symbol): M' moves to the right to the X in track 2 and writes the symbol  $y_1$  on track 1. The X on track 2 is moved in the direction designated by  $d_1$ . The machine then returns to the initial position.
5.  $p2$  (print second symbol): M' moves to the right to the X in track 4 and writes the symbol  $y_2$  on track 3. The X on track 4 is moved in the direction designated by  $d_2$ .
6. The simulation of the transition  $\delta(q_i, x_1, x_2) = [q_j; y_1, d_1; y_2, d_2]$  terminates by returning the tape head to the initial position to process the subsequent transition.

If  $\delta(q_i, x_1, x_2)$  is undefined in the two-tape machine, the simulation halts after returning to the initial position following step 2. A state  $[f2, q_i, x_1, y_1, U, U, U, U]$  is an accepting state of the multitrack machine M' whenever  $q_i$  is an accepting state of M.

The next two  
data in a computa

### Example 8.6.2

The set  $\{a^k \mid k \in \mathbb{N}\}$  is accepted by a three-tape machine. The input is composed of three tapes. The first tape holds a string whose length is  $k$ . The second tape holds a string of  $k$  X's. The third tape holds a string of  $k$  U's. The machine starts with all three tapes empty. It then performs the following steps:

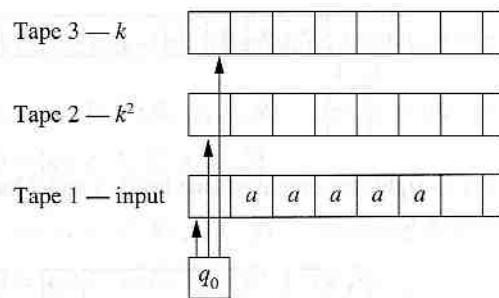
The values of  $k$  are those for which the simulation of the following actions terminates.

1. If the input is the empty string, then 0 and 3 are initial states. The heads on tape 1 and 3 are positioned at the start of the tape, and the head on tape 2 is positioned at the end of the tape.
2. Tape 3 now contains the string  $a^k$ . The heads on tape 1 and 3 are positioned at the start of the tape, and the head on tape 2 is positioned at the end of the tape.
3. If neither of the heads on tape 1 or 3 has reached the end of the tape, then the simulation continues. Otherwise, the simulation halts.
- a) If both heads have reached the end of the tape, then the string is accepted.
- b) If tape head 1 has reached the end of the tape, then the string is rejected.

The next two examples illustrate the use of the additional tapes to store and manipulate data in a computation.

### Example 8.6.2

The set  $\{a^k \mid k \text{ is a perfect square}\}$  is a recursively enumerable language. The design of a three-tape machine that accepts this language is presented. Tape 1 contains the input string. The input is compared with a string of  $X$ 's on tape 2 whose length is a perfect square. Tape 3 holds a string whose length is the square root of the string on tape 2. The initial configuration for a computation with input  $aaaaaa$  is

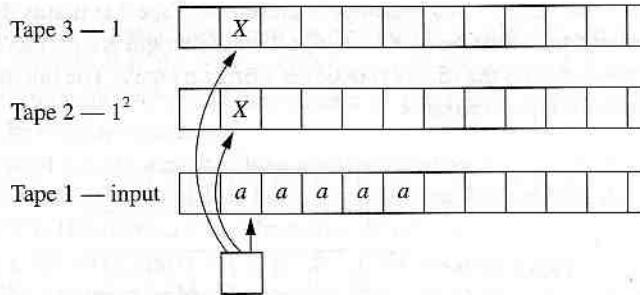


The values of  $k$  and  $k^2$  are incremented until the length of the string on tape 2 is greater than or equal to the length of the input. A machine to perform these comparisons consists of the following actions.

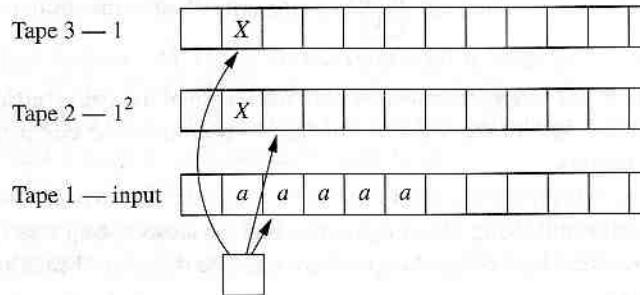
1. If the input is the null string, the computation halts in an accepting state. If not, tapes 2 and 3 are initialized by writing  $X$  in position one. The three tape heads are then moved to position one.
2. Tape 3 now contains a sequence of  $k$   $X$ 's and tape 2 contains  $k^2$   $X$ 's. Simultaneously, the heads on tapes 1 and 2 move to the right while both heads scan nonblank squares. The head reading tape 3 remains at position one.
  - a) If both heads simultaneously read a blank, the computation halts and the string is accepted.
  - b) If tape head 1 reads a blank and tape head 2 an  $X$ , the computation halts and the string is rejected.
3. If neither of the halting conditions occur, the tapes are reconfigured for comparison with the next perfect square.
  - a) An  $X$  is added to the right end of the string of  $X$ 's on tape 2.
  - b) Two copies of the string on tape 3 are added to the right end of the string on tape 2. This constructs a sequence of  $(k + 1)^2$   $X$ 's on tape 2.

- c) An  $X$  is added to the right end of the string of  $X$ 's on tape 3. This constructs a sequence of  $k + 1$   $X$ 's on tape 3.
- d) The tape heads are then repositioned at position one of their respective tapes.
4. The computation continues with step 2.

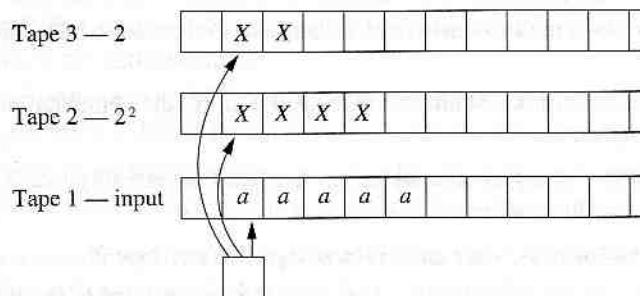
Tracing the computation for the input string  $aaaaaa$ , step 1 produces the configuration



The simultaneous left-to-right movement of tape heads 1 and 2 halts when tape head 2 scans the blank in position two.



Part (c) of step 3 reformats tapes 2 and 3 so that the input string can be compared with the next perfect square.



Another it

A machine that

$\delta(q_0, B, L)$

$\delta(q_1, a, L)$

$\delta(q_2, a, X)$

$\delta(q_2, B, L)$

$\delta(q_2, a, E)$

$\delta(q_4, a, E)$

$\delta(q_4, a, B)$

$\delta(q_5, a, B)$

$\delta(q_6, a, X)$

$\delta(q_6, a, X)$

$\delta(q_6, a, B)$

$\delta(q_6, B, X)$

$\delta(q_6, B, B)$

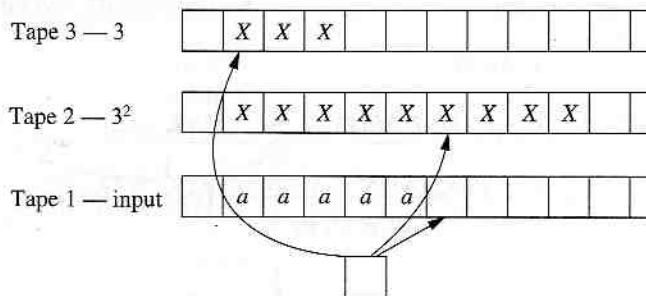
The accepting state  
is a perfect square

Since the machine  
language  $\{a^k \mid k \in \mathbb{N}\}$

constructs a  
tapes.

configuration

Another iteration of step 2 halts and rejects the input.



A machine that performs the preceding computation is defined by the following transitions:

$$\delta(q_0, B, B, B) = [q_1; B, R; B, R; B, R] \quad (\text{initialize the tape})$$

$$\delta(q_1, a, B, B) = [q_2; a, S; X, S; X, S]$$

$$\delta(q_2, a, X, X) = [q_2; a, R; X, R; X, S] \quad (\text{compare strings on tapes 1 and 2})$$

$$\delta(q_2, B, B, X) = [q_3; B, S; B, S; X, S] \quad (\text{accept})$$

$$\delta(q_2, a, B, X) = [q_4; a, S; X, R; X, S]$$

$$\delta(q_4, a, B, X) = [q_5; a, S; X, R; X, S] \quad (\text{rewrite tapes 2 and 3})$$

$$\delta(q_4, a, B, B) = [q_6; a, L; B, L; X, L]$$

$$\delta(q_5, a, B, X) = [q_4; a, S; X, R; X, R]$$

$$\delta(q_6, a, X, X) = [q_6; a, L; X, L; X, L] \quad (\text{reposition tape heads})$$

$$\delta(q_6, a, X, B) = [q_6; a, L; X, L; B, S]$$

$$\delta(q_6, a, B, B) = [q_6; a, L; B, S; B, S]$$

$$\delta(q_6, B, X, B) = [q_6; B, S; X, L; B, S]$$

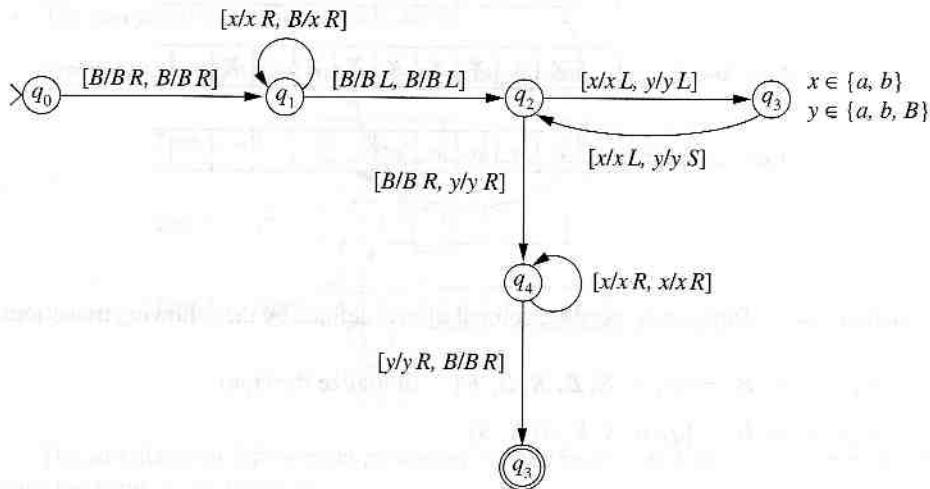
$$\delta(q_6, B, B, B) = [q_2; B, R; B, R; B, R]. \quad (\text{repeat comparison cycle})$$

The accepting states are  $q_1$  and  $q_3$ . The null string is accepted in  $q_1$ , and strings  $a^k$ , where  $k$  is a perfect square greater than zero, are accepted in  $q_3$ .

Since the machine designed above halts for all input strings, we have shown that the language  $\{a^k \mid k \text{ is a perfect square}\}$  is not only recursively enumerable but also recursive.  $\square$

**Example 8.6.3**

The two-tape Turing machine



accepts the language  $\{uu \mid u \in \{a, b\}^*\}$ . The symbols  $x$  and  $y$  on the labels of the arcs represent an arbitrary input symbol.

The computation begins by making a copy of the input on tape 2. When this is complete, both tape heads are to the immediate right of the input. The tape heads now move back to the left, with tape head 1 moving two squares for every one square that tape head 2 moves. If the computation halts in  $q_3$ , the input string has odd length and is rejected. The loop in  $q_4$  compares the first half of the input with the second; if they match, the string is accepted in state  $q_5$ .  $\square$

## 8.7 Nondeterministic Turing Machines

A nondeterministic Turing machine may specify any finite number of transitions for a given configuration. The components of a nondeterministic machine, with the exception of the transition function, are identical to those of the standard Turing machine. Transitions in a nondeterministic machine are defined by a function from  $Q \times \Gamma$  to subsets of  $Q \times \Gamma \times \{L, R\}$ .

Whenever the transition function indicates that more than one action is possible, a computation arbitrarily chooses one of the transitions. An input string is accepted by a nondeterministic machine if there is at least one computation that terminates in an accepting state. The existence of other computations that halt in nonaccepting states or fail to halt altogether is irrelevant. As usual, the language of a machine is the set of strings accepted by the machine.

**Example 8.7.1**

The nondeterm

accepts string  
in state  $q_1$  until  
 $q_1$ , enter state  
preceded by  $a$   
chooses one o

The machine  
standard machine  
state or by halting  
is at least one of  
these alternatives

Nondeterm  
guages accepte  
machines. To a  
equivalent deter  
string can be se

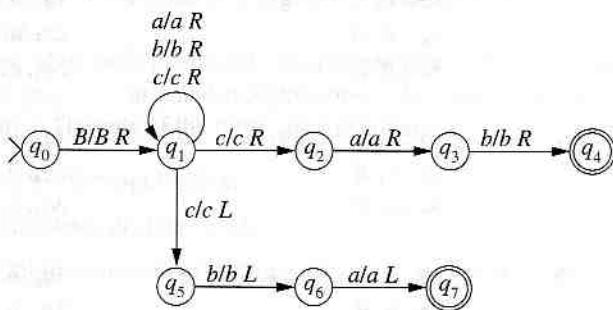
A nondeterm  
input string. The  
transitions for a  
any combination  
not necessarily  
If the transition  
numbers to com

A sequence  
unique computa  
sequence consist  
tape symbol sca  
leaves the machi  
the machine exec

**Example 8.7.1**

The nondeterministic Turing machine

$$\begin{aligned} \Sigma &= \{a, b\} \\ \Gamma &= \{a, b, B\} \end{aligned}$$



accepts strings containing a *c* preceded or followed by *ab*. The machine processes the input in state  $q_1$  until a *c* is encountered. When this occurs, the computation may continue in state  $q_1$ , enter state  $q_2$  to determine if the *c* is followed by *ab*, or enter  $q_5$  to determine if the *c* is preceded by *ab*. In the language of nondeterminism, the computation chooses a *c* and then chooses one of the conditions to check.  $\square$

s of the arcs  
is complete,  
move back to  
end 2 moves.  
The loop in  
g is accepted  
 $\square$

ons for a given  
ception of the  
Transitions in  
ts of  $Q \times \Gamma \times$

is possible, a  
accepted by a  
in an accepting  
or fail to halt  
rings accepted

The machine constructed in Example 8.7.1 accepts strings by final state. As with standard machines, acceptance in nondeterministic Turing machines can be defined by final state or by halting alone. A nondeterministic machine accepts a string *u* by halting if there is at least one computation that halts normally when run with *u*. Exercise 24 establishes that these alternative approaches accept the same languages.

Nondeterminism does not increase the capabilities of Turing computation; the languages accepted by nondeterministic machines are precisely those accepted by deterministic machines. To accomplish the transformation of a nondeterministic Turing machine to an equivalent deterministic machine, we show that the multiple computations for a single input string can be sequentially generated and examined.

A nondeterministic Turing machine may produce multiple computations for a single input string. The computations can be systematically produced by ordering the alternative transitions for a state, symbol pair. Let *n* be the maximum number of transitions defined for any combination of state and tape symbol. The numbering assumes that  $\delta(q_i, x)$  defines *n*, not necessarily distinct, transitions for every state  $q_i$  and tape symbol *x* with  $\delta(q_i, x) \neq \emptyset$ . If the transition function defines fewer than *n* transitions, one transition is assigned several numbers to complete the ordering.

A sequence  $(m_1, \dots, m_i, \dots, m_k)$ , where each  $m_i$  is a number from 1 to *n*, defines a unique computation in the nondeterministic machine. The computation associated with this sequence consists of *k* or fewer transitions. The *j*th transition is determined by the state, the tape symbol scanned, and  $m_j$ , the *j*th number in the sequence. Assume the *j* - 1st transition leaves the machine in state  $q_i$  scanning *x*. If  $\delta(q_i, x) = \emptyset$ , the computation halts. Otherwise, the machine executes the transition in  $\delta(q_i, x)$  numbered  $m_j$ .

