Thomas A. Sudkamp

Second Edition

Languages *and* Machines

An Introduction to the Theory of Computer Science

Languages and Machines

An Introduction to the Theory of Computer Science

Second Edition

Thomas A. Sudkamp

Wright State University

ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Sponsoring Editor	Thomas Stone
Assistant Editor	Kathleen Billus
Senior Production Supervisor	Juliet Silveri
Production Services	Lachina Publishing Services, Inc.
Compositor	Windfall Software, using ZzTEX
Marketing Manager	Tom Ziolkowski
Manufacturing Coordinator	Judy Sullivan
Cover Designer	Peter Blaiwas

Library of Congress Cataloging-in-Publication Data

Sudkamp, Thomas A.
Languages and machines : an introduction to the theory of computer science / Thomas A. Sudkamp. — 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-82136-2
1. Formal languages. 2. Machine theory. 3. Computational complexity. I. Title.
QA267.3.S83 1997
511.3—dc20 95-51366
CIP

Access the latest information about Addison-Wesley books from our World Wide Web page: http://www.aw.com/cseng

1 2 3 4 5 6 7 8 9 10–MA–00 99 98 97 96

⟨*dedication*⟩ → ⟨*parents*⟩

⟨*parents*⟩ → ⟨*first name*⟩ ⟨*last name*⟩

⟨*first name*⟩ → *Donald* | *Mary*

⟨*last name*⟩ → *Sudkamp*

Preface

The objective of the second edition remains the same as that of the first, to provide a mathematically sound presentation of the theory of computer science at a level suitable for junior- and senior-level computer science majors. It is my hope that, like a fine wine, this book has improved with age. The improvement comes with the experience gained from eight years of use by students and faculty who generously shared their observations and suggestions with me. These comments helped to identify areas that would benefit from more comprehensive explanations and topics that would add depth and flexibility to the coverage. A brief description of some of the additions is given below.

An algorithm for mininimizing the number of states in a deterministic finite automaton has been added to Chapter 6. This completes the sequence of actions used to construct automata to accept complex languages: Initially use nondeterminism to assist in the design process, algorithmically construct an equivalent deterministic machine, and, finally, minimize the number of states to obtain the optimal machine. The correctness of the minimization algorithm is proven using the Myhill-Nerode theorem, which has been included in Chapter 7.

Rice's theorem has been added to the presentation of undecidability. This further demonstrates the importance of reduction in establishing undecidability and provides a simple method for showing that many properties of recursively enumerable languages are undecidable.

The coverage of computational complexity has been significantly expanded. Chapter 14 introduces the time complexity of Turing machines and languages. Properties of the

complexity of languages, including the speedup theorem and the existence of arbitrarily complex languages, are established. Chapter 15 is devoted solely to the issues of tractability and NP-completeness.

Organization

Since most students at this level have had little or no background in abstract mathematics, the presentation is designed 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 concepts and theorems 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.

The presentation of formal language and automata theory examines the relationships between the grammars and abstract machines of the Chomsky hierarchy. Parsing context-free languages is presented via standard graph-searching algorithms immediately following the introduction of context-free grammars. Considering parsing at this point reinforces the need for a formal approach to language definition and motivates the development of normal forms. Chapters on LL and LR grammars are included to permit a presentation of formal language theory that serves as a foundation to a course in compiler design.

Finite-state automata and Turing machines provide the framework for the study of effective computation. The equivalence of the languages generated by the grammars and recognized by the machines of the Chomsky hierarchy is established. The coverage of computability includes decidability, the Church-Turing thesis, and the equivalence of Turing computability and μ-recursive functions. The classes $\mathcal{P}$ and $\mathcal{NP}$ of solvable decision problems and the theory of NP-completeness are introduced by analyzing the time complexity of Turing machines.

To make these topics accessible to the undergraduate student, no special mathematical prerequisites are assumed. Instead, Chapter 1 introduces the mathematical tools of the theory of computing: naive set theory, recursive definitions, and proof by mathematical induction. With the exception of the specialized topics in Sections 1.3 and 1.4, Chapters 1 and 2 provide background material that will be used throughout the text. Section 1.3 introduces cardinality and the diagonalization argument, which arise in the counting arguments that establish the existence of uncomputable functions in Chapter 12. Equivalence relations, introduced in Section 1.4, are used in the algorithm that minimizes the number of states of a deterministic finite automaton and in the Myhill-Nerode theorem. For students who have completed a course in discrete mathematics, most of the material in Chapter 1 can be treated as review.

The following table indicates the dependence of each chapter upon the preceding material:

Chapter	Prerequisite Chapters
1	—
2	1
3	2
4	3
5	3
6	2
7	3, 6
8	5, 7
9	2
10	8, 9
11	9
12	11
13	12
14	13
15	14
16	4, 5
17	4, 5, 6

The entire book can be comfortably covered in two semesters. Since courses on the foundations of computing may emphasize different topics, the text has been organized to allow the flexibility needed to design one-term courses that concentrate on one or two specific areas. Suggested outlines for such courses are

- Formal language and automata theory:
  Chapters 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- Computability and complexity theory:
  Chapters 1, 2, 6, 9, 11, 12, 13, 14, 15
- Language design and parsing:
  Chapters 1, 2, 3, 4, 5, 6, 8, 16, 17

Notation

The theory of computer science is a mathematical examination of the capabilities and limitations of effective computation. As with any formal analysis, mathematical notation is used to provide unambiguous definitions of the concepts and operations used in formal

language and automata theory. 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	a, b, abc
functions	italic lowercase letters	f, g, h
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
automata	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. Exercises are referenced simply by the chapter and the number.

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

Acknowledgments

A number of people made significant contributions to this book, both in the original and the current edition. I would like to extend my most sincere appreciation to the students and professors who have used this text over the past eight years. Your comments and suggestions provided the impetus for the inclusion of many of the new topics and more detailed explanations of material previously covered.

The first edition was reviewed by Professors Andrew Astromoff (San Francisco State University), Michael Harrison (University of California at Berkeley), David Hemmendinger (Union College), D. T. Lee (Northwestern University), C. L. Liu (University of Illinois at Urbana-Champaign), Kenneth Williams (Western Michigan University), and Hsu-Chun Yen (Iowa State University). Valuable suggestions for the second edition were provided by Dan Cooke (University of Texas-El Paso), Raymond Gumb (University of Massachusetts-Lowell), Jeffery Shallit (University of Waterloo), and William Ward (University of South Alabama). I would also like to acknowledge the assistance received from the staff of the Addison-Wesley Publishing Co.

Thomas A. Sudkamp *Dayton, Ohio*

Contents

PART IV

Decidability and Computability

PART VI

Deterministic Parsing

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 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 and language may have inadvertent and unwanted consequences for 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 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 Chapters 11 and 13, 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 its similarity to the modern 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 11 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. The time complexity of a Turing machine measures the number of transitions 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 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 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 capabilites of the finite automata. 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 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 syntatic 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 explores the capabilities and limitations of algorithmic problem solving. Formal language theory provides the foundation for the definition of programming languages and compiler design. Abstract machines are built to recognize the syntactic properties of languages and to compute functions. The relationship between the grammatical generation of languages and the recognition of languages by automata is a central theme of this book. Chapter 1 introduces the mathematical concepts, operations, and notation required for the study of formal language theory and automata theory.

Formal language theory has its roots in linguistics, mathematical logic, and computer science. Grammars were developed to provide a mechanism for describing natural (spoken and written) languages and have become the primary tool for the formal specification of programming languages. A set-theoretic definition of language is given in Chapter 2. This definition is sufficiently broad to include both natural and formal languages, but the generality is gained at the expense of not providing a technique for mechanically generating the strings of a language. To overcome this shortcoming, recursive definitions and set operations are used to give finite specifications of languages. This section ends with the development of the regular sets, a family of languages that arises in automata theory, formal language theory, switching circuits, and neural networks.

CHAPTER 1

Mathematical Preliminaries

Set theory provides the mathematical foundation for formal language and automata theory. In this chapter we review the notation and basic operations of set theory. Cardinality is used to compare the size of sets and provide 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 IV 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 consists 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 and automata theory.

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; $x \in \mathrm{X}$ indicates that x is a member or element of the set X. A slash through a symbol represents *not,* so $x \notin \mathrm{X}$ 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, 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

$$\mathrm{X} = \{1, 2, 3\}$$
$$\mathrm{Y} = \{a, b, c, d, e\}$$

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 mentions the set of **natural numbers**. This important set, denoted **N**, consists of the numbers 0, 1, 2, 3, ... 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 of X, Y, and Z describe the same set:

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

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 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

$$\emptyset \quad \{1\} \quad \{2\} \quad \{3\}$$
$$\{1,2\} \quad \{2,3\} \quad \{3,1\} \quad \{1,2,3\}. \qquad \square$$

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, $X_1, X_2, \ldots, X_n$, are defined by

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

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

respectively.

Subsets $X_1, X_2, \ldots, X_n$ 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 **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 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 set U is known, the complement of X with respect to U is denoted $\overline{X}$. 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 $X = \{0, 1, 2, 3\}$ and $Y = \{2, 3, 4, 5\}$. $\overline{X}$ and $\overline{Y}$ denote the complement of X and Y with respect to **N**.

$$X \cup Y = \{0, 1, 2, 3, 4, 5\} \qquad \overline{X} = \{n \mid n > 3\}$$

$$X \cap Y = \{2, 3\} \qquad \overline{Y} = \{0, 1\} \cup \{n \mid n > 5\}$$

$$X - Y = \{0, 1\} \qquad \overline{X} \cap \overline{Y} = \{n \mid n > 5\}$$

$$Y - X = \{4, 5\} \qquad \overline{(X \cup Y)} = \{n \mid n > 5\}$$ □

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 1.1.3

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

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

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 $X = Y$. □

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

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

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 $\mathbf{N} \times \mathbf{N}$. This relation is the subset of $\mathbf{N} \times \mathbf{N}$ defined by

$$LT = \{[i, j] \mid i < j \text{ and } i, j \in \mathbf{N}\}.$$

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 $x_1, x_2, \ldots, x_n$ are n elements, then $[x_1, x_2, \ldots, x_n]$ 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 $X_1, X_2, \ldots, X_n$ is defined by

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

An ***n*-ary relation** on $X_1, X_2, \ldots, X_n$ 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 $Y = \{a, b\}$.

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. Each member of X is mapped to at most one element of Y. A function f from X to Y is denoted $f : X \to Y$. 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. 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 $f : X \to Y$ 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

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

is said to be an ***n*-variable function** or operation. The value of the function with variables $x_1, x_2, \ldots, x_n$ is denoted $f(x_1, x_2, \ldots, x_n)$. Functions with one, two, or three variables are often referred to as *unary*, *binary*, and *ternary* operations. The variables $x_1, x_2, \ldots, x_n$ are also called the *arguments* of the functions. The function $sq : \mathbf{N} \to \mathbf{N}$ 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 **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 $\mathbf{N} \times \mathbf{N}$ 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	$f(x)$
1	a
2	a
3	a

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

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

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

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

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

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

x	$f(x)$
1	b
2	b
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 **N** $\times$ **N** to **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 h are defined from **N** to **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 **N** to **N** $-$ {0} defined by g is clearly onto but not one-to-one. The function s is both one-to-one and onto, defining a correspondence that maps each natural number to its successor. □

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 it satisfies 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 = $\{[i, j] \mid i < j\}$.

Infix notation is often used to express membership in 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 $b \equiv a$

iii) *Transitivity:* $a \equiv b$ and $b \equiv c$ implies $a \equiv c$.

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 $[a]_{\equiv} = \{b \in X \mid a \equiv b\}$.

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 transivity, 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 $a \equiv b$.

Let d be any element $[b]_{\equiv}$. Then $b \equiv d$. Combining $a \equiv b$ with $b \equiv d$ and employing transitivity yields $a \equiv d$. That is, $d \in [a]_{\equiv}$. Thus, we have shown that $[b]_{\equiv}$ is a subset of $[a]_{\equiv}$. 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. ■

Example 1.3.1

Let $\equiv_P$ be the parity relation over $\mathbf{N}$ defined by $n \equiv_P m$ if, and only if, n and m have the same parity (even or odd). To show 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 $\equiv_P$ are $[0]_{\equiv_P} = \{0, 2, 4, \ldots\}$ and $[1]_{\equiv_P} = \{1, 3, 5, \ldots\}$. □

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

$$a \longrightarrow 1$$
$$b \longrightarrow 2$$
$$c \longrightarrow 3$$

demonstrates that the sets $\{a, b, c\}$ and $\{1, 2, 3\}$ have the same size. This approach, comparing 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 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 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.

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.** 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 $\mathbf{N} - \{0\}$ is countably infinite; the function $s(n) = n + 1$ defines a one-to-one mapping from $\mathbf{N}$ onto $\mathbf{N} - \{0\}$. It may seem paradoxical that the set $\mathbf{N} - \{0\}$, obtained by removing an element from $\mathbf{N}$, has the same number of elements of $\mathbf{N}$. 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.

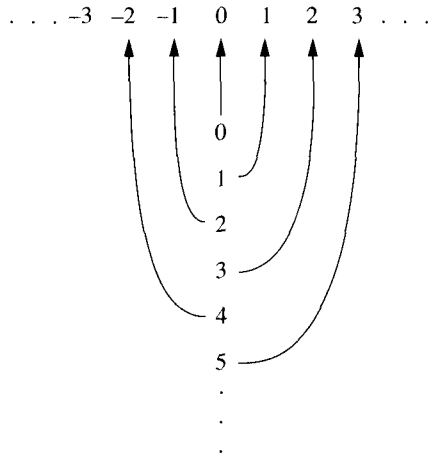
Definition 1.4.3

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 1.2.3 establishes the one-to-one correspondence between $\mathbf{N}$ and the odd numbers. □

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 $\mathbf{N}$ onto a set graphically exhibits the countability of the set. The one-to-one correspondence between the natural numbers and the set of all integers

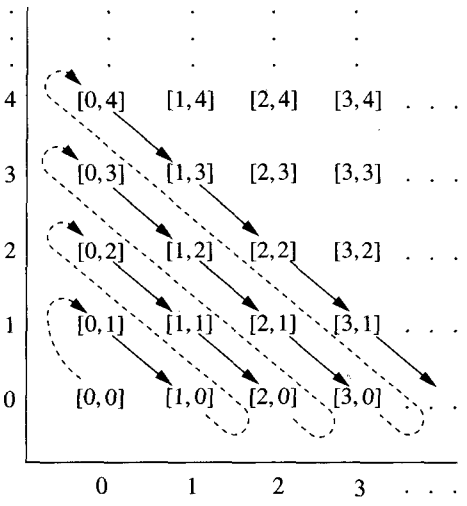


exhibits the countability of the set of integers. This correspondence is defined by the function

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

Example 1.4.2

The points of an infinite two-dimensional grid can be used to show that $\mathbf{N} \times \mathbf{N}$, 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 ith entry on the horizontal axis and the jth 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

$$\begin{array}{cccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & \ldots \\ \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \\ [0,0] & [0,1] & [1,0] & [0,2] & [1,1] & [2,0] & [0,3] & [1,2] & \ldots \end{array}$$

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

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 shows 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 **N** to **N**. Two total functions $f : \mathbf{N} \to \mathbf{N}$ and $g : \mathbf{N} \to \mathbf{N}$ are equal if they assume the same value for every element in the domain. That is, $f = g$ if $f(n) = g(n)$ for all $n \in \mathbf{N}$. 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 $f_0, f_1, f_2, \ldots$ 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	$f_4(0)$	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$	...
⋮	⋮	⋮	⋮	⋮	⋮	

Consider the function $f : \mathbf{N} \to \mathbf{N}$ 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. It follows from the definition of f that for any value i, $f(i) \neq f_i(i)$. Consequently, f is not in the sequence $f_0, f_1, f_2, \ldots$. 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 listing. No conditions are put on the listing of the elements other than that it must contain all the elements of the set. Achieving a contradiction by diagonalization then 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 from **N** to **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 **N** to **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 $f_0, f_1, f_2, \ldots$. To show that the set is uncountable, we must 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 and consequently 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 $f_0, f_1, f_2, \ldots$ is not exhaustive, and we conclude that the number of functions without fixed points is uncountable. □

Example 1.4.4

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

$$j \in \mathrm{D} \text{ if, and only if, } j \notin \mathrm{N}_j.$$

D is clearly a subset of **N**. By our assumption, $N_0, N_1, N_2, \ldots$ is an exhaustive listing of the subsets of **N**. Hence, $\mathrm{D} = \mathrm{N}_i$ for some i. Is the number i in the set D? By definition of D,

$$i \in \mathrm{D} \text{ if, and only if, } i \notin \mathrm{N}_i.$$

But since $\mathrm{D} = \mathrm{N}_i$, this becomes

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

We have shown that $i \in \mathrm{D}$ if, and only if, $i \notin \mathrm{D}$, which is a contradiction. Thus, our assumption that $\mathcal{P}(\mathbf{N})$ is countable must be false and we conclude that $\mathcal{P}(\mathbf{N})$ 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 $N_0, N_1, N_2, \ldots$. 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*. □

1.5 Recursive Definitions

Many of the sets involved in the generation of languages contain an infinite number of elements. We must be able to define an infinite set in a manner that allows its members to be constructed and manipulated. The description of the natural numbers avoided this by 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, an alien unfamiliar with our base 10 arithmetic system and numeric representations would have no idea that the symbol 4 is the next element in the sequence.

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: the 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 natural numbers. This idea is formalized in the following definition.

Definition 1.5.1

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 1.5.1 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 1.5.1 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 1.5.2

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.

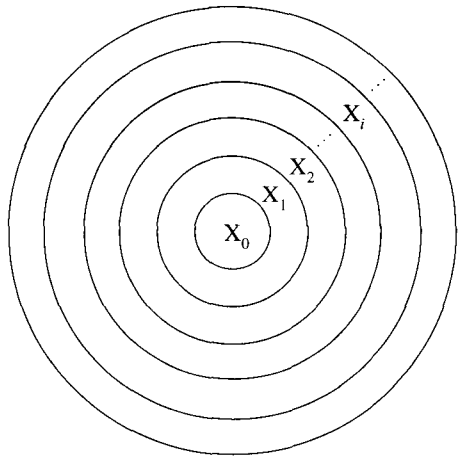
Following the construction given in Definition 1.5.2, the sum of any two natural numbers can be computed using 0 and s, the primitives used in the definition of the natural numbers. Example 1.5.1 traces the recursive computation of $3 + 2$.

Example 1.5.1

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

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

This final value is the representation of the number 5. □



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.

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 $\mathbf{N} \times \mathbf{N}$. The Cartesian product $\mathbf{N} \times \mathbf{N}$ 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.

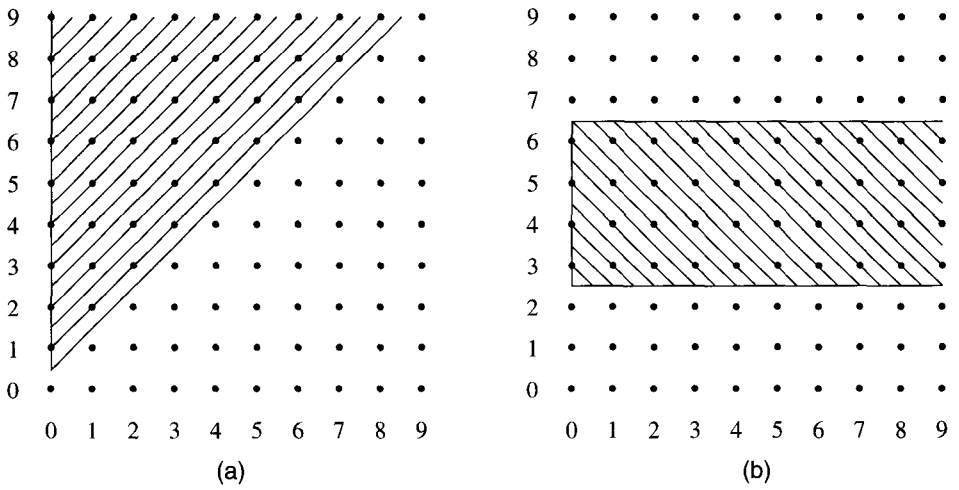


FIGURE 1.2 Relations on $\mathbf{N} \times \mathbf{N}$.

Example 1.5.2

The relation LT is defined as follows:

i) Basis: $[0, 1] \in \mathrm{LT}$.

ii) Recursive step: If $[n, m] \in \mathrm{LT}$, then $[n, s(m)] \in \mathrm{LT}$ and $[s(n), s(m)] \in \mathrm{LT}$.

iii) Closure: $[n, m] \in \mathrm{LT}$ only if it can be obtained from $[0, 1]$ by a finite number of applications of the operations in the recursive step.

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

$$\mathrm{LT}_0 = \{[0, 1]\}$$
$$\mathrm{LT}_1 = \mathrm{LT}_0 \cup \{[0, 2], [1, 2]\}$$
$$\mathrm{LT}_2 = \mathrm{LT}_1 \cup \{[0, 3], [1, 3], [2, 3]\}$$
$$\mathrm{LT}_3 = \mathrm{LT}_2 \cup \{[0, 4], [1, 4], [2, 4], [3, 4]\}$$
$$\vdots$$
$$\mathrm{LT}_i = \mathrm{LT}_{i-1} \cup \{[j, i + 1] \mid j = 0, 1, \ldots, i\}.$$
$$\vdots$$

□

The generation of an element in a recursively defined set may not be unique. The ordered pair $[1, 3] \in \mathrm{LT}_2$ is generated by the two distinct sequences of operations:

Basis:	$[0,1]$	$[0,1]$
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 1.5.3

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 $[n,m] \in \mathrm{X}$, then $[s(n),m] \in \mathrm{X}$.

iii) Closure: $[n,m] \in \mathrm{X}$ 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

$$\mathrm{X}_i = \{[j,3], [j,4], [j,5], [j,6] \mid j = 0,1,\ldots,i\}. \qquad \square$$

1.6 Mathematical Induction

Establishing the relationships between the elements of sets and operations on the sets requires the capability of constructing proofs to 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 $\mathrm{X}_0, \mathrm{X}_1, \mathrm{X}_2, \ldots, \mathrm{X}_i, \ldots$ 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 $\mathrm{X}_0, \mathrm{X}_1, \ldots, \mathrm{X}_i$, **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 by a recursive definition. 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.

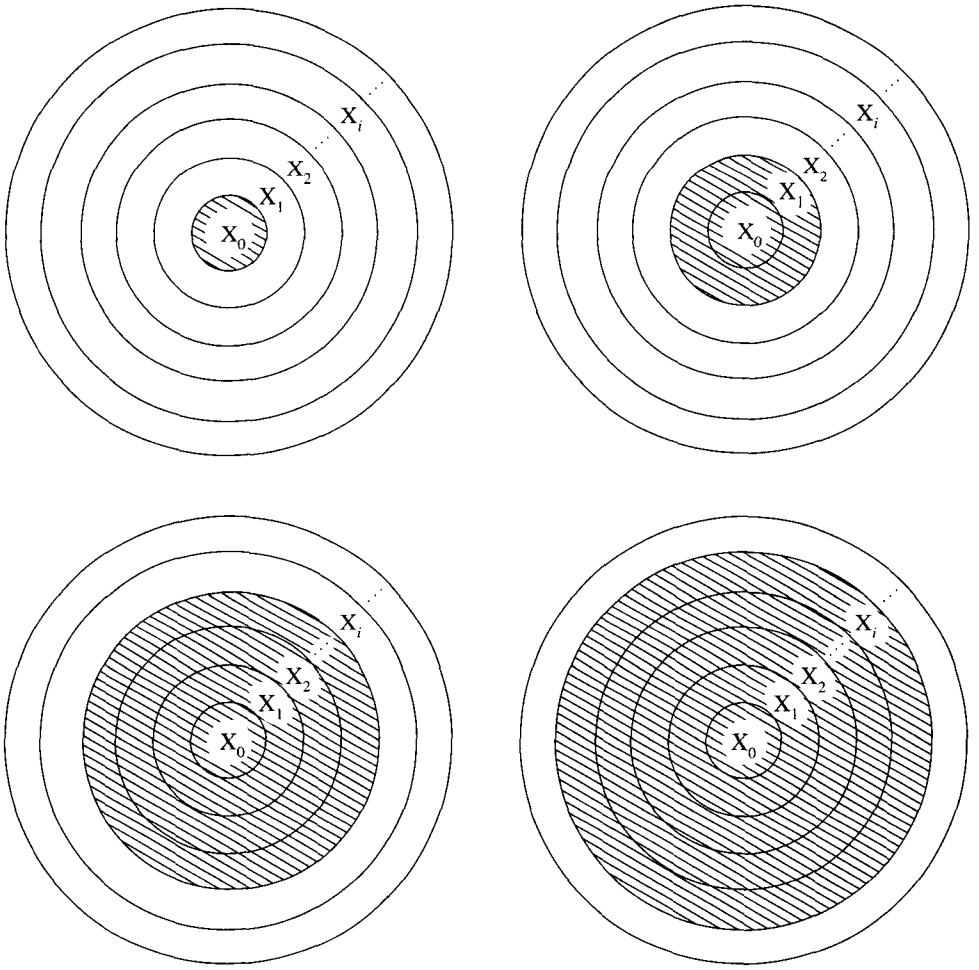


FIGURE 1.3 Principle of mathematical induction.

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 $X_0, X_1, \ldots, X_n$. 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.

To illustrate the steps of an inductive proof we use the natural numbers as the underlying recursively defined set. When establishing properties of natural numbers, it is often the case that the basis consists of the single natural number zero. A recursive definition of this set with basis {0} is given in Definition 1.5.1. Example 1.6.2 shows that this is not necessary; an inductive proof can be initiated using any number n as the basis. The principle of mathematical induction then allows us to conclude that the property holds for all natural numbers greater than or equal to n.

Example 1.6.1

Induction is used to prove that $0 + 1 + \cdots + n = n(n + 1)/2$. 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 $k = 1, 2, \ldots, n$ 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. □

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 operations to produce the desired result.

Example 1.6.2

$n! > 2^n$, for $n \geq 4$.

Basis: $n = 4$. $4! = 24 > 16 = 2^4$.

Inductive Hypothesis: Assume that $k! > 2^k$ for all values $k = 4, 5, \ldots, n$.

Inductive Step: We need to prove that $(n+1)! > 2^{n+1}$.

$$\begin{aligned}
(n+1)! &= n!(n+1) \\
&> 2^n(n+1) && \text{(inductive hypothesis)} \\
&> 2^n 2 && \text{(since } n+1 > 2\text{)} \\
&= 2^{n+1}
\end{aligned}$$

□

The subsets of a finite set can be defined recursively using the binary operation union (Exercise 31). In Example 1.6.3, induction is used to establish the relationship between the cardinality of a finite set and that of its power set.

Example 1.6.3

Induction on the cardinality of X is used to show that $card(\mathcal{P}(X)) = 2^{card(X)}$ for any finite set X.

Basis: $card(X) = 0$. Then $X = \emptyset$ and $\mathcal{P}(X) = \{\emptyset\}$. So $2^{card(X)} = 2^0 = 1 = card(\mathcal{P}(X))$.

Inductive Hypothesis: Assume that $card(\mathcal{P}(X)) = 2^{card(X)}$ for all sets of cardinality $0, 1, \ldots, n$.

Inductive Step: Let X be any set of cardinality $n + 1$. The proof is completed by showing that $card(\mathcal{P}(X)) = 2^{card(X)}$. Since X is nonempty, it must contain at least one element. Choose an element $a \in X$. Let $Y = X - \{a\}$. Then $card(Y) = n$ and $card(\mathcal{P}(Y)) = 2^{card(Y)}$ by the inductive hypothesis.

The subsets of X can be partitioned into two distinct categories, those that contain the element a and those that do not contain a. The subsets of X that do not contain a are precisely the sets in $\mathcal{P}(Y)$. A subset of X containing a can be obtained by augmenting a subset of Y with the element a. Thus

$$\mathcal{P}(X) = \mathcal{P}(Y) \cup \{Y_0 \cup \{a\} \mid Y_0 \in \mathcal{P}(Y)\}.$$

Since the two components of the union are disjoint, the cardinality of the union is the sum of the cardinalities of each of the sets. Employing the inductive hypothesis yields

$$\begin{aligned} card(\mathcal{P}(X)) &= card(\mathcal{P}(Y)) + card(\{Y_0 \cup \{a\} \mid Y_0 \in \mathcal{P}(Y)\}) \\ &= card(\mathcal{P}(Y)) + card(\mathcal{P}(Y)) \\ &= 2^n + 2^n \\ &= 2^{n+1} \\ &= 2^{card(X)}. \end{aligned}$$

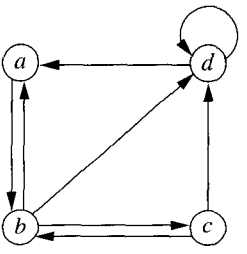
□

1.7 Directed Graphs

A mathematical structure consists of functions and relations on a set or sets and distinguished elements from the sets. A *distinguished element* is an element of a set that has special properties that distinguish it from the other elements. The natural numbers, as defined in Definition 1.5.1, can be expressed as a structure $(\mathbf{N}, s, 0)$. The set $\mathbf{N}$ contains the natural numbers, s is a unary function on $\mathbf{N}$, and 0 is a distinguished element of $\mathbf{N}$. Zero is distinguished because of its explicit role in the definition of the natural numbers.

Graphs are frequently used in both formal language theory and automata theory because they provide the ability 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** of length n from x to y in a directed graph is a sequence of nodes $x_0, x_1, \ldots, x_n$ satisfying



$N = \{a, b, c, d\}$

$A = \{[a,b], [b,a], [b,c], [b,d], [c,b], [c,d], [d,a], [d,d]\}$

Node	In-degree	Out-degree
a	2	1
b	2	3
c	1	2
d	3	2

FIGURE 1.4 Directed graph.

i) x_i is adjacent to x_{i-1}, for $i = 1, 2, \ldots, n$

ii) $x = x_0$

iii) $y = x_n$.

The node x is the initial node of the path and y is the terminal node. 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, c, 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

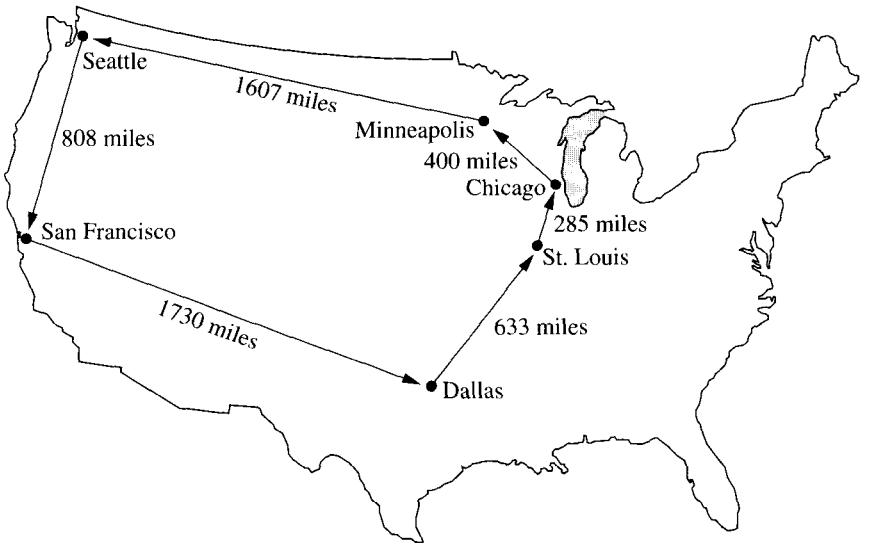


FIGURE 1.5 Labeled directed graph.