**TABLE 8.1** Ordering of Transitions

| State | Symbol | Transition   | State | Symbol | Transition   |
|-------|--------|--------------|-------|--------|--------------|
| $q_0$ | $B$    | $1q_1, B, R$ | $q_2$ | $a$    | $1q_3, a, R$ |
|       |        | $2q_1, B, R$ |       |        | $2q_3, a, R$ |
|       |        | $3q_1, B, R$ |       |        | $3q_3, a, R$ |
| $q_1$ | $a$    | $1q_1, a, R$ | $q_3$ | $b$    | $1q_4, b, R$ |
|       |        | $2q_1, a, R$ |       |        | $2q_4, b, R$ |
|       |        | $3q_1, a, R$ |       |        | $3q_4, b, R$ |
| $q_1$ | $b$    | $1q_1, b, R$ | $q_5$ | $b$    | $1q_6, b, L$ |
|       |        | $2q_1, b, R$ |       |        | $2q_6, b, L$ |
|       |        | $3q_1, b, R$ |       |        | $3q_6, b, L$ |
| $q_1$ | $c$    | $1q_1, c, R$ | $q_6$ | $a$    | $1q_7, a, L$ |
|       |        | $2q_2, c, R$ |       |        | $2q_7, a, L$ |
|       |        | $3q_5, c, L$ |       |        | $3q_7, a, L$ |

The transitions of the nondeterministic machine in Example 8.7.1 can be ordered as shown in Table 8.7.1. The computations defined by the input string  $acab$  and the sequences  $(1, 1, 1, 1, 1)$ ,  $(1, 1, 2, 1, 1)$ , and  $(2, 2, 3, 3, 1)$  are

$$\begin{array}{lll}
 q_0BacabB & 1 & q_0BacabB & 1 & q_0BacabB & 2 \\
 \vdash Bq_1acabB & 1 & \vdash Bq_1acabB & 1 & \vdash Bq_1acabB & 2 \\
 \vdash Baq_1cabB & 1 & \vdash Baq_1cabB & 2 & \vdash Baq_1cabB & 3 \\
 \vdash Bacq_1abB & 1 & \vdash Bacq_2abB & 1 & \vdash Bq_5acabB. \\
 \vdash Bacaq_1bB & 1 & \vdash Bacaq_3bB & 1 & \\
 \vdash Bacabq_1B & & \vdash Bacabq_4B & &
 \end{array}$$

The number on the right designates the transition used to obtain the subsequent configuration. The third computation terminates prematurely since no transition is defined when the machine is in state  $q_5$  scanning an  $a$ . The string  $acab$  is accepted since the computation defined by  $(1, 1, 2, 1, 1)$  terminates in state  $q_4$ .

Using the ability to sequentially produce the computations of a nondeterministic machine, we will now show that every nondeterministic Turing machine can be transformed into an equivalent deterministic machine. Let  $M = (Q, \Sigma, \Gamma, \delta, q_0)$  be a nondeterministic machine that accepts strings by halting. We choose acceptance by halting because this reduces the number of potential outcomes of a computation from three to two—a

computa  
the proo  
scheme,  
three-tap  
defined b

The  
tween se  
tions are

T

T

A compu

1. A sec
2. The i
3. The c
4. If the
5. If the

The si  
machine in  
the symb

in numeri  
begins in s  
contains th

Using

tic three-ta

the genera

the sequen

accept by h

Let  $\Sigma$

Symbols of  
on tape 2 du  
grouped by t

computation halts (and accepts) or does not halt. Thus we have fewer cases to consider in the proof. Assume that the transitions of  $M$  have been numbered according to the previous scheme, with  $n$  the maximum number of transitions for a state, symbol pair. A deterministic three-tape machine  $M'$  is constructed to accept the language of  $M$ . Acceptance in  $M'$  is also defined by halting.

The machine  $M'$  is built to simulate the computations of  $M$ . The correspondence between sequences  $(m_1, \dots, m_k)$  and computations of  $M'$  ensures that all possible computations are examined. The role of the three tapes of  $M'$  are

- Tape 1: stores the input string;
- Tape 2: simulates the tape of  $M$ ;
- Tape 3: holds sequences of the form  $(m_1, \dots, m_k)$  to guide the simulation.

A computation in  $M'$  consists of the actions:

1. A sequence of integers  $(m_1, \dots, m_k)$  from 1 to  $n$  is written on tape 3.
2. The input string on tape 1 is copied to the standard input position on tape 2.
3. The computation of  $M$  defined by the sequence on tape 3 is simulated on tape 2.
4. If the simulation halts prior to executing  $k$  transitions, the computation of  $M'$  halts and accepts the input.
5. If the computation did not halt in step 3, the next sequence is generated on tape 3 and the computation continues at step 2.

The simulation is guided by the sequence of values on tape 3. The deterministic Turing machine in Figure 8.1 generates all finite-length sequences of integers from 1 to  $n$ , where the symbols  $1, 2, \dots, n$  are individual tape symbols. Sequences of length 1 are generated in numeric order, followed by sequences of length 2, length 3, and so on. A computation begins in state  $q_0$  at position zero. When the tape head returns to position zero the tape contains the next sequence of values. The notation  $i/i$  abbreviates  $1/1, 2/2, \dots, n/n$ .

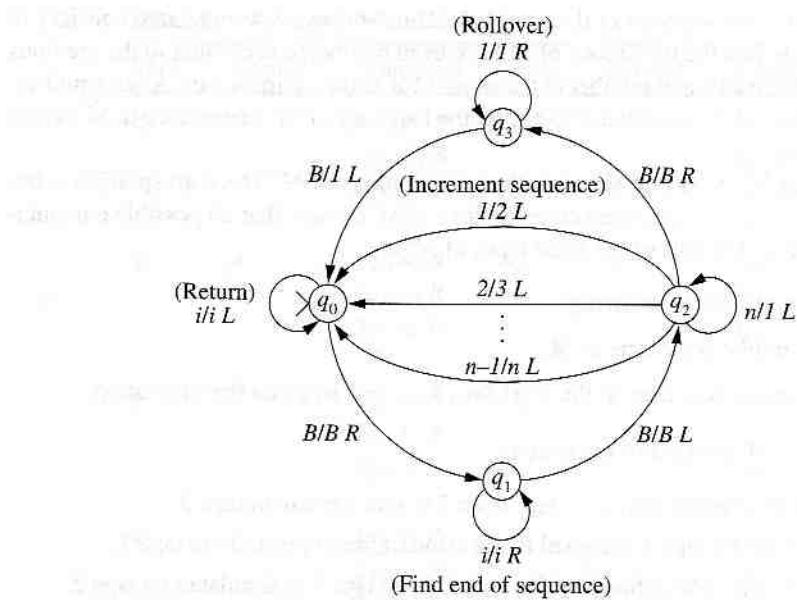
Using the exhaustive generation of numeric sequences, we now construct a deterministic three-tape machine  $M'$  that accepts  $L(M)$ . A computation of the machine  $M'$  interweaves the generation of the sequences on tape 3 with the simulation of  $M$  on tape 2.  $M'$  halts when the sequence on tape 3 defines a computation that halts in  $M$ . Recall that both  $M$  and  $M'$  accept by halting.

Let  $\Sigma$  and  $\Gamma$  be the input and tape alphabets of  $M$ . The alphabets of  $M'$  are

$$\Sigma_{M'} = \Sigma$$

$$\Gamma_{M'} = \{x, \#x \mid x \in \Gamma\} \cup \{1, \dots, n\}.$$

Symbols of the form  $\#x$  represent tape symbol  $x$  and are used to mark the leftmost square on tape 2 during the simulation of the computation of  $M$ . The transitions of  $M'$  are naturally grouped by their function. States labeled  $q_{s,j}$  are used in the generation of a sequence on tape

FIGURE 8.1 Turing machine generating  $\{1, 2, \dots, n\}^+$ .

3. These transitions are obtained from the machine in Figure 8.1. The tape heads reading tapes 1 and 2 remain stationary during this operation.

$$\begin{aligned}
 \delta(q_{s,0}, B, B, B) &= [q_{s,1}; B, S; B, S; B, R] \\
 \delta(q_{s,1}, B, B, t) &= [q_{s,1}; B, S; B, S; i, R] \quad t = 1, \dots, n \\
 \delta(q_{s,1}, B, B, B) &= [q_{s,2}; B, S; B, S; B, L] \\
 \delta(q_{s,2}, B, B, n) &= [q_{s,2}; B, S; B, S; I, L] \\
 \delta(q_{s,2}, B, B, t-1) &= [q_{s,4}; B, S; B, S; t, L] \quad t = 1, \dots, n-1 \\
 \delta(q_{s,2}, B, B, B) &= [q_{s,3}; B, S; B, S; B, R] \\
 \delta(q_{s,3}, B, B, I) &= [q_{s,3}; B, S; B, S; I, R] \\
 \delta(q_{s,3}, B, B, B) &= [q_{s,4}; B, S; B, S; I, L] \\
 \delta(q_{s,4}, B, B, t) &= [q_{s,4}; B, S; B, S; t, L] \quad t = 1, \dots, n \\
 \delta(q_{s,4}, B, B, B) &= [qc,0; B, S; B, S; B, S]
 \end{aligned}$$

The next step is to make a copy of the input on tape 2. The symbol  $\#B$  is written in position zero to designate the left boundary of the tape.

$\delta(q_{c,0}, B, \#B, R)$   
 $\delta(q_{c,0}, B, \#B, L)$   
 $\delta(q_{c,1}, B, \#B, R)$   
 $\delta(q_{c,1}, B, \#B, L)$   
 $\delta(q_{c,2}, B, \#B, R)$   
 $\delta(q_{c,2}, B, \#B, L)$   
 $\delta(q_{c,3}, B, \#B, R)$   
 $\delta(q_{c,3}, B, \#B, L)$

The transitions to  $qc,0$  and  $qc,1$  are obtained from the transitions to  $q_0$  and  $q_1$  in the ordering of the states.

are the corresponding transitions.

If the sequence of transitions. The computation on tape 3 halts. The transitions designed to produce the result of the computation on the tape heads or  $qc,1$ , respectively.

$\delta(q_i, B, x, R)$   
 $\delta(q_i, B, x, L)$   
 $\delta(q_{e,0}, B, x, R)$   
 $\delta(q_{e,0}, B, x, L)$   
 $\delta(q_{e,1}, B, x, R)$   
 $\delta(q_{e,1}, B, x, L)$   
 $\delta(q_{e,2}, B, x, R)$   
 $\delta(q_{e,2}, B, x, L)$   
 $\delta(q_{e,3}, B, x, R)$   
 $\delta(q_{e,3}, B, x, L)$

When a blank symbol is accessed during the initial position of computations.

$$\begin{aligned}
 \delta(q_{c,0}, B, B, B) &= [q_{c,1}; B, R; \#B, R; B, S] \\
 \delta(q_{c,1}, x, B, B) &= [q_{c,1}; x, R; x, R; B, S] \quad \text{for all } x \in \Gamma - \{B\} \\
 \delta(q_{c,1}, B, B, B) &= [q_{c,2}; B, L; B, L; B, S] \\
 \delta(q_{c,2}, x, x, B) &= [q_{c,2}; x, L; x, L; B, S] \quad \text{for all } x \in \Gamma \\
 \delta(q_{c,2}, B, \#B, B) &= [q_0; B, S; \#B, S; B, R]
 \end{aligned}$$

The transitions that simulate the computation of  $M$  on tape 2 of  $M'$  are obtained directly from the transitions of  $M$ . If  $\delta(q_i, x) = [q_j, y, d]$  is a transition of  $M$  assigned the number  $t$  in the ordering, then

$$\begin{aligned}
 \delta(q_i, B, x, t) &= [q_j; B, S; y, d; t, R] \\
 \delta(q_i, B, \#x, t) &= [q_j; B, S; \#y, d; t, R]
 \end{aligned}$$

are the corresponding transitions of  $M'$ .

If the sequence on tape 3 consists of  $k$  numbers, the simulation processes at most  $k$  transitions. The computation of  $M'$  halts if the computation of  $M$  specified by the sequence on tape 3 halts. When a blank is read on tape 3, the simulation has processed all of the transitions designated by the current sequence. Before the next sequence is processed, the result of the simulated computation must be erased from tape 2. To accomplish this, the tape heads on tapes 2 and 3 are repositioned at the leftmost position in state  $q_{e,0}$  and  $q_{e,1}$ , respectively. The head on tape 2 then moves to the right, erasing the tape.

heads reading

$$\begin{aligned}
 \delta(q_i, B, x, B) &= [q_{e,0}; B, S; x, S; B, S] \quad \text{for all } x \in \Gamma \\
 \delta(q_i, B, \#x, B) &= [q_{e,0}; B, S; \#x, S; B, S] \quad \text{for all } x \in \Gamma \\
 \delta(q_{e,0}, B, x, B) &= [q_{e,0}; B, S; x, L; B, S] \quad \text{for all } x \in \Gamma \\
 \delta(q_{e,0}, B, \#x, B) &= [q_{e,1}; B, S; B, S; B, L] \quad \text{for all } x \in \Gamma \\
 \delta(q_{e,1}, B, B, t) &= [q_{e,1}; B, S; B, S; t, L] \quad t = 1, \dots, n \\
 \delta(q_{e,1}, B, B, B) &= [q_{e,2}; B, S; B, R; B, R] \\
 \delta(q_{e,2}, B, x, i) &= [q_{e,2}; B, S; B, R; i, R] \quad \text{for all } x \in \Gamma \text{ and } i = 1, \dots, n \\
 \delta(q_{e,2}, B, B, B) &= [q_{e,3}; B, S; B, L; B, L] \\
 \delta(q_{e,3}, B, B, t) &= [q_{e,3}; B, S; B, L; t, L] \quad t = 1, \dots, n \\
 \delta(q_{e,3}, B, B, B) &= [q_{s,0}; B, S; B, S; B, S]
 \end{aligned}$$

When a blank is read on tape 3, the entire segment of the tape that may have been accessed during the simulated computation has been erased.  $M'$  then returns the tape heads to their initial position and enters  $q_{s,0}$  to generate the next sequence and continue the simulation of computations.

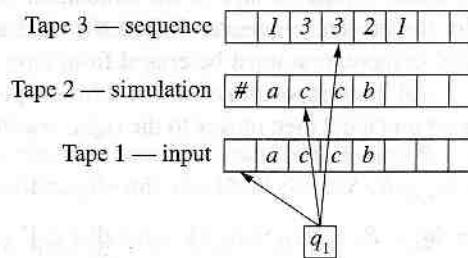
#B is written in

The process of simulating computations of  $M$ , steps 2 through 5 of the algorithm, continues until a sequence of numbers is generated on tape 3 that defines a halting computation. The simulation of this computation causes  $M'$  to halt, accepting the input. If the input string is not in  $L(M)$ , the cycle of sequence generation and computation simulation in  $M'$  will continue indefinitely.

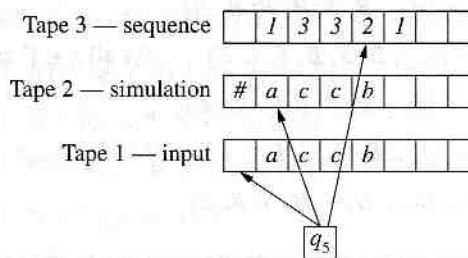
The actions of the deterministic machine constructed following the preceding strategy are illustrated using the nondeterministic machine from Example 8.7.1 and the numbering of the transitions in Table 8.7.1. The first three transitions of the computation  $M$  defined by the sequence  $(1, 3, 3, 2, 1)$  and input string  $accb$  are

$$\begin{aligned} & q_0 B accb B \ 1 \\ \vdash & B q_1 accb B \ 3 \\ \vdash & B a q_1 ccb B \ 3 \\ \vdash & B q_5 accb B. \end{aligned}$$

The sequence  $1, 3, 3, 2, 1$  that designates the particular computation of  $M$  is written on tape 3 of  $M'$ . The configuration of the three-tape machine  $M'$  prior to the execution of the third transition of  $M$  is



Transition 3 from state  $q_1$  with  $M$  scanning a  $c$  causes the machine to print  $c$ , enter state  $q_5$ , and move to the left. This transition is simulated in  $M'$  by the transition  $\delta'(q_1, B, c, 3) = [q_5; B, S; c, L; 3, R]$ . The transition of  $M'$  alters tape 2 as prescribed by the transition of  $M$  and moves the head on tape 3 to designate the number of the subsequent transition.



Nondeterministic Turing machines can be defined with a multitrack tape, two-way tape, or multiple tapes. Machines defined using these alternative configurations can also be shown to accept precisely the recursively enumerable languages.

Like their  
state can be us  
ministic mach  
(Exercice 23)

### Example 8.7.2

The two-tape

M:  $\times$

accepts the set  
 $q_2$  on reading  
loop in state  $q_2$   
it. If a string is  
the input. The  
a  $b$  in the midd

The next e  
machines and t

### Example 8.7.3

Let  $M = (Q, \Sigma, \Delta, \delta, q_0, F)$ . We will design  $M'$  to have a substring  $xz$  where  $x, z \in \Sigma^*$  and  $y \in L\}$ . A

1. Reading the input string to begin computation
2. Copying from the input string to the simulation tape
3. Simulating the computation of  $M$  on the simulation tape

The first two steps check whether

The states of  $M$  and  $M'$  are the same, and  $q_e$ .

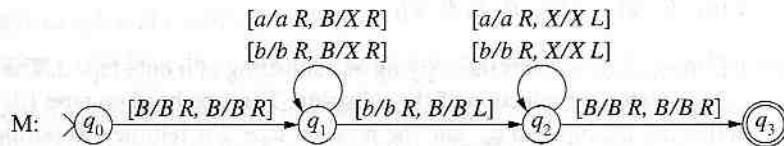
ithm, computation. input string in  $M'$  will

ng strategy numbering defined by

Like their deterministic counterparts, nondeterministic machines that accept by final state can be used to show that a language is recursive. If every computation in the nondeterministic machine halts, so will every computation in the equivalent deterministic machine (Exercise 23).

### Example 8.7.2

The two-tape nondeterministic machine



accepts the set of strings over  $\{a, b\}$  with a  $b$  in the middle. The transition from state  $q_1$  to  $q_2$  on reading a  $b$  on tape 1 represents a guess that the  $b$  is in the middle of the input. The loop in state  $q_2$  compares the number of symbols following the  $b$  to the number preceding it. If a string is in  $L(M)$ , one computation will enter  $q_3$  upon reading the middle  $b$  and accept the input. The computations for strings with no  $b$ 's halt in  $q_1$ , and strings that do not have a  $b$  in the middle halt in either  $q_1$  or  $q_2$ . Since  $M$  halts for all inputs,  $L(M)$  is recursive.  $\square$

The next example illustrates the flexibility afforded by the combination of multitape machines and the guess and check strategy of nondeterminism.