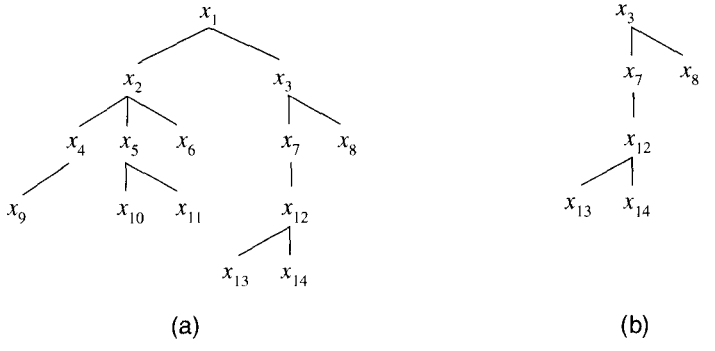


FIGURE 1.6 (a) Tree with root x_1. (b) Subtree generated by x_3.

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.

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 38 and 39). 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 $\mathbf{N} \times \mathbf{N}$. 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 $z_1, z_2, \ldots, z_n$ 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 $[x, y] \in$ LEFTOF; $[y, x] \in$ 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 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 $x_9, x_{10}, x_{11}, x_6, x_{13}, x_{14}, x_8$.

The application of the inductive hypothesis in the proofs by mathematical induction in Section 1.6 used only the assumption that the property in question 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**. We will use strong induction to establish the relationship between the number of leaves and the number of arcs in a strictly binary tree. The process begins with a recursive definition of strictly binary trees.

Example 1.7.1

A tree is called a **strictly binary tree** if every node either is a leaf or has precisely two children. The family of strictly binary trees can be defined recursively as follows:

i) Basis: A directed graph $T_1 = (\{r\}, \emptyset, r)$ is a strictly binary tree.

ii) 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.

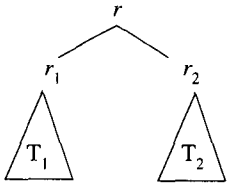
iii) Closure: T is a strictly binary tree only if it can be obtained from the basis elements by a finite number of applications of the construction given in the recursive step.

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 $lv(\mathrm{T})$ and $arc(\mathrm{T})$ denote the number of leaves and arcs in a strictly binary tree T. We prove, by induction on the number of leaves, that $2lv(\mathrm{T}) - 2 = arc(\mathrm{T})$ for all strictly binary trees.

Basis: The basis consists of strictly binary trees containing a single leaf, the trees defined by the basis condition of the recursive definition. 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 $2lv(\mathrm{T}) - 2 = arc(\mathrm{T})$.

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, $lv(\mathrm{T}) = lv(\mathrm{T}_1) + lv(\mathrm{T}_2)$. 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 strong induction to establish the desired equality. By the inductive hypothesis,

$$2lv(\mathrm{T}_1) - 2 = arc(\mathrm{T}_1)$$
$$2lv(\mathrm{T}_2) - 2 = arc(\mathrm{T}_2).$$

Now,

$$\begin{aligned} arc(\mathrm{T}) &= arc(\mathrm{T}_1) + arc(\mathrm{T}_2) + 2 \\ &= 2lv(\mathrm{T}_1) - 2 + 2lv(\mathrm{T}_2) - 2 + 2 \\ &= 2(lv(\mathrm{T}_1) + lv(\mathrm{T}_2)) - 2 \\ &= 2(lv(\mathrm{T})) - 2, \end{aligned}$$

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) below.
   a) $X \cup Y$
   b) $X \cap Y$
   c) $X - Y$
   d) $Y - X$
   e) $\mathcal{P}(X)$
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. Give functions $f : \mathbf{N} \to \mathbf{N}$ that satisfy
   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 not total, but is one-to-one and onto.
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 an example of a binary relation on $\mathbf{N} \times \mathbf{N}$ that is
   a) reflexive and symmetric but not transitive.
   b) reflexive and transitive but not symmetric.
   c) symmetric and transitive but not reflexive.
7. Let $\equiv$ be the binary relation on $\mathbf{N}$ defined by $n \equiv m$ if, and only if, $n = m$. Prove that $\equiv$ is an equivalence relation. Describe the equivalence classes of $\equiv$.
8. Let $\equiv$ be the binary relation on $\mathbf{N}$ defined by $n \equiv m$ for all $n, m \in \mathbf{N}$. Prove that $\equiv$ is an equivalence relation. Describe the equivalence classes of $\equiv$.
9. Show that the binary relation LT, less than, is not an equivalence relation.
10. Let $\equiv_p$ be the binary relation on $\mathbf{N}$ defined by $n \equiv_p m$ if $n \bmod p = m \bmod p$. Prove that $\equiv_p$ is an equivalence relation. Describe the equivalence classes of $\equiv_p$.
11. Let $X_1, \ldots, X_n$ be a partition of a set X. Define an equivalence relation $\equiv$ on X whose equivalence classes are precisely the sets $X_1, \ldots, X_n$.
12. 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.
13. Prove that the set of even natural numbers is denumerable.

14. Prove that the set of even integers is denumerable.
15. Prove that the set of nonnegative rational numbers is denumerable.
16. Prove that the union of two disjoint countable sets is countable.
17. 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.
18. Prove that there are an uncountable number of total functions from $\mathbf{N}$ to $\{0, 1\}$.
19. A total function f from $\mathbf{N}$ to $\mathbf{N}$ is said to be repeating if $f(n) = f(n + 1)$ for some $n \in \mathbf{N}$. 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.
20. A total function f from $\mathbf{N}$ to $\mathbf{N}$ is monotone-increasing if $f(n) < f(n + 1)$ for all $n \in \mathbf{N}$. Prove that there are an uncountable number of monotone increasing functions.
21. Prove that there are uncountably many functions from $\mathbf{N}$ to $\mathbf{N}$ that have a fixed point. See Example 1.4.3 for the definition of a fixed point.
22. Prove that the binary relation on sets defined by $X \equiv Y$ if, and only if, $card(X) = card(Y)$ is an equivalence relation.
23. Prove the Schröder-Bernstein theorem.
24. Give a recursive definition of the relation *is equal to* on $\mathbf{N} \times \mathbf{N}$ using the operator s.
25. Give a recursive definition of the relation *greater than* on $\mathbf{N} \times \mathbf{N}$ using the successor operator s.
26. Give a recursive definition of the set of points $[m, n]$ that lie on the line $n = 3m$ in $\mathbf{N} \times \mathbf{N}$. Use s as the operator in the definition.
27. Give a recursive definition of the set of points $[m, n]$ that lie on or under the line $n = 3m$ in $\mathbf{N} \times \mathbf{N}$. Use s as the operator in the definition.
28. Give a recursive definition of the operation of multiplication of natural numbers using the operations s and addition.
29. Give a recursive definition of the predecessor operation

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

using the operator s.

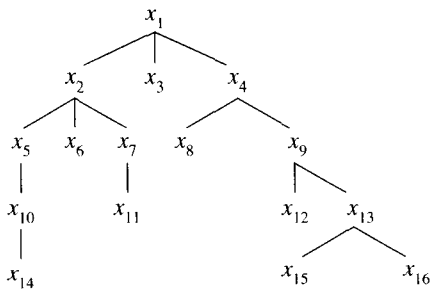
30. Subtraction on the set of natural numbers is defined by

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

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

31. 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.
32. Give a recursive definition of the set of finite subsets of **N**. Use union and the successor s as the operators in the definition.
33. Prove that $2 + 5 + 8 + \cdots + (3n - 1) = n(3n + 1)/2$ for all $n > 0$.
34. Prove that $1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$ for all $n \geq 0$.
35. Prove $1 + 2^n < 3^n$ for all $n > 2$.
36. Prove that 3 is a factor of $n^3 - n + 3$ for all $n \geq 0$.
37. 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.
38. 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.
39. Give a recursive definition of the set of ancestors of a node x in a tree.
40. List the members of the relation LEFTOF for the tree in Figure 1.6(a).
41. 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

42. Prove that a strictly binary tree with n leaves contains $2n - 1$ nodes.
43. A **complete binary tree** of depth n is a strictly binary tree in which every node on levels $1, 2, \ldots, n - 1$ 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

This chapter reviews the topics normally covered in a first course in discrete mathematics. Introductory texts in discrete mathematics include Kolman and Busby [1984], Johnsonbaugh [1984], Gersting [1982], Sahni [1981], and Tremblay and Manohar [1975]. A more sophisticated 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 books that provide detailed presentations of the topics introduced in this chapter. An introduction to naive set theory can be found in Halmos [1974] and Stoll [1963]. The texts by Wilson [1985], Ore [1963], Bondy and Murty [1977], and Busacker and Saaty [1965] introduce the theory of graphs. The diagonalization argument was originally presented by Cantor in 1874 and is reproduced in Cantor [1947]. 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: natural languages, computer languages, and mathematical languages. A general definition of language must encompass 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. This is the broadest possible definition, there are no inherent restrictions on the form of the strings that constitute a language.

Languages of interest are not made up of arbitrary strings, but rather strings that satisfy certain properties. These properties define the syntax of the language. Recursive definitions and set operations are used to enforce syntactic restrictions on the strings of a language. The chapter concludes with the introduction of a family of languages known as the *regular sets.* A regular set is constructed recursively from the empty set and singleton sets. Although we introduce the regular sets via a set-theoretic construction, as we progress we will see that they occur naturally as the languages generated by regular grammars and recognized by finite-state machines.

2.1 Strings and Languages

A **string** over a set X is a finite sequence of elements from X. Strings are the fundamental objects used in the definition of languages. The set of elements from which the strings

are built is called the **alphabet** of the language. An alphabet consists of a finite set of indivisible objects. The alphabet of a language is denoted Σ.

The alphabet of a natural language, like English or French, consists of the words of the language. The words 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. The sentence that you have just read, omitting the punctuation, is such a string. The alphabet of a computer language consists of the permissible keywords, variables, and symbols of the language. A string over this language is a sequence of computer code.

Because the elements of the alphabet of a language are indivisible, they are generally denoted by single characters. Letters a, b, c, d, e, with or without subscripts, are used to represent the elements of 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 $\lambda, a, aa, aaa, \ldots$. 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 $length(w)$. 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: $aaa\ aab\ aac\ aba\ abb\ abc\ aca\ acb\ acc$
$baa\ bab\ bac\ bba\ bbb\ bbc\ bca\ bcb\ bcc$
$caa\ cab\ cac\ cba\ cbb\ cbc\ cca\ ccb\ ccc$ □

A language consists of strings over an alphabet. Usually some restrictions are placed on the strings of the language. The English language consists of those strings of words that we call sentences. Not all strings of words form sentences in a language, only those satisfying certain conditions on the order and type of the constituent words. Consequently, 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 is the binary operation of taking two strings and "gluing them together" to construct a new string. Concatenation is the fundamental operation in the generation of strings. A formal definition is given by recursion on the length of the second string in the concatenation. At this point, the primitive operation of adjoining a single character 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 as follows:

i) Basis: If $length(v) = 0$, then $v = \lambda$ and $uv = u$.

ii) Recursive step: Let v be a string with $length(v) = n > 0$. 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

$$uv = abca \qquad vw = cabb$$
$$(uv)w = abcabb \qquad u(vw) = abcabb.$$ □

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: $length(w) = 0$. 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, etc. 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: $length(u) = 0$. Then $u = \lambda$ and $\lambda^R = \lambda$.

ii) Recursive step: If $length(u) = n > 0$, 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: $length(v) = 0$. 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, for any string v of length $n + 1$, that $(uv)^R = v^R u^R$. 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 having specified forms. The acceptable forms define the syntax of the language.

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 uaa, 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. □

Example 2.2.2

The language L consists of strings over $\{a, b\}$ in which each occurrence of b is immediately preceded by an a. For example, λ, a, $abaab$ are in L and bb, bab, abb are not in L.

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. □

Recursive definitions provide a tool for defining the strings of a language. Examples 2.2.1 and 2.2.2 have shown that requirements on order, positioning, and parity can be obtained using a recursive generation of strings. The process of string generation in a recursive definition, however, is unsuitable for enforcing the syntactic requirements of complex languages such as mathematical 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.

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 2.2.3

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

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

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

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

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

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

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

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

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

Defining languages requires the 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 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. The result of a unary operation on a language or a binary operation on two languages defines another language. Example 2.2.4 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 2.2.4

The language $L = \{a, b\}^*\{bb\}\{a, b\}^*$ consists of the strings over $\{a, b\}$ that contain the substring bb. The concatenation of the set $\{bb\}$ 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. □

Example 2.2.5

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 $L = \{aa\}\{a, b\}^* \cup \{a, b\}^*\{bb\}$. □

Example 2.2.6

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 L_1^* and L_2^*. □

Example 2.2.7

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 $\{a, b\}\{aa, bb, ab, ba\}^*$. □

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 construction process.

A set is regular if it can be generated from the empty set, the set containing the null string, and the elements of the alphabet using union, concatenation, and the Kleene star operation. The regular sets are an important family of languages, occurring in both formal language theory 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:

i) Basis: $\emptyset$, $\{\lambda\}$ and $\{a\}$, for every $a \in \Sigma$, are regular sets over Σ.

ii) Recursive step: Let X and Y be regular sets over Σ. The sets

$$X \cup Y$$
$$XY$$
$$X^*$$

are regular sets over Σ.

iii) 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.

Example 2.3.1

The language from Example 2.2.4, 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.

Applying union and the Kleene star operation produces $\{a, b\}^*$, the set of all strings over $\{a, b\}$. By concatenation, $\{b\}\{b\} = \{bb\}$ is regular. Applying concatenation twice yields $\{a, b\}^*\{bb\}\{a, b\}^*$. □

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. □

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 set $\{b\}$ is represented by $\boldsymbol{b}$, 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 $\boldsymbol{a}$, for every $a \in \Sigma$, are regular expressions over Σ.

ii) Recursive step: Let $\boldsymbol{u}$ and $\boldsymbol{v}$ be regular expressions over Σ. The expressions

$$(\boldsymbol{u} \cup \boldsymbol{v})$$

$$(\boldsymbol{uv})$$

$$(\boldsymbol{u}^*)$$

are regular expressions over Σ.

iii) Closure: $\boldsymbol{u}$ 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 operation 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 $(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{bb}(\boldsymbol{a} \cup \boldsymbol{b})^*$ and $\boldsymbol{a}(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{b}(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{a}$, respectively.

The notation $\boldsymbol{u}^+$ is used to abbreviate the expression $\boldsymbol{uu}^*$. Similarly, $\boldsymbol{u}^2$ denotes the regular expression $\boldsymbol{uu}$, $\boldsymbol{u}^3$ denotes $\boldsymbol{u}^2\,\boldsymbol{u}$, etc.

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

□

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

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

Example 2.3.4

The regular expressions $\boldsymbol{(a \cup b)^*aa(a \cup b)^*}$ and $\boldsymbol{(a \cup b)^*bb(a \cup b)^*}$ represent the regular sets with strings containing aa and bb, respectively. Combining these two expressions with the $\cup$ operator yields the expression $\boldsymbol{(a \cup b)^*aa(a \cup b)^* \cup (a \cup b)^*bb(a \cup b)^*}$ 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 $\boldsymbol{a^*ba^*ba^*}$. □

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. □

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 expression for this set is $a^*(ba^*ba^*)^*$. □

The previous examples show that the regular expression definition of a set is not unique. Two expressions that represent the same set are called **equivalent.** The identities in Table 2.3.1 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 union. Identities 9 and 10 are the distributive laws translated to the regular expression notation. The final set of expressions specifies all sequences of elements from the sets represented by u and v.

Example 2.3.8

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; 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 2.3.1 as follows:

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

□

TABLE 2.3.1 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.	$(u \cup v)^* = (u^* \cup v)^*$ $= u^*(u \cup v)^* = (u \cup vu^*)^*$ $= (u^*v^*)^* = u^*(vu^*)^*$ $= (u^*v)^*u^*$

Example 2.3.9

The regular expression $(a \cup b \cup c)^*bc(a \cup b \cup c)^*$ defines the set of strings containing the substring bc. The expression $(a \cup b \cup c)^*$ is the set of all strings over $\{a, b, c\}$. Following the technique of Example 2.3.5, we use concatenation to explicitly insert the substring bc in the string, preceded and followed by any sequence of a's, b's, and c's. □

Example 2.3.10

Let L be the language defined by $c^*(b \cup ac^*)^*$. 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^*. When an element from ac^* occurs, any number of a's or b's, in any order, can follow. Putting these observations together, we see that L consists of all strings that do not contain the substring bc. 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 $c^*(b \cup (ac^*))^*$. □

The previous two examples show that it is often easier to build a regular set (expression) in which every string satisfies a given condition than one consisting of all strings that do not satisfy the condition. Techniques for constructing a regular expression for a language from an expression defining its complement will be given in Chapter 7.

It is important to note that there are languages that cannot be defined by regular expressions. We will see that there is no regular expression that defines the language $\{a^n b^n \mid n \geq 0\}$.

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. Give a recursive definition of the set of strings over $\{a, b\}$ that contains all and only those strings with an equal number of a's and b's. Use concatenation as the operator.
5. Give a recursive definition of the set $\{a^i b^j \mid 0 < i < j\}$.
6. 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.
7. Prove that every string in the language defined in Example 2.2.2 has at least as many a's as b's. Let $n_a(u)$ and $n_b(u)$ be the number of a's and b's in a string u. The inductive proof should establish the inequality $n_a(u) \geq n_b(u)$.
8. 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 iterations 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 $n_a(u)$ denote the number of a's in a string u and $n_b(u)$ denote the number of b's in u.
9. 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:

   i) Basis: λ and a, for all $a \in \Sigma$, are palindromes.

   ii) Recursive step: If w is a palindrome and $a \in \Sigma$, then awa is a palindrome.

   iii) 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.
10. Let X = $\{aa, bb\}$ and Y = $\{\lambda, b, ab\}$.

    a) List the strings in the set XY.

    b) List the strings of the set Y* of length three or less.

    c) How many strings of length 6 are there in X*?

11. 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 12 through 37, give a regular expression that represents the described set.

12. 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.
13. The same set as Exercise 12 without the null string.
14. The set of strings of length two or more over $\{a,b\}$ in which all the a's precede the b's.
15. The set of strings over $\{a,b\}$ that contain the substring aa and the substring bb.
16. The set of strings over $\{a,b\}$ in which the substring aa occurs at least twice. *Hint:* Beware of the substring aaa.
17. The set of strings over $\{a,b,c\}$ that do not contain the substring aa.
18. The set of strings over $\{a,b\}$ that do not begin with the substring aaa.
19. The set of strings over $\{a,b\}$ that do not contain the substring aaa.
20. The set of strings over $\{a,b\}$ that do not contain the substring aba.
21. The set of strings over $\{a,b\}$ in which the substring aa occurs exactly once.
22. The set of strings over $\{a,b,c\}$ that begin with a, contain exactly two b's, and end with cc.
23. The set of strings over $\{a,b\}$ that contain the substring ab and the substring ba.
24. The set of strings over $\{a,b,c\}$ that contain the substrings aa, bb, and cc.
25. The set of strings over $\{a,b,c\}$ in which every b is immediately followed by at least one c.
26. The set of strings over $\{a,b,c\}$ with length three.
27. The set of strings over $\{a,b,c\}$ with length less than three.
28. The set of strings over $\{a,b,c\}$ with length greater than three.
29. The set of strings over $\{a,b\}$ in which the number of a's is divisible by three.
30. The set of strings over $\{a,b,c\}$ in which the total number of b's and c's is three.
31. 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.
32. The set of strings of odd length over $\{a,b\}$ that contain the substring bb.
33. The set of strings of even length over $\{a,b,c\}$ that contain exactly one a.
34. The set of strings over $\{a,b,c\}$ with an odd number of occurrences of the substring ab.
35. The set of strings of odd length over $\{a,b\}$ that contain exactly two b's.
36. The set of strings over $\{a,b\}$ with an even number of a's or an odd number of b's.

37. 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 7.
38. Use the regular expression identities in Table 2.3.1 to establish the following identities:
    a) $(ba)^+(a^*b^* \cup a^*) = (ba)^*ba^+(b^* \cup \lambda)$
    b) $b^+(a^*b^* \cup \lambda)b = b(b^*a^* \cup \lambda)b^+$
    c) $(a \cup b)^* = (a \cup b)^*b^*$
    d) $(a \cup b)^* = (a^* \cup ba^*)^*$
    e) $(a \cup b)^* = (b^*(a \cup \lambda)b^*)^*$

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

Context-Free Grammars and Parsing

The syntax of a language specifies the permissible forms of the strings of the language. In Chapter 2, set-theoretic operations and recursive definitions were used to construct 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. In this section we introduce a formal system for string generation known as a *context-free grammar.* An element of the language is constructed from the start symbol of the grammar using rules that define permissible string transformations. The derivation of a string consists of a sequence of acceptable transformations.

Context-free grammars, like recursive definitions, generate the strings of a language. The flexibility provided by the rules of a context-free grammar has proven well suited for defining the syntax of programming languages. The grammar that generates the programming language Pascal is used to demonstrate the context-free definition of several common programming language constructs.

The process of analyzing a string for syntactic correctness is known as *parsing.* Defining the syntax of a language by a set of context-free rules facilitates the development of parsing algorithms. Several simple parsers, based on algorithms for traversing directed graphs, are presented in Chapter 4. These algorithms systematically examine derivations to determine if a string is derivable from the start symbol of the grammar.

Context-free grammars are members of a family of the string generation systems known as *phrase-structure grammars.* Another family of grammars, the regular grammars, is introduced as a special case of context-free grammars. These two types of grammars, along with two additional families of grammars, make up the sequence of increasingly powerful string generation systems known as the *Chomsky hierarchy of grammars.* The relationships between the grammars of the Chomsky hierarchy will be examined in Chapter 10.

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, the, John, Jill, hamburger, car, drives, eats, slowly, frequently, big, juicy, 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 rapidly* 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 the brackets ⟨ ⟩.

Since the generation procedure constructs sentences, the initial variable is named ⟨*sentence*⟩. The generation process 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 ⟨*sentence*⟩ are

1. ⟨*sentence*⟩ → ⟨*noun-phrase*⟩⟨*verb-phrase*⟩
2. ⟨*sentence*⟩ → ⟨*noun-phrase*⟩⟨*verb*⟩⟨*direct-object-phrase*⟩

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, ⟨*noun-phrase*⟩

nor ⟨*verb-phrase*⟩, 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. ⟨*noun-phrase*⟩ → ⟨*proper-noun*⟩
4. → ⟨*determiner* ⟩⟨ *common-noun*⟩
5. ⟨*proper-noun*⟩ → *John*
6. → *Jill*
7. ⟨*common-noun*⟩ → *car*
8. → *hamburger*
9. ⟨*determiner*⟩ → *a*
10. → *the*
11. ⟨*verb-phrase*⟩ → ⟨*verb*⟩⟨*adverb*⟩
12. → ⟨*verb*⟩
13. ⟨*verb*⟩ → *drives*
14. → *eats*
15. ⟨*adverb*⟩ → *slowly*
16. → *frequently*

With the exception of ⟨*direct-object-phrase*⟩, rules have been defined for each of the variables that have been introduced. The generation of a sentence consists of repeated rule applications to transform the variable ⟨*sentence*⟩ into a string of terminal symbols. For example, the sentence *Jill drives frequently* is generated by the following transformations:

Derivation	**Rule Applied**
⟨*sentence*⟩ ⇒ ⟨*noun-phrase*⟩⟨*verb-phrase*⟩	1
⇒ ⟨*proper-noun*⟩⟨*verb-phrase*⟩	3
⇒ *Jill* ⟨*verb-phrase*⟩	6
⇒ *Jill* ⟨*verb*⟩⟨*adverb*⟩	11
⇒ *Jill drives* ⟨*adverb*⟩	13
⇒ *Jill drives frequently*	16

The application of a rule transforms one string to another. The symbol $\Rightarrow$, used to designate a rule application, is read "derives." The column on the right indicates the number of the rule that was applied to achieve the transformation. The derivation of a sentence terminates when all variables have been removed from the derived string. The set of terminal strings derivable from the variable ⟨*sentence*⟩ is the language generated by the rules of the example.

To complete the set of rules, the transformations for ⟨*direct-object-phrase*⟩ 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

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.

17. ⟨*adjective-list*⟩ → ⟨*adjective*⟩⟨*adjective-list*⟩
18. → λ
19. ⟨*adjective*⟩ → *big*
20. → *juicy*
21. → *brown*

The definition of ⟨*adjective-list*⟩ follows the standard recursive pattern. Rule 17 defines ⟨*adjective-list*⟩ 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 ⟨*adjective-list*⟩ with the null string. Repeated applications of rule 17 generate a sequence of adjectives. Rules for ⟨*direct-object-phrase*⟩ are constructed using ⟨*adjective-list*⟩:

22. ⟨*direct-object-phrase*⟩ → ⟨*adjective-list*⟩⟨*proper-noun*⟩
23. → ⟨*determiner*⟩⟨*adjective-list*⟩⟨*common-noun*⟩

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

Derivation	Rule Applied
⟨*sentence*⟩ ⇒ ⟨*noun-phrase*⟩ ⟨*verb*⟩ ⟨*direct-object-phrase*⟩	2
⇒ ⟨*proper-noun*⟩ ⟨*verb*⟩ ⟨*direct-object-phrase*⟩	3
⇒ *John* ⟨*verb*⟩ ⟨*direct-object-phrase*⟩	5
⇒ *John eats* ⟨*direct-object-phrase*⟩	14
⇒ *John eats* ⟨*determiner*⟩ ⟨ *adjective-list*⟩ ⟨*common-noun*⟩	23
⇒ *John eats a* ⟨*adjective-list*⟩ ⟨ *common-noun*⟩	9
⇒ *John eats a* ⟨*adjective*⟩ ⟨ *adjective-list*⟩ ⟨*common-noun*⟩	17
⇒ *John eats a big* ⟨*adjective-list*⟩ ⟨*common-noun*⟩	19
⇒ *John eats a big* ⟨*adjective*⟩ ⟨ *adjective-list*⟩ ⟨*common-noun*⟩	17
⇒ *John eats a big juicy* ⟨*adjective-list*⟩ ⟨*common-noun*⟩	20
⇒ *John eats a big juicy* ⟨*common-noun*⟩	18
⇒ *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 ⟨*noun-phrase*⟩ ⟨*verb-phrase*⟩ 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 ⟨*adjective-list*⟩ may be incorporated into the ⟨*noun-phrase*⟩ rules, permitting adjectives to modify a noun:

3′. ⟨*noun-phrase*⟩ → ⟨ *adjective-list*⟩ ⟨ *proper-noun*⟩

4′. → ⟨ *determiner*⟩ ⟨ *adjective-list*⟩ ⟨*common-noun*⟩

With this modification, the string *big John eats frequently* can be derived from the variable ⟨*sentence*⟩.

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

A **rule**, often called a *production,* is an element of the set $V \times (V \cup \Sigma)^*$. The rule $[A, w]$ is usually written $A \rightarrow w$. 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 **lambda**, **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, $a, b, c, \ldots$. 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 uwv. This is denoted $uAv \Rightarrow uwv$. 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

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

can be constructed from the rules of the grammar. The derivability of w from v is denoted $v \overset{*}{\Rightarrow} w$. The set of strings derivable from v, being constructed by a finite but (possibly) 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:

i) Basis: v is derivable from v.

ii) Recursive step: If $u = xAy$ is derivable from v and $A \rightarrow w \in P$, then xwy is derivable from v.

iii) Closure: Precisely those strings constructed from v by finitely many applications of the recursive step are derivable from v.

Note that the definition of a rule uses the $\rightarrow$ notation while its application uses $\Rightarrow$. This is because the two symbols represent relations on different sets. A rule is a member of a relation on $V \times (V \cup \Sigma)^*$, while an application of a rule transforms one string into another and is a member of $(V \cup \Sigma)^+ \times (V \cup \Sigma)^*$. The symbol $\overset{+}{\Rightarrow}$ 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 $v \overset{n}{\Rightarrow} w$. When

more than one grammar is being considered, the notation $v \overset{*}{\underset{G}{\Rightarrow}} w$ will be used to explicitly indicate that the derivation utilizes the 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 $S \overset{*}{\Rightarrow} w$ in G.

ii) A string $w \in \Sigma^*$ is a **sentence** of G if there is a derivation $S \overset{*}{\Rightarrow} w$ in G.

iii) The **language** of G, denoted L(G), is the set $\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$.

The sentential forms are the strings derivable from the start symbol of the grammar. The sentences are the sentential forms that contain only terminal symbols. The language of a grammar is the set of sentences generated by the grammar. A set of strings over an alphabet Σ is called a **context-free language** if there is a context-free grammar that generates it. Two grammars are said to be equivalent if they generate the same language.

Recursion is necessary for a finite set of rules to generate infinite languages and strings of arbitrary length. An A rule of the form $A \rightarrow uAv$ is called **directly recursive**. This rule can generate any number of copies of the string u followed by an A and an equal number of v's. A nonrecursive A rule may then be employed to halt the recursion. A variable A is called recursive if there is a derivation $A \overset{+}{\Rightarrow} uAv$. A derivation $A \Rightarrow w \overset{+}{\Rightarrow} uAv$, where A is not in w, is said to be **indirectly recursive**.

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) 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 decomposed. This decomposition can be graphically depicted by a derivation or parse tree. The tree structure specifies the rule that is applied to each variable

$G = (V, \Sigma, P, S)$
$V = \{S, A\}$
$\Sigma = \{a, b\}$
P: $S \rightarrow AA$
$A \rightarrow AAA \mid bA \mid Ab \mid a$

(a)	(b)	(c)	(d)
$S \Rightarrow AA$	$S \Rightarrow AA$	$S \Rightarrow AA$	$S \Rightarrow AA$
$\Rightarrow aA$	$\Rightarrow AAAA$	$\Rightarrow Aa$	$\Rightarrow aA$
$\Rightarrow aAAA$	$\Rightarrow aAAA$	$\Rightarrow AAAa$	$\Rightarrow aAAA$
$\Rightarrow abAAA$	$\Rightarrow abAAA$	$\Rightarrow AAbAa$	$\Rightarrow aAAa$
$\Rightarrow abaAA$	$\Rightarrow abaAA$	$\Rightarrow AAbaa$	$\Rightarrow abAAa$
$\Rightarrow ababAA$	$\Rightarrow ababAA$	$\Rightarrow AbAbaa$	$\Rightarrow abAbAa$
$\Rightarrow ababaA$	$\Rightarrow ababaA$	$\Rightarrow Ababaa$	$\Rightarrow ababAa$
$\Rightarrow ababaa$	$\Rightarrow ababaa$	$\Rightarrow ababaa$	$\Rightarrow ababaa$

FIGURE 3.1 Sample derivations of *ababaa* in G.