### Example 8.7.3

inter state  $q_5$ ,  
 $i_1, B, c, 3) =$   
 ansition of  $M$   
 sition.

Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a standard Turing machine that accepts a language  $L$ . We will design a two-tape nondeterministic machine  $M'$  that accepts strings over  $\Sigma^*$  that have a substring of length two or more in  $L$ . That is,  $L(M') = \{u \mid u = xyz, \text{length}(y) \geq 2 \text{ and } y \in L\}$ . A computation of  $M'$  with input  $u$  consists of the following steps:

1. Reading the input on tape 1 and nondeterministically choosing a position in the string to begin copying to tape 2;
2. Copying from tape 1 to tape 2 and nondeterministically choosing a position to stop copying;
3. Simulating the computation of  $M$  on tape 2.

The first two steps constitute the nondeterministic guess of a substring of  $u$  and the third checks whether the substring is in  $L$ .

The states of  $M'$  are  $Q \cup \{q_s, q_b, q_c, q_d, q_e\}$  with start state  $q_s$ . The alphabets and final states are the same as those of  $M$ . The transitions for steps 1 and 2 use states  $q_s, q_b, q_c, q_d$ , and  $q_e$ .

two-way tape,  
 also be shown

$$\begin{aligned}
 \delta'(q_s, B, B) &= \{ [q_b; B, R; B, R] \} \\
 \delta'(q_b, x, B) &= \{ [q_b; x, R; B, S], [q_c; x, R; x, R] \} \quad \text{for all } x \in \Sigma \\
 \delta'(q_c, x, B) &= \{ [q_c; x, R; x, R], [q_d; x, R; x, R] \} \quad \text{for all } x \in \Sigma \\
 \delta'(q_d, x, B) &= \{ [q_d; x, R; B, S] \} \quad \text{for all } x \in \Sigma \\
 \delta'(q_d, B, B) &= \{ [q_e; B, S; B, L] \} \\
 \delta'(q_e, B, x) &= \{ [q_e; B, S; x, L] \} \quad \text{for all } x \in \Sigma \\
 \delta'(q_e, B, B) &= \{ [q_0; B, S; B, S] \}
 \end{aligned}$$

The transition from  $q_b$  to  $q_c$  initiates the copying of a substring of  $u$  onto tape 2. The second transition in  $q_c$  completes the selection of the substring. The tape head on tape 1 is moved to the blank following the input in  $q_d$ , and the head on tape 2 is returned to position zero in  $q_e$ .

After the nondeterministic selection of a substring, the transitions of  $M$  are run on tape 2 to check whether the “guessed” substring is in  $L$ . The transitions for this part of the computation are obtained directly from  $\delta$ , the transition function of  $M$ :

$$\delta'(q_i, B, x) = \{ [q_j; B, S; y, d] \} \quad \text{whenever } \delta(q_i, x) = [q_j, y, d] \text{ is a transition of } M.$$

The tape head reading tape 1 remains stationary while the computation of  $M$  is run on tape 2.  $\square$

### Definition 8.8.1

- A  $k$ -tape Turing machine is defined by:
- the computation is run on  $k$  tapes
  - with each tape having a head that moves to the right
  - at any point in time, the heads are at positions  $u_1, u_2, \dots, u_k$  where  $u_i \in L$

where  $u_i \in L$

iv) a string  $u$  will be produced

The last condition eventually writes a string  $u$  which must be produced by the computation. Such a machine models an enumerator.

### Example 8.8.1

The machine  $E$  enumerates the language  $L = \{a^n b^n \mid n \geq 0\}$ .

$\times q_0 \quad [B/B, B/B]$

## 8.8 Turing Machines as Language Enumerators

In the preceding sections Turing machines have been formulated as language acceptors: A machine is provided with an input string, and the result of the computation indicates the acceptability of the input. Turing machines may also be designed to enumerate a language. The computation of such a machine sequentially produces an exhaustive listing of the elements of the language. An enumerating machine has no input; its computation continues until it has generated every string in the language.

Like Turing machines that accept languages, there are a number of equivalent ways to define an enumerating machine. We will use a  $k$ -tape deterministic machine,  $k \geq 2$ , as the underlying Turing machine model in the definition of enumerating machines. The first tape is the output tape and the remaining tapes are work tapes. A special tape symbol # is used on the output tape to separate the elements of the language that are generated during the computation.

The machines considered in this section perform two distinct tasks, acceptance and enumeration. To distinguish them, a machine that accepts a language will be denoted  $M$  while an enumerating machine will be denoted  $E$ .

The computation of  $E$  generates strings in  $L$  simultaneously, an infinite sequence of strings.

**Definition 8.8.1**

A  $k$ -tape Turing machine  $E = (Q, \Sigma, \Gamma, \delta, q_0)$  **enumerates** the language  $L$  if

- the computation begins with all tapes blank;
- with each transition, the tape head on tape 1 (the output tape) remains stationary or moves to the right;
- at any point in the computation, the nonblank portion of tape 1 has the form

$$B\#u_1\#u_2\#\dots\#u_k\# \quad \text{or} \quad B\#u_1\#u_2\#\dots\#u_k\#v,$$

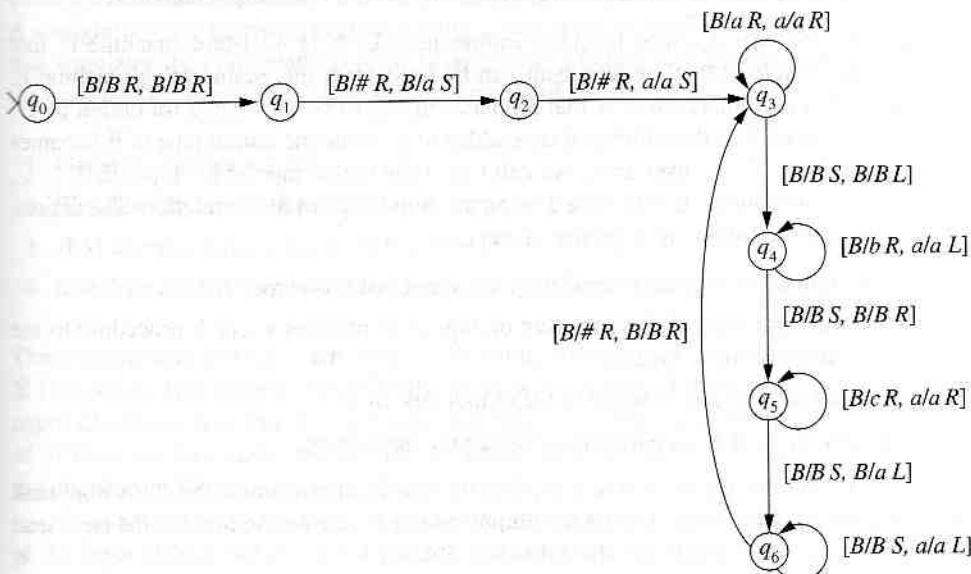
where  $u_i \in L$  and  $v \in \Sigma^*$ ;

- a string  $u$  will be written on tape 1 preceded and followed by  $\#$  if, and only if,  $u \in L$ .

The last condition indicates that the computation of a machine  $E$  that enumerates  $L$  eventually writes every string in  $L$  on the output tape. Since all of the elements of a language must be produced, a computation enumerating an infinite language will never halt. The definition does not require a machine to halt even if it is enumerating a finite language. Such a machine may continue indefinitely after writing the last element on the output tape.

**Example 8.8.1**

The machine  $E$  enumerates the language  $L = \{a^i b^j c^i \mid i \geq 0\}$ . A Turing machine accepting this language was given in Example 8.2.2.



The computation of  $E$  begins by writing  $\#\#$  on the output tape, indicating that  $\lambda \in L$ . Simultaneously, an  $a$  is written in position one of tape 2, with the head returning to tape

position zero. At this point,  $E$  enters the nonterminating loop described by the following actions.

1. The tape heads move to the right, writing an  $a$  on the output tape for every  $a$  on the work tape.
2. The head on the work tape then moves right to left through the  $a$ 's and a  $b$  is written on the output tape for each  $a$ .
3. The tape heads move to the right, writing a  $c$  on the output tape for every  $a$  on the work tape.
4. An  $a$  is added to the end of the work tape and the head is moved to position one.
5. A  $\#$  is written on the output tape.

After a string is completed on the output tape, the work tape contains the information required to construct the next string in the enumeration.  $\square$

The definition of enumeration requires that each string in the language appear on the output tape but permits a string to appear multiple times. Theorem 8.8.2 shows that any language that is enumerated by a Turing machine can be enumerated by one in which each string is written only once on the output tape.

### Theorem 8.8.2

Let  $L$  be a language enumerated by a Turing machine  $E$ . Then there is a Turing machine  $E'$  that enumerates  $L$  and each string in  $L$  appears only once on the output tape of  $E'$ .

**Proof.** Assume  $E$  is a  $k$ -tape machine enumerating  $L$ . A  $(k + 1)$ -tape machine  $E'$  that satisfies the "single output" requirement can be built from the enumerating machine  $E$ . Intuitively,  $E$  is a submachine of  $E'$  that produces strings to be considered for output by  $E'$ . The output tape of  $E'$  is the additional tape added to  $E$ , while the output tape of  $E$  becomes a work tape for  $E'$ . For convenience, we call tape 1 the output tape of  $E'$ . Tapes 2, 3, ...,  $k + 1$  are used to simulate  $E$ , with tape 2 being the output tape of the simulation. The actions of  $E'$  consist of the following sequence of steps:

1. The computation begins by simulating the actions of  $E$  on tapes 2, 3, ...,  $k + 1$ .
2. When the simulation of  $E$  writes  $\#u\#$  on tape 2,  $E'$  initiates a search procedure to see if  $u$  already occurs on tape 2.
3. If  $u$  is not on tape 2, it is added to the output tape of  $E'$ .
4. The simulation of  $E$  is restarted to produce the next string.

Searching for another occurrence of  $u$  requires the tape head to examine the entire nonblank portion of tape 2. Since tape 2 is not the output tape of  $E'$ , the restriction that the tape head on the output tape never move to the left is not violated.  $\blacksquare$

Theorem 8.8.2 justifies the selection of the term *enumerate* to describe this type of computation. The computation sequentially and exhaustively lists the strings in the

language  
sequence  
first string  
may pro  
Turi  
acceptan  
language

**Lemma 8.8.3**  
If  $L$  is en

**Proof.** A  
M accept  
remaining  
with a strin  
of  $E$  write  
if  $u = w$ .  
comparis  
accepted  $L$

The p  
cated by t  
A straigh  
that simul  
the output

1. Gener
2. Simul
3. If  $M$  a
4. Contin

The genera  
 $\Sigma$  for testin  
approach c  
of  $M$  does  
membership

To con  
of the input  
check eve  
alphabet  $\Sigma$   
in  $\Sigma^*$ .

ed by the following  
e for every  $a$  on the  
i's and a  $b$  is written  
r every  $a$  on the work  
to position one.

tains the information  
□

nguage appear on the  
8.8.2 shows that any  
by one in which each

is a Turing machine  $E'$   
put tape of  $E'$ .

-tape machine  $E'$  that  
umerating machine  $E$ .  
dered for output by  $E'$ .  
put tape of  $E$  becomes  
of  $E'$ . Tapes 2, 3, . . . ,  
imulation. The actions

s 2, 3, . . . ,  $k + 1$ .  
earch procedure to see

nine the entire nonblank  
ction that the tape head  
■

e to describe this type  
lists the strings in the

language. The order in which the strings are produced defines a mapping from an initial sequence of the natural numbers onto  $L$ . Thus we can talk about the zeroth string in  $L$ , the first string in  $L$ , and so on. This ordering is machine-specific; another enumerating machine may produce a completely different ordering.

Turing machine computations now have two distinct ways of defining a language: by acceptance and by enumeration. We show that these two approaches produce the same languages.

### Lemma 8.8.3

If  $L$  is enumerated by a Turing machine, then  $L$  is recursively enumerable.

**Proof.** Assume that  $L$  is enumerated by a  $k$ -tape Turing machine  $E$ . A  $(k + 1)$ -tape machine  $M$  accepting  $L$  can be constructed from  $E$ . The additional tape of  $M$  is the input tape; the remaining  $k$  tapes allow  $M$  to simulate the computation of  $E$ . The computation of  $M$  begins with a string  $u$  on its input tape. Next  $M$  simulates the computation of  $E$ . When the simulation of  $E$  writes  $\#$ , a string  $w \in L$  has been generated.  $M$  then compares  $u$  with  $w$  and accepts  $u$  if  $u = w$ . Otherwise, the simulation of  $E$  is used to generate another string from  $L$  and the comparison cycle is repeated. If  $u \in L$ , it will eventually be produced by  $E$  and consequently accepted by  $M$ . ■

The proof that any recursively enumerable language  $L$  can be enumerated is complicated by the fact that a Turing machine  $M$  that accepts  $L$  need not halt for every input string. A straightforward approach to enumerating  $L$  would be to build an enumerating machine that simulates the computations of  $M$  to determine whether a string should be written on the output tape. The actions of such a machine would be to

1. Generate a string  $u \in \Sigma^*$ .
2. Simulate the computation of  $M$  with input  $u$ .
3. If  $M$  accepts, write  $u$  on the output tape.
4. Continue at step 1 until all strings in  $\Sigma^*$  have been tested.

The generate-and-test approach requires the ability to generate the entire set of strings over  $\Sigma$  for testing. This presents no difficulty, as we will see later. However, step 2 of this naive approach causes it to fail. It is possible to produce a string  $u$  for which the computation of  $M$  does not terminate. In this case, no strings after  $u$  will be generated and tested for membership in  $L$ .

To construct an enumerating machine, we first introduce the lexicographical ordering of the input strings and provide a strategy to ensure that the enumerating machine  $E$  will check every string in  $\Sigma^*$ . The lexicographical ordering of the set of strings over a nonempty alphabet  $\Sigma$  defines a one-to-one correspondence between the natural numbers and the strings in  $\Sigma^*$ .

**Definition 8.8.4**

Let  $\Sigma = \{a_1, \dots, a_n\}$  be an alphabet. The lexicographical ordering  $lo$  of  $\Sigma^*$  is defined recursively as follows:

- i) Basis:  $lo(\lambda) = 0$ ,  $lo(a_i) = i$  for  $i = 1, 2, \dots, n$ .
- ii) Recursive step:  $lo(a_i u) = lo(u) + i \cdot n^{length(u)}$ .

The values assigned by the function  $lo$  define a total ordering on the set  $\Sigma^*$ . Strings  $u$  and  $v$  are said to satisfy  $u < v$ ,  $u = v$ , and  $u > v$  if  $lo(u) < lo(v)$ ,  $lo(u) = lo(v)$ , and  $lo(u) > lo(v)$ , respectively.

**Example 8.8.2**

Let  $\Sigma = \{a, b, c\}$  and let  $a$ ,  $b$ , and  $c$  be assigned the values 1, 2, and 3, respectively. The lexicographical ordering produces

$$\begin{array}{llllll} lo(\lambda) = 0 & lo(a) = 1 & lo(aa) = 4 & lo(ba) = 7 & lo(ca) = 10 & lo(aaa) = 13 \\ lo(b) = 2 & lo(ab) = 5 & lo(bb) = 8 & lo(cb) = 11 & lo(aab) = 14 & \\ lo(c) = 3 & lo(ac) = 6 & lo(bc) = 9 & lo(cc) = 12 & lo(aac) = 15. & \square \end{array}$$

**Lemma 8.8.5**

For any alphabet  $\Sigma$ , there is a Turing machine  $E_{\Sigma^*}$  that enumerates  $\Sigma^*$  in lexicographical order.

The construction of a machine that enumerates the set of strings over the alphabet  $\{0, 1\}$  is left as an exercise.

The lexicographical ordering and a dovetailing technique are used to show that a recursively enumerable language  $L$  can be enumerated by a Turing machine. Let  $M$  be a Turing machine that accepts  $L$ . Recall that  $M$  need not halt for all input strings. The lexicographical ordering produces a listing  $u_0 = \lambda, u_1, u_2, u_3, \dots$  of the strings of  $\Sigma^*$ . A two-dimensional table is constructed whose columns are labeled by the strings of  $\Sigma^*$  and rows by the natural numbers.

|   | $\lambda$      | $u_1$      | $u_2$      | $\dots$ |
|---|----------------|------------|------------|---------|
| 3 | $[\lambda, 3]$ | $[u_1, 3]$ | $[u_2, 3]$ | $\dots$ |
| 2 | $[\lambda, 2]$ | $[u_1, 2]$ | $[u_2, 2]$ | $\dots$ |
| 1 | $[\lambda, 1]$ | $[u_1, 1]$ | $[u_2, 1]$ | $\dots$ |
| 0 | $[\lambda, 0]$ | $[u_1, 0]$ | $[u_2, 0]$ | $\dots$ |

The  $[i, j]$  entry in this table is interpreted to mean “run machine M on input  $u_i$  for  $j$  steps.” Using the technique presented in Example 1.4.2, the ordered pairs in the table can be enumerated in a “diagonal by diagonal” manner (Exercise 33).

The machine E built to enumerate L interleaves the enumeration of the ordered pairs with the computations of M. The computation of E is a loop that consists of the following steps:

1. Generate an ordered pair  $[i, j]$ .
2. Run a simulation of M with input  $u_i$  for  $j$  transitions or until the simulation halts.
3. If M accepts, write  $u_i$  on the output tape.
4. Continue with step 1.

If  $u_i \in L$ , then the computation of M with input  $u_i$  halts and accepts after  $k$  transitions, for some number  $k$ . Thus  $u_i$  will be written to the output tape of E when the ordered pair  $[i, k]$  is processed. The second element in an ordered pair  $[i, j]$  ensures that the simulation M is terminated after  $j$  steps. Consequently, no nonterminating computations are allowed and each string in  $\Sigma^*$  is examined.

Combining the preceding argument with Lemma 8.8.3 yields

### Theorem 8.8.6

A language is recursively enumerable if, and only if, it can be enumerated by a Turing machine.

A Turing machine that accepts a recursively enumerable language halts and accepts every string in the language but is not required to halt when an input is a string that is not in the language. A language L is recursive if it is accepted by a machine that halts for all input. Since every computation halts, such a machine provides a decision procedure for determining membership in L. The family of recursive languages can also be defined by enumerating Turing machines.

The definition of an enumerating Turing machine does not impose any restrictions on the order in which the strings of the language are generated. Requiring the strings to be generated in a predetermined computable order provides the additional information needed to obtain negative answers to the membership question. Intuitively, the strategy to determine whether a string  $u$  is in the language is to begin the enumerating machine and compare  $u$  with each string that is produced. Eventually either  $u$  is output, in which case it is accepted, or some string beyond  $u$  in the ordering is generated. Since the output strings are produced according to the ordering,  $u$  has been passed and is not in the language. Thus we are able to decide membership, and the language is recursive. Theorem 8.8.7 shows that recursive languages may be characterized as the family of languages whose elements can be enumerated in order.

### Theorem 8.8.7

L is recursive if, and only if, L can be enumerated in lexicographical order.

**Proof.** We first show that every recursive language can be enumerated in lexicographical order. Let  $L$  be a recursive language over an alphabet  $\Sigma$ . Then it is accepted by some machine  $M$  that halts for all input strings. A machine  $E$  that enumerates  $L$  in lexicographical order can be constructed from  $M$  and the machine  $E_{\Sigma^*}$  that enumerates  $\Sigma^*$  in lexicographical order. The machine  $E$  is a hybrid, interleaving the computations of  $M$  and  $E_{\Sigma^*}$ . The computation of  $E$  consists of the following loop:

1. The machine  $E_{\Sigma^*}$  is run, producing a string  $u \in \Sigma^*$ .
2.  $M$  is run with input  $u$ .
3. If  $M$  accepts  $u$ ,  $u$  is written on the output tape of  $E$ .
4. The generate-and-test loop continues with step 1.

Since  $M$  halts for all inputs,  $E$  cannot enter a nonterminating computation in step 2. Thus each string  $u \in \Sigma^*$  will be generated and tested for membership in  $L$ .

Now we show that any language  $L$  that can be enumerated in lexicographical order is recursive. This proof is divided into two cases based on the cardinality of  $L$ .

Case 1:  $L$  is finite. Then  $L$  is recursive since every finite language is recursive.

Case 2:  $L$  is infinite. The argument is similar to that given in Theorem 8.8.2 except that the ordering is used to terminate the computation. As before, a  $(k+1)$ -tape machine  $M$  accepting  $L$  can be constructed from a  $k$ -tape machine  $E$  that enumerates  $L$  in lexicographical order. The additional tape of  $M$  is the input tape; the remaining  $k$  tapes allow  $M$  to simulate the computations of  $E$ . The ordering of the strings produced by  $E$  provides the information needed to halt  $M$  when the input is not in the language. The computation of  $M$  begins with a string  $u$  on its input tape. Next  $M$  simulates the computation of  $E$ . When the simulation produces a string  $w$ ,  $M$  compares  $u$  with  $w$ . If  $u = w$ , then  $M$  halts and accepts. If  $w$  is greater than  $u$  in the ordering,  $M$  halts rejecting the input. Finally, if  $w$  is less than  $u$  in the ordering, then the simulation of  $E$  is restarted to produce another element of  $L$  and the comparison cycle is repeated. ■

## Exercises

1. Let  $M$  be the Turing machine defined by

| $\delta$ | $B$         | $a$         | $b$         | $c$         |
|----------|-------------|-------------|-------------|-------------|
| $q_0$    | $q_1, B, R$ |             |             |             |
| $q_1$    | $q_2, B, L$ | $q_1, a, R$ | $q_1, c, R$ | $q_1, c, R$ |
| $q_2$    |             | $q_2, c, L$ |             | $q_2, b, L$ |

- a) Trace the computation for the input string  $aabca$ .
- b) Trace the computation for the input string  $bcbc$ .

- c) Give the state diagram of M.
- d) Describe the result of a computation in M.
2. Let M be the Turing machine defined by
- | $\delta$ | $B$         | $a$         | $b$         | $c$         |
|----------|-------------|-------------|-------------|-------------|
| $q_0$    | $q_1, B, R$ |             |             |             |
| $q_1$    | $q_1, B, R$ | $q_1, a, R$ | $q_1, b, R$ | $q_2, c, L$ |
| $q_2$    |             | $q_2, b, L$ | $q_2, a, L$ |             |
- a) Trace the computation for the input string  $abcab$ .
- b) Trace the first six transitions of the computation for the input string  $abab$ .
- c) Give the state diagram of M.
- d) Describe the result of a computation in M.
3. Construct a Turing machine with input alphabet  $\{a, b\}$  to perform each of the following operations. Note that the tape head is scanning position zero in state  $q_f$  whenever a computation terminates.
- Move the input one space to the right. Input configuration  $q_0BuB$ , result  $q_fBBuB$ .
  - Concatenate a copy of the reversed input string to the input. Input configuration  $q_0BuB$ , result  $q_fBuu^RB$ .
  - Insert a blank between each of the input symbols. For example, input configuration  $q_0BabaB$ , result  $q_fBaBbBaB$ .
  - Erase the  $b$ 's from the input. For example, input configuration  $q_0BbabaahabB$ , result  $q_fBaaaB$ .
4. Construct a Turing machine with input alphabet  $\{a, b, c\}$  that accepts strings in which the first  $c$  is preceded by the substring  $aaa$ . A string must contain a  $c$  to be accepted by the machine.
5. Construct a Turing machine with input alphabet  $\{a, b\}$  to accept each of the following languages by final state.
- $\{a^i b^j \mid i \geq 0, j \geq i\}$
  - $\{a^i b^j a^i b^j \mid i, j > 0\}$
  - Strings with the same number of  $a$ 's and  $b$ 's
  - $\{uu^R \mid u \in \{a, b\}^*\}$
  - $\{uu \mid u \in \{a, b\}^*\}$
6. Modify your solution to Exercise 5(a) to obtain a Turing machine that accepts the language  $\{a^i b^j \mid i \geq 0, j \geq i\}$  by halting.
7. An alternative method of acceptance by final state can be defined as follows: A string  $u$  is accepted by a Turing machine M if the computation of M with input  $u$  enters

(but does not necessarily terminate in) a final state. With this definition, a string may be accepted even though the computation of the machine does not terminate. Prove that the languages accepted by this definition are precisely the recursively enumerable languages.

8. The transitions of a one-tape deterministic Turing machine may be defined by a partial function from  $Q \times \Gamma$  to  $Q \times \Gamma \times \{L, R, S\}$ , where  $S$  indicates that the tape head remains stationary. Prove that machines defined in this manner accept precisely the recursively enumerable languages.
9. An **atomic** Turing machine is one in which every transition consists of a change of state and one other action. The transition may write on the tape or move the tape head, but not both. Prove that the atomic Turing machines accept precisely the recursively enumerable languages.
- \* 10. A **context-sensitive** Turing machine is one in which the applicability of a transition is determined not only by the symbol scanned but also by the symbol in the tape square to the right of the tape head. A transition has the form

$$\delta(q_i, xy) = [q_j, z, d] \quad x, y, z \in \Gamma; \quad d \in \{L, R\}.$$

When the machine is in state  $q_i$  scanning an  $x$ , the transition may be applied only when the tape position to the immediate right of the tape head contains a  $y$ . In this case the  $x$  is replaced by  $z$ , the machine enters state  $q_j$ , and the tape head moves in direction  $d$ .

- a) Let  $M$  be a standard Turing machine. Define a context-sensitive Turing machine  $M'$  that accepts  $L(M)$ . Hint: Define the transition function of  $M'$  from that of  $M$ .
- b) Let  $\delta(q_i, xy) = [q_j, z, d]$  be a context-sensitive transition. Show that the result of the application of this transition can be obtained by a sequence of standard Turing machine transitions. You must consider the case both when transition  $\delta(q_i, xy)$  is applicable and when it isn't.
- c) Use parts (a) and (b) to conclude that context-sensitive machines accept precisely the recursively enumerable languages.
11. Prove that every recursively enumerable language is accepted by a Turing machine with a single accepting state.
12. Construct a Turing machine with two-way tape and input alphabet  $\{a\}$  that halts if the tape contains a nonblank square. The symbol  $a$  may be anywhere on the tape, not necessarily to the immediate right of the tape head.
13. A **two-dimensional** Turing machine is one in which the tape consists of a two-dimensional array of tape squares.

A tra  
adjac  
The t  
d is l  
with

14. Let L

- a) B  
b) B  
sh

15. Cons  
guage  
right.

16. Const  
langu

17. Const  
with t  
more

18. Const  
by an

where

19. Const  
 $\{a, b\}$   
i) le  
ii) u

20. Constr  
 $\{a, b\}^5$

in, a string may terminate. Prove it is not recursively enumerable.

ined by a partial function. At the tape head position precisely the

s of a change of state of the tape head, given the recursively

of a transition is given on the tape square

plied only when  $y = x$ . In this case the tape moves in direction  $d$ . Using machine  $M'$  that of  $M$ .

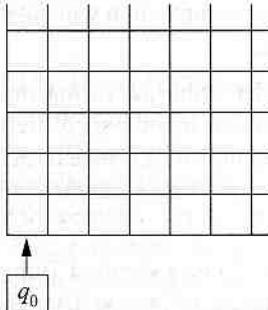
that the result of a standard Turing computation  $\delta(q_i, xy)$  is

s accept precisely

ring machine with

et  $\{a\}$  that halts if there are no  $a$ 's on the tape, not

consists of a two-



A transition consists of rewriting a square and moving the head to any one of the four adjacent squares. A computation begins with the tape head reading the corner position. The transitions of the two-dimensional machine are written  $\delta(q_i, x) = [q_j, y, d]$ , where  $d$  is  $U$  (up),  $D$  (down),  $L$  (left), or  $R$  (right). Design a two-dimensional Turing machine with input alphabet  $\{a\}$  that halts if the tape contains a nonblank square.

14. Let  $L$  be the set of palindromes over  $\{a, b\}$ .
  - a) Build a standard Turing machine that accepts  $L$ .
  - b) Build a two-tape machine that accepts  $L$  in which the computation with input  $u$  should take no more than  $3 \text{length}(u) + 4$  transitions.
15. Construct a two-tape Turing machine with input alphabet  $\{a, b\}$  that accepts the language  $\{a^i b^{2i} \mid i \geq 0\}$  in which the tape head on the input tape only moves from left to right.
16. Construct a two-tape Turing machine with input alphabet  $\{a, b, c\}$  that accepts the language  $\{a^i b^i c^i \mid i \geq 0\}$ .
17. Construct a two-tape Turing machine with input alphabet  $\{a, b\}$  that accepts strings with the same number of  $a$ 's and  $b$ 's. The computation with input  $u$  should take no more than  $2 \text{length}(u) + 3$  transitions.
18. Construct a two-tape Turing machine that accepts strings in which each  $a$  is followed by an increasing number of  $b$ 's; that is, the strings are of the form

$$ab^{n_1}ab^{n_2}\dots ab^{n_k}, k > 0,$$

where  $n_1 < n_2 < \dots < n_k$ .

19. Construct a nondeterministic Turing machine whose language is the set of strings over  $\{a, b\}$  that contain a substring  $u$  satisfying the following two properties:
  - i)  $\text{length}(u) \geq 3$ ;
  - ii)  $u$  contains the same number of  $a$ 's and  $b$ 's.
20. Construct a two-tape nondeterministic Turing machine that accepts  $L = \{uvuw \mid u \in \{a, b\}^5, v, w \in \{a, b\}^*\}$ . A string is in  $L$  if it contains two nonoverlapping identical

substrings of length 5. Every computation with input  $w$  should terminate after at most  $2 \text{length}(w) + 2$  transitions.

21. Construct a two-tape nondeterministic Turing machine that accepts  $L = \{uu \mid u \in \{a, b\}^*\}$ . Every computation with input  $w$  should terminate after at most  $2 \text{length}(w) + 2$  transitions. Using the deterministic machine from Example 8.6.2 that accepts  $L$ , what is the maximum number of transitions required for a computation with an input of length  $n$ ?
22. Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a standard Turing machine that accepts a language  $L$ . Design a Turing machine  $M'$  (of any variety) that accepts a string  $w \in \Sigma^*$  if, and only if, there is a substring of  $w$  in  $L$ .
23. Let  $L$  be a language accepted by a nondeterministic Turing machine in which every computation terminates. Prove that  $L$  is recursive.
24. Prove the equivalent of Theorem 8.3.2 for nondeterministic Turing machines.
25. Prove that every finite language is recursive.
26. Prove that a language  $L$  is recursive if, and only if,  $L$  and  $\bar{L}$  are recursively enumerable.
27. Prove that the recursive languages are closed under union, intersection, and complement.
28. A machine that generates all sequences made up of integers from 1 to  $n$  was given in Figure 8.1. Trace the first seven cycles of the machine for  $n = 3$ . A cycle consists of the tape head returning to the initial position in state  $q_0$ .
29. Build a Turing machine that enumerates the set of even length strings over  $\{a\}$ .
30. Build a Turing machine that enumerates the set  $\{a^i b^j \mid 0 \leq i \leq j\}$ .
31. Build a Turing machine that enumerates the set  $\{a^{2^n} \mid n \geq 0\}$ .
32. Build a Turing machine  $E_{\Sigma^*}$  that enumerates  $\Sigma^*$  where  $\Sigma = \{0, 1\}$ . Note: This machine may be thought of as enumerating all finite-length bit strings.
- \* 33. Build a machine that enumerates the ordered pairs  $N \times N$ . Represent a number  $n$  by a string of  $n + 1$  1's. The output for ordered pair  $[i, j]$  should consist of the representation of the number  $i$  followed by a blank followed by the representation of  $j$ . The markers # should surround the entire ordered pair.
34. In Theorem 8.8.7, the proof that every recursive language can be enumerated in lexicographical order considered the cases of finite and infinite languages separately. The argument for an infinite language may not be sufficient for a finite language. Why?
35. Define the components of a two-track nondeterministic Turing machine. Prove that these machines accept precisely the recursively enumerable languages.
36. Prove that every context-free language is recursive. Hint: Construct a two-tape nondeterministic Turing machine that simulates the computation of a pushdown automaton.

## Bibliography

The Turing  
tation. Turi  
single tape L  
the same co  
The use  
The capabil  
Chapters 10  
[1974], and

---

## Bibliographic Notes

ter at most

$\{uu \mid u \in$   
 $ngth(w) +$   
 $pts L$ , what  
 $ut of length$

a language  
 $\Sigma^*$  if, and

which every

nes.

enumerable.  
 and comple-

was given in  
 e consists of

er  $\{a\}$ .

This machine

umber  $n$  by a  
 epresentation  
 The markers

numerated in  
 es separately.  
 iaguage. Why?

ne. Prove that

o-tape nonde-  
 vn automaton.

The Turing machine was introduced by Turing [1936] as a model for algorithmic computation. Turing's original machine was deterministic, consisting of a two-way tape and a single tape head. Independently, Post [1936] introduced a family of abstract machines with the same computational capabilities as Turing machines.

The use of Turing machines for the computation of functions is presented in Chapter 9. The capabilities and limitations of Turing machines as language acceptors are examined in Chapters 10 and 11. The books by Kleene [1952], Minsky [1967], Brainerd and Landweber [1974], and Hennie [1977] give an introduction to computability and Turing machines.

## CHAPTER 9

# Turing Computable Functions

In the preceding chapter Turing machines provided the computational framework for accepting languages. The result of a computation was determined by final state or by halting. In either case there are only two possible outcomes: accept or reject. The result of a Turing machine computation can also be defined in terms of the symbols written on the tape when the computation terminates. Defining the result in terms of the halting tape configuration permits an infinite number of possible outcomes. In this manner, the computations of a Turing machine produce a mapping between input strings and output strings; that is, the Turing machine computes a function. When the strings are interpreted as natural numbers, Turing machines can be used to compute number-theoretic functions. We will show that several important number-theoretic functions are Turing computable and that computability is closed under the composition of functions. In Chapter 13 we will categorize the entire family of functions that can be computed by Turing machines.