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, Σ, P, S) be a context-free grammar and $S \overset{*}{\underset{G}{\Rightarrow}} w$ a derivation. The **derivation tree**, DT, of $S \overset{*}{\underset{G}{\Rightarrow}} w$ is an ordered tree that can be built iteratively as follows:

i) Initialize DT with root S.

ii) If $A \rightarrow x_1 x_2 \ldots x_n$ with $x_i \in (V \cup \Sigma)$ is the rule in the derivation applied to the string uAv, then add $x_1, x_2, \ldots, x_n$ 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 $A \rightarrow x_1 x_2 \ldots x_n$ 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 $x_1, x_2, \ldots, x_n$. 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 1.7. 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

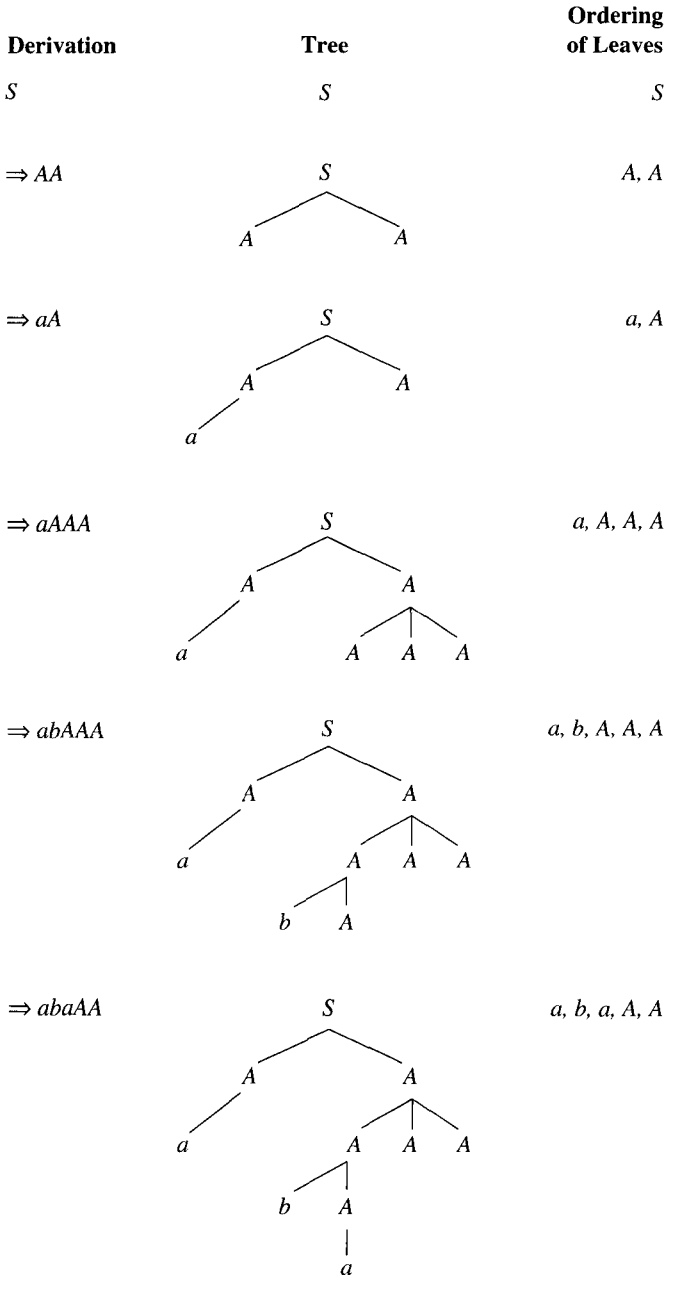


FIGURE 3.2 Construction of derivation tree.

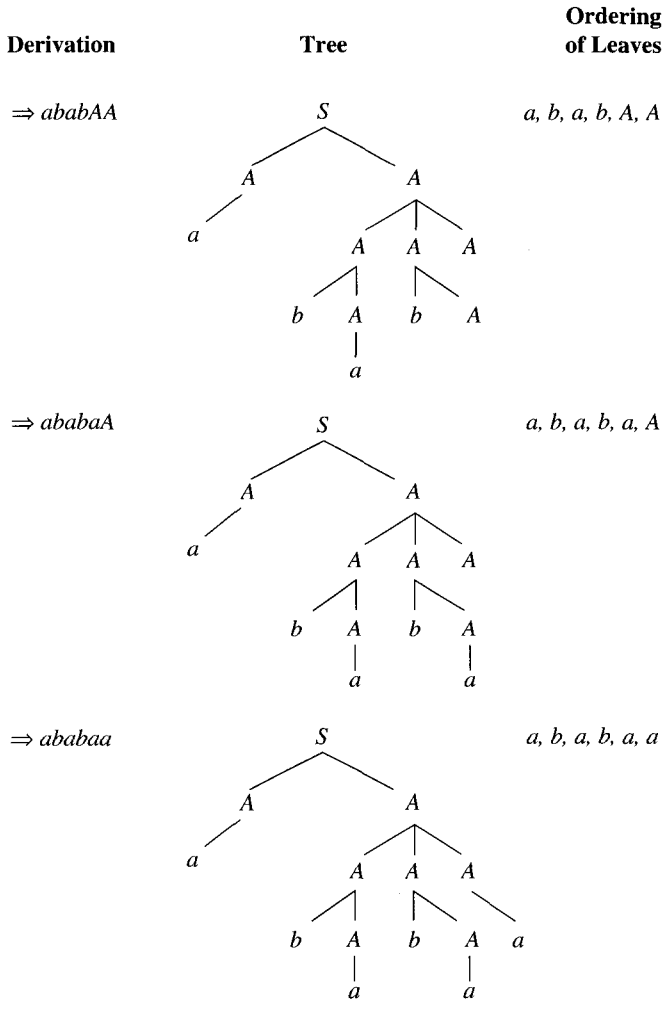


FIGURE 3.2 (Cont.)

decomposed 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. For a node containing a variable A, the rule applied to A can be reconstructed from the children of A in the tree. The rightmost derivation

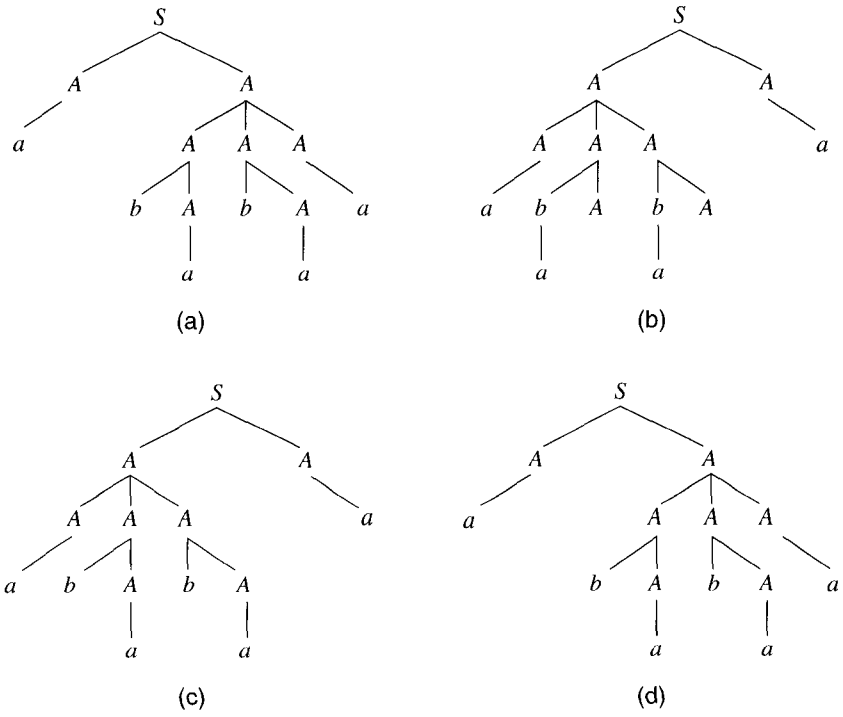


FIGURE 3.3 Trees corresponding to the derivations in Figure 3.1.

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

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 derivation above. The two trees illustrate the distinct decompositions.

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. Mathematical induction

provides a proof technique for establishing that a property holds for all derivations from a given string.

Lemma 3.1.5
Let G be a context-free grammar and $v \stackrel{n}{\Rightarrow} w$ 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) $A_i \stackrel{t_i}{\Rightarrow} p_i$

ii) $w = w_1 p_1 w_2 p_2 \dots w_k p_k w_{k+1}$

iii) $\sum_{i=1}^{k} t_i = n.$

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 $v \stackrel{0}{\Rightarrow} w$. In this case, $w = v$ and each A_i is equal to the corresponding p_i. The desired derivations have the form $A_i \stackrel{0}{\Rightarrow} p_i$.

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

Inductive Step: Let $v \stackrel{n+1}{\Longrightarrow} w$ 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 $v \Rightarrow u \stackrel{n}{\Rightarrow} w$. This reduces the original derivation to a derivation of length n, which is in the correct form for the invocation of the inductive hypothesis, and the application of a single rule.

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

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

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) $A_i \stackrel{t_i}{\Rightarrow} p_i$ for $i = 1, \dots, j-1, j+1, \dots, k$
$B_i \stackrel{s_i}{\Rightarrow} q_i$ for $i = 1, \dots, m$,

ii) $w = w_1 p_1 w_2 \dots p_{j-1} w_j u_1 q_1 u_2 \dots u_m q_m u_{m+1} w_{j+1} p_{j+1} \dots w_k p_k w_{k+1}$,

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 $A_j \to u_1 B_1 u_2 B_2 \ldots u_m B_m u_{m+1}$ with the derivations $B_i \overset{*}{\Rightarrow} q_i$, we obtain a derivation

$$A_j \overset{*}{\Rightarrow} u_1 q_1 u_2 q_2 \ldots u_m q_m u_{m+1} = p_j$$

whose length is the sum of lengths of the derivations from the B_i's plus one. The derivations $A_i \overset{*}{\Rightarrow} p_i, i = 1, \ldots, k$, 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 appropriately combined by additional rules of the grammar.

3.2 Examples of Grammars and Languages

Context-free grammars have been described as generators of languages. Formal languages, like computer languages and natural languages, have requirements that the strings must satisfy in order to be syntactically correct. Grammars must be designed to generate precisely the desired strings and no others. Two approaches may be taken to develop the relationship between grammars and languages. One is to specify a language and then construct a grammar that generates it. Conversely, the rules of a given grammar can be analyzed to determine the language generated.

Initially both of these tasks may seem difficult. With experience, techniques will evolve for generating certain frequently occurring patterns. Building grammars and observing the interactions of the variables and the terminals is the only way to increase one's proficiency with grammars and the formal definition of languages. No proofs will be given in this section; the goal is to use the examples to develop an intuitive understanding of grammars and languages.

In each of the examples, the 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 productions

$$S \to aSa \mid aBa$$
$$B \to bB \mid b.$$

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. □

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

$$S \rightarrow aSdd \mid A$$
$$A \rightarrow bAc \mid bc$$

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. □

Example 3.2.3

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 given in Exercise 2.9. 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. □

Example 3.2.4

The first recursive rule of G generates a trailing b for every a while the second generates two b's for each a.

$$\text{G: } S \rightarrow aSb \mid aSbb \mid \lambda$$

Thus there is at least one b for every a and at most two. The language of the grammar is $\{a^n b^m \mid 0 \leq n \leq m \leq 2n\}$. □

Example 3.2.5

Consider the grammar

$$S \to abScB \mid \lambda$$
$$B \to bB \mid b.$$

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

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

the first occurrence of B generates a single b and the second occurrence produces bb. The language of the grammar consists of 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 $i \neq j$. □

Example 3.2.6

Let G_1 and G_2 be the grammars

$$\begin{aligned} G_1\text{:}\ & S \to AB \\ & A \to aA \mid a \\ & B \to bB \mid \lambda \end{aligned} \qquad \begin{aligned} G_2\text{:}\ & S \to aS \mid aB \\ & B \to bB \mid \lambda. \end{aligned}$$

Both of these grammars generate the language $\boldsymbol{a^+b^*}$. The A rules in G_1 provide the standard method of generating a nonnull string of a's. The use of the lambda rule to terminate the recursion allows the possibility of having no b's. Grammar G_2 builds the same language in a left-to-right manner.

This example shows that there may be many grammars that generate the same language. There may not even be a best grammar. In later chapters, however, we will see that some rules have certain desirable forms that facilitate the mechanical determination of the syntactic correctness of strings. □

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 $\boldsymbol{a^*ba^*ba^*}$.

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

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. □

Example 3.2.8

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

$$\begin{array}{ll} G_1\colon\ S \to AbAbA & G_2\colon\ S \to aS \mid bA \\ \quad A \to aA \mid bA \mid \lambda. & \quad\ A \to aA \mid bC \\ & \quad\ C \to aC \mid bC \mid \lambda \end{array}$$

In G_2, any string can be generated before, between, and after the two b's produced by the S rule. G_2 allows any string to be generated after the second b is produced by the A rule. □

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.

$$S \to aO \mid bO \mid \lambda$$
$$O \to aS \mid bS$$

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

Example 3.2.10

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

$$S \to aS \mid bB \mid \lambda$$
$$B \to aB \mid bS \mid bC$$
$$C \to aC \mid \lambda$$

Deleting all rules containing C yields another grammar that generates L. □

Example 3.2.11

Exercise 2.37 requested a regular expression for the language L 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 with language L straightforward. As in Example 3.2.9, the variables represent the current status of 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

$$S \rightarrow aB \mid bA \mid \lambda$$
$$A \rightarrow aC \mid bS$$
$$B \rightarrow aS \mid bC$$
$$C \rightarrow aA \mid bB.$$

When the variable S is present, the derived string satisfies the specification of 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 in L. □

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

A context-free grammar is regular if the right-hand side of every rule satisfies certain prescribed conditions. The regular grammars constitute an important subclass of the context-free grammars that form the most restrictive family in the Chomsky hierarchy of grammars. In Chapter 7 we will show that regular grammars generate precisely the languages that can be defined by regular expressions.

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) $A \rightarrow aB$

iii) $A \rightarrow \lambda$,

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

A language is said to be regular if it can be generated by a regular grammar. A regular language may be generated by both regular and nonregular grammars. The grammars G_1 and G_2 from Example 3.2.6 generate the language a^+b^*. The grammar G_1 is not regular because the rule $S \rightarrow AB$ does not have the specified form. G_2, however, is a regular grammar. 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.

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.

Example 3.3.1

We will construct a regular grammar that generates the language of the grammar

$$\text{G: } S \rightarrow abSA \mid \lambda$$
$$A \rightarrow Aa \mid \lambda.$$

The language of G is given by the regular expression $\lambda \cup (ab)^+a^*$. An equivalent regular grammar must generate the strings in a left-to-right manner. In the grammar below, 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.

$$S \to aB \mid \lambda$$
$$B \to bS \mid bA$$
$$A \to aA \mid \lambda$$ □

3.4 Grammars and Languages Revisited

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

$$G: \quad S \to AASB \mid AAB$$
$$A \to a$$
$$B \to bbb$$

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 element in L. Every element of the form $a^{2n}b^{3n}$, for $n > 0$, can be derived by the following sequence of rule applications:

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

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 is the result 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 $aASB$, derivable in G, contains one a and no b's.

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 $aASB$, derived above, satisfies this condition since $n_a(aASB) + n_A(aASB) = 2$ and $n_b(aASB) + 3n_B(aASB) = 3$.

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.

ii) $n_A(u) + n_a(u) > 1$.

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

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 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.

Now assume that (i), (ii), and (iii) are satisfied by all strings derivable by n or fewer rule applications. Let w be a string derivable from S by a derivation of length $n + 1$. We must show that the three conditions hold for the string w. A derivation of length $n + 1$ consists of a derivation of length n followed by a single rule application. $S \overset{n+1}{\Longrightarrow} w$ can be written as $S \overset{n}{\Rightarrow} u \Rightarrow w$. For any $v \in (V \cup \Sigma)^*$, 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 u 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 $L(G) = \{a^{2n}b^{3n} \mid n > 0\}$.

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 relationships are sufficient if, when all references to the variables are removed, they yield the desired structure of the terminal strings.

Example 3.4.1

Let G be the grammar

$$S \rightarrow aSb \mid ab.$$

Then $\{a^nb^n \mid n > 0\} \subseteq L(G)$. An arbitrary string in this set has the form a^nb^n with $n > 0$. A derivation for this string is

Derivation	Rule Applied
$S \overset{n-1}{\Longrightarrow} a^{n-1}Sb^{n-1}$	$S \to aSb$
$\Longrightarrow a^n b^n$	$S \to ab$

□

Example 3.4.2

Let G be the grammar

$$S \to aS \mid bB \mid \lambda$$
$$B \to aB \mid bS \mid bC$$
$$C \to aC \mid \lambda$$

given in Example 3.2.10. We will prove that $L(G) = \boldsymbol{a^*(a^*ba^*ba^*)^*}$, the set of all strings over $\{a, b\}$ with 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:

$$S \Rightarrow aS$$
$$S \Rightarrow bB$$
$$S \Rightarrow \lambda.$$

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 $S \overset{n+1}{\Longrightarrow} w$. The key step is to reformulate the derivation to apply the inductive hypothesis. A derivation of w of length $n + 1$ can be written $S \overset{n}{\Rightarrow} u \Rightarrow w$.

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 following table indicates 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. The table shows that 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.

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

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

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

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

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

□

Example 3.4.3

Let G be the grammar

$$S \to aASB \mid \lambda$$
$$A \to ad \mid d$$
$$B \to bb.$$

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	k
$S \Rightarrow aASB$	2	2
$S \Rightarrow \lambda$	0	0

and $j \leq k$ 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 $S \overset{n+1}{\Longrightarrow} w$. The derivation of w can be rewritten $S \overset{n}{\Rightarrow} u \Rightarrow w$. 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 $w \in \mathrm{L(G)}$, the inequality yields $n_a(w) \leq n_b(w)$ as desired. □

Example 3.4.4

The grammar

$$\mathrm{G:}\ S \rightarrow aSdd \mid A$$
$$A \rightarrow bAc \mid bc$$

was constructed in Example 3.2.2 to generate the language $\mathrm{L} = \{a^n b^m c^m d^{2n} \mid n \geq 0,\ m > 0\}$. We develop relationships among the variables and terminals that are sufficient to prove that $\mathrm{L(G)} \subseteq \mathrm{L}$. 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 three conditions:

i) $2n_a(u) = n_d(u)$

ii) $n_b(u) = n_c(u)$

iii) $n_S(u) + n_A(u) + n_b(u) > 0$

for every sentential form u of G.

The equalities guarantee that the terminals occur in correct numerical relationships. The description of the language demands more, that the terminals occur in a specified order. The additional requirement, that the a's (if any) precede the b's (if any) which precede the S or A (if present) which precede the c's (if any) which precede the d's (if any), must be established to ensure the correct order of the components in a terminal string. □

3.5 A Context-Free Grammar for Pascal

In the preceding sections context-free grammars were used to generate small "toy" languages. These examples were given to illustrate the use of context-free grammars as a tool for defining languages. The design of programming languages must contend with a complicated syntax and larger alphabets, increasing the complexity of the rules needed to generate the language. John Backus [1959] and Peter Naur [1963] used a system of rules to define the programming language ALGOL 60. The method of definition employed is now referred to as *Backus-Naur form,* or *BNF.* The syntax of the programming language Pascal, designed by Niklaus Wirth [1971], was also defined using this technique.

A BNF description of a language is a context-free grammar, the only difference being the notation used to define the rules. The definition of Pascal in its BNF form is given in Appendix III. The notational conventions are the same as the natural language example at the beginning of the chapter. The names of the variables are chosen to indicate the components of the language that they generate. Variables are enclosed in ⟨ ⟩. Terminals are represented by character strings delimited by blanks.

The design of a programming language, like the design of a complex program, is greatly simplified by utilizing modularity. The rules for modules, subsets of the grammar that are referenced repeatedly by other rules, can be developed independently. To illustrate the principles of language design and the importance of precise language definition, the syntax of several important constituents of the Pascal language is examined. The rules are given in the notation of context-free grammars.

Numeric constants in Pascal include positive and negative integers and real numbers. The simplest numeric constants, the unsigned integers, are defined first.

⟨*digit*⟩ → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨*unsigned integer*⟩ → ⟨*digit*⟩ ⟨*unsigned integer*⟩ | ⟨*digit*⟩

This definition places no limit on the size of an integer. Overflow conditions are properties of implementations, not of the language. Notice that this definition allows leading zeros in unsigned integers.

Real numbers are defined using the variable ⟨*unsigned integer*⟩. Real number constants in Pascal have several possible forms. Examples include 12.34, 1.1E23, and 2E–3. A separate rule is designed to generate each of these different forms.

⟨*unsigned real*⟩ → ⟨*unsigned integer*⟩ . ⟨*unsigned integer*⟩ |
⟨*unsigned integer*⟩ . ⟨*unsigned integer*⟩ E ⟨*scale factor*⟩ |
⟨*unsigned integer*⟩ E ⟨*scale factor*⟩

⟨*scale factor*⟩ → ⟨*unsigned integer*⟩ | ⟨*sign*⟩ ⟨ *unsigned integer*⟩

⟨*sign*⟩ → + | −

The rules for the numeric constants can be easily constructed utilizing the Pascal definitions of integers and real numbers.

⟨*unsigned number*⟩ → ⟨*unsigned integer*⟩ | ⟨*unsigned real*⟩

⟨*unsigned constant*⟩ → ⟨*unsigned number*⟩

⟨*constant*⟩ → ⟨*unsigned number*⟩ | ⟨*sign*⟩ ⟨ *unsigned number*⟩

Another independent portion of a programming language is the definition of identifiers. Identifiers have many uses: variable names, constant names, procedure and function names. A Pascal identifier consists of a letter followed by any string of letters and digits. These strings can be generated by the following rules:

⟨*letter*⟩ → *a* | *b* | *c* | . . . | *y* | *z*

⟨*identifier*⟩ → ⟨*letter* ⟩ ⟨*identifier-tail*⟩

⟨*identifier-tail*⟩ → ⟨*letter*⟩ ⟨*identifier-tail*⟩ | ⟨*digit*⟩ ⟨*identifier-tail*⟩ | λ

A constant is declared and its value assigned in the constant definition part of a block in Pascal. This is accomplished by the rule

⟨*constant definition*⟩ → ⟨*identifier*⟩ = ⟨*constant*⟩.

The string "*pi* = 3.1415" is a syntactically correct constant definition. What about the string "*plusone* = − − 1"? Are two signs allowed in a constant definition? To determine your answer, attempt to construct a derivation of the string.

3.6 Arithmetic Expressions

The evaluation of expressions is the key to numeric computation. Expressions may be written in prefix, postfix, or infix notation. The postfix and prefix forms are computationally

the most efficient since they require no parentheses; the precedence of the operators is determined completely by their position in the string. The more familiar infix notation utilizes a precedence relationship defined on the operators. Parentheses are used to override the positional properties. Many of the common programming languages, including Pascal, use the infix notation for arithmetic expressions.

We will examine the rules that define the syntax of arithmetic expressions in Pascal. The interactions between the variables ⟨*term*⟩ and ⟨*factor*⟩ specify the relationship between subexpressions and the precedence of the operators. The derivations begin with the rules

⟨*expression*⟩ → ⟨*simple expression*⟩

⟨*simple expression*⟩ → ⟨*term*⟩ |
⟨*sign*⟩ ⟨ *term*⟩ |
⟨*simple expression*⟩ ⟨ *adding operator*⟩ ⟨*term*⟩.

The arithmetic operators for numeric computation are defined by

⟨*adding operator*⟩ → + | −

⟨*multiplying operator*⟩ → ∗ | / | **div** | **mod**.

A ⟨*term*⟩ represents a subexpression of an adding operator. The variable ⟨*term*⟩ follows ⟨*adding operator*⟩ and is combined with the result of the ⟨*simple expression*⟩. Since additive operators have the lowest priority of the infix operators, the computation within a term should be completed before the adding operator is invoked.

⟨*term*⟩ → ⟨*factor*⟩ | ⟨ *term*⟩ ⟨ *multiplying operator*⟩ ⟨ *factor*⟩

When a multiplying operator is present, it is immediately evaluated. The rule for factor shows that its subexpressions are either variables, constants, or other expressions enclosed in parentheses.

⟨*factor*⟩ → ⟨*variable*⟩ | ⟨*unsigned constant*⟩ | (⟨*expression*⟩)

The subgrammar for generating expressions is used to construct the derivations of several expressions. Because of the complexity of the infix notation, simple expressions often require lengthy derivations.

Example 3.6.1

The constant 5 is generated by the derivation

$$
\begin{aligned}
\langle \textit{expression} \rangle &\Rightarrow \langle \textit{simple expression} \rangle \\
&\Rightarrow \langle \textit{term} \rangle \\
&\Rightarrow \langle \textit{factor} \rangle \qquad \text{(since no addition occurs)} \\
&\Rightarrow \langle \textit{unsigned constant} \rangle \\
&\Rightarrow \langle \textit{unsigned number} \rangle \\
&\Rightarrow \langle \textit{unsigned integer} \rangle \\
&\Rightarrow \langle \textit{digit} \rangle \\
&\Rightarrow 5. \qquad \square
\end{aligned}
$$

Example 3.6.2

The expression $x + 5$ consists of two subexpressions: x and 5. The presence of the adding operator dictates the decomposition of $\langle \textit{simple expression} \rangle$, with the subexpressions x and 5 being the operands.

$$
\begin{aligned}
\langle \textit{expression} \rangle &\Rightarrow \langle \textit{simple expression} \rangle \\
&\Rightarrow \langle \textit{simple expression} \rangle \ \langle \textit{adding operator} \rangle \langle \textit{term} \rangle \\
&\Rightarrow \langle \textit{term} \rangle \ \langle \ \textit{adding operator} \rangle \ \langle \textit{term} \rangle \\
&\Rightarrow \langle \textit{factor} \rangle \ \langle \ \textit{adding operator} \rangle \langle \textit{term} \rangle \\
&\Rightarrow \langle \textit{variable} \rangle \ \langle \ \textit{adding operator} \rangle \langle \ \textit{term} \rangle \\
&\Rightarrow \langle \textit{entire variable} \ \rangle \langle \ \textit{adding operator} \rangle \langle \textit{term} \ \rangle \\
&\Rightarrow \langle \textit{variable identifier} \ \rangle \langle \ \textit{adding operator} \rangle \langle \textit{term} \ \rangle \\
&\Rightarrow \langle \textit{identifier} \rangle \ \langle \ \textit{adding operator} \rangle \ \langle \ \textit{term} \ \rangle \\
&\Rightarrow \langle \textit{letter} \rangle \ \langle \ \textit{identifier-tail} \rangle \ \langle \ \textit{adding operator} \rangle \ \langle \ \textit{term} \ \rangle \\
&\Rightarrow x \ \langle \textit{identifier-tail} \rangle \ \langle \textit{adding operator} \rangle \ \langle \ \textit{term} \ \rangle \\
&\Rightarrow x \ \langle \textit{adding operator} \ \rangle \langle \ \textit{term} \ \rangle \\
&\Rightarrow x + \langle \textit{term} \ \rangle \\
&\overset{*}{\Rightarrow} x + 5 \qquad \text{(from Example 3.6.1)}
\end{aligned}
$$

The rules for deriving $\langle \textit{identifier} \rangle$ from $\langle \textit{variable} \rangle$ can be found in Appendix III. The derivation of 5 from $\langle \textit{term} \rangle$ was given in the previous example. □

Example 3.6.3

A derivation of the expression $5 * (x + 5)$ can be constructed using the derivations of the subexpressions 5 and $x + 5$.

$$
\begin{aligned}
\langle \textit{expression} \rangle &\Rightarrow \langle \textit{ simple expression } \rangle \\
&\Rightarrow \langle \textit{term} \rangle \\
&\Rightarrow \langle \textit{term} \rangle \ \langle \textit{ multiplying operator } \rangle \ \langle \textit{ factor } \rangle \\
&\overset{*}{\Rightarrow} 5 \ \langle \textit{multiplying operator} \rangle \ \langle \textit{ factor} \rangle \qquad \text{(from Example 3.6.1)} \\
&\Rightarrow 5 * \langle \textit{factor } \rangle \\
&\Rightarrow 5 * (\langle \textit{expression } \rangle) \\
&\overset{*}{\Rightarrow} 5 * (x + 5) \qquad \text{(from Example 3.6.2)}
\end{aligned}
$$

This example illustrates the relationship between terms and factors. The operands of a multiplicative operator consist of a term and a factor. □

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 $abaabbbabbaa$.
   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

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

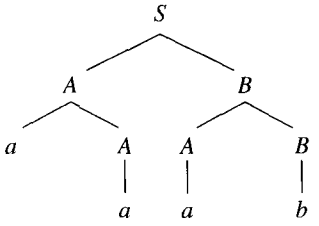
a) Give a leftmost derivation of $abbaab$.

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. 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) $S \to aA$
    $A \to aA \mid bA \mid b$

    b) $S \to aA$
    $A \to aA \mid bB$
    $B \to bB \mid \lambda$

    c) $S \to aS \mid bA$
    $A \to bB$
    $B \to aB \mid \lambda$

    d) $S \to aS \mid bA \mid \lambda$
    $A \to aA \mid bS$

15–39 For each of the languages described in Exercises 12 through 36 in Chapter 2, give a regular grammar G whose language is the specified set.

40. The grammar in Figure 3.1 generates $(\boldsymbol{b^*ab^*ab^*})^+$, the set of all strings with a positive, even number of a's. Prove this.
41. Prove that the grammar given in Example 3.2.2 generates the prescribed language.
42. Let G be the grammar

$$S \to aSb \mid B$$
$$B \to bB \mid b.$$

    Prove that L(G) = $\{a^n b^m \mid 0 \leq n < m\}$.
43. Let G be the grammar

$$S \to aSaa \mid B$$
$$B \to bbBdd \mid C$$
$$C \to bd.$$

    a) What is L(G)?
    b) Prove that L(G) is the set given in part (a).
44. Let G be the grammar

$$S \to aSbS \mid aS \mid \lambda.$$

Prove that every prefix of a string in L(G) has at least as many a's as b's.

For Exercises 45 through 48, use the definition of Pascal in Appendix III to construct the derivations.

45. Construct a derivation of the string $x\,1\,y$ from the variable ⟨*variable*⟩.
46. Construct a derivation of $(x\,1\,y)$ from ⟨*expression*⟩.
47. Construct a derivation for the expression $(x * y * 5)$ from the variable ⟨*expression*⟩.
48. For the not-faint-of-heart: Construct a derivation of $(x + y * (12 + z))$ from the variable ⟨*expression*⟩.
49. Let G_1 and G_2 be the following grammars:

$$\begin{array}{lll} G_1\colon & S \to aABb & \qquad G_2\colon \quad S \to AABB \\ & A \to aA \mid a & \qquad\qquad A \to AA \mid a \\ & B \to bB \mid b & \qquad\qquad B \to BB \mid b. \end{array}$$

    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 $L(G_1) = L(G_2)$.
50. A **right-linear grammar** is a context-free grammar each of whose rules has one of the following forms:
    i) $A \to w$
    ii) $A \to wB$
    iii) $A \to \lambda$,

    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.
51. 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.
52. 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 language Pascal, a descendant of ALGOL, was also defined using the BNF notation. The BNF definition of Pascal is given in Appendix III. The equivalence of context-free languages and the languages generated by BNF definitions was noted by Ginsburg and Rice [1962].

CHAPTER 4

Parsing: An Introduction

Derivations in a context-free grammar provide a mechanism for generating the strings of the language of the grammar. The language of the Backus-Naur definition of Pascal is the set of syntactically correct Pascal programs. An important question remains: How can we determine whether a sequence of Pascal code is a syntactically correct program? The syntax is correct if the string is derivable from the start symbol using the rules of the grammar. Algorithms must be designed to generate derivations for strings in the language of the grammar. When an input string is not in the language, these procedures should discover that no derivation exists. A procedure that performs this function is called a *parsing algorithm* or *parser.*

This chapter introduces several simple parsing algorithms. These parsers are variations of classical algorithms for traversing directed graphs. The parsing algorithms presented in this chapter are valid but incomplete; the answers that they produce are correct, but it is possible that they may enter a nonterminating computation and fail to produce an answer. The potential incompleteness is a consequence of the occurrence of certain types of derivations allowed by the grammar. In Chapter 5 we present a series of rule transformations that produce an equivalent grammar for which the parsing algorithms are guaranteed to terminate.

Grammars that define programming languages often require additional restrictions on the form of the rules to efficiently parse the strings of the language. Grammars specifically designed for efficient syntax analysis are presented in Chapters 16 and 17.

4.1 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. Four distinct derivations of the string *ababaa* are given in the grammar given in Figure 3.1. Any one of these derivations is sufficient to exhibit the syntactic correctness of the string.

The sample derivations generating Pascal expressions in Chapter 3 were given in a leftmost form. This is a natural technique for readers of English since the leftmost variable is the first encountered when scanning a string. To reduce the number of derivations that must be considered by a parser, we prove that every string in the language of a grammar is derivable in a leftmost manner. It follows that a parser that constructs only leftmost derivations is sufficient for deciding whether a string is generated by a grammar.

Theorem 4.1.1

Let G = (V, Σ, 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

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

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. If k is less than n, a new derivation of w with length n is constructed in which the first nonleftmost rule application occurs after step k. This procedure can be repeated, $n - k$ times if necessary, to produce a leftmost derivation.