The current chapter ends by outlining how a high-level programming language could be defined using the Turing machine architecture. This brings Turing machine computations closer to the computational paradigm with which we are most familiar—the modern-day computer.

### 9.1 Computation of Functions

A function  $f : X \rightarrow Y$  is a mapping that assigns at most one value from the set  $Y$  to each element of the domain  $X$ . Adopting a computational viewpoint, we refer to the variables of  $f$  as the input of the function. The definition of a function does not specify how to obtain

$f(x)$ , the value assigned to  $x$  by the function  $f$ , from the input  $x$ . Turing machines will be designed to compute the values of functions. The domain and range of a function computed by a Turing machine consist of strings over the input alphabet of the machine.

A Turing machine that computes a function has two distinguished states: the initial state  $q_0$  and the halting state  $q_f$ . A computation begins with a transition from state  $q_0$  that positions the tape head at the beginning of the input string. The state  $q_0$  is never reentered; its sole purpose is to initiate the computation. All computations that terminate do so in state  $q_f$  with the value of the function written on the tape beginning at position one. These conditions are formalized in Definition 9.1.1.

### Definition 9.1.1

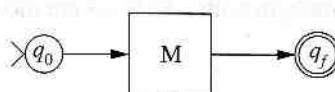
A deterministic one-tape Turing machine  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$  computes the unary function  $f : \Sigma^* \rightarrow \Sigma^*$  if

- i) there is only one transition from the state  $q_0$  and it has the form  $\delta(q_0, B) = [q_i, B, R]$ ;
- ii) there are no transitions of the form  $\delta(q_i, x) = [q_0, y, d]$  for any  $q_i \in Q, x, y \in \Gamma$ , and  $d \in \{L, R\}$ ;
- iii) there are no transitions of the form  $\delta(q_f, B)$ ;
- iv) the computation with input  $u$  halts in the configuration  $q_f B v B$  whenever  $f(u) = v$ ; and
- v) the computation continues indefinitely whenever  $f(u) \uparrow$ .

A function is said to be **Turing computable** if there is a Turing machine that computes it. A Turing machine that computes a function  $f$  may fail to halt for an input string  $u$ . In this case,  $f$  is undefined for  $u$ . Thus Turing machines can compute both total and partial functions.

An arbitrary function need not have the same domain and range. Turing machines can be designed to compute functions from  $\Sigma^*$  to a specific set  $R$  by designating an input alphabet  $\Sigma$  and a range  $R$ . Condition (iv) is then interpreted as requiring the string  $v$  to be an element of  $R$ .

To highlight the distinguished states  $q_0$  and  $q_f$ , a Turing machine  $M$  that computes a function is depicted by the diagram



Intuitively, the computation remains inside the box labeled  $M$  until termination. This diagram is somewhat simplistic since Definition 9.1.1 permits multiple transitions to state  $q_f$  and transitions from  $q_f$ . However, condition (iii) ensures that there are no transitions from  $q_f$  when the machine is scanning a blank. When this occurs, the computation terminates with the result written on the tape.

Exam  
The T

compu

Th  
moves  
state q2  
input w  
configuThe  
should b  
ments o  
of M ini  
ing confi  
correct p  
out" priFun  
is placed  
computaIf  $f(abc)$   
 $q_f B f(ab$ The conse  
is the null

ies will be computed

the initial state  $q_0$  that reentered; it do so in one. These

es the unary

$= [q_i, B, R];$   
 $, y \in \Gamma$ , and

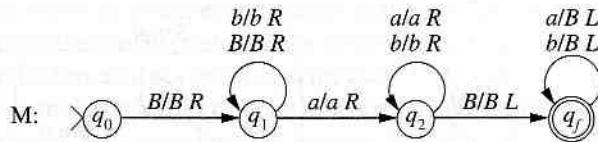
er  $f(u) = v$ ;  
hat computes  
it string  $u$ . In  
al and partial

machines can be  
input alphabet  
be an element  
at computes a

mination. This  
ions to state  $q_f$   
transitions from  
tion terminates

### Example 9.1.1

The Turing machine



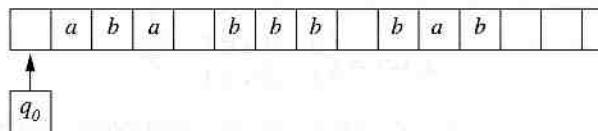
computes the partial function  $f$  from  $\{a, b\}^*$  to  $\{a, b\}^*$  defined by

$$f(u) = \begin{cases} \lambda & \text{if } u \text{ contains an } a \\ \uparrow & \text{otherwise.} \end{cases}$$

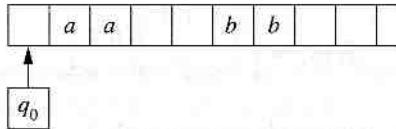
The function  $f$  is undefined if the input does not contain an  $a$ . In this case the machine moves indefinitely to the right in state  $q_1$ . When an  $a$  is encountered, the machine enters state  $q_2$  and reads the remainder of the input. The computation is completed by erasing the input while returning to the initial position. A computation that terminates produces the configuration  $q_f BB$  designating the null string as the result.  $\square$

The machine  $M$  in Example 9.1.1 was designed to compute the unary function  $f$ . It should be neither surprising nor alarming that computations of  $M$  do not satisfy the requirements of Definition 9.1.1 when the input does not have the anticipated form. A computation of  $M$  initiated with input  $BbBbBaB$  terminates in the configuration  $BbBbq_f B$ . In this halting configuration, the tape does not contain a single value and the tape head is not in the correct position. This is just another manifestation of the time-honored “garbage in, garbage out” principle of computer science.

Functions with more than one argument are computed in a similar manner. The input is placed on the tape with the arguments separated by blanks. The initial configuration of a computation of a ternary function  $f$  with input  $aba$ ,  $bbb$ , and  $bab$  is



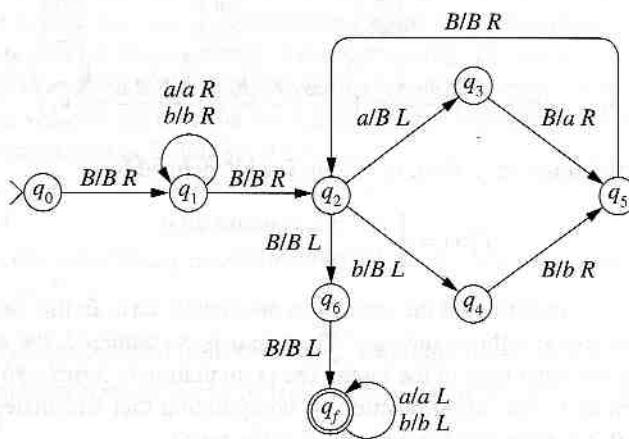
If  $f(aba, bbb, bab)$  is defined, the computation terminates with the configuration  $q_f Bf(aba, bbb, bab)B$ . The initial configuration for the computation of  $f(aa, \lambda, bb)$  is



The consecutive blanks in tape positions three and four indicate that the second argument is the null string.

**Example 9.1.2**

The Turing machine



computes the binary function of concatenation of strings over  $\{a, b\}$ . The initial configuration of a computation with input strings  $u$  and  $v$  has the form  $q_0BuBvB$ . Either or both of the input strings may be null.

The initial string is read in state  $q_1$ . The cycle formed by states  $q_2, q_3, q_5, q_2$  translates an  $a$  one position to the left. Similarly,  $q_2, q_4, q_5, q_2$  shift a  $b$  to the left. These cycles are repeated until the entire second argument has been translated one position to the left, producing the configuration  $q_fBuBvB$ .  $\square$

Turing machines that compute functions can also be used to accept languages. The **characteristic function** of a language  $L$  is a function  $\chi_L : \Sigma^* \rightarrow \{0, 1\}$  defined by

$$\chi_L(u) = \begin{cases} 1 & \text{if } u \in L \\ 0 & \text{if } u \notin L. \end{cases}$$

A language  $L$  is recursive if there is a Turing machine  $M$  that computes the characteristic function  $\chi_L$ . The results of the computations of  $M$  indicate the acceptability of strings. A machine that computes the partial characteristic function

$$\hat{\chi}_L(u) = \begin{cases} 1 & \text{if } u \in L \\ 0 \text{ or } \uparrow & \text{if } u \notin L \end{cases}$$

shows that  $L$  is recursively enumerable. Exercises 2, 3, and 4 establish the equivalence between acceptance of a language by a Turing machine and the computability of its characteristic function.

We have seen that a many-one reduction maps a domain and range into one another. A number-theoretic function consists of a set of numbers defined by some rule, such as addition and multiplication.

The transition function of a Turing machine is represented by a directed graph. This is known as the state transition diagram of the machine. The input alphabet is the set  $\{I\}$ .

The computation of a function is a theoretical function.

If  $f(2, 0, 3) = 1$ ,

A  $k$ -variable function is called a  $k$ -ary relation.

The function  $\chi_L$  is computable if  $L$  is recursive.

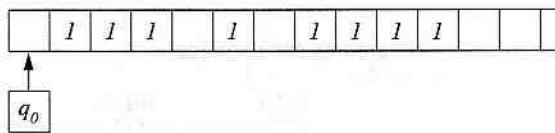
We will now show that every number-theoretic function is computable. We start with a corresponding machine.

## 9.2 Numeric Computation

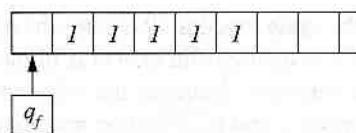
We have seen that Turing machines can be used to compute the values of functions whose domain and range consist of strings over the input alphabet. In this section we turn our attention to numeric computation, in particular the computation of number-theoretic functions. A **number-theoretic function** is a function of the form  $f : \mathbf{N} \times \mathbf{N} \times \dots \times \mathbf{N} \rightarrow \mathbf{N}$ . The domain consists of natural numbers or  $n$ -tuples of natural numbers. The function  $sq : \mathbf{N} \rightarrow \mathbf{N}$  defined by  $sq(n) = n^2$  is a unary number-theoretic function. The standard operations of addition and multiplication are binary number-theoretic functions.

The transition from symbolic to numeric computation requires only a change of perspective since numbers are represented by strings of symbols. The input alphabet of the Turing machine is determined by the representation of the natural numbers used in the computation. We will represent the natural number  $n$  by the string  $I^{n+1}$ . The number zero is represented by the string  $I$ , the number one by  $II$ , and so on. This notational scheme is known as the *unary representation* of the natural numbers. The unary representation of a natural number  $n$  is denoted  $\bar{n}$ . When numbers are encoded using the unary representation, the input alphabet for a machine that computes a number-theoretic function is the singleton set  $\{I\}$ .

The computation of  $f(2, 0, 3)$  in a Turing machine that computes a ternary number-theoretic function  $f$  begins with the machine configuration



If  $f(2, 0, 3) = 4$ , the computation terminates with the configuration



A  $k$ -variable total number-theoretic function  $r : \mathbf{N} \times \mathbf{N} \times \dots \times \mathbf{N} \rightarrow \{0, 1\}$  defines a  $k$ -ary relation  $R$  on the domain of the function. The relation is defined by

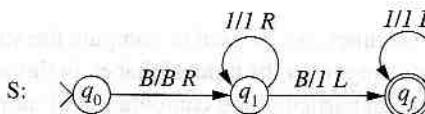
$$[n_1, n_2, \dots, n_k] \in R \text{ if } r(n_1, n_2, \dots, n_k) = 1$$

$$[n_1, n_2, \dots, n_k] \notin R \text{ if } r(n_1, n_2, \dots, n_k) = 0.$$

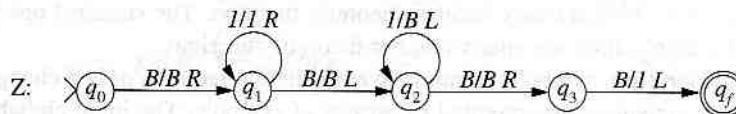
The function  $r$  is called the **characteristic function** of the relation  $R$ . A relation is Turing computable if its characteristic function is Turing computable.

We will now construct Turing machines that compute several simple, but important, number-theoretic functions. The functions are denoted by lowercase letters and the corresponding machines by capital letters.

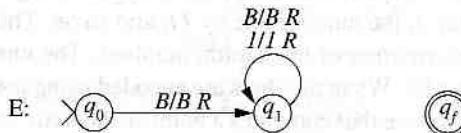
The successor function:  $s(n) = n + 1$



The zero function:  $z(n) = 0$

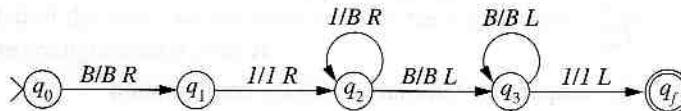


The empty function:  $e(n) \uparrow$



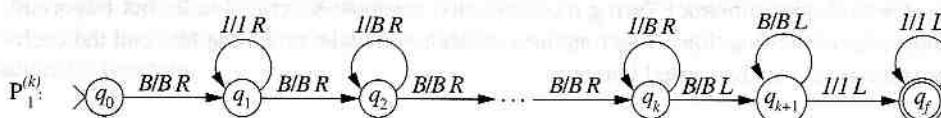
The machine that computes the successor simply adds a  $I$  to the right end of the input string. The zero function is computed by erasing the input and writing  $I$  in tape position one. The empty function is undefined for all arguments; the machine moves indefinitely to the right in state  $q_1$ .

The zero function is also computed by the machine



That two machines compute the same function illustrates the difference between functions and algorithms. A function is a mapping from elements in the domain to elements in the range. A Turing machine mechanically computes the value of the function whenever the function is defined. The difference is that of definition and computation. In Section 9.5 we will see that there are number-theoretic functions that cannot be computed by any Turing machine.

The value of the  $k$ -variable projection function  $p_i^{(k)}$  is defined as the  $i$ th argument of the input,  $p_i^{(k)}(n_1, n_2, \dots, n_i, \dots, n_k) = n_i$ . The superscript  $k$  specifies the number of arguments and the subscript designates the argument that defines the result of the projection. The superscript is placed in parentheses so that it is not mistaken for an exponent. The machine that computes  $p_1^{(k)}$  leaves the first argument unchanged and erases the remaining arguments.



The function *function* and is example 9.3.1.

### Example 9.2.1

The Turing machine numbers.

A: 

The unary representation of numbers is represented by the arguments w.

### Example 9.2.2

The predecessor

is computed by t

D: 

For input greater than

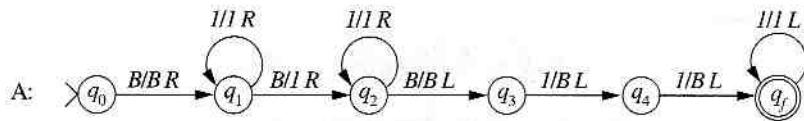
### 9.3 Sequen

Turing machines that perform running the machine the succeeding m

The function  $p_1^{(1)}$  maps a single input to itself. This function is also called the *identity function* and is denoted *id*. Machines  $P_i^{(k)}$  that compute  $p_i^{(k)}$  will be designed in Example 9.3.1.

### Example 9.2.1

The Turing machine A computes the binary function defined by the addition of natural numbers.



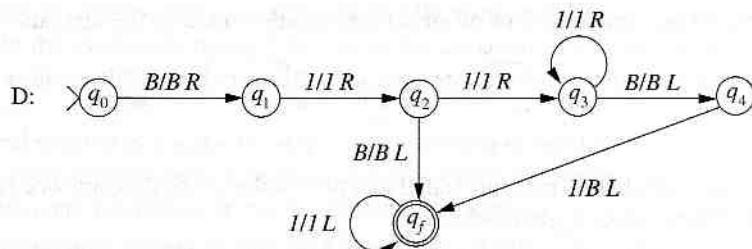
The unary representations of natural numbers  $n$  and  $m$  are  $I^{n+1}$  and  $I^{m+1}$ . The sum of these numbers is represented by  $I^{n+m+1}$ . This string is generated by replacing the blank between the arguments with a  $I$  and erasing two  $I$ 's from the right end of the second argument.  $\square$

### Example 9.2.2

The predecessor function

$$\text{pred}(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$$

is computed by the machine D (decrement):



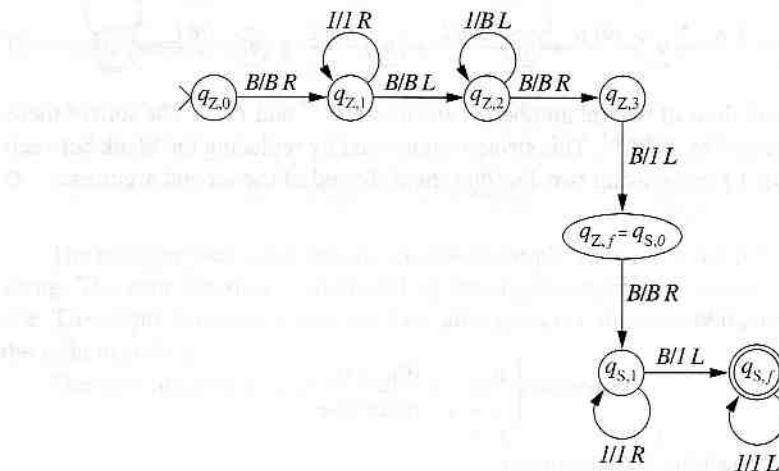
For input greater than zero, the computation erases the rightmost  $I$  on the tape.  $\square$

## 9.3 Sequential Operation of Turing Machines

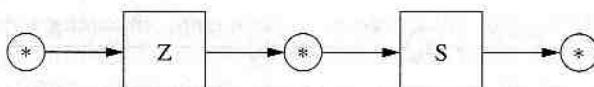
Turing machines designed to accomplish a single task can be combined to construct machines that perform complex computations. Intuitively, the combination is obtained by running the machines sequentially. The result of one computation becomes the input for the succeeding machine. A machine that computes the constant function  $c(n) = 1$  can be

constructed by combining the machines that compute the zero and the successor functions. Regardless of the input, a computation of the machine Z terminates with the value zero on the tape. Running the machine S on this tape configuration produces the number one.

The computation of Z terminates with the tape head in position zero scanning a blank. These are precisely the input conditions for the machine S. The initiation and termination conditions of Definition 9.1.1 were introduced to facilitate this coupling of machines. The handoff between machines is accomplished by identifying the final state of Z with the initial state of S. Except for this handoff, the states of the two machines are assumed to be distinct. This can be ensured by subscripting each state of the composite machine with the name of the original machine.



The sequential combination of two machines is represented by the diagram



The state names are omitted from the initial and final nodes in the diagram since they may be inferred from the constituent machines.

There are certain sequences of actions that frequently occur in a computation of a Turing machine. Machines can be constructed to perform these recurring tasks. These machines are designed in a manner that allows them to be used as components in more complicated machines. Borrowing terminology from assembly language programming, we call a machine constructed to perform a single simple task a **macro**.

The computations of a macro adhere to several of the restrictions introduced in Definition 9.1.1. The initial state  $q_0$  is used strictly to initiate the computation. Since these machines are combined to construct more complex machines, we do not assume that a computation must begin with the tape head at position zero. We do assume, however, that each computation begins with the machine scanning a blank. Depending upon the operation, the

segment of the computation. As with machine of the form  $\delta(q,$

A family of head to the right is defined by the

$MR_k$  is construct numbers.

$MR_k: \times(q_0)$

The move macro computation of M the configuration

Macros, like having a specified numbers to the in of a composite m each macro.

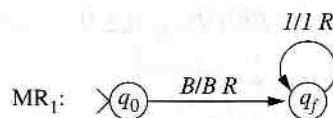
Several fami machine. The con by the initial and access nor alter a is indicated by th before and after c

$ML_k$  (move left):

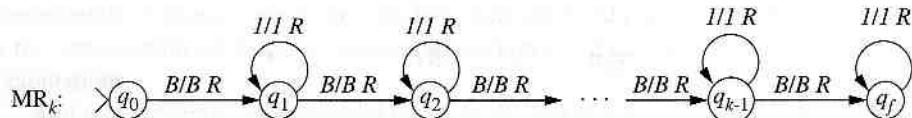
functions.  
be zero on  
one.  
g a blank.  
mination  
ines. The  
the initial  
e distinct.  
e name of

segment of the tape to the immediate right or left of the tape head will be examined by the computation. A macro may contain several states in which a computation may terminate. As with machines that compute functions, a macro is not permitted to contain a transition of the form  $\delta(q_f, B)$  from any halting state  $q_f$ .

A family of macros is often described by a schema. The macro  $MR_i$  moves the tape head to the right through  $i$  consecutive natural numbers (sequences of 1's) on the tape.  $MR_1$  is defined by the machine



$MR_k$  is constructed by adding states to move the tape head through the sequence of  $k$  natural numbers.



The move macros do not affect the tape to the left of the initial position of the tape head. A computation of  $MR_2$  that begins with the configuration  $B\bar{n}_1q_0B\bar{n}_2B\bar{n}_3B\bar{n}_4B$  terminates in the configuration  $B\bar{n}_1B\bar{n}_2B\bar{n}_3q_fB\bar{n}_4B$ .

Macros, like Turing machines that compute functions, expect to be run with the input having a specified form. The move right macro  $MR_i$  requires a sequence of at least  $i$  natural numbers to the immediate right of the tape at the initiation of a computation. The design of a composite machine must ensure that the appropriate input configuration is provided to each macro.

Several families of macros are defined by describing the results of a computation of the machine. The computation of each macro remains within the segment of the tape indicated by the initial and final blank in the description. The application of the macro will neither access nor alter any portion of tape outside of these bounds. The location of the tape head is indicated by the underscore. The double arrows indicate identical tape positions in the before and after configurations.

$ML_k$  (move left):

$$\begin{array}{ccc} B\bar{n}_1B\bar{n}_2B \dots B\bar{n}_kB & & k \geq 0 \\ \uparrow & & \uparrow \\ B\bar{n}_1B\bar{n}_2B \dots B\bar{n}_kB & & \end{array}$$

FR (find right):

$$\begin{array}{c} \underline{B} B^i \bar{n} B \quad i \geq 0 \\ \uparrow \quad \downarrow \\ B^i \underline{\bar{n} B} \end{array}$$

FL (find left):

$$\begin{array}{c} B \bar{n} B^i \underline{B} \quad i \geq 0 \\ \uparrow \quad \downarrow \\ \underline{B} \bar{n} B^i B \end{array}$$

 $E_k$  (erase):

$$\begin{array}{c} \underline{B} \bar{n}_1 B \bar{n}_2 B \dots B \bar{n}_k B \quad k \geq 1 \\ \uparrow \quad \uparrow \\ \underline{B} B \quad \dots \quad B B \end{array}$$

 $CPY_k$  (copy):

$$\begin{array}{c} \underline{B} \bar{n}_1 B \bar{n}_2 B \dots B \bar{n}_k B B B \quad \dots \quad B B \quad k \geq 1 \\ \uparrow \quad \uparrow \quad \uparrow \\ \underline{B} \bar{n}_1 B \bar{n}_2 B \dots B \bar{n}_k B \bar{n}_1 B \bar{n}_2 B \dots B \bar{n}_k B \end{array}$$

 $CPY_{k,i}$  (copy through  $i$  numbers):

$$\begin{array}{c} \underline{B} \bar{n}_1 B \bar{n}_2 B \dots B \bar{n}_k B \bar{n}_{k+1} \dots B \bar{n}_{k+i} B B \quad \dots \quad B B \quad k \geq 1 \\ \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \\ \underline{B} \bar{n}_1 B \bar{n}_2 B \dots B \bar{n}_k B \bar{n}_{k+1} \dots B \bar{n}_{k+i} B \bar{n}_1 B \bar{n}_2 B \dots B \bar{n}_k B \end{array}$$

T (translate):

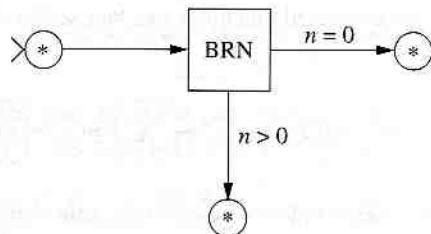
$$\begin{array}{c} \underline{B} B^i \bar{n} B \quad i \geq 0 \\ \uparrow \quad \downarrow \\ \underline{B} \bar{n} B^i B \end{array}$$

The find macros move the tape head into a position to process the first natural number to the right or left of the current position.  $E_k$  erases a sequence of  $k$  natural numbers and halts with the tape head in its original position.

The copy machines produce a copy of the designated number of integers. The segment of the tape on which the copy is produced is assumed to be blank.  $CPY_{k,i}$  expects a sequence

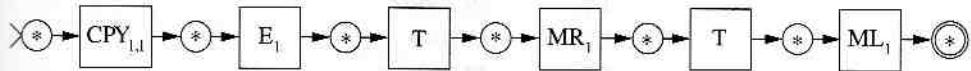
of  $k + i$  numbers followed by a blank segment large enough to hold a copy of the first  $k$  numbers. The translate macro changes the location of the first natural number to the right of the tape head. A computation terminates with the head in the position it occupied at the beginning of the computation with the translated string to its immediate right.

The BRN (branch on zero) macro has two possible terminating states. The input to the macro BRN, a single natural number, is used to select the halting state of the macro. The branch macro is depicted



The computation of BRN does not alter the tape nor change the position of the tape head. Consequently, it may be run in any configuration  $B\bar{n}B$ . The branch macro is often used in the construction of loops in composite machines and in the selection of alternative computations.

Additional macros can be created using those already defined. The machine



interchanges the order of two numbers. The tape configurations for this macro are INT (interchange):

$$\underline{B\bar{n}B\bar{m}BB^{n+1}B}$$

↑              ↓

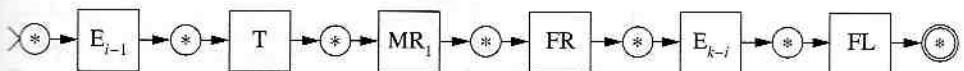
$$\underline{B\bar{m}B\bar{n}BB^{n+1}B}$$

$$k \geq 1$$

In Exercise 6, you are asked to construct a Turing machine for the macro INT that does not leave the tape segment  $B\bar{n}B\bar{m}B$ .

### Example 9.3.1

The computation of a machine that evaluates the projection function  $p_i^{(k)}$  consists of three distinct actions: erasing the initial  $i - 1$  arguments, translating the  $i$ th argument to tape position one, and erasing the remainder of the input. A machine to compute  $p_i^{(k)}$  can be designed using the macros FR, FL, E<sub>i</sub>, MR<sub>1</sub>, and T.



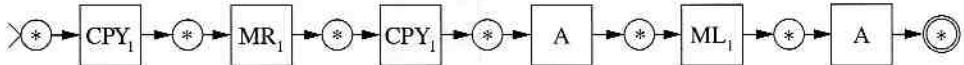
□

st natural number to  
al numbers and halts  
ntegers. The segment  
, expects a sequence

Turing machines defined to compute functions can be used like macros in the design of composite machines. Unlike the computations of the macros, there is no a priori bound on the amount of tape required by a computation of such a machine. Consequently, these machines should be run only when the input is followed by a completely blank tape.

### Example 9.3.2

The macros and previously constructed machines can be used to design a Turing machine that computes the function  $f(n) = 3n$ .



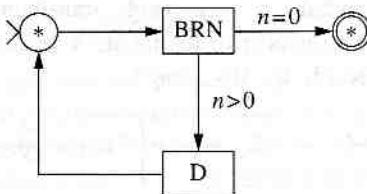
The machine A, constructed in Example 9.2.1, adds two natural numbers. The computation of  $f(n)$  combines the copy macro with A to add three copies of  $n$ . A computation with input  $\bar{n}$  generates the following sequence of tape configurations.

| Machine          | Configuration                                                                                            |
|------------------|----------------------------------------------------------------------------------------------------------|
| CPY <sub>1</sub> | <u>B</u> <u><math>\bar{n}</math></u> <u>B</u>                                                            |
| MR <sub>1</sub>  | <u>B</u> <u><math>\bar{n}</math></u> <u><math>\bar{n}</math></u> <u>B</u>                                |
| CPY <sub>1</sub> | <u>B</u> <u><math>\bar{n}</math></u> <u><math>\bar{n}</math></u> <u><math>\bar{n}</math></u> <u>B</u>    |
| A                | <u>B</u> <u><math>\bar{n}</math></u> <u><math>\bar{n}</math></u> <u><math>\bar{n}</math></u> <u>+ nB</u> |
| ML <sub>1</sub>  | <u>B</u> <u><math>\bar{n}</math></u> <u><math>B</math></u> <u><math>n</math></u> <u>B</u>                |
| A                | <u>B</u> <u><math>n</math></u> <u><math>+ n</math></u> <u>B</u>                                          |

Note that the addition machine A is run only when its arguments are the two rightmost encoded numbers on the tape. □

### Example 9.3.3

The one-variable constant function zero defined by  $z(n) = 0$ , for all  $n \in \mathbb{N}$ , can be built from the BRN macro and the machine D that computes the predecessor function.

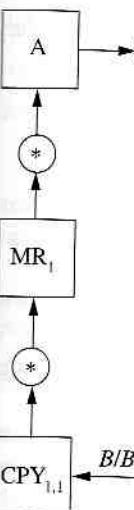


### Example 9.3.3

A Turing machine  
Macros can  
machine. Th  
upon the pro  
with a state

Since the ma  
defined for s

MULT: X



in the design priori bound  
quently, these  
ik tape.

ring machine



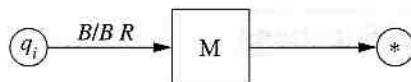
e computation  
ion with input

two rightmost

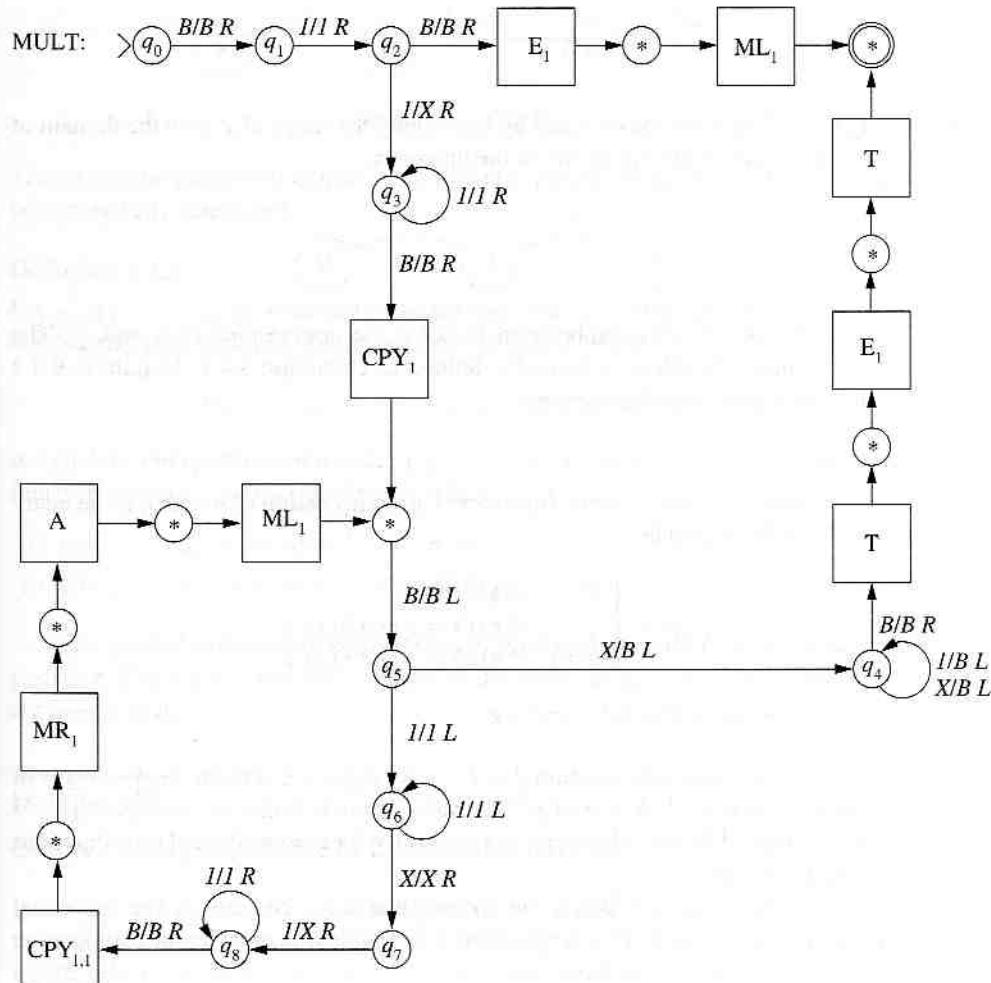
an be built from  
b:

**Example 9.3.4**

A Turing machine  $MULT$  is constructed to compute the multiplication of natural numbers. Macros can be mixed with standard Turing machine transitions when designing a composite machine. The conditions on the initial state of a macro permit the submachine to be entered upon the processing of a blank from any state. The identification of the start state of a macro with a state  $q_i$  is depicted



Since the macro is entered only upon the processing of a blank, transitions may also be defined for state  $q_i$  with the tape head scanning nonblank tape symbols.



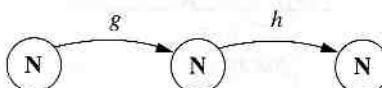
If the first argument is zero, the computation erases the second argument, returns to the initial position, and halts. Otherwise, a computation of MULT adds  $m$  to itself  $n$  times. The addition is performed by copying  $\bar{m}$  and then adding the copy to the previous total. The number of iterations is recorded by replacing a 1 in the first argument with an  $X$  when a copy is made.  $\square$

## 9.4 Composition of Functions

Using the interpretation of a function as a mapping from its domain to its range, we can represent the unary number-theoretic functions  $g$  and  $h$  by the diagrams



A mapping from  $N$  to  $N$  can be obtained by identifying the range of  $g$  with the domain of  $h$  and sequentially traversing the arrows in the diagrams.



The function obtained by this combination is called the composition of  $h$  with  $g$ . The composition of unary functions is formally defined in Definition 9.4.1. Definition 9.4.2 extends the notion to  $n$ -variable functions.