By the choice of w_k, the derivation $S \overset{k}{\Rightarrow} w_k$ 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 $u_1Au_2Bu_3$ with $u_1 \in \Sigma^*$. The application of a rule $B \rightarrow v$ to w_k has the form

$$w_k = u_1Au_2Bu_3 \Rightarrow u_1Au_2vu_3 = w_{k+1}.$$

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 A occur at the $j + 1$st step in the original derivation. Then the application of the rule $A \rightarrow p$ can be written

$$w_j = u_1Aq \Rightarrow u_1pq = w_{j+1}.$$

The rules applied in steps $k+2$ to j transform the string u_2vu_3 into q. The derivation is completed by the subderivation

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

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 $A \to p$ 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:

$$\begin{aligned}
S &\overset{k}{\Rightarrow} w_k = u_1Au_2Bu_3 \\
&\Rightarrow u_1pu_2Bu_3 && \text{(applying } A \to p\text{)} \\
&\Rightarrow u_1pu_2vu_3 && \text{(applying } B \to v\text{)} \\
&\overset{j-k-1}{\Longrightarrow} u_1pq = w_{j+1} && \text{(using the derivation } u_2vu_3 \overset{*}{\Rightarrow} q\text{)} \\
&\overset{n-j-1}{\Longrightarrow} w_n. && \text{('using the derivation } w_{j+1} \overset{*}{\Rightarrow} w_n\text{)}
\end{aligned}$$

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 4.1.1 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

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

that generates a^*b^*. The string 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 1) establishes the sufficiency of using rightmost derivations for the generation of terminal strings. Leftmost and rightmost derivations of w from v are explicitly denoted $v \overset{*}{\underset{L}{\Rightarrow}} w$ and $v \overset{*}{\underset{R}{\Rightarrow}} w$.

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* are 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 *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 4.1.2

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 4.1.1

Let G be the grammar

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

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 of G is a^+. This language 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 $S \rightarrow a$. □

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 generating inherently ambiguous languages.

Example 4.1.2

Let G be the grammar

$$S \to bS \mid Sb \mid a.$$

The language of G is b^*ab^*. The leftmost derivations

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

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

$$\begin{aligned} G_1\colon\ & S \to bS \mid aA & G_2\colon\ & S \to bS \mid A \\ & A \to bA \mid \lambda & & A \to Ab \mid a. \end{aligned}$$

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

$$\begin{aligned} S &\overset{n}{\Rightarrow} b^nS \\ &\Rightarrow b^naA \\ &\overset{m}{\Rightarrow} b^nab^mA \\ &\Rightarrow b^nab^m. \end{aligned}$$

A derivation in G_2 initially generates the leading b's, followed by the trailing b's and finally the a. □

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 that the application of any other rule makes it impossible to complete a derivation of the string.

Consider the possibilities encountered in constructing a leftmost derivation of the string *bbabb* using the grammar G_2 from Example 4.1.2. There are two S rules that can initiate a derivation. Derivations initiated with the rule $S \to A$ generate strings beginning with a. Consequently, a derivation of *bbabb* must begin with the application of the rule $S \to bS$. The second b is generated by another application of the same rule. At this point, the derivation continues using $S \to A$. Another application of $S \to bS$ would generate the prefix *bbb*. The suffix *bb* is generated by two applications of $A \to Ab$. The derivation is successfully completed with an application of $A \to a$. Since the terminal string specifies the exact sequence of rule applications, the grammar is unambiguous.

Example 4.1.3

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

$$\begin{aligned} S &\Rightarrow aSb & S &\Rightarrow aSbb \\ &\Rightarrow aaSbbb & &\Rightarrow aaSbbb \\ &\Rightarrow aabbb & &\Rightarrow aabbb \end{aligned}$$

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

$$\begin{aligned} S &\rightarrow aSb \mid A \mid \lambda \\ A &\rightarrow aAbb \mid abb \end{aligned}$$

□

A derivation tree depicts the decomposition 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* in Figure 3.1 generate distinct derivation trees.

4.2 The Graph of a Grammar

The leftmost derivations of a context-free grammar G can be represented by a labeled directed graph $g(G)$, the leftmost graph of the grammar G. The nodes of the graph are the left sentential forms of the grammar. A **left sentential form** is a string that can be derived from the start symbol by a leftmost derivation. A string w is adjacent to v in $g(G)$ if $v \underset{L}{\Rightarrow} w$, that is, if w can be obtained from v by one leftmost rule application.

Definition 4.2.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. The **leftmost graph of the grammar** G, $g(G)$, is the labeled directed graph (N, P, A) where the nodes and arcs are defined by

i) $N = \{w \in (V \cup \Sigma)^* \mid S \overset{*}{\underset{L}{\Rightarrow}} w\}$

ii) $A = \{[v, w, r] \in N \times N \times P \mid v \underset{L}{\Rightarrow} w$ by application of rule $r\}$.

A path from S to w in $g(G)$ represents a derivation of w from S in the grammar G. The label on the arc from v to w specifies the rule applied to v to obtain w. The problem of

deciding whether a string w is in L(G) is reduced to that of finding a path from S to w in $g(G)$.

The relationship between leftmost derivations in a grammar and paths in the graph of the grammar can be seen in Figure 4.1. The number of rules that can be applied to the leftmost variable of a sentential form determines the number of children of the node. Since a context-free grammar has finitely many rules, each node has only finitely many children. A graph with this property is called *locally finite.* Graphs of all interesting grammars, however, have infinitely many nodes. The repeated applications of the directly recursive S rule and the indirectly recursive S and B rules generate arbitrarily long paths in the graph of the grammar depicted in Figure 4.1.

The graph of a grammar may take many forms. If every string in the graph has only one leftmost derivation, the graph is a tree with the start symbol as the root. Figure 4.2 gives a portion of the graphs of two ambiguous grammars. The lambda rules in Figure 4.2(b) generate cycles in the graph.

A rightmost graph of a grammar G can be constructed by defining adjacency using rightmost rule applications. The complete graph of G is defined by replacing $v \underset{L}{\Rightarrow} w$ with $v \Rightarrow w$ in condition (ii). Theorem 4.1.1 and Exercise 1 guarantee that every terminal string in the complete graph also occurs in both the leftmost and rightmost graphs. When referring to the graph of a grammar G, unless stated otherwise, we mean the leftmost graph defined in Definition 4.2.1.

Using the graph of a grammar, the generation of derivations is reduced to the construction of paths in a locally finite graph. Standard graph searching techniques will be used to develop several simple parsing algorithms. In graph searching terminology, the graph of a grammar is called an *implicit graph* since its nodes have not been constructed prior to the invocation of the algorithm. The search consists of building the graph as the paths are examined. An important feature of the search is to explicitly construct as little of the implicit graph as possible.

Two distinct strategies may be employed to find a derivation of w from S. The search can begin with the node S and attempt to find the string w. An algorithm utilizing this approach is called a **top-down parser**. An algorithm that begins with the terminal string w and searches for the start symbol is called a **bottom-up** parsing algorithm. An important difference between these strategies is the ease with which the adjacent nodes can be generated while constructing the explicit search structure.

4.3 A Breadth-First Top-Down Parser

The objective of a parser is to determine whether an input string is derivable from the rules of a grammar. A top-down parser constructs derivations by applying rules to the leftmost variable of a sentential form. The application of a rule $A \rightarrow w$ in a leftmost derivation has the form $uAv \Rightarrow uwv$. The string u, consisting solely of terminals, is called the **terminal**

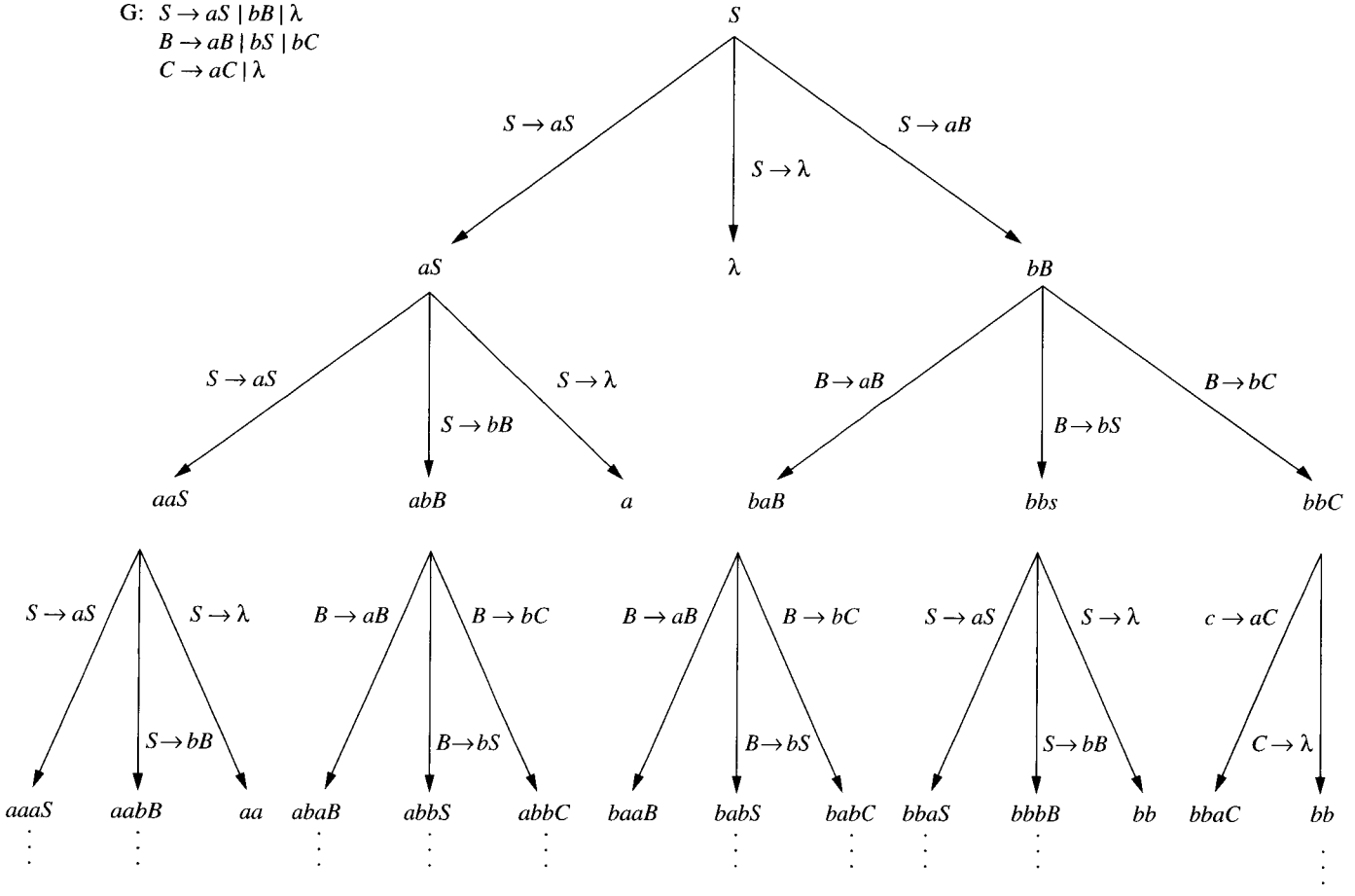
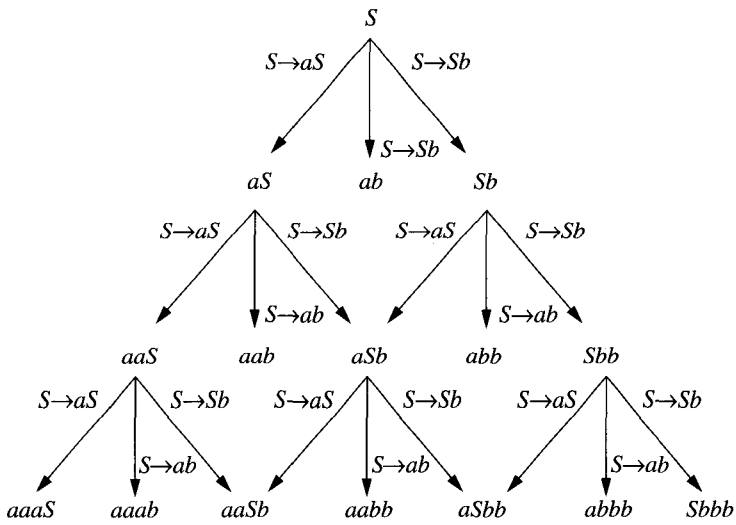
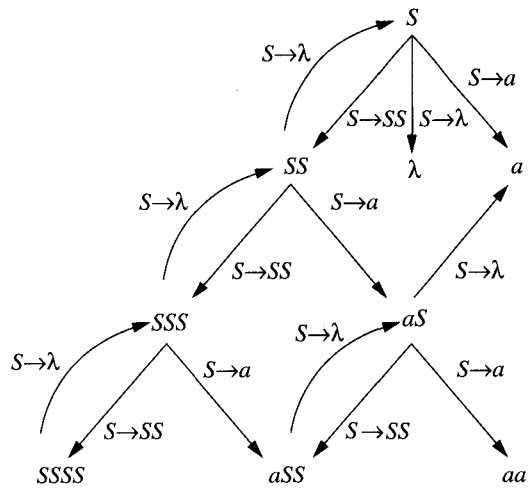


FIGURE 4.1 Graph of grammar G.



(a)



(b)

FIGURE 4.2 Graphs of ambiguous grammars. (a) $S \rightarrow aS \mid Sb \mid ab$. (b) $S \rightarrow SS \mid a \mid \lambda$.

prefix of the sentential form uAv. The parsing algorithms use the terminal prefix of the derived string to identify dead-ends. A dead-end is a string that the parser can determine does not occur in a derivation of the input string.

Parsing is an inherently nondeterministic process. In constructing a derivation, there may be several rules that can be applied to a sentential form. It is not known whether the application of a particular rule will lead to a derivation of the input string, a dead-end, or an unending computation. A parse is said to terminate successfully when it produces a derivation of the input string.

Paths beginning with S in the graph of a grammar represent the leftmost derivations of the grammar. The arcs emanating from a node represent the possible rule applications. The parsing algorithm presented in Algorithm 4.3.1 employs a breadth-first search of the implicit graph of a grammar. The algorithm terminates by accepting or rejecting the input string. An input string p is accepted if the parser constructs a derivation of p. If the parser determines that no derivation of p is possible, the string is rejected.

To implement a breadth-first search, the parser builds a search tree T containing nodes from $g(\mathrm{G})$. The **search tree** is the portion of the implicit graph of the grammar that is explicitly examined during the parse. The rules of the grammar are numbered to facilitate the construction of the search tree. The children of a node uAv are added to the tree according to the ordering of the A rules. The search tree is built with directed arcs from each child to its parent (parent pointers).

A queue is used to implement the first-in, first-out memory management strategy required for a breadth-first graph traversal. The queue **Q** is maintained by three functions: *INSERT*(x, **Q**) places the string x at the rear of the queue, *REMOVE*(**Q**) returns the item at the front and deletes it from the queue, and *EMPTY*(**Q**) is a Boolean function that returns true if the queue is empty, false otherwise.

Algorithm 4.3.1
Breadth-First Top-Down Parsing Algorithm

input: context-free grammar $\mathrm{G} = (\mathrm{V}, \Sigma, \mathrm{P}, S)$
string $p \in \Sigma^*$
queue **Q**

1. initialize T with root S
   INSERT(S, **Q**)
2. **repeat**
   2.1. $q :=$ *REMOVE*(**Q**)
   2.2. $i := 0$
   2.3. done:=*false*
   Let $q = uAv$ where A is the leftmost variable in q.
   2.4. **repeat**
      2.4.1. **if** there is no A rule numbered greater than i **then** done := *true*
      2.4.2. **if** not done **then**
         Let $A \rightarrow w$ be the first A rule with number greater than i and

let j be the number of this rule.
2.4.2.1. **if** $uwv \notin \Sigma^*$ and the terminal prefix of uwv matches a prefix of p **then**
2.4.2.1.1. *INSERT*$(uwv, \mathbf{Q})$
2.4.2.1.2. Add node uwv to T. Set a pointer from uwv to q.
end if
end if
2.4.3. $i := j$
until done **or** $p = uwv$
until *EMPTY*$(\mathbf{Q})$ **or** $p = uwv$
3. **if** $p = uwv$ **then** accept **else** reject

The search tree is initialized with root S since a top-down algorithm attempts to find a derivation of p from S. The repeat-until loop in step 2.4 generates the successors of the node with sentential form q in the order specified by the numbering of the rules. The process of generating the successors of a node and adding them to the search tree is called *expanding the node.* Utilizing the queue, nodes are expanded level by level, resulting in the breadth-first construction of T.

The terminal prefix of a string can be used to determine dead-ends in the search. Let uAv be a string in T with terminal prefix u. If u is not a prefix of p, no sequence of rule applications can derive p from uAv. The condition in step 2.4.2.1 checks each node for a prefix match before inserting it into the queue. The node-expansion phase is repeated until the input string is generated or the queue is emptied. The latter occurs only when all possible derivations have been examined and have failed.

Throughout the remainder of this chapter, the grammar AE (additive expressions) is used to demonstrate the construction of derivations by the parsers presented in this chapter. The language of AE consists of arithmetic expressions with the single variable b, the $+$ operator, and parentheses. Strings generated by AE include b, (b), $(b + b)$, and $(b) + b$. The variable S is the start symbol of AE.

$$
\begin{aligned}
\text{AE}: \ & \text{V} = \{S, A, T\} \\
& \Sigma = \{b, +, (,)\} \\
\text{P}: \ & 1.\ S \rightarrow A \\
& 2.\ A \rightarrow T \\
& 3.\ A \rightarrow A + T \\
& 4.\ T \rightarrow b \\
& 5.\ T \rightarrow (A)
\end{aligned}
$$

The search tree constructed by the parse of $(b + b)$ using Algorithm 4.3.1 is given in Figure 4.3. The sentential forms that are generated but not added to the search tree because of the prefix matching conditions are indicated by dotted lines.

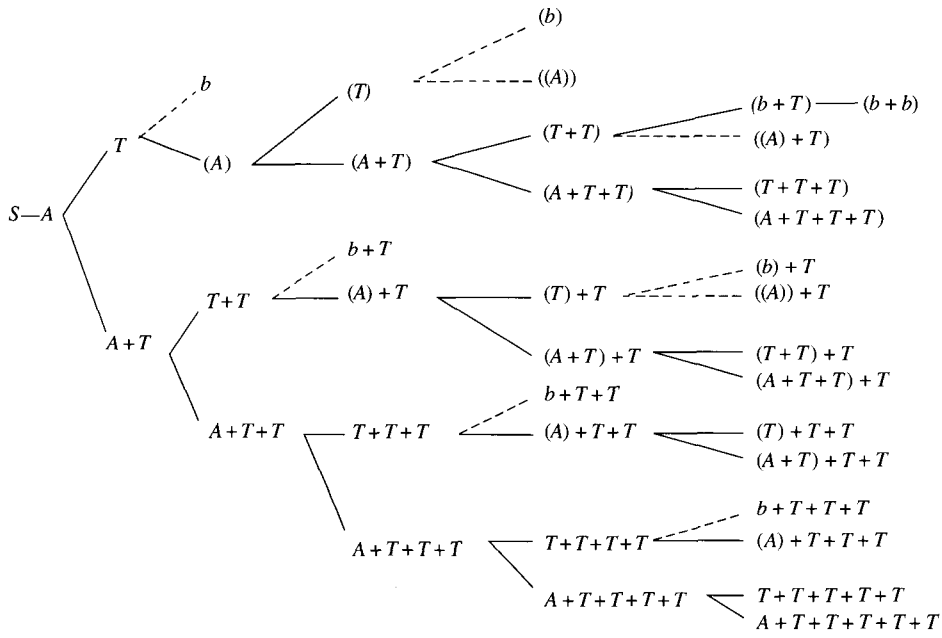


FIGURE 4.3 Breadth-first top-down parse of $(b + b)$.

The comparison in step 2.4.2.1 matches the terminal prefix of the sentential form generated by the parser to the input string. To obtain the information required for the match, the parser "reads" the input string as it builds derivations. Like a human reading a string of text, the parser scans the input string in a left-to-right manner. The growth of the terminal prefix causes the parser to read the entire input string. The derivation of $(b + b)$ exhibits the correspondence between the initial segment of the string scanned by the parser and the terminal prefix of the derived string:

Derivation	Input Read by Parser
$S \Rightarrow A$	λ
$\Rightarrow T$	λ
$\Rightarrow (A)$	(
$\Rightarrow (A + T)$	(
$\Rightarrow (T + T)$	(
$\Rightarrow (b + T)$	$(b+$
$\Rightarrow (b + b)$	$(b + b)$

A parser must not only be able to generate derivations for strings in the language; it must also determine when strings are not in the language. The bottom branch of the search

tree in Figure 4.3 can potentially grow forever. The direct recursion of the rule $A \rightarrow A + T$ builds strings with any number of $+ T$'s as a suffix. In the search for a derivation of a string not in the language, the directly recursive A rule will never generate a prefix capable of terminating the search.

It may be argued that the string $A + T + T$ cannot lead to a derivation of $(b + b)$ and should be declared a dead-end. It is true that the presence of two $+$'s guarantees that no sequence of rule applications can transform $A + T + T$ to $(b + b)$. However, such a determination requires a knowledge of the input string beyond the initial segment that has been scanned by the parser.

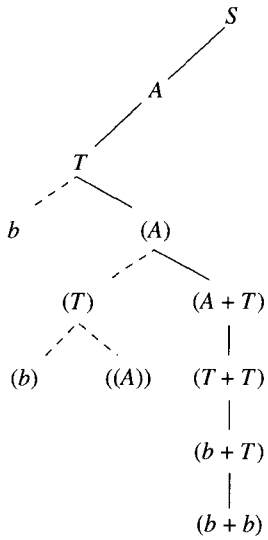
More complicated parsers may scan the input string several times. Multipass parsers may use other criteria, such as the total length of the derived string, to determine dead-ends. An initial pass can be used to obtain the length of the input string. A sentential form derived by the parser containing more terminals than the length of the input string can be declared a dead-end. This condition can be strengthened for grammars without lambda rules. Derivations in these grammars are *noncontracting;* the application of a rule does not reduce the length of a sentential form. It follows that an input string is not derivable from any sentential form of greater length. These conditions cannot be used by a single-pass parser since the length of the input string is not known until the parse is completed.

Although the breadth-first algorithm succeeds in constructing a derivation for any string in the language, the practical application of this approach has several shortcomings. Lengthy derivations and grammars with a large number of rules cause the size of the search tree to increase rapidly. The exponential growth of the search tree is not limited to parsing algorithms but is a general property of breadth-first graph searches. If the grammar can be designed to utilize the prefix matching condition quickly or if other conditions can be developed to find dead-ends in the search, the combinatorial problems associated with growth of the search tree may be delayed.

4.4 A Depth-First Top-Down Parser

A depth-first search of a graph avoids the combinatorial problems associated with a breadth-first search. The traversal moves through the graph examining a single path. In a graph defined by a grammar, this corresponds to exploring a single derivation at a time. When a node is expanded, only one successor is generated and added to the search structure. The choice of the descendant added to the path is arbitrary, and it is possible that the alternative chosen will not produce a derivation of the input string.

The possibility of incorrectly choosing a successor adds two complications to a depth-first parser. The algorithm must be able to determine that an incorrect choice has been made. When this occurs, the parser must have the ability to backtrack and generate the alternative derivations. Figure 4.4 shows the sentential forms constructed by a depth-first parse of the string $(b + b)$ in the graph of the grammar AE. The rules are applied in the order specified by the numbering. The sentential forms connected by dotted lines are those



AE: 1. $S \rightarrow A$
2. $A \rightarrow T$
3. $A \rightarrow A + T$
4. $T \rightarrow b$
5. $T \rightarrow (A)$

$S \Rightarrow A$
$\Rightarrow T$
$\Rightarrow (A)$
$\Rightarrow (A + T)$
$\Rightarrow (T + T)$
$\Rightarrow (b + T)$
$\Rightarrow (b + b)$

FIGURE 4.4 Path and derivation generated by depth-first search for $(b + b)$ in the graph of AE.

that have been generated and determined to be dead-ends. The prefix matching conditions introduced in Algorithm 4.3.1 are used to make these determinations.

The successors of a string are generated in the order specified by the numbering of the rules. In Figure 4.4, the string T is initially expanded with the rule $T \rightarrow b$ since it precedes $T \rightarrow (A)$ in the ordering. When this choice results in a dead-end, the next T rule is applied. Using the prefix matching condition, the parser eventually determines that the rule $A \rightarrow T$ applied to (A) cannot lead to a derivation of $(b + b)$. At this point the search returns to the node (A) and constructs derivations utilizing the A rule $A \rightarrow A + T$.

Backtracking algorithms can be implemented by using a stack to store the information required for generating the alternative paths. A stack **S** is maintained using the procedures *PUSH, POP,* and *EMPTY.* A stack element is an ordered pair $[u, i]$ where u is a sentential form and i the number of the rule applied to u to generate the subsequent node in the path. *PUSH*($[u, i]$, **S**) places the stack item $[u, i]$ on the top of the stack **S**. *POP*(**S**) returns the

top item and deletes it from the stack. *EMPTY*(**S**) is a Boolean function that returns true if the stack is empty, false otherwise.

A stack provides a last-in, first-out memory management strategy. A top-down backtracking algorithm is given in Algorithm 4.4.1. The input string is p, and q is the sentential form contained in the final node of the path currently being explored. When the parse successfully terminates, a derivation can be obtained from the elements stored in the stack.

Algorithm 4.4.1
Depth-First Top-Down Parsing Algorithm

input: context-free grammar G = (V, Σ, P, S)
string $p \in \Sigma^*$
stack **S**

1. *PUSH*([S, 0], **S**)
2. **repeat**
   2.1. [q, i] := *POP*(**S**)
   2.2. dead-end := *false*
   2.3. **repeat**
   Let $q = uAv$ where A is the leftmost variable in q.
   2.3.1. **if** u is not a prefix of p **then** dead-end := *true*
   2.3.2. **if** there are no A rules numbered greater than i **then**
   dead-end := *true*
   2.3.3. **if not** dead-end **then**
   Let $A \rightarrow w$ be the first A rule with number greater than i
   and let j be the number of this rule.
   2.3.3.1. *PUSH*([q, j], **S**)
   2.3.3.2. $q := uwv$
   2.3.3.3. $i := 0$
   end if
   until dead-end **or** $q \in \Sigma^*$
   until $q = p$ **or** *EMPTY*(**S**)
3. **if** $q = p$ **then** accept **else** reject

The algorithm consists of two repeat-until loops. The interior loop extends the current derivation by expanding the final node of the path. This loop terminates when the most recently constructed node completes the derivation of a terminal string or is determined to be a dead-end. There are three ways in which dead-ends can be detected. The terminal prefix of q may not match an initial substring of p (step 2.3.1). There may be no rules to apply to the leftmost variable in the q (step 2.3.2). This occurs when all the appropriate rules have been examined and have produced dead-ends. Finally, a terminal string other than p may be derived.

When a dead-end is discovered, the outer loop pops the stack to implement the backtracking. The order in which the rules are applied is determined by the numbering of the rules and the integer stored in the stack. When the stack is popped, step 2.1 restores the sentential form in the preceding node. If the top stack element is $[uAv, i]$, with A the leftmost variable in the string, the first A rule with number greater than i is applied to extend the derivation.

Example 4.4.1

The top-down backtracking algorithm is used to construct a derivation of $(b + b)$. The action of the parser is demonstrated by tracing the sequence of nodes generated. The strings at the bottom of each column, except the final one, are dead-ends that cause the parser to backtrack. The items popped from the stack are indicated by the slash. A stack item to the immediate right of a popped item is the alternative generated by backtracking. For example, the string T is expanded with rule 5, $T \rightarrow (A)$, after the expansion with rule 4 produces b.

~~[S, 0]~~	[S, 1]			
	[A, 2]			
	~~[T, 4]~~	[T, 5]		
	b	~~[(A), 2]~~		[(A), 3]
		~~[(T), 4]~~	~~[(T), 5]~~	[(A + T), 2]
		(b)	$((A))$	[(T + T), 4]
				[(b + T), 4]
				$(b + b)$

A derivation of the input string can be constructed from the items on the stack when the search successfully terminates. The first element is the sentential form being expanded and the second is the number of the rule applied to generate the successor.

Stack	Derivation
$[S, 1]$	$S \Rightarrow A$
$[A, 2]$	$\Rightarrow T$
$[T, 5]$	$\Rightarrow (A)$
$[(A), 3]$	$\Rightarrow (A + T)$
$[(A + T), 2]$	$\Rightarrow (T + T)$
$[(T + T), 4]$	$\Rightarrow (b + T)$
$[(b + T), 4]$	$\Rightarrow (b + b)$

□

The exhaustive nature of the breadth-first search guarantees that Algorithm 4.3.1 produces a derivation whenever the input is in the language of the grammar. We have already noted that prefix matching does not provide sufficient information to ensure that the breadth-first search terminates for strings not in the language. Unfortunately, the backtracking algorithm may fail to find derivations for strings in the language. Since it employs a depth-first strategy, the parser may explore an infinitely long path. If this exploration fails to trigger one of the dead-end conditions, the search will never backtrack to examine other paths in the graph. Example 4.4.2 shows that Algorithm 4.4.1 fails to terminate when attempting to parse the string $(b) + b$ using the rules of the grammar AE.

Example 4.4.2

The actions of the top-down backtracking algorithm are traced for input string $(b) + b$.

~~[S, 0]~~	[S, 1]						
	[A, 2]						
	~~[T, 4]~~	[T, 5]					
	b	~~[(A), 2]~~		[(A), 3]			
		~~[(T), 4]~~	~~[(T), 5]~~	~~[(A+T), 2]~~		[(A+T), 3]	
		(b)	$((A))$	~~[(T+T), 4]~~	~~[(T+T), 5]~~	[(A+T+T), 2]	
				$(b+T)$	$((A)+T)$	~~[(T+T+T), 4]~~	[(T+T+T), 5]
						$(b+T+T)$	$((A)+T+T)$

A pattern emerges in columns five and six and in columns seven and eight. The next step of the algorithm is to pop the stack and restore the string $(A + T + T)$. The stack item $[(A + T + T), 2]$ indicates that all A rules numbered two or less have previously been examined. Rule 3 is then applied to $(A + T + T)$, generating $(A + T + T + T)$. The directly recursive rule $A \rightarrow A + T$ can be applied repeatedly without increasing the length of the terminal prefix. The path consisting of $(A), (A + T), (A + T + T), (A + T + T + T), \ldots$ will be explored without producing a dead-end. □

The possibility of the top-down parsers producing a nonterminating computation is a consequence of left recursion in the grammar. This type of behavior must be eliminated to ensure the termination of a parse. In the next chapter, a series of transformations that change the rules of the grammar but preserve the language is presented. Upon the completion of the rule transformations, the resulting grammar will contain no left recursive derivations. With these modifications to the grammar, the top-down parsers provide complete algorithms for analyzing the syntax of strings in context-free languages.

4.5 Bottom-Up Parsing

A derivation of a string p is obtained by building a path from the start symbol S to p in the graph of a grammar. When the explicit search structure begins with input string p, the resulting algorithm is known as a *bottom-up parser.* To limit the size of the implicit graph, top-down parsers generate only leftmost derivations. The bottom-up parsers that we examine will construct rightmost derivations.

The top-down parsers of Sections 4.3 and 4.4 are fundamentally exhaustive search algorithms. The strategy is to systematically produce derivations until the input string is found or until all possible derivations are determined to be incapable of producing the input. This exhaustive approach produces many derivations that do not lead to the input string. For example, the entire subtree with root $A + T$ in Figure 4.3 consists of derivations that cannot produce $(b + b)$.

By beginning the search with the input string p, the only derivations that are produced by a bottom-up parser are those that can generate p. This serves to focus the search and limit the size of the search tree generated by the parser. Bottom-up parsing may be considered to be a search of an implicit graph consisting of all strings that derive p by rightmost derivations.