### Definition 9.4.1

Let  $g$  and  $h$  be unary number-theoretic functions. The **composition** of  $h$  with  $g$  is the unary function  $f : N \rightarrow N$  defined by

$$f(x) = \begin{cases} \uparrow & \text{if } g(x) \uparrow \\ \uparrow & \text{if } g(x) = y \text{ and } h(y) \uparrow \\ h(y) & \text{if } g(x) = y \text{ and } h(y) \downarrow. \end{cases}$$

The composite function is denoted  $f = h \circ g$ .

The value of the composite function  $f = h \circ g$  for input  $x$  is written  $f(x) = h(g(x))$ . The latter expression is read “ $h$  of  $g$  of  $x$ .” The value  $h(g(x))$  is defined whenever  $g(x)$  is defined and  $h$  is defined for the value  $g(x)$ . Consequently, the composition of total functions produces a total function.

From a computational viewpoint, the composition  $h \circ g$  consists of the sequential evaluation of functions  $g$  and  $h$ . The computation of  $g$  provides the input for the computation of  $h$ :

The composite function can be successfully computed.

### Definition 9.4.2

Let  $g_1, g_2, \dots, g_n$  be unary number-theoretic functions. The composite function

$f(x)$

is called the composite function  $f(x_1, \dots, x_n)$ .

- i)  $g_i(x_1, \dots, x_k)$
- ii)  $g_i(x_1, \dots, x_k)$

The general definition of the composite function  $f(x_1, \dots, x_n)$  is given by the interpretation. The input arguments of  $h$  are the outputs of  $g_1, g_2, \dots, g_{n-1}$ .

### Example 9.4.1

Consider the mapping

where  $add(n, m) =$

, returns to itself  $n$  times. Its total. The  $X$  when a  $\square$

range, we can

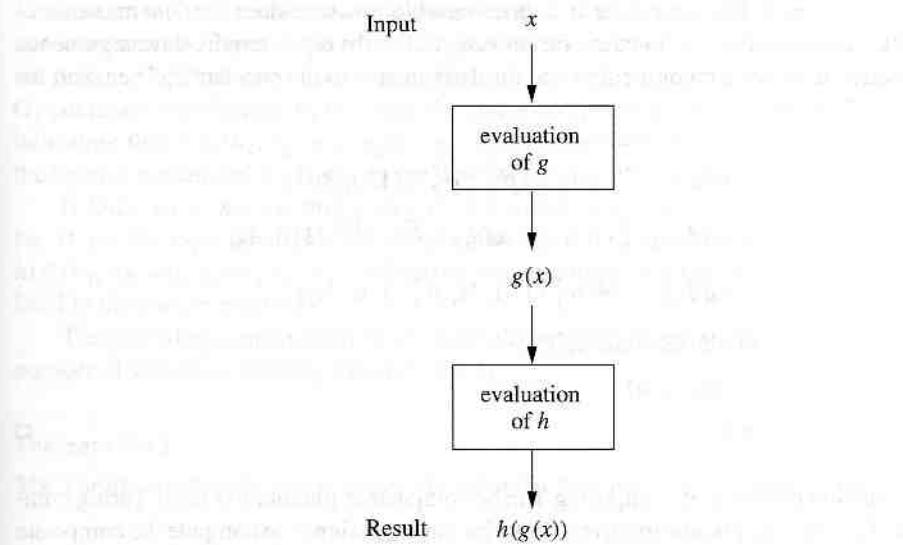
the domain of

with  $g$ . The definition 9.4.2

$g$  is the unary

$(x) = h(g(x))$ . whenever  $g(x)$  is total functions

the sequential he computation



The composite function is defined only when the preceding sequence of computations can be successfully completed.

#### Definition 9.4.2

Let  $g_1, g_2, \dots, g_n$  be  $k$ -variable number-theoretic functions and let  $h$  be an  $n$ -variable number-theoretic function. The  $k$ -variable function  $f$  defined by

$$f(x_1, \dots, x_k) = h(g_1(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k))$$

is called the **composition** of  $h$  with  $g_1, g_2, \dots, g_n$  and written  $f = h \circ (g_1, \dots, g_n)$ . The function  $f(x_1, \dots, x_k)$  is undefined if either

- i)  $g_i(x_1, \dots, x_k) \uparrow$  for some  $1 \leq i \leq n$ , or
- ii)  $g_i(x_1, \dots, x_k) = y_i$  for  $1 \leq i \leq n$  and  $h(y_1, \dots, y_n) \uparrow$ .

The general definition of composition of functions also admits a computational interpretation. The input is provided to each of the functions  $g_i$ . These functions generate the arguments of  $h$ .

---

#### Example 9.4.1

Consider the mapping defined by the composite function

$$\text{add} \circ (c_2^{(3)}, \text{add} \circ (p_1^{(3)}, p_3^{(3)})),$$

where  $\text{add}(n, m) = n + m$  and  $c_2^{(3)}$  is the three-variable constant function defined by

$c_2^{(3)}(n_1, n_2, n_3) = 2$ . The composite is a three-variable function since the innermost functions of the composition, the functions that directly utilize the input, require three arguments. The function adds the sum of the first and third arguments to the constant 2. The result for input 1, 0, 3 is

$$\begin{aligned} & \text{add} \circ (c_2^{(3)}, \text{add} \circ (p_1^{(3)}, p_3^{(3)}))(1, 0, 3) \\ &= \text{add} \circ (c_2^{(3)}(1, 0, 3), \text{add} \circ (p_1^{(3)}, p_3^{(3)})(1, 0, 3)) \\ &= \text{add}(2, \text{add}(p_1^{(3)}(1, 0, 3), p_3^{(3)}(1, 0, 3))) \\ &= \text{add}(2, \text{add}(1, 3)) \\ &= \text{add}(2, 4) \\ &= 6. \end{aligned}$$

□

A function obtained by composing Turing computable functions is itself Turing computable. The argument is constructive; a machine can be designed to compute the composite function by combining the machines that compute the constituent functions and the macros developed in the previous section.

Let  $g_1$  and  $g_2$  be three-variable Turing computable functions and let  $h$  be a Turing computable two-variable function. Since  $g_1$ ,  $g_2$ , and  $h$  are computable, there are machines  $G_1$ ,  $G_2$ , and  $H$  that compute them. The actions of a machine that computes the composite function  $h \circ (g_1, g_2)$  are traced for input  $n_1$ ,  $n_2$ , and  $n_3$ .

| Machine            | Configuration                                                                                                                                                                                                                                                                                                               |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CPY <sub>3</sub>   | <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u>                                                                                                                                                                                                                       |
| MR <sub>3</sub>    | <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u>                                                                                                                          |
| G <sub>1</sub>     | <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u> <u>g</u> <sub>1</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> ) <u>B</u>                                                                                                              |
| ML <sub>3</sub>    | <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u> <u>g</u> <sub>1</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> ) <u>B</u>                                                                                                              |
| CPY <sub>3,1</sub> | <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u> <u>g</u> <sub>1</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> ) <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u>                 |
| MR <sub>4</sub>    | <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u> <u>g</u> <sub>1</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> ) <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u>                 |
| G <sub>2</sub>     | <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u> <u>g</u> <sub>1</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> ) <u>B</u> <u>g</u> <sub>2</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> ) <u>B</u>     |
| ML <sub>1</sub>    | <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u> <u>g</u> <sub>1</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> ) <u>B</u> <u>g</u> <sub>2</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> ) <u>B</u>     |
| H                  | <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u> <u>h</u> ( <u>g</u> <sub>1</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> ), <u>g</u> <sub>2</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> )) <u>B</u> |
| ML <sub>3</sub>    | <u>B</u> <u>ñ</u> <sub>1</sub> <u>B</u> <u>ñ</u> <sub>2</sub> <u>B</u> <u>ñ</u> <sub>3</sub> <u>B</u> <u>h</u> ( <u>g</u> <sub>1</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> ), <u>g</u> <sub>2</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> )) <u>B</u> |
| E <sub>3</sub>     | <u>BB</u> . . . <u>B</u> <u>h</u> ( <u>g</u> <sub>1</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> ), <u>g</u> <sub>2</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> )) <u>B</u>                                                                              |
| T                  | <u>B</u> <u>h</u> ( <u>g</u> <sub>1</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> ), <u>g</u> <sub>2</sub> ( <u>n</u> <sub>1</sub> , <u>n</u> <sub>2</sub> , <u>n</u> <sub>3</sub> )) <u>B</u>                                                                                              |

The computation as the argument the original in  $G_1$  continues indicating that the input is cop

If both  $g_1$  for  $H$  on the  $h(g_1(n_1, n_2, n_3)$  lated to the cor

The preced number of vari

### Theorem 9.4.3

The Turing con

Theorem 9 explicitly const of Turing comp

### Example 9.4.2

The  $k$ -variable c Turing computa

The projection f function. The co of the functions computable by T

### Example 9.4.3

The binary functio function can be v

innermost function  
as arguments.  
The result for

The computation copies the input and computes the value of  $g_1$  using the newly created copy as the arguments. Since the machine  $G_1$  does not move to the left of its starting position, the original input remains unchanged. If  $g_1(n_1, n_2, n_3)$  is undefined, the computation of  $G_1$  continues indefinitely. In this case the entire computation fails to terminate, correctly indicating that  $h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))$  is undefined. Upon the termination of  $G_1$ , the input is copied and  $G_2$  is run on the new copy.

If both  $g_1(n_1, n_2, n_3)$  and  $g_2(n_1, n_2, n_3)$  are defined,  $G_2$  terminates with the input for  $H$  on the tape preceded by the original input. The machine  $H$  is run computing  $h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))$ . When the computation of  $H$  terminates, the result is translated to the correct position.

The preceding construction easily generalizes to the composition of functions of any number of variables, yielding Theorem 9.4.3.

□

If Turing computers  
the composite  
and the macros

$h$  be a Turing  
are machines  
the composite

### Theorem 9.4.3

The Turing computable functions are closed under the operation of composition.

Theorem 9.4.3 can be used to show that a function  $f$  is Turing computable without explicitly constructing a machine that computes it. If  $f$  can be defined as the composition of Turing computable functions then, by Theorem 9.4.3,  $f$  is also Turing computable.

---

### Example 9.4.2

The  $k$ -variable constant functions  $c_i^{(k)}$  whose values are given by  $c_i^{(k)}(n_1, \dots, n_k) = i$  are Turing computable. The function  $c_i^{(k)}$  can be defined by

$$c_i^{(k)} = \underbrace{s \circ s \circ \cdots \circ s}_{i \text{ times}} \circ z \circ p_1^{(k)}.$$

The projection function accepts the  $k$ -variable input and passes the first value to the zero function. The composition of  $i$  successor functions produces the desired value. Since each of the functions in the composition is Turing computable, the function  $c_i^{(k)}$  is Turing computable by Theorem 9.4.3. □

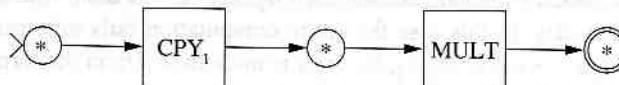
---

### Example 9.4.3

The binary function  $smsq(n, m) = n^2 + m^2$  is Turing computable. The sum-of-squares function can be written as the composition of functions

$$smsq = add \circ (sq \circ p_1^{(2)}, sq \circ p_2^{(2)}),$$

where  $sq$  is defined by  $sq(n) = n^2$ . The function  $add$  is computed by the machine constructed in Example 9.2.1 and  $sq$  by

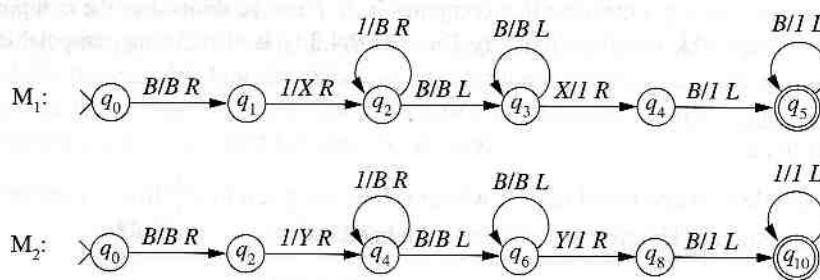


□

## 9.5 Uncomputable Functions

A function is Turing computable only if there is a Turing machine that computes it. The existence of number-theoretic functions that are not Turing computable can be demonstrated by a simple counting argument. We begin by showing that the set of computable functions is countably infinite.

A Turing machine is completely defined by its transition function. The states and tape alphabet used in computations of the machine can be extracted from the transitions. Consider the machines  $M_1$  and  $M_2$  defined by



Both  $M_1$  and  $M_2$  compute the unary constant function  $c_1^{(1)}$ . The two machines differ only in the names given to the states and the markers used during the computation. These symbols have no effect on the result of a computation and hence the function computed by the machine.

Since the names of the states and tape symbols other than  $B$  and  $I$  are immaterial, we adopt the following conventions concerning the naming of the components of a Turing machine:

- The set of states is a finite subset of  $Q_0 = \{q_i \mid i \geq 0\}$ .
- The input alphabet is  $\{I\}$ .
- The tape alphabet is a finite subset of the set  $\Gamma_0 = \{B, I, X_i \mid i \geq 0\}$ .
- The initial state is  $q_0$ .

The transitions of a Turing machine have been specified using functional notation; the transition defined for state  $q_i$  and tape symbol  $x$  is represented by  $\delta(q_i, x) = [q_j, y, d]$ . This information can also be represented by the quintuple

With the preceding, the set  $T = Q_0$  is the product of countably many

The transition function first two components of such subsets is countably infinite, at least countably many Turing computa-

### Theorem 9.5.1

The set of Turing computable functions is countably infinite.

In Section 1.2, we saw that there are ably many total functions. We can obtain Corollary 9.5.2.

### Corollary 9.5.2

The set of total computable functions is countably infinite.

Corollary 9.5.2 shows that there are many more computable functions than there are total functions.

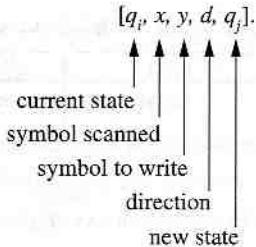
## 9.6 Towards a High-level Programming Language

High-level programming languages are designed to make it easier for programmers to write programs. They provide a way to express algorithms in a programming language. The basic idea behind a high-level programming language is to abstract away the details of the machine architecture and focus on the logic of the program. This makes it easier for programmers to write programs that are portable across different machines and easier to maintain and modify.

con-

□

t. The  
strated  
ctions  
d tape  
nsider



With the preceding naming conventions, a transition of a Turing machine is an element of the set  $T = Q_0 \times \Gamma_0 \times \Gamma_0 \times \{L, R\} \times Q_0$ . The set  $T$  is countable since it is the Cartesian product of countable sets.

The transitions of a deterministic Turing machine form a finite subset of  $T$  in which the first two components of every element are distinct. There are only a countable number of such subsets. It follows that the number of Turing computable functions is at most countably infinite. On the other hand, the number of Turing computable functions is at least countably infinite since there are countably many constant functions, all of which are Turing computable by Example 9.4.2. These observations yield

### Theorem 9.5.1

The set of Turing computable number-theoretic functions is countably infinite.

In Section 1.4, the diagonalization technique was used to prove that there are uncountably many total unary number-theoretic functions. Combining this with Theorem 9.5.1, we obtain Corollary 9.5.2.

### Corollary 9.5.2

There is a total unary number-theoretic function that is not Turing computable.

Corollary 9.5.2 vastly understates the relationship between computable and uncomputable functions. The former constitute a countable set and the latter an uncountable set.

: only in  
symbols  
1 by the

material,  
a Turing

---

## 9.6 Toward a Programming Language

ition; the  
 $q_j, y, d]$

High-level programming languages are the most commonly employed type of computational system. A program defines a mechanistic and deterministic process, the hallmark of algorithmic computation. The intuitive argument that the computation of a program written in a programming language and executed on a computer can be simulated by a Turing machine rests in the fact that a machine (computer) instruction simply changes the bits in some location of memory. This is precisely the type of action performed by a Turing machine, writing 0's and 1's in memory. Although it may take a large number of Turing machine transitions to accomplish the task, it is not at all difficult to envision a sequence of transitions that will access the correct position and rewrite the memory.

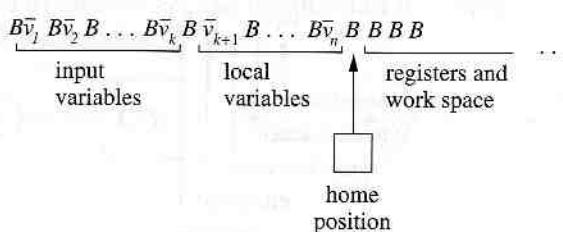


FIGURE 9.1 Turing machine architecture for high-level computation.

In this section we will explore the possibility of using the Turing machine architecture as the underlying framework for high-level programming. The development of a programming language based on the Turing machine architecture further demonstrates the power of the Turing machine model. In describing our assembly language, we use Turing machines and macros to define the operations. The objective of this section is not to create a functional assembly language, but rather to demonstrate further the universality of the Turing machine architecture.

The standard Turing machine provides the computational framework used throughout this section. We will design an assembly language TM to bridge the gap between the Turing machine architecture and programming languages. The first objective of the assembly language is to provide a sequential description of the actions of the Turing machine. The "program flow" of a Turing machine is determined by the arcs in the state diagram of the machine. The flow of an assembly language program consists of the sequential execution of the instructions unless this pattern is specifically altered by an instruction that redirects the flow. In assembly language, branch and goto instructions are used to alter sequential program flow. The second objective of the assembly language is to provide instructions that simplify memory management.