Let G = (V, Σ, P, S) be a context-free grammar. The nodes of the implicit graph to be searched by a bottom-up parser with input p are the strings w for which there are derivations $w \underset{R}{\overset{*}{\Rightarrow}} p$. A node w is adjacent to a node v if there is a derivation $v \underset{R}{\Rightarrow} w \underset{R}{\overset{*}{\Rightarrow}} p$. That is, w is adjacent to v if p is derivable from w, $v = u_1 A u_2$ with $u_2 \in \Sigma^*$, $A \rightarrow q$ is a rule of the grammar, and $w = u_1 q u_2$. The process of obtaining v from w is known as a **reduction** because the substring q in w is reduced to the variable A in v. A bottom-up parse repeatedly applies reductions until the input string is transformed into the start symbol of the grammar.

Example 4.5.1

A reduction of the string $(b) + b$ to S is given using the rules of the grammar AE.

Reduction	Rule
$(b) + b$	
$(T) + b$	$T \rightarrow b$
$(A) + b$	$A \rightarrow T$
$T + b$	$T \rightarrow (A)$
$A + b$	$A \rightarrow T$
$A + T$	$T \rightarrow b$
A	$A \rightarrow A + T$
S	$S \rightarrow A$

Reversing the order of the sentential forms that constitute the reduction of w to S produces the rightmost derivation

$$\begin{aligned} S &\Rightarrow A \\ &\Rightarrow A + T \\ &\Rightarrow A + b \\ &\Rightarrow T + b \\ &\Rightarrow (A) + b \\ &\Rightarrow (T) + b \\ &\Rightarrow (b) + b. \end{aligned}$$

For this reason, bottom-up parsers are often said to construct rightmost derivations in reverse. □

To generate all possible reductions of a string w, a bottom-up parser utilizes a pattern matching scheme. The string w is divided into two substrings, $w = uv$. The initial division sets u to the null string and v to w. The right-hand side of each rule is compared with the suffixes of u. A match occurs when u can be written $u_1 q$ and $A \rightarrow q$ is a rule of the grammar. This combination produces the reduction of w to $u_1 A v$.

When all the rules have been compared with u for a given pair uv, w is divided into a new pair of substrings $u'v'$ and the process is repeated. The new decomposition is obtained by setting u' to u concatenated with the first element of v; v' is v with its first element removed. The process of updating the division is known as a *shift*. The shift and compare operations are used to generate all possible reductions of the string $(A + T)$ in the grammar AE.

	u	v	**Rule**	**Reduction**
	λ	$(A + T)$		
Shift	$($	$A + T)$		
Shift	$(A$	$+ T)$	$S \rightarrow A$	$(S + T)$
Shift	$(A +$	$T)$		
Shift	$(A + T$	$)$	$A \rightarrow A + T$	(A)
			$A \rightarrow T$	$(A + A)$
Shift	$(A + T)$	λ		

In generating the reductions of the string, the right-hand side of the rule must match a suffix of u. All other reductions in which the right-hand side of a rule occurs in u would have been discovered prior to the most recent shift.

Algorithm 4.5.1 is a breadth-first bottom-up parser. As with the breadth-first top-down parser, a search tree T is constructed using the queue operations *INSERT, REMOVE,* and *EMPTY.*

Algorithm 4.5.1
Breadth-First Bottom-Up Parser

input: context-free grammar G = (V, Σ, P, S)
string $p \in \Sigma^*$
queue **Q**

1. initialize T with root p
   $INSERT(p, \mathbf{Q})$
2. **repeat**
   $q := REMOVE(\mathbf{Q})$
   2.1. **for** each rule $A \rightarrow w$ in P **do**
   2.1.1. **for** each decomposition uwv of q with $v \in \Sigma^*$ **do**
   2.1.1.1. $INSERT(uAv, \mathbf{Q})$
   2.1.1.2. Add node uAv to T. Set a pointer from uAv to q.
   end for
   end for
   until $q = S$ **or** $EMPTY(\mathbf{Q})$
3. **if** $q = S$ **then** accept **else** reject

Step 2.1.1.1 inserts reductions into the queue. Bottom-up parsers are designed to generate only rightmost derivations. A reduction of uwv to uAv is added to the search tree only if v is a terminal string. As in a top-down parser, the left-hand side of the string is transformed first. The bottom-up parser, however, is building the derivation "in reverse." The first reduction is the last rule application of the derivation. Consequently, the left side of the string is the last to be transformed when the rules are applied, producing in a rightmost derivation.

Figure 4.5 shows the search tree built when the string $(b + b)$ is analyzed by the breadth-first bottom-up parsing algorithm. Following the path from S to $(b + b)$ yields the rightmost derivation

$$\begin{aligned} S &\Rightarrow A \\ &\Rightarrow T \\ &\Rightarrow (A) \\ &\Rightarrow (A + T) \\ &\Rightarrow (A + b) \\ &\Rightarrow (T + b) \\ &\Rightarrow (b + b). \end{aligned}$$

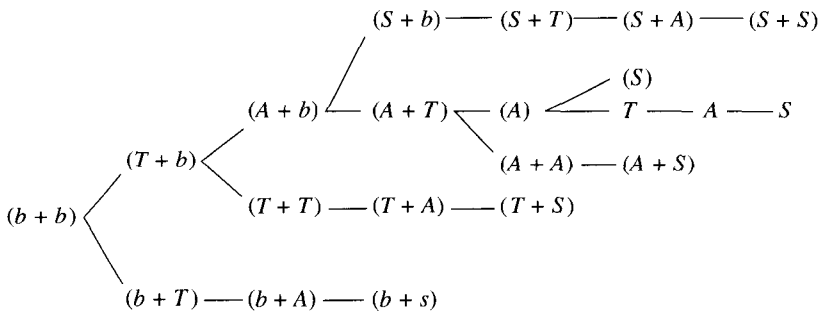


FIGURE 4.5 Breadth-first bottom-up parse of $(b + b)$.

Compare the search produced by the bottom-up parse of $(b + b)$ in Figure 4.5 with that produced by the top-down parse in Figure 4.3. Restricting the search to derivations that can produce $(b + b)$ significantly reduced the number of nodes generated.

Does a breadth-first bottom-up parser halt for every possible input string or is it possible for the algorithm to continue indefinitely in the repeat-until loop? If the string p is in the language of the grammar, a rightmost derivation will be found. If the length of the right-hand side of each rule is greater than one, the reduction of a sentential form creates a new string of strictly smaller length. For grammars satisfying this condition, the depth of the search tree cannot exceed the length of the input string, assuring the termination of a parse with either a derivation or a failure. This condition, however, is not satisfied by grammars with rules of the form $A \rightarrow B$, $A \rightarrow a$, and $A \rightarrow \lambda$.

4.6 A Depth-First Bottom-Up Parser

Bottom-up parsing can also be implemented in a depth-first manner. The reductions are generated by the shift and compare technique described for the breadth-first algorithm. The order in which the reductions are processed is determined by the number of shifts required to produce the match and the ordering of the rules. The right-hand sides of the rules $A \rightarrow T$ and $T \rightarrow b$ both occur in the string $(T + b)$. Reducing $(T + b)$ with the rule $A \rightarrow T$ produces $(A + b)$ with terminal suffix $+ b)$. The other reduction produces the string to $(T + T)$ with terminal suffix). The parser examines the reduction to $(A + b)$ first since it is discovered after only two shifts while four shifts are required to find the other.

The stack elements of the backtracking bottom-up parser presented in Algorithm 4.6.1 are triples $[u, i, v]$ where $w = uv$ is the sentential form that was reduced and i the number of the rule used in the reduction. The strings u and v specify the decomposition of w created by the shifting process; u is the substring whose suffixes are compared with the right-hand side of the rules. There are two distinct reductions of the string $(b + b)$ using

the rule $T \rightarrow b$. The reduction of the first b pushes $[(b, 4, +b)]$ onto the stack. The stack element $[(b + b, 4,)]$ is generated by the second reduction. The possibility of a string admitting several reductions demonstrates the need for storing the decomposition of the sentential form in the stack element.

Algorithm 4.6.1 is designed for parsing strings in a context-free grammar in which the start symbol is nonrecursive. The start symbol is nonrecursive if it does not occur on the right-hand side of any rule. A reduction using a rule $S \rightarrow w$ in a grammar with a nonrecursive start symbol is a dead-end unless $u = w$ and $v = \lambda$. In this case, the reduction successfully terminates the parse.

Algorithm 4.6.1 utilizes an auxiliary procedure *shift*. If v is not null, $shift(u, v)$ removes the first symbol from v and concatenates it to the end of u.

Algorithm 4.6.1
Depth-First Bottom-Up Parsing Algorithm

input: context-free grammar G = (V, Σ, P, S) with nonrecursive start symbol
string $p \in \Sigma^*$
stack **S**

1. $PUSH([\lambda, 0, p], \mathbf{S})$
2. **repeat**
   2.1. $[u, i, v] := POP(\mathbf{S})$
   2.2. dead-end := *false*
   2.3. **repeat**
      Find the first $j > i$ with rule number j that satisfies
      i) $A \rightarrow w$ with $u = qw$ and $A \neq S$ or
      ii) $S \rightarrow w$ with $u = w$ and $v = \lambda$
      2.3.1. **if** there is such a j **then**
         2.3.1.1. $PUSH([u, j, v], \mathbf{S})$
         2.3.1.2. $u := qA$
         2.3.1.3. $i := 0$
      end if
      2.3.2. **if** there is no such j **and** $v \neq \lambda$ **then**
         2.3.2.1. $shift(u, v)$
         2.3.2.2. $i := 0$
      end if
      2.3.3. **if** there is no such j **and** $v = \lambda$ **then** dead-end := *true*
   until $(u = S)$ **or** dead-end
until $(u = S)$ **or** $EMPTY(\mathbf{S})$
3. **if** $EMPTY(\mathbf{S})$ **then** reject **else** accept

---

The condition that detects dead-ends and forces the parser to backtrack occurs in step 2.3.3. When the string v is empty, an attempted shift indicates that all reductions of uv have been examined.

Algorithm 4.6.1 assumes that the grammar has a nonrecursive start symbol. This restriction does not limit the languages that can be parsed. In Section 5.1 we will see that any context-free grammar can easily be transformed into an equivalent grammar with a nonrecursive start symbol. Algorithm 4.6.1 can be modified to parse arbitrary context-free grammars. This requires removing the condition that restricts the processing of reductions using S rules. The modification is completed by changing the terminating condition of the repeat-until loops from $(u = S)$ to $(u = S$ **and** $v = \lambda)$.

Example 4.6.1

Using Algorithm 4.6.1 and the grammar AE, we can construct a derivation of the string $(b + b)$. The stack is given in the second column, with the stack top being the top triple. The decomposition of the string and current rule numbers are in the columns labeled u, v, and i. The operation that produced the new configuration is given on the left. At the beginning of the computation the stack contains the single element $[\lambda, 0, (b + b)]$. The configuration consisting of an empty stack and $u = \lambda$, $i = 0$, and $v = (b + b)$ is obtained by popping the stack. Two shifts are required before a reduction pushes $[(b, 4, +b)]$ onto the stack.

Operation	**Stack**	u	i	v
	$[\lambda, 0, (b + b)]$			
pop		λ	0	$(b + b)$
shift		$($	0	$b + b)$
shift		$(b$	0	$+ b)$
reduction	$[(b, 4, + b)]$	$(T$	0	$+ b)$
	$[(T, 2, + b)]$			
reduction	$[(b, 4, + b)]$	$(A$	0	$+ b)$
	$[(T, 2, + b)]$			
shift	$[(b, 4, + b)]$	$(A+$	0	$b)$
	$[(T, 2, + b)]$			
shift	$[(b, 4, + b)]$	$(A + b$	0	$)$

Continued

Operation	**Stack**	u	i	v
	$[(A+b,4,)]$			
	$[(T,2,+b)]$			
reduction	$[(b,4,+b)]$	$(A+T$	0	)
	$[(A+T,2,)]$			
	$[(A+b,4,)]$			
	$[(T,2,+b)]$			
reduction	$[(b,4,+b)]$	$(A+A$	0	)
	$[(A+T,2,)]$			
	$[(A+b,4,)]$			
	$[(T,2,+b)]$			
shift	$[(b,4,+b)]$	$(A+A)$	0	λ
	$[(A+b,4,)]$			
	$[(T,2,+b)]$			
pop	$[(b,4,+b)]$	$(A+T$	2	)
	$[(A+T,3,)]$			
	$[(A+b,4,)]$			
	$[(T,2,+b)]$			
reduction	$[(b,4,+b)]$	$(A$	0	)
	$[(A+T,3,)]$			
	$[(A+b,4,)]$			
	$[(T,2,+b)]$			
shift	$[(b,4,+b)]$	(A)	0	λ
	$[(A),5,\lambda]$			
	$[(A+T,3,)]$			
	$[(A+b,4,)]$			
	$[(T,2,+b)]$			
reduction	$[(b,4,+b)]$	T	0	λ

Continued

Operation	Stack	u	i	v
	$[T, 2, \lambda]$			
	$[(A), 5, \lambda]$			
	$[(A + T, 3,)]$			
	$[(A + b, 4,)]$			
	$[(T, 2, + b)]$			
reduction	$[(b, 4, + b)]$	A	0	λ
	$[A, 1, \lambda]$			
	$[T, 2, \lambda]$			
	$[(A), 5, \lambda]$			
	$[(A + T, 3,)]$			
	$[(A + b, 4,)]$			
	$[(T, 2, + b)]$			
reduction	$[(b, 4, + b)]$	S	0	λ

The rightmost derivation produced by this parse is identical to that obtained by the breadth-first parse. □

Exercises

1. Let G be a grammar and $w \in L(G)$. Prove that there is a rightmost derivation of w in G.
2. Build the subgraph of the grammar of G consisting of the left sentential forms that are generated by derivations of length three or less.

$$\begin{aligned} \text{G: } & S \rightarrow aS \mid AB \mid B \\ & A \rightarrow abA \mid ab \\ & B \rightarrow BB \mid ba \end{aligned}$$

3. Build the subgraph of the grammar of G consisting of the left sentential forms that are generated by derivations of length four or less.

$$\begin{aligned} \text{G: } & S \rightarrow aSA \mid aB \\ & A \rightarrow bA \mid \lambda \\ & B \rightarrow cB \mid c \end{aligned}$$

Is G ambiguous?

4. Construct two regular grammars, one ambiguous and one unambiguous, that generate the language a^*.

5. Let G be the grammar

$$S \rightarrow aS \mid Sb \mid ab \mid SS.$$

   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.

6. Let G be the grammar

$$S \rightarrow ASB \mid ab \mid SS$$
$$A \rightarrow aA \mid \lambda$$
$$B \rightarrow bB \mid \lambda.$$

   a) Give a leftmost derivation of *aaabb*.
   b) Give a rightmost derivation of *aaabb*.
   c) Show that G is ambiguous.
   d) Construct an unambiguous grammar equivalent to G.

7. Let G be the grammar

$$S \rightarrow aSA \mid \lambda$$
$$A \rightarrow bA \mid \lambda.$$

   a) Give a regular expression for L(G).
   b) Show that G is ambiguous.
   c) Construct an unambiguous grammar equivalent to G.

8. Show that the grammar

$$S \rightarrow aaS \mid aaaaaS \mid \lambda$$

   is ambiguous. Give an unambiguous grammar that generates L(G).

9. Let G be the grammar

$$S \rightarrow aSb \mid aAb$$
$$A \rightarrow cAd \mid B$$
$$B \rightarrow aBb \mid \lambda.$$

   a) Give a set-theoretic definition of L(G).
   b) Show that G is ambiguous.
   c) Construct an unambiguous grammar equivalent to G.

10. Let G be the grammar

$$S \rightarrow AaSbB \mid \lambda$$
$$A \rightarrow aA \mid a$$
$$B \rightarrow bB \mid \lambda.$$

a) Give a set-theoretic definition of L(G).

b) Show that G is ambiguous.

c) Construct an unambiguous grammar equivalent to G.

11. Let G be the grammar

$$S \rightarrow ASB \mid AB$$
$$A \rightarrow aA \mid a$$
$$B \rightarrow bB \mid \lambda.$$

a) Give a regular expression for L(G).

b) Show that G is ambiguous.

c) Construct an unambiguous regular grammar equivalent to G.

12. Let G be the grammar

$$S \rightarrow aA \mid \lambda$$
$$A \rightarrow aA \mid bB$$
$$B \rightarrow bB \mid b.$$

a) Give a regular expression for L(G).

b) Prove that G is unambiguous.

13. Let G be the grammar

$$S \rightarrow aS \mid aA \mid a$$
$$A \rightarrow aAb \mid ab.$$

a) Give a set-theoretic definition of L(G).

b) Prove that G is unambiguous.

14. Construct unambiguous grammars for the languages $L_1 = \{a^n b^n c^m \mid n, m > 0\}$ and $L_2 = \{a^n b^m c^m \mid n, m > 0\}$. 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.

In Exercises 15 through 23, trace the actions of the algorithm as it parses the input string. For the breadth-first algorithms, build the tree constructed by the parser. For the depth-first algorithms, trace the stack as in Examples 4.4.1 and 4.6.1. If the input string is in the language, give the derivation constructed by the parser.

15. Algorithm 4.3.1 with input $(b) + b$.
16. Algorithm 4.3.1 with input $b + (b)$.
17. Algorithm 4.3.1 with input $((b))$.
18. Algorithm 4.4.1 with input $((b))$.
19. Algorithm 4.4.1 with input $b + (b)$.
20. Algorithm 4.5.1 with input $(b) + b$.
21. Algorithm 4.5.1 with input $(b))$.
22. Algorithm 4.6.1 with input $(b) + b$.
23. Algorithm 4.6.1 with input $(b))$.
24. Give the first five levels of the search tree generated by Algorithms 4.3.1 and 4.5.1 when parsing the string $b) + b$.
25. Let G be the grammar

$$
\begin{aligned}
&1.\ S \rightarrow aS \\
&2.\ S \rightarrow AB \\
&3.\ A \rightarrow bAa \\
&4.\ A \rightarrow a \\
&5.\ B \rightarrow bB \\
&6.\ B \rightarrow b.
\end{aligned}
$$

    a) Trace the stack (as in Example 4.4.1) of the top-down depth-first parse of $baab$.
    b) Give the tree built by the breadth-first bottom-up parse of $baab$.
    c) Trace the stack (as in Example 4.6.1) of the bottom-up depth-first parse of $baab$.
26. Let G be the grammar

$$
\begin{aligned}
&1.\ S \rightarrow A \\
&2.\ S \rightarrow AB \\
&3.\ A \rightarrow abA \\
&4.\ A \rightarrow b \\
&5.\ B \rightarrow baB \\
&6.\ B \rightarrow a.
\end{aligned}
$$

    a) Give a regular expression for L(G).
    b) Trace the stack of the top-down depth-first parse of $abbbaa$.
    c) Give the tree built by the breadth-first bottom-up parse of $abbbaa$.
    d) Trace the stack of the bottom-up depth-first parse of $abbbaa$.
27. Construct a grammar G and string $p \in \Sigma^*$ such that Algorithm 4.5.1 loops indefinitely in attempting to parse p.

28. Construct a grammar G and string $p \in L(G)$ such that Algorithm 4.6.1 loops indefinitely in attempting to parse p.

Bibliographic Notes

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 14 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 nondeterministic parsing algorithms are essentially graph searching algorithms modified for this particular application. The depth-first algorithms presented here are from Denning, Dennis, and Qualitz [1978]. A thorough exposition of graph and tree traversals is given in Knuth [1968] and in many texts on data structures and algorithms.

The analysis of the syntax of a string is an essential feature in the construction of compilers for computer languages. Grammars amenable to deterministic parsing techniques are presented in Chapters 16 and 17. For references to parsing, see the bibliographic notes following those chapters.

CHAPTER 5

Normal Forms

A normal form is defined by imposing restrictions on the form of the rules allowed in the grammar. The grammars in a normal form generate the entire set of context-free languages. Two important normal forms for context-free grammars, the Chomsky and Greibach normal forms, are introduced in this chapter. Transformations are developed to convert an arbitrary context-free grammar into an equivalent grammar in normal form. The transformations consist of a series of techniques that add and delete rules. Each of these replacement schema preserves the language generated by the grammar.

Restrictions imposed on the rules by a normal form often ensure that derivations of the grammar have certain desirable properties. The transformations in this chapter produce grammars for which both the top-down and bottom-up parsers presented in Chapter 4 are guaranteed to terminate. The ability to transform a grammar into an equivalent Greibach normal form will be used in Chapter 8 to establish a characterization of the languages that can be generated by context-free grammars.

5.1 Elimination of Lambda Rules

The transformation of a grammar begins by imposing a restriction of the start symbol of the grammar. Given a grammar G, an equivalent grammar G′ is constructed in which the role of the start symbol is limited to the initiation of derivations. A recursive derivation of the form $S \stackrel{*}{\Rightarrow} uSv$ permits the start symbol to occur in sentential forms in intermediate

steps of a derivation. The restriction is satisfied whenever the start symbol of G′ is a nonrecursive variable.

Lemma 5.1.1

Let G = (V, Σ, P, S) be a context-free grammar. There is a grammar G′ = (V′, Σ, P′, S') that satisfies

i) L(G) = L(G′).

ii) The rules of P′ are of the form

$$A \to w,$$

where $A \in V'$ and $w \in ((V - \{S'\}) \cup \Sigma)^*$.

Proof If the start symbol S does not occur on the right-hand side of a rule of G, then 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 $G' = (V \cup \{S'\}, \Sigma, P \cup \{S' \to S\}, S')$ is constructed by designating a new start symbol S' and adding $S' \to S$ to the rules of G. The two grammars generate the same language since any string u derivable in G by a derivation $S \overset{*}{\underset{G}{\Rightarrow}} u$ can be obtained by the derivation $S' \underset{G'}{\Rightarrow} S \overset{*}{\underset{G'}{\Rightarrow}} u$. 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. ■

Example 5.1.1

The start symbol of the grammar G is recursive. The technique outlined in Lemma 5.1.1 is used to construct an equivalent grammar G′ with a nonrecursive start symbol. The start symbol of G′ is the variable S'.

G: $S \to aS \mid AB \mid AC$
$A \to aA \mid \lambda$
$B \to bB \mid bS$
$C \to cC \mid \lambda$

G′: $S' \to S$
$S \to aS \mid AB \mid AC$
$A \to aA \mid \lambda$
$B \to bB \mid bS$
$C \to cC \mid \lambda$

The variable S is still recursive in G′, but it is not the start symbol of the new grammar. □

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 lambda rules. This property is illustrated by the derivation of the string $aaaa$ in the grammar

$$S \to SaB \mid aB$$
$$B \to bB \mid \lambda.$$

The language generated by this grammar is $(a^+b^*)^+$. The leftmost derivation of *aaaa* generates four B's, each of which is removed by the application of the rule $B \rightarrow \lambda$.

$$\begin{aligned} S &\Rightarrow SaB \\ &\Rightarrow SaBaB \\ &\Rightarrow SaBaBaB \\ &\Rightarrow aBaBaBaB \\ &\Rightarrow aaBaBaB \\ &\Rightarrow aaaBaB \\ &\Rightarrow aaaaB \\ &\Rightarrow aaaa \end{aligned}$$

A more efficient approach would be to avoid the generation of variables that are subsequently removed by lambda rules. Another grammar, without lambda rules, that generates $(a^+b^*)^+$ is given below. The string *aaaa* is generated using half the number of rule applications. This efficiency is gained at the expense of increasing the number of rules of the grammar.

$$S \rightarrow SaB \mid Sa \mid aB \mid a \qquad B \rightarrow bB \mid b$$

$$\begin{aligned} S &\Rightarrow Sa \\ &\Rightarrow Saa \\ &\Rightarrow Saaa \\ &\Rightarrow aaaa \end{aligned}$$

A variable that can derive the null string is called **nullable.** The length of a sentential form can be reduced by a sequence of rule applications if the string contains nullable variables. A grammar without nullable variables is called **noncontracting** since the application of a rule cannot decrease the length of the sentential form. Algorithm 5.1.2 iteratively constructs the set of nullable variables in G. The algorithm utilizes two sets; the set NULL collects the nullable variables and PREV triggers the halting condition.

Algorithm 5.1.2
Construction of the Set of Nullable Variables

input: context-free grammar G = (V, Σ, P, S)

1. NULL := $\{A \mid A \rightarrow \lambda \in P\}$
2. **repeat**
   2.1. PREV := NULL
   2.2. **for** each variable $A \in V$ **do**
      if there is an A rule $A \rightarrow w$ and $w \in$ PREV* **then**
         NULL := NULL ∪ $\{A\}$

until NULL = PREV

The set PREV 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. Induction is used to show that the set NULL contains exactly the nullable variables of G at the termination of the computation.

Lemma 5.1.3

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. Algorithm 5.1.2 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

$$A \rightarrow A_1 A_2 \ldots A_k$$

with each A_i in PREV at the $n + 1$st iteration. By the inductive hypothesis, $A_i \stackrel{*}{\Rightarrow} \lambda$ for $i = 1, 2, \ldots, k$. These derivations can be used to construct the derivation

$$\begin{aligned} A &\Rightarrow A_1 A_2 \ldots A_k \\ &\stackrel{*}{\Rightarrow} A_2 \ldots A_k \\ &\stackrel{*}{\Rightarrow} A_3 \ldots A_k \\ &\vdots \\ &\stackrel{*}{\Rightarrow} A_k \\ &\stackrel{*}{\Rightarrow} \lambda, \end{aligned}$$

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 $A \stackrel{1}{\Rightarrow} \lambda$, 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

$$A \Rightarrow A_1A_2 \ldots A_k$$
$$\stackrel{n}{\Rightarrow} \lambda.$$

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

$$A \rightarrow A_1A_2 \ldots A_k$$

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 5.1.2 provides a decision procedure for determining whether the null string is in the language of a grammar.

Example 5.1.2

The set of nullable variables of the grammar

$$\begin{aligned} \text{G: } & S \rightarrow ACA \\ & A \rightarrow aAa \mid B \mid C \\ & B \rightarrow bB \mid b \\ & C \rightarrow cC \mid \lambda \end{aligned}$$

is constructed using Algorithm 5.1.2. 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. □

Example 5.1.3

The sets of nullable variables of the grammars G and G′ from Example 5.1.1 are $\{S, A, C\}$ and $\{S', S, A, C\}$, respectively. The start symbol S' produced by the construction of a grammar with a nonrecursive start symbol is nullable only if the start symbol of the original grammar is nullable. □

The process of transforming grammars into the normal forms consists of removing and adding rules to the grammar. With each alteration, the language generated by the grammar should remain unchanged. Lemma 5.1.4 establishes a simple criterion by which rules may be added to a grammar without altering the language.

Lemma 5.1.4

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. If $A \overset{*}{\underset{G}{\Rightarrow}} w$, 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 results from the observation that the application of the rule $A \rightarrow w$ in a derivation in G' can be simulated in G by the derivation $A \overset{*}{\underset{G}{\Rightarrow}} w$. ■

A grammar with lambda 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 lambda rules. Assume that B is a nullable variable. There are two distinct roles that B can play in a derivation that is initiated by the application of the rule $A \rightarrow BAa$; it can derive a nonnull terminal string or it can derive the null string. In the latter case, the derivation has the form

$$\begin{aligned} A &\Rightarrow BAa \\ &\overset{*}{\Rightarrow} Aa \\ &\overset{*}{\Rightarrow} u. \end{aligned}$$

The string u can be derived without lambda rules by augmenting the grammar with the rule $A \rightarrow Aa$. Lemma 5.1.4 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 lambda 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 lambda rules. This process is used to construct a grammar without lambda 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 $S \overset{*}{\Rightarrow} \lambda$

must eventually disappear. To handle this special case, the rule $S \to \lambda$ is allowed in the new grammar but all other lambda 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 lambda rules is denoted G_L.

Theorem 5.1.5

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) $A \to \lambda$ is in P_L if, and only if, $\lambda \in L(G)$ and $A = S_L$.

Proof The start symbol can be made nonrecursive by the technique presented in Lemma 5.1.1. The set of variables V_L is simply V with a new start symbol added, if necessary. The productions of G_L are defined as follows:

i) If $\lambda \in L(G)$, then $S_L \to \lambda \in P_L$.

ii) Let $A \to w$ be a rule in P. If w can be written

$$w_1 A_1 w_2 A_2 \ldots w_k A_k w_{k+1},$$

where $A_1, A_2, \ldots, A_k$ are a subset of the occurrences of the nullable variables in w, then

$$A \to w_1 w_2 \ldots w_k w_{k+1}$$

is a rule of P_L.

iii) $A \to \lambda \in P_L$ only if $\lambda \in L(G)$ and $A = S_L$.

The process of removing lambda rules creates a set of rules from each rule of the original grammar. A rule with n occurrences of nullable variables in the right-hand side produces 2^n rules. Condition (iii) deletes all lambda rules other than $S_L \to \lambda$ from P_L.

Derivations in G_L utilize rules from G and those created by condition (ii), each of which is derivable in 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 $A \overset{n}{\underset{G}{\Rightarrow}} w$ be a derivation in G with $w \in \Sigma^+$. We prove that $A \overset{*}{\underset{G_L}{\Rightarrow}} w$ by induction on n, the length of the derivation of w in G. If $n = 1$, then $A \to w$ is a rule in P and, since $w \neq \lambda$, $A \to w$ is in P_L.

Assume that all terminal strings derivable from A by n or fewer rule applications can be derived from A in G_L. Note that this makes no claim concerning the length of the

derivation in G_L. Let $A \overset{n+1}{\underset{G}{\Longrightarrow}} w$ be a derivation of a terminal string. If we explicitly specify the first rule application, the derivation can be written

$$A \Rightarrow w_1 A_1 w_2 A_2 \dots w_k A_k w_{k+1} \overset{n}{\underset{G}{\Rightarrow}} w,$$

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 $A_i \overset{*}{\underset{G_L}{\Rightarrow}} p_i$. If $p_j = \lambda$, the variable A_j is nullable in G. Condition (ii) 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 derive 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 5.1.4

Let G be the grammar given in Example 5.1.2. The nullable variables of G are $\{S, A, C\}$. The equivalent essentially noncontracting grammar G_L is given below.

G:	G_L:
$S \rightarrow ACA$	$S \rightarrow ACA \mid CA \mid AA \mid AC \mid A \mid C \mid \lambda$
$A \rightarrow aAa \mid B \mid C$	$A \rightarrow aAa \mid aa \mid B \mid C$
$B \rightarrow bB \mid b$	$B \rightarrow bB \mid b$
$C \rightarrow cC \mid \lambda$	$C \rightarrow cC \mid c$

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 lambda 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 5.1.4, while the lambda rule in G_L generates it immediately. Leftmost derivations of the string *aba* are given in each of the grammars.

$$\begin{aligned} \text{G: } S &\Rightarrow ACA & \text{G}_\text{L}\text{: } S &\Rightarrow A \\ &\Rightarrow aAaCA & &\Rightarrow aAa \\ &\Rightarrow aBaCA & &\Rightarrow aBa \\ &\Rightarrow abaCA & &\Rightarrow aba \\ &\Rightarrow abaA \\ &\Rightarrow abaC \\ &\Rightarrow aba \end{aligned}$$

The first rule application of the derivation in G_L generates only variables that eventually derive terminals. Thus, the application of lambda rules is avoided.

Example 5.1.5

Let G be the grammar

$$\begin{aligned} \text{G: } S &\rightarrow ABC \\ A &\rightarrow aA \mid \lambda \\ B &\rightarrow bB \mid \lambda \\ C &\rightarrow cC \mid \lambda \end{aligned}$$

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

$$\begin{aligned} \text{G}_\text{L}\text{: } S &\rightarrow ABC \mid AB \mid BC \mid AC \mid A \mid B \mid C \mid \lambda \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b \\ C &\rightarrow cC \mid c. \end{aligned}$$

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. □

5.2 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 having this form are called **chain rules**. The removal of chain rules requires the addition of rules that allow the revised grammar to generate the same strings. The idea behind the removal process 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

$$A \rightarrow aA \mid a \mid bB \mid b \mid C$$
$$B \rightarrow bB \mid b \mid C.$$

Unfortunately, another chain rule was created by this replacement.

A derivation $A \stackrel{*}{\Rightarrow} C$ consisting solely of chain rules is called a **chain**. Algorithm 5.2.1 generates all variables that can be derived by chains from a variable A in an essentially noncontracting grammar. This set is denoted CHAIN(A). The set NEW contains the variables that were added to CHAIN(A) on the previous iteration.

Algorithm 5.2.1
Construction of the Set CHAIN(A)

input: essentially noncontracting context-free grammar G = (V, Σ, P, S)

1. CHAIN(A) := $\{A\}$
2. PREV := $\emptyset$
3. **repeat**
   3.1. NEW := CHAIN(A) − PREV
   3.2. PREV := CHAIN(A)
   3.3. **for** each variable $B \in$ NEW **do**
      for each rule $B \rightarrow C$ **do**
         CHAIN(A) := CHAIN(A) $\cup$ $\{C\}$

until CHAIN(A) = PREV

Algorithm 5.2.1 is fundamentally different from the algorithm that generates the nullable variables. The difference is similar to the difference between bottom-up and top-down parsing strategies. 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 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 5.2.1 generates CHAIN(A) is left as an exercise.

Lemma 5.2.2
Let $G = (V, \Sigma, P, S)$ be an essentially noncontracting context-free grammar. Algorithm 5.2.1 generates the set of variables derivable from A using only chain rules.

The variables in 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 5.2.3
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 has no chain rules.

Proof The A rules of G_C are constructed from the set CHAIN(A) and the rules of G. The rule $A \to w$ is in P_C if there is a variable B and a string w that satisfy

i) $B \in \text{CHAIN}(A)$

ii) $B \to w \in P$

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 5.1.4, 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 $A \overset{*}{\underset{G}{\Rightarrow}} B$ be a maximal sequence of chain rules used in the derivation of w. The derivation of w has the form

$$S \overset{*}{\underset{G}{\Rightarrow}} uAv \overset{*}{\underset{G}{\Rightarrow}} uBv \underset{G}{\Rightarrow} upv \overset{*}{\underset{G}{\Rightarrow}} w,$$

where $B \to p$ is a rule, but not a chain rule, in G. The rule $A \to p$ 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 5.2.1

The grammar G_C is constructed from the grammar G_L in Example 5.1.4.

$$\begin{aligned} G_L\colon\ & S \to ACA \mid CA \mid AA \mid AC \mid A \mid C \mid \lambda \\ & A \to aAa \mid aa \mid B \mid C \\ & B \to bB \mid b \\ & C \to cC \mid c \end{aligned}$$

Since G_L is essentially noncontracting, Algorithm 5.2.1 generates the variables derivable using chain rules. The computations construct the sets

$$\text{CHAIN}(S) = \{S, A, C, B\}$$
$$\text{CHAIN}(A) = \{A, B, C\}$$
$$\text{CHAIN}(B) = \{B\}$$
$$\text{CHAIN}(C) = \{C\}.$$

These sets are used to generate the rules of G_C.

$$\begin{aligned} P_C\colon\ & S \to ACA \mid CA \mid AA \mid AC \mid aAa \mid aa \mid bB \mid b \mid cC \mid c \mid \lambda \\ & A \to aAa \mid aa \mid bB \mid b \mid cC \mid c \\ & B \to bB \mid b \\ & C \to cC \mid c \end{aligned}$$ □

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 $A \to w$ be a rule created by the removal of chain rules. This implies that there is a rule $B \to w$ for some variable $B \in \text{CHAIN}(A)$. Since the original grammar was essentially noncontracting, the only lambda rule is $S \to \lambda$. The start symbol, being nonrecursive, is not a member of CHAIN(A) for any $A \neq S$. It follows that no additional lambda 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) $S \to \lambda$
ii) $A \to a$
iii) $A \to w$,

where $w \in (V \cup \Sigma)^*$ is of length at least two. The rule $S \to \lambda$ 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.

Using a grammar G that has been modified to satisfy the preceding conditions with the bottom-up parsers presented in Chapter 4 gives a complete algorithm for determining membership in L(G). For any string p of length n greater than zero, at most n reductions using rules of the form $A \to a$ may be applied. A reduction by a rule of type (iii) decreases the length of the string. Consequently, at most $n - 1$ reductions of this form are possible. Thus, after at most $2n - 1$ reductions, the bottom-up parser will either successfully complete the parse of the input string or determine that the current string is a dead-end.

5.3 Useless Symbols

A grammar is designed to generate a language. Variables are introduced to assist the string-generation process. Each variable in the 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

$$\begin{aligned} \text{G: } & S \to AC \mid BS \mid B \\ & A \to aA \mid aF \\ & B \to CF \mid b \\ & C \to cC \mid D \\ & D \to aD \mid BD \mid C \\ & E \to aA \mid BSA \\ & F \to bB \mid b. \end{aligned}$$

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 5.3.1
Let G be a context-free grammar. A symbol $x \in (V \cup \Sigma)$ is **useful** if there is a derivation

$$S \underset{G}{\overset{*}{\Rightarrow}} uxv \underset{G}{\overset{*}{\Rightarrow}} w,$$

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. A variable is useful if it occurs in a derivation that begins with the start symbol and generates a terminal string. 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, there must be a derivation of a terminal string (the null string is considered to be a terminal string) from the variable. A variable that occurs in a sentential form is said to be **reachable** from S. A two-part procedure to eliminate useless variables is presented. Each construction establishes one of the requirements for the variables to be useful.

Algorithm 5.3.2 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 5.3.2 generates the desired set follows the techniques presented in the proof of Lemma 5.1.3 and is left as an exercise.

Algorithm 5.3.2
Construction of the Set of Variables That Derive Terminal Strings

input: context-free grammar $G = (V, \Sigma, P, S)$

1. TERM := $\{A \mid$ there is a rule $A \to w \in P$ with $w \in \Sigma^*\}$
2. **repeat**
   2.1. PREV := TERM
   2.2. **for** each variable $A \in V$ **do**
      if there is an A rule $A \to w$ and $w \in (\text{PREV} \cup \Sigma)^*$ **then**
         TERM := TERM $\cup \{A\}$
   until PREV = 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 5.3.3
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 − TERM.

$$V_T = \text{TERM}$$

$$P_T = \{A \to w \mid A \to w \text{ is a rule in P, } A \in \text{TERM, and } w \in (\text{TERM} \cup \Sigma)^*\}$$

$$\Sigma_T = \{a \in \Sigma \mid a \text{ occurs in the right-hand side of a rule in } P_T\}$$

The set 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 − TERM has no effect on the set of terminal strings generated. Let $S \overset{*}{\underset{G}{\Rightarrow}} w$ be a derivation of a string $w \in L(G)$. This is also a derivation in G_T. If not, a variable from V − TERM must occur in an intermediate step in the derivation. A derivation from a sentential form containing a variable in V − TERM cannot produce a terminal string. Consequently, all the rules in the derivation are in P_T and $w \in L(G_T)$. ■

Example 5.3.1

The grammar G_T is constructed for the grammar G introduced at the beginning of this section.

$$
\begin{aligned}
\text{G: } & S \to AC \mid BS \mid B \\
& A \to aA \mid aF \\
& B \to CF \mid b \\
& C \to cC \mid D \\
& D \to aD \mid BD \mid C \\
& E \to aA \mid BSA \\
& F \to bB \mid b
\end{aligned}
$$

Algorithm 5.3.2 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

$$
\begin{aligned}
& V_T = \{S, A, B, E, F\} \\
& \Sigma_T = \{a, b\} \\
\text{P}_T\text{: } & S \to BS \mid B \\
& A \to aA \mid aF \\
& B \to b \\
& E \to aA \mid BSA \\
& F \to bB \mid b.
\end{aligned}
$$

The indirectly recursive loops generated by the variables C and D, which can never be exited once entered, are discovered by the algorithm. All rules containing these variables are deleted. □

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 5.3.4
Construction of the Set of Reachable Variables

input: context-free grammar G = (V, Σ, P, S)

1. REACH := $\{S\}$
2. PREV := $\emptyset$
3. **repeat**
   3.1. NEW := REACH − PREV
   3.2. PREV := REACH
   3.3. **for** each variable $A \in$ NEW **do**
   for each rule $A \rightarrow w$ **do** add all variables in w to REACH

until REACH = PREV

Algorithm 5.3.4, like Algorithm 5.2.1, 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 5.3.5
Let G = (V, Σ, P, S) be a context-free grammar. Algorithm 5.3.4 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 $S \stackrel{*}{\Rightarrow} xAy$. 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 $S \stackrel{n}{\Rightarrow} uAv$, 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 $S \stackrel{n}{\Rightarrow} xAy \Rightarrow xuBvy$ 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 5.3.6
Let G = (V, Σ, 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 5.3.4 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.

$$V_U = \text{REACH}$$

$$P_U = \{A \to w \mid A \to w \in P_T, A \in \text{REACH}, \text{ and } w \in (\text{REACH} \cup \Sigma)^*\}$$

$$\Sigma_U = \{a \in \Sigma \mid a \text{ occurs in the right-hand side of a rule in } P_U\}$$

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 5.3.2

The grammar G_U is constructed from the grammar G_T in Example 5.3.1. The set of reachable variables of G_T is obtained using Algorithm 5.3.4.

Iteration	REACH	PREV	NEW
0	$\{S\}$	$\emptyset$	
1	$\{S, B\}$	$\{S\}$	$\{S\}$
2	$\{S, B\}$	$\{S, B\}$	$\{B\}$

Removing all references to the variables A, E, and F produces the grammar

$$G_U: \; S \to BS \mid B$$

$$B \to b.$$

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 5.3.6. The first step is the removal of variables that do not generate terminal strings. 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 5.3.3

Let G be the grammar

$$G: \; S \to a \mid AB$$

$$A \to b.$$

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:	Remove unreachable symbols:
$S \to a$	$S \to a \mid AB$
$A \to b$	$A \to b$
Remove unreachable symbols:	Remove variables that do not generate terminal strings:
$S \to a$	$S \to a$
	$A \to b$

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 noncontracting grammar. This phenomenon is illustrated by the transformations in Excercises 8 and 17.

5.4 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 5.4.1

A context-free grammar $G = (V, \Sigma, P, S)$ is in **Chomsky normal form** if each rule has one of the following forms:

i) $A \to BC$

ii) $A \to a$

iii) $S \to \lambda$,

where $B, C \in V - \{S\}$.

The derivation tree associated with a derivation in a Chomsky normal form grammar is a binary tree. The application of a rule $A \to BC$ 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 8 to establish repetition properties of strings in a context-free languages. For the present, we will use the Chomsky normal form as another step in the sequence of transformations leading to the Greibach normal form.

The conversion of a grammar to Chomsky normal form continues the sequence of modifications presented in the previous sections.

Theorem 5.4.2
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 We assume

i) the start symbol of G is nonrecursive

ii) G does not contain lambda rules other than $S \rightarrow \lambda$

iii) G does not contain chain rules

iv) G does not contain useless symbols.

If these conditions are not satisfied by G, an equivalent grammar can be constructed that satisfies conditions (i) to (iv) and the transformation to Chomsky normal form will utilize the modified grammar. A rule in a grammar satisfying these conditions has the form $S \rightarrow \lambda$, $A \rightarrow a$, or $A \rightarrow w$, where $w \in ((V \cup \Sigma) - \{S\})^*$ and $length(w) > 1$. 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 $length(w)$ 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. The rule

$$A \rightarrow bDcF$$

can be replaced by the three rules

$$A \rightarrow B'DC'F$$
$$B' \rightarrow b$$
$$C' \rightarrow c.$$

After this transformation, the right-hand side of a rule consists of the null string, a terminal, or a string in V^+. 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

$$A \rightarrow B'T_1$$
$$T_1 \rightarrow DT_2$$
$$T_2 \rightarrow C'F.$$

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 5.4.1

Let G be the grammar

$$S \rightarrow aABC \mid a$$
$$A \rightarrow aA \mid a$$
$$B \rightarrow bcB \mid bc$$
$$C \rightarrow cC \mid c.$$

This grammar already satisfies the conditions placed on the start symbol and lambda 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.

$$\begin{aligned} \text{G}': \quad & S \rightarrow A'T_1 \mid a \\ & A' \rightarrow a \\ & T_1 \rightarrow AT_2 \\ & T_2 \rightarrow BC \\ & A \rightarrow A'A \mid a \\ & B \rightarrow B'T_3 \mid B'C' \\ & T_3 \rightarrow C'B \\ & C \rightarrow C'C \mid c \\ & B' \rightarrow b \\ & C' \rightarrow c \end{aligned}$$

□

Example 5.4.2

The preceding techniques are used to transform the grammar AE (Section 4.3) to an equivalent grammar in Chomsky normal form. The start symbol of AE is nonrecursive and

the grammar does not contain lambda rules. Removing the chain rules yields

$$S \rightarrow A + T \mid b \mid (A)$$
$$A \rightarrow A + T \mid b \mid (A)$$
$$T \rightarrow b \mid (A).$$

The transformation to Chomsky normal form requires the introduction of new variables. The resulting grammar, along with a brief description of the role of each new variable, is given below.

$S \rightarrow AY \mid b \mid LZ$	Y represents $+T$
$Z \rightarrow AR$	Z represents $A)$
$A \rightarrow AY \mid b \mid LZ$	L represents (
$T \rightarrow b \mid LZ$	R represents)
$Y \rightarrow PT$	P represents $+$
$P \rightarrow +$	
$L \rightarrow ($	
$R \rightarrow)$	

□

5.5 Removal of Direct Left Recursion

The halting conditions of the top-down parsing algorithms depend upon the generation of terminal prefixes to discover dead-ends. The directly left recursive rule $A \rightarrow A + T$ in the grammar AE introduced the possibility of unending computations in both the breadth-first and depth-first algorithms. Repeated applications of this rule fail to generate a prefix that can terminate the parse.

Consider derivations using the rules $A \rightarrow Aa \mid b$. Repeated applications of the directly left recursive rule $A \rightarrow Aa$ produce strings of the form Aa^i, $i \geq 0$. The derivation terminates with the application of the nonrecursive rule $A \rightarrow b$, generating ba^*. The derivation of $baaa$ has the form

$$\begin{aligned} A &\Rightarrow Aa \\ &\Rightarrow Aaa \\ &\Rightarrow Aaaa \\ &\Rightarrow baaa. \end{aligned}$$

Applications of the directly left recursive rule generate a string of a's but do not increase the length of the terminal prefix. The prefix grows only when the nondirectly left recursive rule is applied.

To avoid the possibility of a nonterminating parse, directly left recursive rules must be removed from the grammar. Recursion itself cannot be removed; it is necessary to generate strings of arbitrary length. It is the left recursion that causes the problem, not recursion in general. The general technique for replacing directly left recursive rules is illustrated by the following examples.

a) $A \rightarrow Aa \mid b$ b) $A \rightarrow Aa \mid Ab \mid b \mid c$ c) $A \rightarrow AB \mid BA \mid a$
$B \rightarrow b \mid c$

The sets generated by these rules are ba^*, $(b \cup c)(a \cup b)^*$, and $(b \cup c)^*a(b \cup c)^*$, respectively. The direct left recursion builds a string to the right of the recursive variable. The recursive sequence is terminated by an A rule that is not directly 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) $A \rightarrow bZ \mid b$
$Z \rightarrow aZ \mid a$

b) $A \rightarrow bZ \mid cZ \mid b \mid c$
$Z \rightarrow aZ \mid bZ \mid a \mid b$

c) $A \rightarrow BAZ \mid aZ \mid BA \mid a$
$Z \rightarrow BZ \mid B$
$B \rightarrow b \mid c$

The rules in (a) generate ba^* with left recursion replaced by direct 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 directly 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 directly left recursive rules

$$A \rightarrow Au_1 \mid Au_2 \mid \ldots \mid Au_j$$

and the rules

$$A \rightarrow v_1 \mid v_2 \mid \ldots \mid v_k$$

in which the first symbol of each v_i is not A. A leftmost derivation from these rules consists of applications of directly left recursive rules followed by the application of a rule $A \rightarrow v_i$, which ends the direct 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 initially place one of the v_i's on the left-hand side of the derived string:

$$A \rightarrow v_1 \mid \ldots \mid v_k \mid v_1 Z \mid \ldots \mid v_k Z.$$

If the string contains a sequence of u_i's, they are generated by the Z rules

$$Z \rightarrow u_1 Z \mid \ldots \mid u_j Z \mid u_1 \mid \ldots \mid u_j$$

using right recursion.

Example 5.5.1

A set of rules without direct left recursion is constructed to generate the same strings as

$$A \rightarrow Aa \mid Aab \mid bb \mid b.$$

These rules generate $(b \cup bb)(a \cup ab)^*$. The left recursion in 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 to generate the leftmost symbol of the string.

$$A \rightarrow bb \mid b \mid bbZ \mid bZ$$

The Z rules generate $(a \cup ab)^+$ using the right recursive rules

$$Z \rightarrow aZ \mid abZ \mid a \mid ab.$$ □

Lemma 5.5.1

Let G = (V, Σ, 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′, Σ, P′, S') in which A is not directly left recursive.

Proof We assume that the start symbol of G is nonrecursive, the only lambda 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 directly left recursive since the first symbol of each of the v_i's is not A. The Z rules are also not directly 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 directly left recursive rules while preserving the language of the grammar. Will this eliminate the possibility of unending computations in the top-down parsing algorithms? A derivation using the rules

$A \to Bu$ and $B \to Av$ can generate the sentential forms

$$\begin{aligned} A &\Rightarrow Bu \\ &\Rightarrow Avu \\ &\Rightarrow Buvu \\ &\Rightarrow Avuvu \\ &\vdots \end{aligned}$$

The difficulties associated with direct left recursion can also be caused by indirect left recursion. Additional transformations are required to remove all possible occurrences of left recursion.

5.6 Greibach Normal Form

The construction of terminal prefixes facilitates the discovery of dead-ends by the top-down parsing algorithms. A normal form, the Greibach normal form, is presented in which the application of every rule increases the length of the terminal prefix of the derived string. This ensures that left recursion, direct or indirect, cannot occur.

Definition 5.6.1

A context-free grammar $G = (V, \Sigma, P, S)$ is in **Greibach normal form** if each rule has one of the following forms:

i) $A \to aA_1A_2 \ldots A_n$

ii) $A \to a$

iii) $S \to \lambda$,

where $a \in \Sigma$ and $A_i \in V - \{S\}$ for $i = 1, 2, \ldots, n$.

There are several alternative definitions of the 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.

Lemma 5.6.2 provides a schema for removing a rule while preserving the language generated by the grammar that is used in the construction of a Greibach normal form grammar.

Lemma 5.6.2

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. Let $A \to uBv$ be a rule in P and $B \to w_1 \mid w_2 \mid \ldots \mid w_n$ be the B rules of P. The grammar $G' = (V, \Sigma, P', S)$ where

$$P' = (P - \{A \to uBv\}) \cup \{A \to uw_1v \mid uw_2v \mid \ldots \mid uw_nv\}$$

is equivalent to G.

Proof Since each rule $A \to uw_iv$ is derivable in G, the inclusion $L(G') \subseteq L(G)$ follows from Lemma 5.1.4.

The opposite inclusion is established by showing that every terminal string derivable in G using the rule $A \to uBv$ is also derivable in G'. The derivation of a terminal string that utilizes this rule has the form $S \overset{*}{\Rightarrow} pAq \Rightarrow puBvq \Rightarrow puw_ivq \overset{*}{\Rightarrow} w$. The same string can be generated in G' using the rule $A \to uw_iv$. ■

The conversion of a Chomsky normal form grammar to Greibach normal form uses two rule transformation techniques: the rule replacement scheme of Lemma 5.6.2 and the transformation that removes directly 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) $S \to \lambda$
ii) $A \to aw$
iii) $A \to Bw$,

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.

$$\begin{aligned} \text{G: } & S \to AB \mid \lambda \\ & A \to AB \mid CB \mid a \\ & B \to AB \mid b \\ & C \to AC \mid c \end{aligned}$$

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 directly left recursive rule $A \to AB$ violates these restrictions. Lemma 5.5.1 can be used to remove the direct left recursion, yielding

$$
\begin{aligned}
S &\to AB \mid \lambda \\
A &\to CBR_1 \mid aR_1 \mid CB \mid a \\
B &\to AB \mid b \\
C &\to AC \mid c \\
R_1 &\to BR_1 \mid B.
\end{aligned}
$$

Now the B rules must be transformed to the appropriate form. The rule $B \to AB$ 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 5.6.2 permits the leading A in the right-hand side of the rule $B \to AB$ to be replaced by the right-hand side of the A rules, producing

$$
\begin{aligned}
S &\to AB \mid \lambda \\
A &\to CBR_1 \mid aR_1 \mid CB \mid a \\
B &\to CBR_1B \mid aR_1B \mid CBB \mid aB \mid b \\
C &\to AC \mid c \\
R_1 &\to BR_1 \mid B.
\end{aligned}
$$

Applying the replacement techniques of Lemma 5.6.2 to the C rules creates two directly left recursive rules.

$$
\begin{aligned}
S &\to AB \mid \lambda \\
A &\to CBR_1 \mid aR_1 \mid CB \mid a \\
B &\to CBR_1B \mid aR_1B \mid CBB \mid aB \mid b \\
C &\to CBR_1C \mid aR_1C \mid CBC \mid aC \mid c \\
R_1 &\to BR_1 \mid B
\end{aligned}
$$

The left recursion can be removed, introducing the new variable R_2.

$$
\begin{aligned}
S &\to AB \mid \lambda \\
A &\to CBR_1 \mid aR_1 \mid CB \mid a \\
B &\to CBR_1B \mid aR_1B \mid CBB \mid aB \mid b \\
C &\to aR_1C \mid aC \mid c \mid aR_1CR_2 \mid aCR_2 \mid cR_2 \\
R_1 &\to BR_1 \mid B \\
R_2 &\to BR_1CR_2 \mid BCR_2 \mid BR_1C \mid BC
\end{aligned}
$$

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 5.6.2. Making this transformation, we obtain the rules

$$
\begin{aligned}
S &\to AB \mid \lambda \\
A &\to CBR_1 \mid aR_1 \mid CB \mid a \\
B &\to aR_1B \mid aB \mid b \\
&\to aR_1CBR_1B \mid aCBR_1B \mid cBR_1B \mid aR_1CR_2BR_1B \mid aCR_2BR_1B \mid cR_2BR_1B \\
&\to aR_1CBB \mid aCBB \mid cBB \mid aR_1CR_2BB \mid aCR_2BB \mid cR_2BB \\
C &\to aR_1C \mid aC \mid c \mid aR_1CR_2 \mid aCR_2 \mid cR_2 \\
R_1 &\to BR_1 \mid B \\
R_2 &\to BR_1CR_2 \mid BCR_2 \mid BR_1C \mid BC.
\end{aligned}
$$

The second list of B rules is obtained by substituting for C in the rule $B \to CBR_1B$ and the third in the rule $B \to CBB$. 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

$$
\begin{aligned}
S &\to \lambda \\
&\to aR_1B \mid aB \\
&\to aR_1CBR_1B \mid aCBR_1B \mid cBR_1B \mid aR_1CR_2BR_1B \mid aCR_2BR_1B \mid cR_2BR_1B \\
&\to aR_1CBB \mid aCBB \mid cBB \mid aR_1CR_2BB \mid aCR_2BB \mid cR_2BB \\
A &\to aR_1 \mid a \\
&\to aR_1CBR_1 \mid aCBR_1 \mid cBR_1 \mid aR_1CR_2BR_1 \mid aCR_2BR_1 \mid cR_2BR_1 \\
&\to aR_1CB \mid aCB \mid cB \mid aR_1CR_2B \mid aCR_2B \mid cR_2B \\
B &\to aR_1B \mid aB \mid b \\
&\to aR_1CBR_1B \mid aCBR_1B \mid cBR_1B \mid aR_1CR_2BR_1B \mid aCR_2BR_1B \mid cR_2BR_1B \\
&\to aR_1CBB \mid aCBB \mid cBB \mid aR_1CR_2BB \mid aCR_2BB \mid cR_2BB \\
C &\to aR_1C \mid aC \mid c \mid aR_1CR_2 \mid aCR_2 \mid cR_2 \\
R_1 &\to BR_1 \mid B \\
R_2 &\to BR_1CR_2 \mid BCR_2 \mid BR_1C \mid BC.
\end{aligned}
$$

Finally, the substitution process must be applied to each of the variables added in the removal of direct recursion. Rewriting these rules yields

$$
\begin{aligned}
R_1 &\to aR_1BR_1 \mid aBR_1 \mid bR_1 \\
&\to aR_1CBR_1BR_1 \mid aCBR_1BR_1 \mid cBR_1BR_1 \mid aR_1CR_2BR_1BR_1 \mid aCR_2BR_1BR_1 \mid cR_2BR_1 \\
&\to aR_1CBBR_1 \mid aCBBR_1 \mid cBBR_1 \mid aR_1CR_2BBR_1 \mid aCR_2BBR_1 \mid cR_2BBR_1 \\
R_1 &\to aR_1B \mid aB \mid b \\
&\to aR_1CBR_1B \mid aCBR_1B \mid cBR_1B \mid aR_1CR_2BR_1B \mid aCR_2BR_1B \mid cR_2BR_1B \\
&\to aR_1CBB \mid aCBB \mid cBB \mid aR_1CR_2BB \mid aCR_2BB \mid cR_2BB \\
R_2 &\to aR_1BR_1CR_2 \mid aBR_1CR_2 \mid bR_1CR_2 \\
&\to aR_1CBR_1BR_1CR_2 \mid aCBR_1BR_1CR_2 \mid cBR_1BR_1CR_2 \mid aR_1CR_2BR_1BR_1CR_2 \mid \\
&\quad aCR_2BR_1BR_1CR_2 \mid cR_2BR_1BR_1CR_2 \\
&\to aR_1CBBR_1CR_2 \mid aCBBR_1CR_2 \mid cBBR_1CR_2 \mid aR_1CR_2BBR_1CR_2 \mid aCR_2BBR_1CR_2 \\
&\quad cR_2BBR_1CR_2 \\
R_2 &\to aR_1BCR_2 \mid aBCR_2 \mid bCR_2 \\
&\to aR_1CBR_1BCR_2 \mid aCBR_1BCR_2 \mid cBR_1BCR_2 \mid aR_1CR_2BR_1BCR_2 \mid aCR_2BR_1BCR_2 \\
&\quad cR_2BR_1BCR_2 \\
&\to aR_1CBBCR_2 \mid aCBBCR_2 \mid cBBCR_2 \mid aR_1CR_2BBCR_2 \mid aCR_2BBCR_2 \mid cR_2BBCR_2 \\
R_2 &\to aR_1BR_1C \mid aBR_1C \mid bR_1C \\
&\to aR_1CBR_1BR_1C \mid aCBR_1BR_1C \mid cBR_1BR_1C \mid aR_1CR_2BR_1BR_1C \mid aCR_2BR_1BR_1C \mid \\
&\quad cR_2BR_1BR_1C \\
&\to aR_1CBBR_1C \mid aCBBR_1C \mid cBBR_1C \mid aR_1CR_2BBR_1C \mid aCR_2BBR_1C \mid cR_2BBR_1C \\
R_2 &\to aR_1BC \mid aBC \mid bC \\
&\to aR_1CBR_1BC \mid aCBR_1BC \mid cBR_1BC \mid aR_1CR_2BR_1BC \mid aCR_2BR_1BC \mid cR_2BR_1BC \\
&\to aR_1CBBC \mid aCBBC \mid cBBC \mid aR_1CR_2BBC \mid aCR_2BBC \mid cR_2BBC.
\end{aligned}
$$

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 appropriate computer program. The input to such a program consists of the rules of a 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 specified by Lemma 5.6.2. The variable A is a useful symbol of G, occurring in the derivation

$$S \Rightarrow AB \Rightarrow aB \Rightarrow ab.$$

In the conversion to Griebach 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 5.6.3

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 directly left recursive or in the correct form. All directly left recursive variables can be transformed using the technique of Lemma 5.5.1. ■

Example 5.6.1

The Chomsky and Greibach normal forms are constructed for the grammar

$$S \rightarrow SaB \mid aB$$
$$B \rightarrow bB \mid \lambda.$$

Adding a nonrecursive start symbol S' and removing lambda and chain rules yields

$$S' \rightarrow SaB \mid Sa \mid aB \mid a$$
$$S \rightarrow SaB \mid Sa \mid aB \mid a$$
$$B \rightarrow bB \mid b.$$

The Chomsky normal form is obtained by transforming the preceding rules. Variables A and C are used as aliases for a and b, respectively. T represents the string aB.

$$S' \rightarrow ST \mid SA \mid AB \mid a$$
$$S \rightarrow ST \mid SA \mid AB \mid a$$
$$B \rightarrow CB \mid b$$
$$T \rightarrow AB$$
$$A \rightarrow a$$
$$C \rightarrow b$$

The variables are ordered by S', S, B, T, A, C. Removing the directly left recursive S rules produces

$$S' \rightarrow ST \mid SA \mid AB \mid a$$
$$S \rightarrow ABZ \mid aZ \mid AB \mid a$$
$$B \rightarrow CB \mid b$$
$$T \rightarrow AB$$
$$A \rightarrow a$$
$$C \rightarrow b$$
$$Z \rightarrow TZ \mid AZ \mid T \mid A.$$

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

$$S' \rightarrow aBZT \mid aZT \mid aBT \mid aT \mid aBZA \mid aZA \mid aBA \mid aA \mid aB \mid a$$
$$S \rightarrow aBZ \mid aZ \mid aB \mid a$$
$$B \rightarrow bB \mid b$$
$$T \rightarrow aB$$
$$A \rightarrow a$$
$$C \rightarrow b$$
$$Z \rightarrow aBZ \mid aZ \mid aB \mid a.$$

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	$\Rightarrow$	$abZA$
$\Rightarrow$	$SaBaBaB$	$\Rightarrow$	$SATA$	$\Rightarrow$	$abaZA$
$\Rightarrow$	$aBaBaBaB$	$\Rightarrow$	$ABATA$	$\Rightarrow$	$abaaBA$
$\Rightarrow$	$abBaBaBaB$	$\Rightarrow$	$aBATA$	$\Rightarrow$	$abaabA$
$\Rightarrow$	$abaBaBaB$	$\Rightarrow$	$abATA$	$\Rightarrow$	$abaaba$
$\Rightarrow$	$abaaBaB$	$\Rightarrow$	$abaTA$		
$\Rightarrow$	$abaabBaB$	$\Rightarrow$	$abaABA$		
$\Rightarrow$	$abaabaB$	$\Rightarrow$	$abaaBA$		
$\Rightarrow$	$abaaba$	$\Rightarrow$	$abaabA$		
		$\Rightarrow$	$abaaba$		

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. □

The top-down parsing algorithms presented in Chapter 4 terminate for all input strings when using the rules of a grammar in Greibach normal form. The derivation of a string of length n, where n is greater than zero, requires exactly n rule applications. Each application adds one terminal symbol to the terminal prefix of the derived string. The construction of a path of length n in the graph of a grammar will either successfully terminate the parse or the derived string will be declared a dead-end.

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 5.2.2.