The underlying architecture of the Turing machine used to evaluate an assembly language program is pictured in Figure 9.1. The input values are assigned to variables  $v_1, \dots, v_k$ , and  $v_{k+1}, \dots, v_n$  are the local variables used in the program. The values of the variables are stored sequentially and separated by blanks. The input variables are in the standard input position for a Turing machine evaluating a function. A TM program begins by declaring the local variables used in the program. Each local variable is initialized to 0 at the start of a computation.

When the initialization is complete, the tape head is stationed at the blank separating the variables from the remainder of the tape. This will be referred to as the *home position*. Between the evaluation of instructions, the tape head returns to the home position. To the right of the home position is the Turing machine version of registers. The first value to the right is considered to be in register 1, the second value in register 2, and so on. The registers must be assigned sequentially; that is, register  $i$  may be written to or read from

| TABLE 9.1       |  |
|-----------------|--|
| TM Instructions |  |
| INIT $v_i$      |  |
| HOME $t$        |  |
| LOAD $v_i, t$   |  |
| STOR $v_i, t$   |  |
| RETURN $v_i$    |  |
| CLEAR $t$       |  |
| BRN L, $t$      |  |
| GOTO L          |  |
| NOP             |  |
| INC $t$         |  |
| DEC $t$         |  |
| ZERO $t$        |  |

only if registers  
are given in Tab

The tape in  
reserves the loc  
are stored sequ  
beginning of a T  
HOME instruct  
by

where ZR is the  
(Exercise 6). Th  
would produce t

**TABLE 9.1** TM Instructions

| TM Instruction | Interpretation                                                            |
|----------------|---------------------------------------------------------------------------|
| INIT $v_i$     | Initialize local variable $v_i$ to 0.                                     |
| HOME $t$       | Move the tape head to the home position when $t$ variables are allocated. |
| LOAD $v_i, t$  | Load value of variable $v_i$ into register $t$ .                          |
| STOR $v_i, t$  | Store value in register $t$ into location of $v_i$ .                      |
| RETURN $v_i$   | Erase the variables and leave the value of $v_i$ in the output position.  |
| CLEAR $t$      | Erase value in register $t$ .                                             |
| BRN L, $t$     | Branch to instruction labeled L if value in register $t$ is 0.            |
| GOTO L         | Execute instruction labeled L.                                            |
| NOP            | No operation (used in conjunction with GOTO commands).                    |
| INC $t$        | Increment the value of register $t$ .                                     |
| DEC $t$        | Decrement the value of register $t$ .                                     |
| ZERO $t$       | Replace value in register $t$ with 0.                                     |

architecture as  
programming  
the power of the  
machines and  
to a functional  
Turing machine

sed throughout  
een the Turing  
the assembly  
machine. The  
diagram of the  
ntial execution  
n that redirects  
alter sequential  
nstructions that

e an assembly  
ed to variables  
am. The values  
ut variables are  
A TM program  
ble is initialized

blank separating  
: *home position*.  
position. To the  
he first value to  
and so on. The  
to or read from

only if registers 1, 2, ...,  $i - 1$  are assigned values. The instructions of the language TM are given in Table 9.1.

The tape initialization is accomplished using the INIT and HOME commands. INIT  $v_i$  reserves the location for local variable  $v_i$  and initializes the value to 0. Since variables are stored sequentially on the tape, local variables must be initialized in order at the beginning of a TM program. Upon completion of the initialization of the local variables, the HOME instruction moves the tape head to the home position. These instructions are defined by

| Instruction | Definition |
|-------------|------------|
| INIT $v_i$  | $MR_{i-1}$ |
|             | ZR         |
|             | $ML_{i-1}$ |
| HOME $t$    | $MR_t$     |

where ZR is the macro that writes the value 0 to the immediate right of the tape head position (Exercise 6). The initialization phase of a program with one input and two local variables would produce the following sequence of Turing machine configurations:

|        | Instruction | Configuration       |
|--------|-------------|---------------------|
|        |             | <u>B̄B</u>          |
| INIT 2 |             | <u>B̄iB̄B̄B</u>     |
| INIT 3 |             | <u>B̄iB̄B̄B̄B̄B</u> |
| HOME 3 |             | <u>B̄iB̄B̄B̄B̄B</u> |

where  $i$  is the value of the input to the computation. The position of the tape head is indicated by the underscore.

In TM, the LOAD and STOR instructions are used to access and store the values of the variables. The objective of these instructions is to make the details of memory management transparent to the user. In Turing machines there is no upper bound to the amount of tape that may be required to store the value of a variable. The lack of a preassigned limit to the amount of tape allotted to each variable complicates the memory management of a Turing machine. This omission, however, is intentional, allowing maximum flexibility in Turing machine computations. Assigning a fixed amount of memory to a variable, the standard approach used by conventional compilers, causes an overflow error when the memory required to store a value exceeds the preassigned allocation.

The STOR command takes the value from register  $t$  and stores it in the specified variable location. The command may be used only when  $t$  is the largest register that has an assigned value. In storing the value of register  $t$  in a variable  $v_i$ , the proper spacing is maintained for all the variables. The Turing machine implementation of the store command utilizes the macro INT to move the value in the register to the proper position. The macro INT is assumed to stay within the tape segment  $B̄x B̄y B$  (Exercise 6).

The STOR command is defined by

| Instruction   | Definition                       | Instruction   | Definition                         |
|---------------|----------------------------------|---------------|------------------------------------|
| STOR $v_i, 1$ | $(ML_1)^{n-i+1}$<br>$\text{INT}$ | STOR $v_i, t$ | $MR_{t-2}$<br>INT                  |
|               | $(MR_1)^{n-i}$<br>$\text{INT}$   |               | $(ML_1)^{t+n-i-1}$<br>$\text{INT}$ |
|               | $MR_1$                           |               | $(MR_1)^{t+n-i-1}$<br>$\text{INT}$ |
|               | $ER_1$                           |               | $MR_1$                             |
|               |                                  |               | $ER_1$                             |
|               |                                  |               | $ML_{t-1}$                         |

where  $t >$   
and  $n - i$  in  
the register

The co  
STOR  $v_2$ , 1  
the executio

The Tur  
of variable v

As previously  
to be filled. T

for the instruc

The instru  
the computati  
position and ne  
machine output

where  $t > 1$  and  $n$  is the total number of input and local variables. The exponents  $n - i + 1$  and  $n - i$  indicate repetition of the sequence of macros. After the value of register  $t$  is stored, the register is erased.

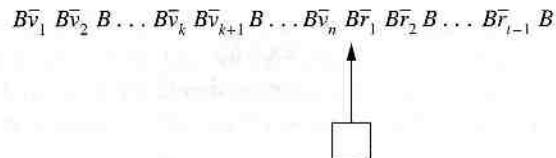
The configurations of a Turing machine obtained by the execution of the instruction STOR  $v_2$ , 1 are traced to show the role of the macros in TM memory management. Prior to the execution of the instruction, the tape head is at the home position.

| Machine         | Configuration                                   |
|-----------------|-------------------------------------------------|
|                 | $B\bar{v}_1B\bar{v}_2B\bar{v}_3B\bar{r}B$       |
| ML <sub>1</sub> | $B\bar{v}_1B\bar{v}_2\bar{B}\bar{v}_3B\bar{r}B$ |
| INT             | $B\bar{v}_1B\bar{v}_2B\bar{r}B\bar{v}_3B$       |
| ML <sub>1</sub> | $B\bar{v}_1\bar{B}\bar{v}_2B\bar{r}B\bar{v}_3B$ |
| INT             | $B\bar{v}_1B\bar{r}B\bar{v}_2B\bar{v}_3B$       |
| MR <sub>1</sub> | $B\bar{v}_1B\bar{r}\bar{B}\bar{v}_2B\bar{v}_3B$ |
| INT             | $B\bar{v}_1B\bar{r}B\bar{v}_3B\bar{v}_2B$       |
| MR <sub>1</sub> | $B\bar{v}_1B\bar{r}B\bar{v}_3\bar{B}\bar{v}_2B$ |
| E <sub>1</sub>  | $B\bar{v}_1B\bar{r}B\bar{v}_3\bar{B}B$          |

The Turing machine implementation of the LOAD instruction simply copies the value of variable  $v_i$  to the specified register.

| Instruction   | Definition        |
|---------------|-------------------|
| LOAD $v_i, t$ | $ML_{n-i+1}$      |
|               | $CPY_{1,n-i+1+t}$ |
|               | $MR_{n-i+1}$      |

As previously mentioned, to load a value into register  $t$  requires registers 1, 2, ...,  $t - 1$  to be filled. Thus the Turing machine must be in configuration



for the instruction LOAD  $v_i, t$  to be executed.

The instructions RETURN and CLEAR reconfigure the tape to return the result of the computation. When the instruction RETURN  $v_i$  is run with the tape head in the home position and no registers allocated, the tape is rewritten placing the value of  $v_i$  in the Turing machine output position. CLEAR simply erases the value in the register.

| Instruction  | Definition  |
|--------------|-------------|
| RETURN $v_i$ | $ML_n$      |
|              | $E_{i-1}$   |
|              | T           |
|              | $MR_1$      |
|              | FR          |
|              | $E_{n-i+1}$ |
|              | FL          |
| CLEAR $t$    | $MR_{t-1}$  |
|              | $E_1$       |
|              | $ML_{t-1}$  |

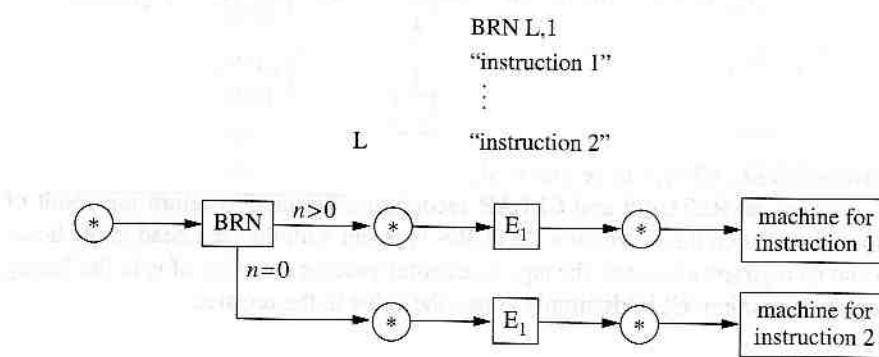
The value is tested. The instruction are the

### Example 9.6.1

The TM program the function  $f(n)$   $v_2$  and  $v_3$ .

Arithmetic operations alter the values in the registers. INC, DEC, and ZERO are defined by the machines computing the successor, predecessor, and zero functions. Additional arithmetic operations may be defined for our assembly language by creating a Turing machine that computes the operation. For example, an assembly language instruction ADD could be defined using the Turing machine implementation of addition given by the machine A in Example 9.2.1. The resulting instruction ADD would add the values in registers 1 and 2 and store the result in register 1. While we could greatly increase the number of assembly language instructions by adding additional arithmetic operations, INC, DEC, and ZERO will be sufficient for purposes of developing our language.

The execution of assembly language instructions consists of the sequential operation of the Turing machines and macros that define each of the instructions. The BRN and GOTO instructions interrupt the sequential evaluation by explicitly specifying the next instruction to be executed. GOTO L indicates that the instruction labeled L is the next to be executed. BRN  $L,t$  tests register  $t$  before indicating the subsequent instruction. If the register is nonzero, the instruction immediately following the branch is executed. Otherwise, the statement labeled by L is executed. The Turing machine implementation of the branch is illustrated by



The variable  $v_2$  defined by the label incremented. The loop, the value is in

The objective of Turing machines, like higher-level languages and compilation, the assembly language closer in form to the

The value is tested, the register erased, and the machines that define the appropriate instruction are then executed.

### Example 9.6.1

The TM program with one input variable and two local variables defined below computes the function  $f(n) = 2n + 1$ . The input variable is  $v_1$ , and the computation uses local variables  $v_2$  and  $v_3$ .

```

INIT v2
INIT v3
HOME 3
LOAD v1,1
STOR v2,1
L1 LOAD v2,1
 BRN L2,1
 LOAD v1,1
 INC
 STOR v1,1
 LOAD v2,1
 DEC
 STOR v2,1
 GOTO L1
L2 LOAD v1,1
 INC
 STOR v1,1
 RETURN v1

```

The variable  $v_2$  is used as a counter, which is decremented each time through the loop defined by the label L1 and the GOTO instruction. In each iteration, the value of  $v_1$  is incremented. The loop is exited after  $n$  iterations, where  $n$  is the input. Upon exiting the loop, the value is incremented again and the result  $2v_1 + 1$  is left on the tape.  $\square$

The objective of constructing the TM assembly language is to show that instructions of Turing machines, like those of conventional machines, can be formulated as commands in a higher-level language. Utilizing the standard approach to programming language definition and compilation, the commands of a high-level language may be defined by a sequence of the assembly language instructions. This would bring Turing machine computations even closer in form to the algorithmic systems most familiar to many of us.

---

**Exercises**

1. Construct Turing machines with input alphabet  $\{a, b\}$  that compute the specified functions. The symbols  $u$  and  $v$  represent arbitrary strings over  $\{a, b\}^*$ .
  - a)  $f(u) = aaa$
  - b)  $f(u) = \begin{cases} a & \text{if } \text{length}(u) \text{ is even} \\ b & \text{otherwise} \end{cases}$
  - c)  $f(u) = u^R$
  - d)  $f(u, v) = \begin{cases} u & \text{if } \text{length}(u) > \text{length}(v) \\ v & \text{otherwise} \end{cases}$
2. Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$  be a Turing machine that computes the partial characteristic function of the language  $L$ . Use  $M$  to build a standard Turing machine that accepts  $L$ .
3. Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be a standard Turing machine that accepts a language  $L$ . Construct a machine  $M'$  that computes the partial characteristic function of  $L$ . Recall that the tape of  $M'$  must have the form  $q_f B 0 B$  or  $q_f B 1 B$  upon the completion of a computation of  $\chi_L$ .
4. Let  $L$  be a language over  $\Sigma$  and let

$$\chi_L(w) = \begin{cases} 1 & \text{if } w \in L \\ 0 & \text{otherwise} \end{cases}$$

be the characteristic function of  $L$ .

- a) If  $\chi_L$  is Turing computable, prove that  $L$  is recursive.
- b) If  $L$  is recursive, prove that there is a Turing machine that computes  $\chi_L$ .
5. Construct Turing machines that compute the following number-theoretic functions and relations. Do not use macros in the design of these machines.
  - a)  $f(n) = 2n + 3$
  - b)  $\text{half}(n) = \lfloor n/2 \rfloor$  where  $\lfloor x \rfloor$  is the greatest integer less than or equal to  $x$
  - c)  $f(n_1, n_2, n_3) = n_1 + n_2 + n_3$
  - d)  $\text{even}(n) = \begin{cases} 1 & \text{if } n \text{ is even} \\ 0 & \text{otherwise} \end{cases}$
  - e)  $\text{eq}(n, m) = \begin{cases} 1 & \text{if } n = m \\ 0 & \text{otherwise} \end{cases}$
  - f)  $\text{lt}(n, m) = \begin{cases} 1 & \text{if } n < m \\ 0 & \text{otherwise} \end{cases}$

- g)  $n \perp m = \begin{cases} 1 & \text{if } n \neq m \\ 0 & \text{otherwise} \end{cases}$
6. Construct TMs that compute the following functions. The configurations are given as  $(q, \Gamma)$ , where  $q$  is the state and  $\Gamma$  is the tape configuration.
  - a) ZR; input  $0^n 1^m$
  - b) FL; input  $0^n 1^m$
  - c) E<sub>2</sub>; input  $0^n 1^m$
  - d) T; input  $0^n 1^m$
  - e) BRN; input  $0^n 1^m$
  - f) INT; input  $0^n 1^m$
7. Use the macro facilities of your system to construct TMs that compute the following functions.
  - a)  $f(n) = 2^n$
  - b)  $f(n) = n!$
  - c)  $f(n_1, n_2) = n_1 + n_2$
  - d)  $f(n, m) = n^m$
  - e)  $f(n_1, n_2) = n_1 \cdot n_2$
8. Design machines that compute the following functions. You may use the machines that you constructed in Exercise 5.
  - a)  $gt(n, m) = \begin{cases} 1 & \text{if } n > m \\ 0 & \text{otherwise} \end{cases}$
  - b)  $\text{persq}(n) = \begin{cases} 1 & \text{if } n \text{ is a perfect square} \\ 0 & \text{otherwise} \end{cases}$
  - c)  $\text{divides}(n, m) = \begin{cases} 1 & \text{if } n \mid m \\ 0 & \text{otherwise} \end{cases}$
9. Trace the action of the macro  $\text{mult}$  for the following values of  $n$  and  $m$ .
  - a)  $n = 0, m = 0$
  - b)  $n = 1, m = 0$
  - c)  $n = 2, m = 0$
10. Describe the macro  $\text{mult}$ .
  - a)  $\text{add} \circ (\text{mult} \circ \text{add})$
  - b)  $p_1^{(2)} \circ (s \circ p_1^{(2)})$
  - c)  $\text{mult} \circ (c_2^{(3)} \circ \text{add})$
  - d)  $\text{mult} \circ (\text{mult} \circ \text{add})$