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 lambda 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: $S \rightarrow A \mid B \mid C$
   $A \rightarrow aa \mid B$
   $B \rightarrow bb \mid C$
   $C \rightarrow cc \mid A$
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 5.3.2 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: $S \rightarrow AA \mid CD \mid bB$
    $A \rightarrow aA \mid a$

$B \to bB \mid bC$
$C \to cB$
$D \to dD \mid d$

15. G: $S \to aA \mid BD$
$A \to aA \mid aAB \mid aD$
$B \to aB \mid aC \mid BF$
$C \to Bb \mid aAC \mid E$
$D \to bD \mid bC \mid b$
$E \to aB \mid bC$
$F \to aF \mid aG \mid a$
$G \to a \mid b$

16. G: $S \to ACH \mid BB$
$A \to aA \mid aF$
$B \to CFH \mid b$
$C \to aC \mid DH$
$D \to aD \mid BD \mid Ca$
$F \to bB \mid b$
$H \to dH \mid d$

17. Show that all the symbols of G are useful. Construct an equivalent grammar G_C by removing the chain rules from G. Show that G_C contains useless symbols.

$$\begin{aligned} \text{G: } & S \to A \mid CB \\ & A \to C \mid D \\ & B \to bB \mid b \\ & C \to cC \mid c \\ & D \to dD \mid d \end{aligned}$$

18. Convert the grammar G to Chomsky normal form. G already satisfies the conditions on the start symbol S, lambda rules, useless symbols, and chain rules.

$$\begin{aligned} \text{G: } & S \to aA \mid ABa \\ & A \to AA \mid a \\ & B \to AbB \mid bb \end{aligned}$$

19. Convert the grammar G to Chomsky normal form. G already satisfies the conditions on the start symbol S, lambda rules, useless symbols, and chain rules.

$$\begin{aligned} \text{G: } & S \to aAbB \mid ABC \mid a \\ & A \to aA \mid a \\ & B \to bBcC \mid b \\ & C \to abc \end{aligned}$$

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

$$\begin{aligned} \text{G: } & S \to A \mid ABa \mid AbA \\ & A \to Aa \mid \lambda \\ & B \to Bb \mid BC \\ & C \to CB \mid CA \mid bB \end{aligned}$$

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 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. Let G be the grammar

$$\begin{aligned} \text{G: } & S \to A \mid B \\ & A \to aaB \mid Aab \mid Aba \\ & B \to bB \mid Bb \mid aba. \end{aligned}$$

    a) Give a regular expression for L(G).
    b) Construct a grammar G′ that contains no directly left recursive rules and is equivalent to G.
26. Construct a grammar G′ that contains no directly left recursive rules and is equivalent to

$$\begin{aligned} \text{G: } & S \to A \mid C \\ & A \to AaB \mid AaC \mid B \mid a \\ & B \to Bb \mid Cb \\ & C \to cC \mid c. \end{aligned}$$

    Give a leftmost derivation of the string $aaccacb$ in the grammars G and G′.
27. Construct a grammar G′ that contains no directly left recursive rules and is equivalent to

$$\begin{aligned} \text{G: } & S \to A \mid B \\ & A \to AAA \mid a \mid B \\ & B \to BBb \mid b. \end{aligned}$$

28. Construct a Greibach normal form grammar equivalent to

$$S \to aAb \mid a$$
$$A \to SS \mid b.$$

29. Convert the Chomsky normal form grammar

$$S \to BB$$
$$A \to AA \mid a$$
$$B \to AA \mid BA \mid b$$

to Greibach normal form. Process the variables according to the order S, A, B.

30. Convert the Chomsky normal form grammar

$$S \to AB \mid BC$$
$$A \to AB \mid a$$
$$B \to AA \mid CB \mid b$$
$$C \to a \mid b$$

to Greibach normal form. Process the variables according to the order S, A, B, C.

31. Convert the Chomsky normal form grammar

$$S \to BA \mid AB \mid \lambda$$
$$A \to BB \mid AA \mid a$$
$$B \to AA \mid b$$

to Greibach normal form. Process the variables according to the order S, A, B.

32. Convert the Chomsky normal form grammar

$$S \to AB$$
$$A \to BB \mid CC$$
$$B \to AD \mid CA$$
$$C \to a$$
$$D \to b$$

to Greibach normal form. Process the variables according to the order S, A, B, C, D.

33. Use the Chomsky normal form of the grammar AE given in Example 5.4.2 to construct the Greibach normal form. Use the ordering S, Z, A, T, Y, P, L, R. Remove all useless symbols that are created by the transformation to Greibach normal form.

34. Use the breadth-first top-down parsing algorithm and the grammar from Exercise 33 to construct a derivation of $(b + b)$. Compare this with the parse given in Figure 4.3.

35. Use the depth-first top-down parsing algorithm and the grammar from Exercise 33 to construct a derivation of $(b) + b$. Compare this with the parse given in Example 4.4.2.

36. 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) $A \rightarrow aB$

    iv) $A \rightarrow aBC$,

    where $A \in \mathrm{V}, B, C \in \mathrm{V} - \{S\}$, and $a \in \Sigma$.

Bibliographic Notes

The constructions for removing lambda rules and chain rules were presented in Bar-Hillel, Perles, and Shamir [1961]. Chomsky normal form was introduced in Chomsky [1959]. Greibach normal form is from Greibach [1965]. A grammar whose rules satisfy the conditions of Exercise 36 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].

PART III

Automata and Languages

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 is defined by a finite set of instructions that specify the operations that make up the procedure. The execution of an instruction is mechanical; it requires no cleverness or ingenuity on the part of the machine or person doing the computation. The computation defined by an effective procedure consists of sequentially executing a finite number of instructions and terminating. These properties can be summarized as follows: An effective procedure is a deterministic discrete process that halts for all possible input values.

This section introduces a series of increasingly powerful abstract computing machines whose computations satisfy our notion of an effective process. The input to a machine consists of a string over an alphabet and the result of a computation indicates the acceptability of the input string. The set of accepted strings is the language recognized by the machine. Thus we have associated languages and machines, the two topics that are the focus of this book.

The study of abstract machines begins with 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. Kleene's theorem shows that finite automata accept precisely the languages generated by regular grammars.

A more powerful class of read-once machines, the 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.

In 1936 British mathematician Alan Turing designed a family of abstract machines that he believed capable of performing every intuitively effective procedure. As with a finite automaton, the applicable Turing machine instruction is determined by the state of the machine and the symbol being read. However, a Turing machine may read its input multiple times and an instruction may write information to memory. The read-write capability of Turing machines increases the number of languages that can be recognized and provides a theoretical prototype for the modern computer. The relationship between effective computation and the capabilities of Turing machines will be discussed in Chapters 11 and 13.

The correspondence between generation of languages by grammars and acceptance by machines extends to the languages accepted by 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. Unrestricted grammars are defined by rules in which there are no restrictions on the form or applicability of the rules. To establish the correspondence between recognition by a Turing machine and generation by an unrestricted grammar, we will show that the computation of a Turing machine can be simulated by a derivation in an unrestricted grammar. The families of machines, grammars, and languages examined in this section make up the Chomsky hierarchy, which associates languages with the grammars that generate them and the machines that accept them.

CHAPTER 6

Finite Automata

An effective procedure that determines whether an input string is in a language is called a *language acceptor.* When parsing with a grammar in Greibach normal form, the top-down parsers of Chapter 4 can be thought of as acceptors of context-free languages. The generation of prefixes guarantees that the computation of the parser produces an answer for every input string. The objective of this chapter is to define a class of abstract machines whose computations, like those of a parser, determine the acceptability of an input string.

Properties common to all machines include the processing of input and the generation of 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 the machines introduced in this chapter consists of a string over an alphabet. The result of the computation indicates whether the input string is acceptable.

The previous examples 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 machines to be deterministic. This condition will be relaxed to examine the effects of nondeterminism on the capabilities of computation.

6.1 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 a parser is a program that is run on a computer. 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.

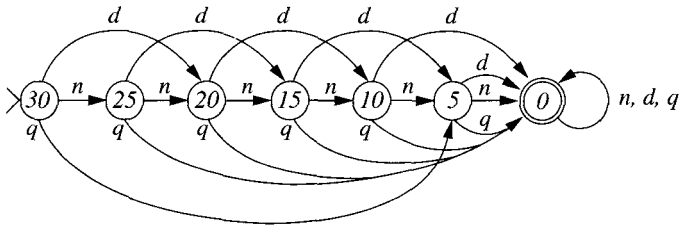


FIGURE 6.1 State diagram of newspaper vending machine.

The state of the machine at the beginning of a computation is designated ⊃◯. 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 6.1 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 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 is accepted if the computation terminates in the accepting state. The string *dndn* is accepted by the vending machine while the string *nndn* is not accepted since the computation terminates in state *5*.

6.2 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 6.2.1

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

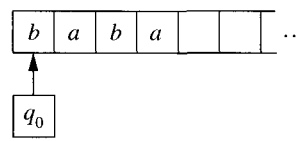
as the *start state*, F a subset of Q called the *final* or *accepting states*, and δ a total function from $Q \times \Sigma$ to Q known as the *transition function*.

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 $q_0, q_1, q_2, \ldots, q_n$. 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 constructed 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 DFA.

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 6.2. 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

M: $Q = \{q_0, q_1\}$ $\quad \delta(q_0, a) = q_1$
$\Sigma = \{a, b\}$ $\quad \delta(q_0, b) = q_0$
$F = \{q_1\}$ $\quad \delta(q_1, a) = q_1$
$\delta(q_1, b) = q_0$

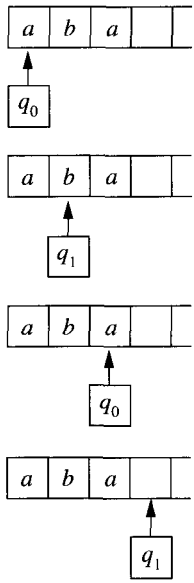


FIGURE 6.2 Computation in a DFA.

the computation terminates in an accepting state; otherwise it is rejected. The computation in Figure 6.2 exhibits the acceptance of the string aba.

Definition 6.2.2

Let M = (Q, Σ, δ, 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 recognized by the machine is simply the set of strings accepted by its computations. The language of the machine in Figure 6.2 is the set of all strings over $\{a, b\}$ that end in a. Two machines that accept the same language are said to be **equivalent.**

A DFA reads the input in a left-to-right manner; once an input symbol has been processed, 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 combination is called an **instantaneous 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 $[q_i, aw] \vdash_M [q_j, w]$ 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 $\vdash_M$, 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 6.2.3

The function $\vdash_M$ on $Q \times \Sigma^+$ is defined by

$$[q_i, aw] \vdash_M [\delta(q_i, a), w]$$

for $a \in \Sigma$ and $w \in \Sigma^*$, where δ is the transition function of the DFA M.

The notation $[q_i, u] \vdash^* [q_j, v]$ is used to indicate that configuration $[q_j, v]$ can be obtained from $[q_i, u]$ by zero or more transitions.

Example 6.2.1

The DFA M defined below accepts the set of strings over $\{a, b\}$ that contain the substring bb. That is, $L(M) = (a \cup b)^*bb(a \cup b)^*$.

$$M : Q = \{q_0, q_1, q_2\}$$
$$\Sigma = \{a, b\}$$
$$F = \{q_2\}$$

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 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. □

Example 6.2.2

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 the vending machine consists of all strings that represent a sum of 30 cents or more. Construct a regular expression that defines the language of this machine.

□

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.

Definition 6.2.4

The **extended transition function** $\hat{\delta}$ of a DFA with transition function δ is a function from $Q \times \Sigma^*$ to Q defined by recursion on the length of the input string.

i) Basis: $length(w) = 0$. Then $w = \lambda$ and $\hat{\delta}(q_i, \lambda) = q_i$.
$length(w) = 1$. 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 $L(M) = \{w \mid \hat{\delta}(q_0, w) \in F\}$.

An effective procedure has been described as an intuitively computable process consisting of a finite sequence of instructions that defines the procedure. The execution of an instruction by a DFA is a purely mechanical action determined by the input symbol and the machine state. The computation of a DFA, which consists of a sequence of these actions, clearly satisfies the conditions required of an effective procedure.

6.3 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 6.1 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.

Definition 6.3.1

The state diagram of a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is a labeled 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, depicted $\rightarrow\bigcirc$.

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 $\mathbf{p}_w$ be a path beginning at q_0 that spells w and let q_w be the terminal node of $\mathbf{p}_w$. Theorem 6.3.2 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 6.3.2

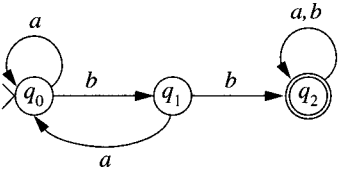
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $w \in \Sigma^*$. Then w determines a unique path $\mathbf{p}_w$ in the state diagram of M and $\hat{\delta}(q_0, w) = q_w$.

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 $\mathbf{p}_u$ that spells u and $\hat{\delta}(q_0, u) = q_u$. The path $\mathbf{p}_w$ is constructed by following the arc labeled a from q_u. This is the only path from q_0 that spells w since $\mathbf{p}_u$ is the unique path that spells u and there is only one arc leaving q_u labeled a. The terminal state of the path $\mathbf{p}_w$ 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. ■

Example 6.3.1

The state diagram of the DFA in Example 6.2.1 is



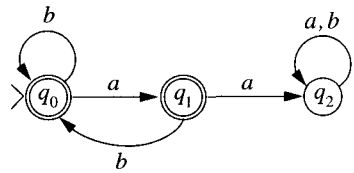
The states are used to record the number of consecutive b's processed. The state q_2 is entered when a substring bb is encountered. Once the machine enters q_2 the remainder of the input is processed, leaving the state unchanged. The computation of the DFA with input $ababb$ and the corresponding path 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. □

Example 6.3.2

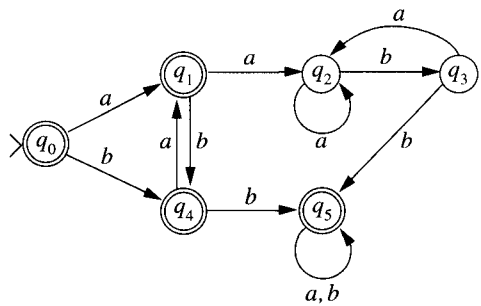
The DFA



accepts $(b \cup ab)^*(a \cup \lambda)$, the set of strings over $\{a, b\}$ that do not contain the substring aa. □

Example 6.3.3

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.

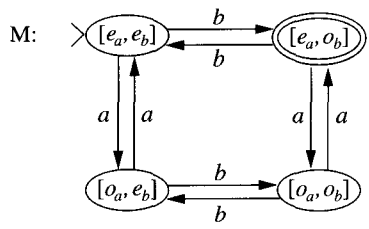


□

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 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 6.3.4

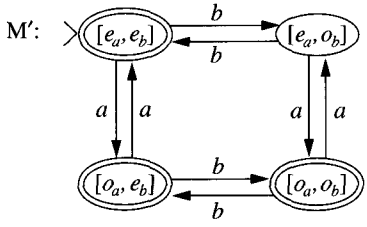
The DFA M accepts the language consisting of all strings over $\{a, b\}$ that contain an even number of a's and an odd number of b's.



At any step of the computation, there are four possibilities for the parities of the input symbols processed: even number of a's and even number of b's, even number of a's and odd number of b's, odd number of a's and even number of b's, 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 6.3.5

Let M be the DFA constructed in Example 6.3.4. 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 machines that accept complementary sets of strings. This relationship is formalized by the following result.

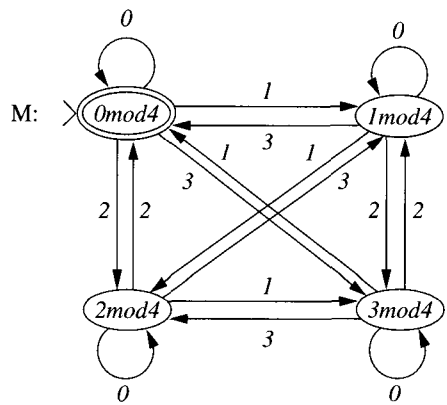
Theorem 6.3.3

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 $w \in L(M')$. ■

Example 6.3.6

Let $\Sigma = \{0, 1, 2, 3\}$. A string in Σ^* is a sequence of integers from Σ. The DFA M determines whether the sum of elements of a string is divisible by 4. The strings *1 2 3 0 2* and *0 1 3 0* should be accepted and *0 1 1 1* rejected by M. The states represent the value of the sum of the processed input modulo 4.



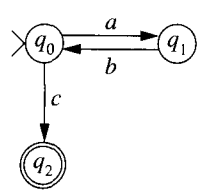
□

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 $imod4$, the machine in Example 6.3.6 acts as a modulo 4 adder.

By definition, a DFA must process the entire input even if the result has already been established. Example 6.3.7 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 6.3.7

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.

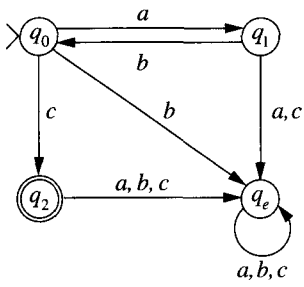


The computation with input $abcc$ is rejected since the machine is unable process the final c from state q_2. □

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 6.3.8

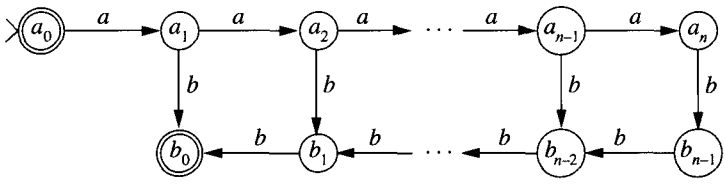
The DFA



accepts the same language as the incompletely specified DFA in Example 6.3.7. The state q_e is the error state that ensures the processing of the entire string. □

Example 6.3.9

The incompletely specified DFA defined by the state diagram given below accepts the language $\{a^i b^i \mid i \leq n\}$, for a fixed integer n. The states a_k count the number of a's, and then the b_k's ensure an equal number of b's.

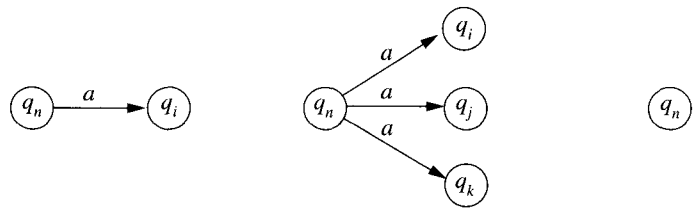


This technique cannot be extended to accept $\{a^i b^i \mid i \geq 0\}$ since an infinite number of states would be needed. In the next chapter we will show that this language is not accepted by any finite automaton. □

6.4 Nondeterministic Finite Automata

We now alter our definition of machine to allow nondeterministic computation. 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.



$\delta(q_n, a) = \{q_i\}$ $\quad$ $\delta(q_n, a) = \{q_i, q_j, q_k\}$ $\quad$ $\delta(q_n, a) = \emptyset$

With the exception of the transition function, the components of an NFA are identical to those of a DFA.

Definition 6.4.1

A **nondeterministic finite automaton** 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*.

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 state that may be entered from a given state and input symbol while an NFA allows zero, one, or more states. By interpreting the transition function of a DFA as a function from $Q \times \Sigma$ to singleton sets of alphabet elements, the family of DFAs may be considered to be a subset of the family of NFAs.

A string input to an NFA may generate several distinct computations. The notion of halting and string acceptance must be revised to conform to the flexibility of nondeterministic computation. Computations in an NFA are traced using the $\vdash$ notation introduced in Section 6.2.

Example 6.4.1

An NFA M is given, along with three distinct computations for the string *ababb*.

$$M : Q = \{q_0, q_1, q_2\}$$
$$\Sigma = \{a, b\}$$
$$F = \{q_2\}$$

δ	a	b
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$

$[q_0, ababb]$	$[q_0, ababb]$	$[q_0, ababb]$
$\vdash [q_0, babb]$	$\vdash [q_0, babb]$	$\vdash [q_0, babb]$
$\vdash [q_0, abb]$	$\vdash [q_1, abb]$	$\vdash [q_0, abb]$
$\vdash [q_0, bb]$		$\vdash [q_0, bb]$
$\vdash [q_0, b]$		$\vdash [q_1, b]$
$\vdash [q_0, \lambda]$		$\vdash [q_2, \lambda]$

The second computation of the machine M 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. □

An input string is accepted if there is a computation that processes the entire string and halts in an accepting state. A string is in the language of a nondeterministic machine if there is one computation that accepts it; the existence of other computations that do not accept the string is irrelevant. The third computation given in Example 6.4.1 demonstrates that *ababb* is in the language of machine M.

Definition 6.4.2

The **language** of an NFA M, denoted L(M), is the set of strings accepted by the M. That is, $L(M) = \{w \mid$ there is a computation $[q_0, w] \vdash^* [q_i, \lambda]$ with $q_i \in F\}$.

Definition 6.4.3

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.

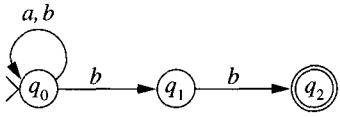
iv) F is the set of accepting nodes.

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 6.4.3 is obtained from Definition 6.3.1 by omitting condition (vi), which translates the deterministic property of the DFA transition function into its graphic representation.

Example 6.4.2

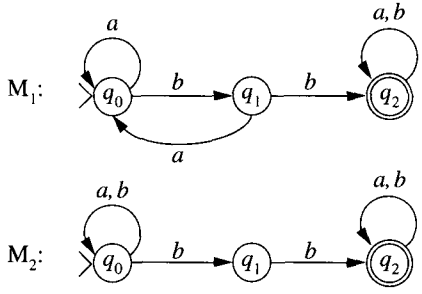
The state diagram for the NFA M from Example 6.4.1 is



Pictorially, it is clear that the language accepted by M is $(a \cup b)^*bb$. □

Example 6.4.3

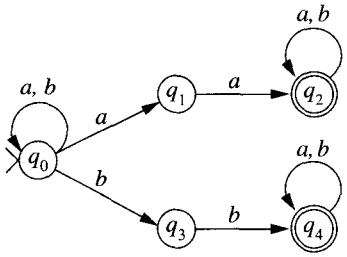
The state diagrams M_1 and M_2 define finite automata that accept $(a \cup b)^*bb(a \cup b)^*$.



M_1 is the DFA from Example 6.3.1. 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. □

Example 6.4.4

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 6.4.3) with a similar machine that accepts strings with aa.

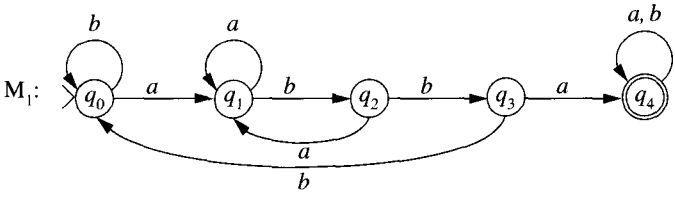


A path exhibiting the acceptance of a string reads the input in state q_0 until an occurrence of the substring aa or bb is encountered. At this point, the path branches to either q_1 or q_3, depending upon the substring. There are three distinct paths that exhibit the acceptance of the string $abaaabb$. □

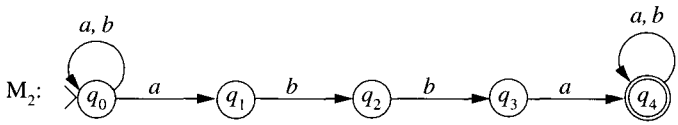
The flexibility permitted by the use of nondeterminism does not always simplify the problem of constructing a machine that accepts $L(M_1) \cup L(M_2)$ from the machines M_1 and M_2. This can be seen by attempting to construct an NFA that accepts the language of the DFA in Example 6.3.3.

Example 6.4.5

The strings over $\{a, b\}$ containing the substring $abba$ are accepted by the machines M_1 and M_2. The states record the progress made in obtaining the desired substring. The DFA M_1 must back up 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, q_0 is entered since the last four symbols processed, $abbb$, indicate no progress has been made toward recognizing $abba$.



The halting conditions of an NFA can be used to avoid the necessity of backing up when the current substring is discovered not to have the desired form. M_2 uses this technique to accept the same language as M_1.



□

6.5 Lambda Transitions

The transitions from state to state, in both deterministic and nondeterministic automata, were initiated by the processing of an input symbol. The definition of NFA is relaxed to allow state transitions without requiring input to be processed. A transition of this form is called a **lambda transition**. The class of nondeterministic machines that utilize lambda transitions is denoted NFA-λ.

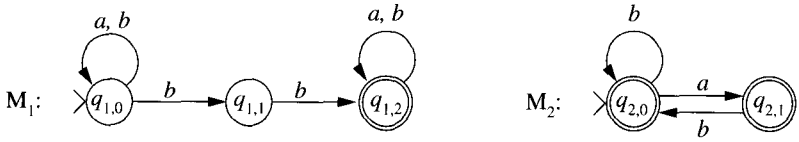
Lambda transitions represent another step away from the deterministic effective computations of a DFA. They do, however, provide a useful tool for the design of machines to accept complex languages.

Definition 6.5.1

A nondeterministic finite automaton with lambda 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 lambda 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 6.4.3 with lambda transitions represented by arcs labeled by lambda.

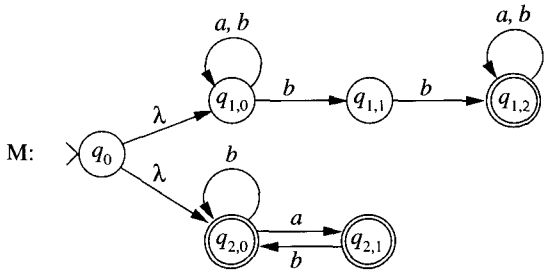
Lambda moves can be used to construct complex machines from simpler machines. Let M_1 and M_2 be the machines



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 6.5.1

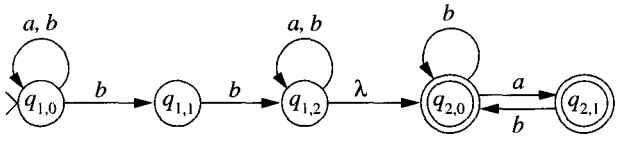
The language of the NFA-λ M is $L(M_1) \cup L(M_2)$.



A computation in the composite machine M begins by following a lambda 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 lambda 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 6.3.3. □

Example 6.5.2

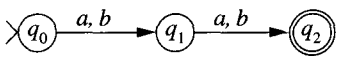
An NFA-λ that accepts $L(M_1)L(M_2)$, the concatenation of the languages of M_1 and M_2, is constructed by joining the two machines with a lambda arc.



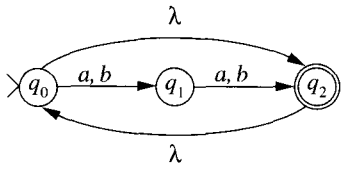
An input string is accepted only if it consists of a string from $L(M_1)$ concatenated with one from $L(M_2)$. The lambda transition allows the computation to enter M_2 whenever a prefix of the input string is accepted by M_1. □

Example 6.5.3

Lambda transitions are used to construct an NFA-λ that accepts all strings of even length over $\{a, b\}$. First we build the state diagram for a machine that accepts strings of length two.



To accept the null string, a lambda arc is added from q_0 to q_2. Strings of any positive, even length are accepted by following the lambda arc from q_2 to q_0 to repeat the sequence q_0, q_1, q_2.



□

The constructions presented in Examples 6.5.1, 6.5.2, and 6.5.3 can be generalized to construct machines that accept the union, concatenation, and Kleene star of languages accepted by existing machines. We begin by transforming an NFA-λ into a form amenable to these constructions.

Lemma 6.5.2

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 q_f is zero.

Proof The transition function of M' is constructed from that of M by adding the lambda transitions

$$\delta(q_0', \lambda) = \{q_0\}$$

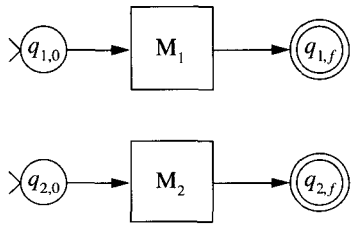
$$\delta(q_i, \lambda) = \{q_f\} \text{ for every } q_i \in F$$

for the new states q_0' and q_f. The lambda 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 lambda transition from the accepting state of M to the accepting state q_f of M'. ■

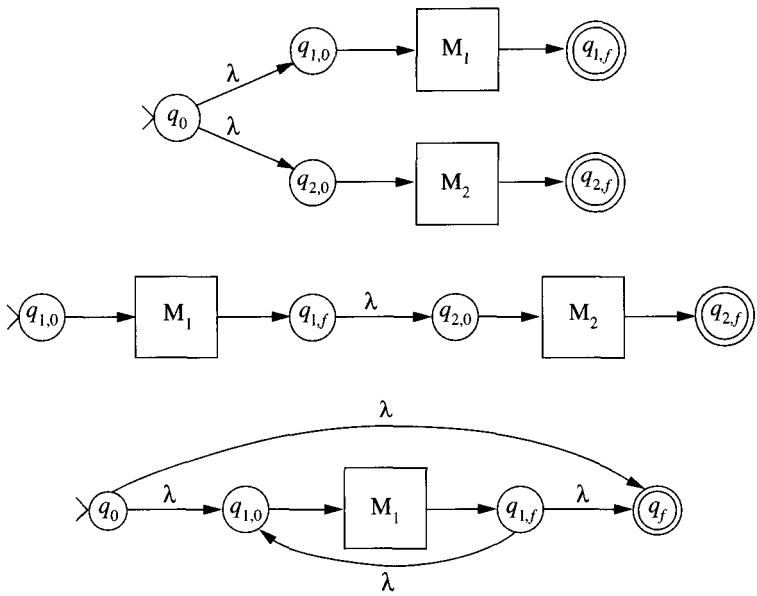
Theorem 6.5.3

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 6.5.2. Because of the restrictions on the start and final states, the machines M_1 and M_2 are depicted



The languages $L(M_1) \cup L(M_2)$, $L(M_1)L(M_2)$, and $L(M_1)^*$ are accepted by the composite machines



respectively. ■

6.6 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.

Path	String Processed
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_0	a

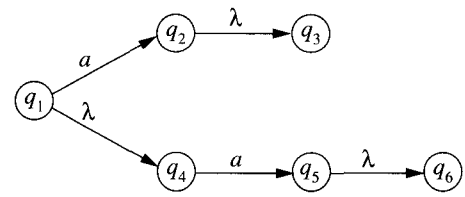


FIGURE 6.3 Paths with lambda transitions.

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 6.3 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 these 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. The node q_j is said to be in the lambda closure of q_i if there is a path from q_i to q_j that spells the null string.

Definition 6.6.1

The **lambda closure** of a state q_i, denoted $\lambda\text{-}closure(q_i)$, is defined recursively by

i) Basis: $q_i \in \lambda\text{-}closure(q_i)$.

ii) Recursive step: Let q_j be an element of $\lambda\text{-}closure(q_i)$. If $q_k \in \delta(q_j, \lambda)$, then $q_k \in \lambda\text{-}closure(q_i)$.

iii) Closure: q_j is in $\lambda\text{-}closure(q_i)$ only if it can be obtained from q_i by a finite number of applications of the recursive step.

The set $\lambda\text{-}closure(q_i)$ can be constructed following the techniques presented in Algorithm 5.2.1, which determines the chains in a context-free grammar. The input transition function is defined using the lambda closure.

Definition 6.6.2

The **input transition function** t of an NFA-λ M is a function from $Q \times \Sigma$ to $\mathcal{P}(Q)$ defined by

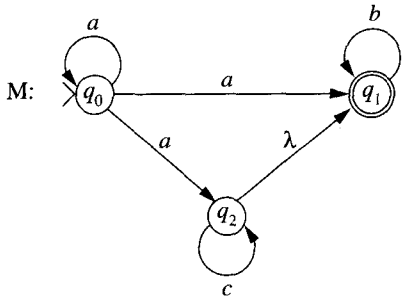
$$t(q_i, a) = \bigcup_{q_j \in \lambda\text{-}closure(q_i)} \lambda\text{-}closure(\delta(q_j, a)),$$

where δ is the transition function of M.

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 lambda transitions, the input transition function t is identical to the transition function δ of the automaton.

Example 6.6.1

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 $a^+c^*b^*$.



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 a nondeterministic automaton is used to construct an equivalent deterministic automaton. The procedure uses the state diagram of the NFA-λ to construct the state diagram of the 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 simulates the simultaneous exploration of all possible computations in the NFA-λ.

The nodes of the DFA are sets of nodes from the NFA-λ. The key to the algorithm is step 2.1.1, which generates the nodes of the equivalent DFA. The set Y consists of 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 the deterministic equivalent by an arc from X to Y labeled a. The paths in the state diagram of an NFA-λ M are used to construct the state diagram of an equivalent DFA DM (deterministic equivalent of M).

The nodes of DM are generated in the set Q'. The start node is lambda closure of the start node of the original NFA-λ.

Algorithm 6.6.3
Construction of DM, a DFA Equivalent to NFA-λ M

input: an NFA-λ M = (Q, Σ, δ, 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
   else done := *true*
   until done
3. the set of accepting states of DM is $F' = \{X \in Q' \mid X \text{ contains an element } q_i \in F\}$

The NFA-λ from Example 6.6.1 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 6.4(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

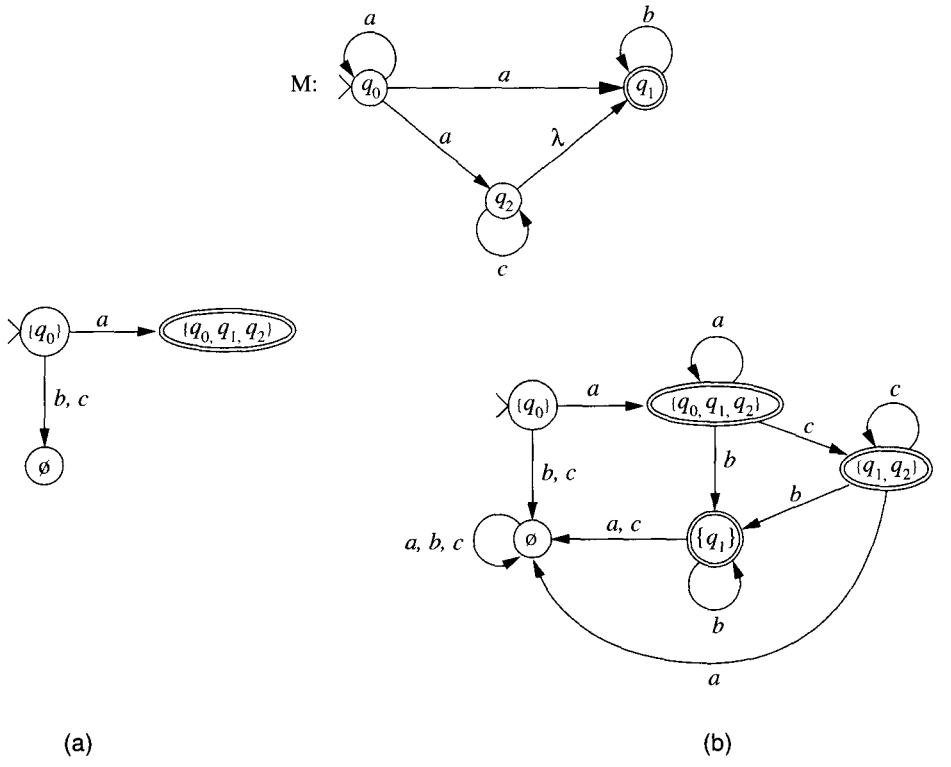


FIGURE 6.4 Construction of equivalent deterministic automaton.

$t(q_0, a)$, $t(q_1, a)$, 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.

Figure 6.4(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 $card(\mathcal{P}(Q))$ nodes can be constructed. Algorithm 6.6.3 always terminates since $card(\mathcal{P}(Q))card(\Sigma)$ is an upper bound on the number of iterations of the repeat-until loop. Theorem 6.6.4 establishes the equivalence of M and DM.

Theorem 6.6.4

Let $w \in \Sigma^*$ and $Q_w = \{q_{w_1}, q_{w_2}, \ldots, q_{w_j}\}$ 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 on length n and let $w = ua$ be a string of length $n + 1$. Let $Q_u = \{q_{u_1}, q_{u_2}, \ldots, q_{u_k}\}$ 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. ■

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.

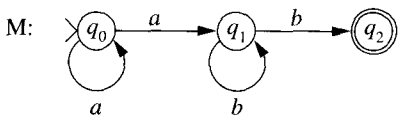
Conversely, let w be a string accepted by DM. Then Q_w contains an accepting state of M. The construction of Q_w guarantees the existence of a computation in M that processes w and terminates in that accepting state. These observations provide the justification for Corollary 6.6.5.

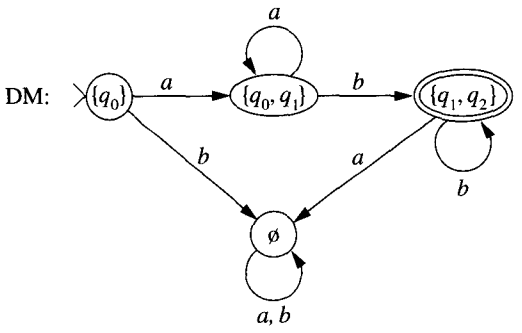
Corollary 6.6.5

The finite automata M and DM are equivalent.

Example 6.6.2

The NFA M accepts the language a^+b^+. Algorithm 6.6.3 is used to construct an equivalent DFA.

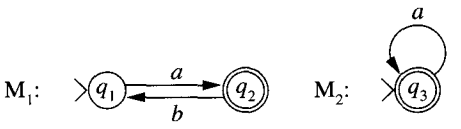




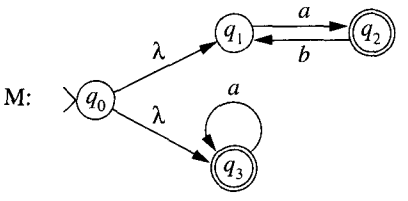
□

Example 6.6.3

The machines M_1 and M_2 accept $a(ba)^*$ and a^* respectively.



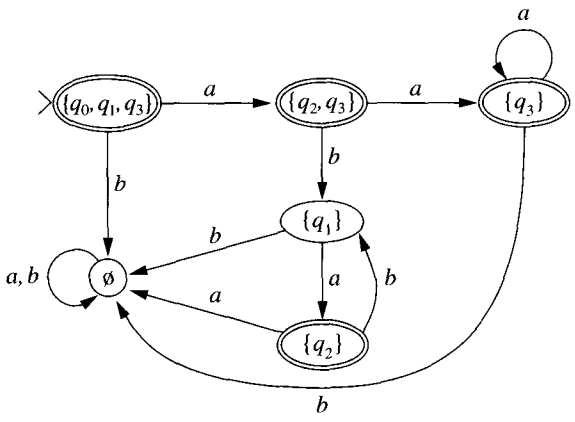
Using lambda arcs to connect a new start state to the start states of the original machines creates an NFA-λ M that accepts $a(ba)^* \cup a^*$.



The input transition function for M is

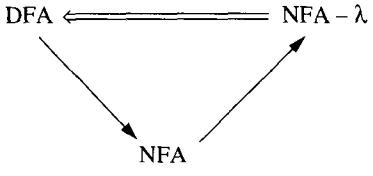
t	a	b
q_0	$\{q_2, q_3\}$	$\emptyset$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\{q_1\}$
q_3	$\{q_3\}$	$\emptyset$

The equivalent DFA obtained from Algorithm 6.6.3 is



□

Algorithm 6.6.3 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.

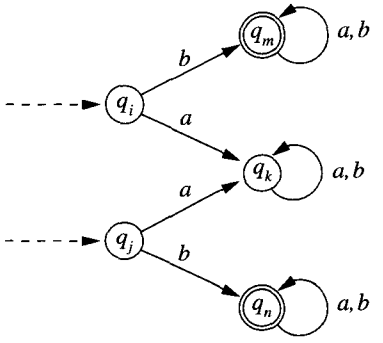
6.7 DFA Minimization

The preceding sections established that the family of languages accepted by DFAs is the same as that accepted by NFAs and NFA-λs. The flexibility of nondeterministic and lambda 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 6.6.3. 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 6.7.1
Let M = (Q, Σ, δ, 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 all $u \in \Sigma^*$.

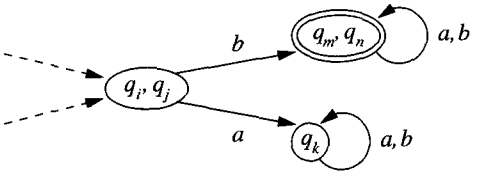
Two states that are equivalent are called *indistinguishable.* The binary relation over Q defined by indistinguishability of states is an equivalence relation, that is, the relation is reflexive, symmetric, and transitive. Two states that are not equivalent are said to be *distinquishable.* States q_i 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, since all computations leading from them produce the same membership value in L. Applying this to the preceding example yields



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, $i < j$, have associated with them 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 can be determined from that of q_m and q_n.

The algorithm to determine distinguishability uses a call to a recursive routine *DIST* to mark states as distinguishable.

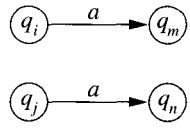
Algorithm 6.7.2
Determination of Equivalent States of DFA

input: DFA M = (Q, Σ, δ, q_0, F)

1. (Initialization)
   for every pair of states q_i and $q_j, i < j$, **do**
   1.1 $D[i, j] := 1$
   1.2 $S[i, j] := 0$
   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, with $i < j$, with $D[i, j] = 0$ **do**
   3.1 **if**, for any $a \in \Sigma$, $\delta(q_i, a) = q_m$ and $\delta(q_j, a) = q_n$ 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 algorithm to identify 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 determine the distinguishability of q_i and q_j. These ideas are formalized in the proof of Theorem 6.7.3.

Theorem 6.7.3

States q_i and q_j are distinguishable if, and only if, $D[i, j] = 1$ at the termination of Algorithm 6.7.2.

Proof First we show that every pair of states q_i and q_j for which $D[i, j] = 1$ is distinguishable. If $D[i, j]$ 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.

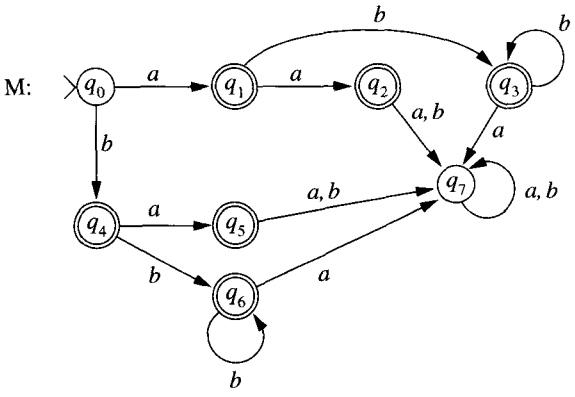
Now assume that every pair of states distinguishable by a string of length k or less is marked by the algorithm. Let q_i and q_j be states for which the shortest distinguishing string u has length $k + 1$. Then u can be written av and the computations with input u have the form $\hat{\delta}(q_i, u) = \hat{\delta}(q_i, av) = \hat{\delta}(q_m, v) = q_s$ and $\hat{\delta}(q_j, u) = \hat{\delta}(q_j, av) = \hat{\delta}(q_n, v) = q_t$. Moreover, 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 to *DIST*. If q_i and q_j are examined in the loop in step 3.1 and $D[i, j] \neq 1$ 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 by a call *DIST* with arguments m and n. $D[i, j]$ will also be set to 1 at this time by the recursive calls in *DIST* since $[i, j]$ is in $S[m, n]$. ■

A new DFA M′ can be built from the original DFA M = (Q, Σ, δ, q_0, F) and the indistinguishability relation defined above. 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 δ' of M′ is defined by $\delta'([q_i], a) = [\delta(q_i, a)]$. In Exercise 38, δ' 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.

Example 6.7.1

The minimization process is exhibited using the DFA M

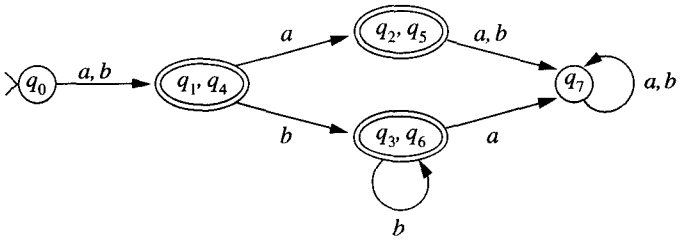


that accepts the language $(a \cup b)(a \cup b^*)$.

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]	$S[2,5] = \{[1,4]\}$	
	$S[3,6] = \{[1,4]\}$	
[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
[2,4]	$D[2,4] = 1$	distinguished by a
[2,5]		no action since $\hat{\delta}(q_2, x) = \hat{\delta}(q_5, x)$ 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

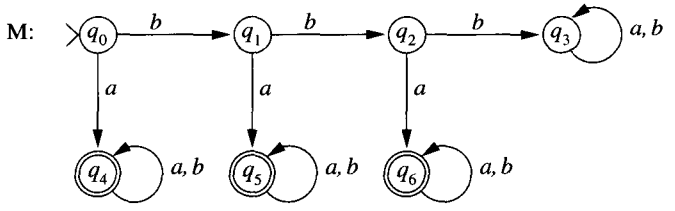
After each pair of indices is examined, [1,4], [2,5] and [3,6] are left as equivalent pairs of states. Merging these states produces the minimal state DFA M′ that accepts $(a \cup b)(a \cup b^*)$.



□

Example 6.7.2

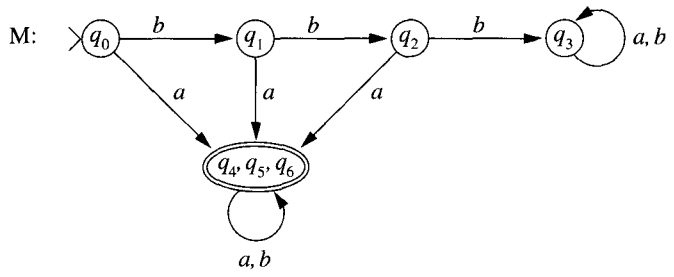
Minimizing the DFA M illustrates the recursive marking of states by the call to *DIST*. The language of M is $a(a \cup b)^* \cup ba(a \cup b)^* \cup bba(a \cup b)^*$.



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

Index	Action	Reason
[0, 1]	$S[4,5] = \{[0,1]\}$	
	$S[1,2] = \{[0,1]\}$	
[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 produces



□

The minimization algorithm completes the sequence of algorithms required for the construction of optimal DFAs. Nondeterminism and λ transitions provide tools for designing finite automata to solve complex problems or accept complex languages. Algorithm 6.6.3 can then be used to transform the nondeterministic machine into a DFA. The resulting deterministic machine need not be minimal. Algorithm 6.7.2 completes the process by producing the minimal state DFA.

Using the characterization of languages accepted by finite automata established in Section 7.7, we will prove that the resulting machine M′ is the unique minimal state DFA that accepts L.

Exercises

1. Let M be the deterministic finite automaton

$$Q = \{q_0, q_1, q_2\}$$
$$\Sigma = \{a, b\}$$
$$F = \{q_2\}$$

δ	a	b
q_0	q_0	q_1
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
      i) *abaa*
      ii) *bbbabb*
      iii) *bababa*
      iv) *bbbaa*
   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

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{a, b\}$

$F = \{q_0\}$

δ	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_0

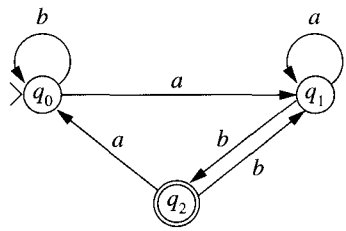
a) Give the state diagram of M.

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 whose state diagram is given below.



a) Construct the transition table of M.

b) Which of the strings *baba*, *baab*, *abab*, *abaaab* are accepted by M?

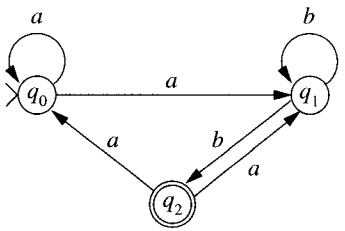
c) Give a regular expression for L(M).

4. The recursive step in the definition of the extended transition function (Definition 6.2.4) 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 $\hat{\delta} = \hat{\delta}'$.

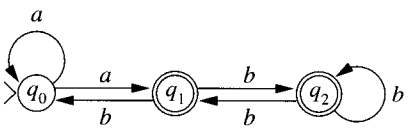
For Exercises 5 through 15, build a DFA that accepts the described language.

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 3.

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 of even length over $\{a, b, c\}$ that contain exactly one a.
15. The set of strings over $\{a, b, c\}$ with an odd number of occurrences of the substring ab.
16. 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) $(ab^*a)^*$
17. Let M be the nondeterministic finite automaton whose state diagram is given below.



    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).
18. Let M be the nondeterministic finite automaton whose state diagram is given below.



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.

19. For each of the following languages, give the state diagram of an NFA that accepts the languages.

a) $(ab)^* \cup a^*$

b) $(abc)^*a^*$

c) $(ba \cup bb)^* \cup (ab \cup aa)^*$

d) $(ab^+a)^+$

For Exercises 20 through 27 give the state diagram of an NFA that accepts the given language. Remember that an NFA may be deterministic.

20. The set of strings over {*1, 2, 3*} the sum of whose elements is divisible by six.

21. 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.

22. The set of strings over $\{a, b\}$ in which every substring of length four has at least one b.

23. The set of strings over $\{a, b, c\}$ in which every substring of length four has exactly one b.

24. The set of strings over $\{a, b\}$ whose third-to-the-last symbol is b.

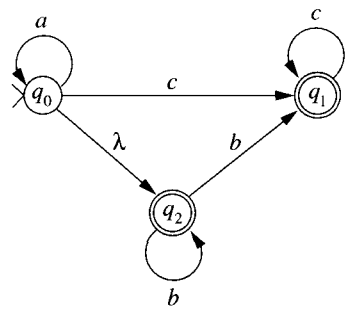
25. The set of strings over $\{a, b\}$ that contain an even number of substrings ba.

26. The set of strings over $\{a, b\}$ that have both or neither aa and bb as substrings.

27. The set of strings over $\{a, b\}$ whose third and third-to-last symbols are both b. For example, *aababaa*, *abbbbbbbb*, and *abba* are in the language.

28. 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.

29. Let M be the NFA-λ



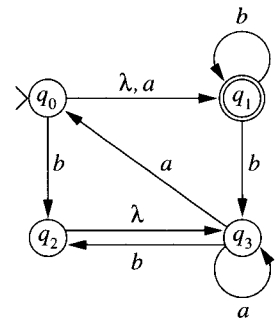
a) Compute λ-$closure(q_i)$ for $i = 0, 1, 2$.

b) Give the input transition function t for M.

c) Use Algorithm 6.6.3 to construct a state diagram of a DFA that is equivalent to M.

d) Give a regular expression for L(M).

30. Let M be the NFA-λ



a) Compute λ-$closure(q_i)$ for $i = 0, 1, 2, 3$.

b) Give the input transition function t for M.

c) Use Algorithm 6.6.3 to construct a state diagram of a DFA that is equivalent to M.

d) Give a regular expression for L(M).

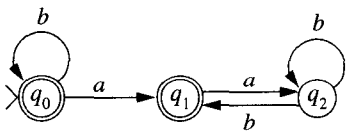
31. Give a recursive definition of the extended transition function $\hat{\delta}$ of an NFA-λ. The value $\hat{\delta}(q_i, w)$ is the set of states that can be reached by computations that begin at node q_i and completely process the string w.

32. Use Algorithm 6.6.3 to construct the state diagram of a DFA equivalent to the NFA in Example 6.5.2.

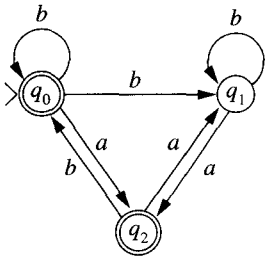
33. Use Algorithm 6.6.3 to construct the state diagram of a DFA equivalent to the NFA in Exercise 17.

34. For each of the following NFAs, use Algorithm 6.6.3 to construct the state diagram of an equivalent DFA.

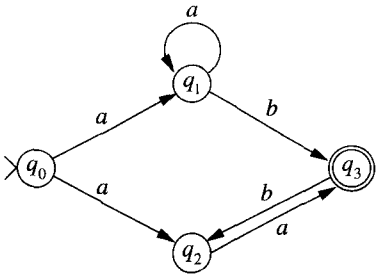
a)



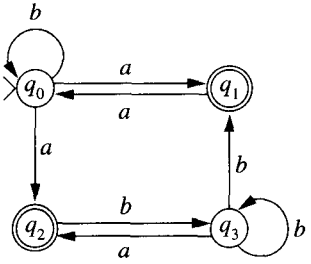
b)



c)

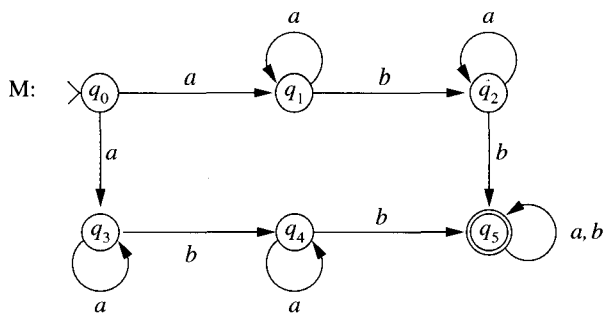


d)



35. Build an NFA M_1 that accepts $(ab)^*$ and an NFA M_2 that accepts $(ba)^*$. Use λ transitions to obtain a machine M that accepts $(ab)^*(ba)^*$. Give the input transition function of M. Use Algorithm 6.6.3 to construct the state diagram of a DFA that accepts L(M).

36. Build an NFA M_1 that accepts $(aba)^+$ and an NFA M_2 that accepts $(ab)^*$. Use λ transitions to obtain a machine M that accepts $(aba)^+ \cup (ab)^*$. Give the input transition function of M. Use Algorithm 6.6.3 to construct the state diagram of a DFA that accepts L(M).
37. Assume that q_i and q_j are equivalent states (as in Definition 6.7.1) and $\hat{\delta}(q_i, u) = q_m$ and $\hat{\delta}(q_j, u) = q_n$. Prove that q_m and q_n are equivalent.
38. Show that the transition function δ' 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 $a \in \Sigma$.
39. Let M be the DFA

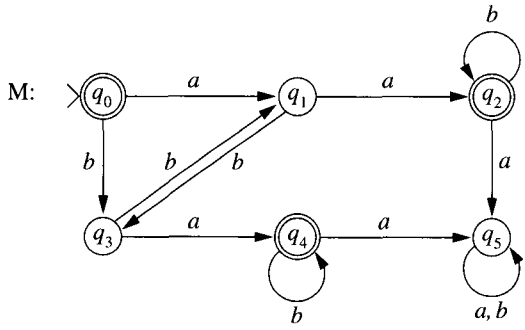


a) Trace the actions of Algorithm 6.7.2 to determine the equivalent states of M. Give the values of $D[i, j]$ and $S[i, j]$ computed by the algorithm.

b) Give the equivalence classes of states.

c) Give the state diagram of the minimal state DFA that accepts L(M).

40. Let M be the DFA



a) Trace the actions of Algorithm 6.7.2 to determine the equivalent states of M. Give the values of $D[i, j]$ and $S[i, j]$ computed by the algorithm.

b) Give the equivalence classes of states.

c) Give the state diagram of the minimal state DFA that accepts L(M).

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].

CHAPTER 7

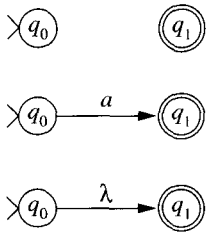
Regular Languages and Sets

Grammars were introduced as language generators and finite automata as language acceptors. This chapter develops the relationship between the generation and the acceptance of regular languages and explores the limitations of finite automata as language acceptors.

7.1 Finite Automata and Regular Sets

Every finite automaton with alphabet Σ accepts a language over Σ. We will show that the family of languages accepted by automata consists of precisely the regular sets over Σ.

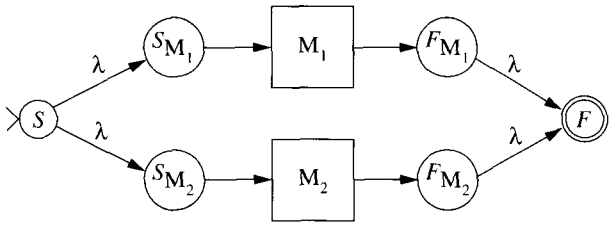
First we show that every regular set is accepted by some NFA-λ. The proof follows the recursive definition of the regular sets (Definition 2.3.1). The regular sets are built from the basis elements $\emptyset$, λ, and singleton sets containing elements from the alphabet. State diagrams for machines that accept these sets are



Note that each of these machines satisfies the restrictions described in Lemma 6.5.2. That is, the machines contain a single accepting state. Moreover, there are no arcs entering the start state or leaving the accepting state.

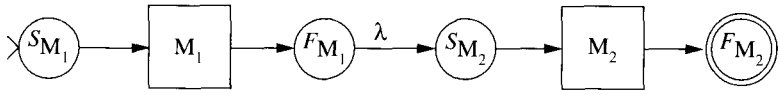
Regular sets are constructed from the basis elements using union, concatenation, and the Kleene star operation. Lambda transitions can be used to combine existing machines to construct more complex composite machines. Let M_1 and M_2 be two finite automata whose start and final states satisfy the prescribed criteria. Composite machines that accept $L(M_1) \cup L(M_2)$, $L(M_1)L(M_2)$, and $L(M_1)^*$ are constructed from M_1 and M_2. The techniques used are similar to those presented in Theorem 6.5.3. Like the machines that accept the basis elements, the composite machines will also have a start state with in-degree zero and a single accepting state with out-degree zero. Using repeated applications of these constructions, an automaton can be constructed to accept any regular set. Let S_{M_1}, F_{M_1} and S_{M_2}, F_{M_2} be the start and accepting states of M_1 and M_2, respectively. The start state and final state of the composite machine are denoted S and F.

The language $L(M_1) \cup L(M_2)$ is accepted by the machine



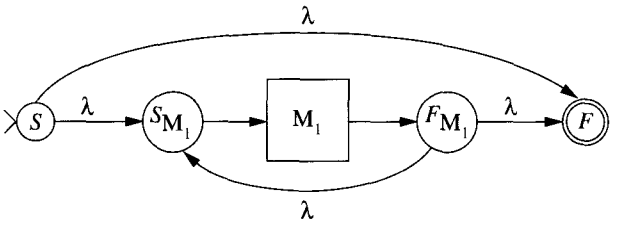
A string is processed by following a lambda arc to M_1 or M_2. If the string is accepted by either of these machines, the lambda arc can be traversed to reach the accepting state of the appropriate submachine. This construction may be thought of as building a machine that runs M_1 and M_2 in parallel. The input is accepted if either of the machines successfully processes the input.

Concatenation can be obtained by operating the component machines sequentially. The start state of the composite machine is S_{M_1} and the accepting state is F_{M_2}. The machines are joined by adding a lambda arc from the final state of M_1 to the start state of M_2.



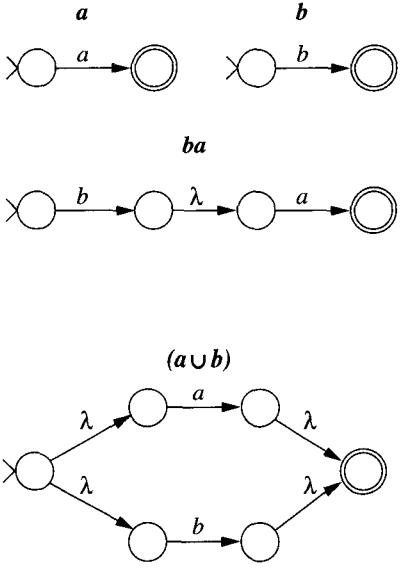
When a prefix of the input string is accepted by M_1, the lambda arc transfers the processing to M_2. If the remainder of the string is accepted by M_2, the processing terminates in F_{M_2}, the accepting state of the composite machine.

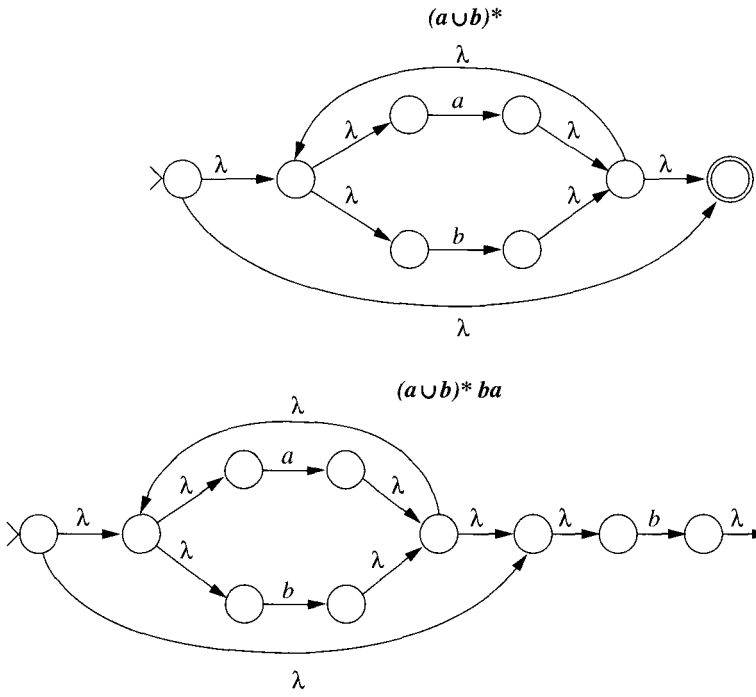
A machine that accepts $L(M_1)^*$ must be able to cycle through M_1 any number of times. The lambda arc from F_{M_1} to S_{M_1} permits the necessary cycling. Another lambda arc is added from S to F to accept the null string. These arcs are added to M_1 producing



Example 7.1.1

The techniques presented above are used to construct an NFA-λ that accepts $(a \cup b)^*ba$. The machine is built following the recursive definition of the regular expression. The language accepted by each intermediate machine is indicated by the expression above the state diagram.





□

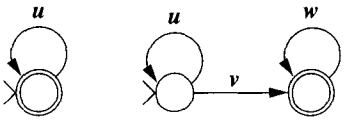
7.2 Expression Graphs

The constructions in the previous section demonstrate that every regular set is recognized by a finite automaton. We will now show that every language accepted by an automaton is a regular set. To accomplish this, we extend the notion of a state diagram.

Definition 7.2.1

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 an automaton with alphabet Σ can be considered an expression graph. The labels consist of lambda and expressions corresponding to the elements of Σ. Arcs in an expression graph can also be labeled by $\emptyset$. Paths in expression graphs generate regular expressions. The language of an expression graph is the union of the sets represented by the accepted regular expressions. The graphs



accept the regular expressions u^* and u^*vw^*.

Because of the simplicity of the graphs, the expressions accepted by the previous examples are obvious. A procedure is developed to reduce an arbitrary expression graph to one of two simple forms. This reduction is accomplished by repeatedly removing nodes from the graph in a manner that produces the expression accepted by 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. To determine the language of an automaton, the previous observation allows us to consider the accepting states separately. The algorithm to construct a regular expression from a state diagram does exactly this; it builds an expression for the set of strings accepted by each individual accepting state.

The nodes of the NFA-λ in Algorithm 7.2.2 are assumed to be numbered. The label of an arc from node i to node j is denoted $w_{i,j}$. If there is no arc from node i to j, $w_{i,j} = \emptyset$.

Algorithm 7.2.2
Construction of Regular Expression from a Finite Automaton

input: state diagram G of a finite automaton
the nodes of G are numbered $1, 2, \ldots, n$

1. Let m be the number of accepting states of G. Make m copies of G, each of which has one accepting state. Call these graphs $G_1, G_2, \ldots, G_m$. Each accepting state of G is the accepting state of some G_t, for $t = 1, 2, \ldots, m$.
2. **for** each G_t **do**
   - 2.1. **repeat**
     - 2.1.1. choose a node i in G_t that is neither the start nor the accepting node of G_t
     - 2.1.2. delete the node i from G_t according to the following procedure:
       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 j to node k labeled $w_{j,i}(w_{i,i})^*w_{i,k}$
       - iii) **if** nodes j and k have arcs labeled $w_1, w_2, \ldots, w_s$ connecting them **then** replace them by a single arc labeled $w_1 \cup w_2 \cup \cdots \cup w_s$
       - iv) remove the node i and all arcs incident to it in G_t

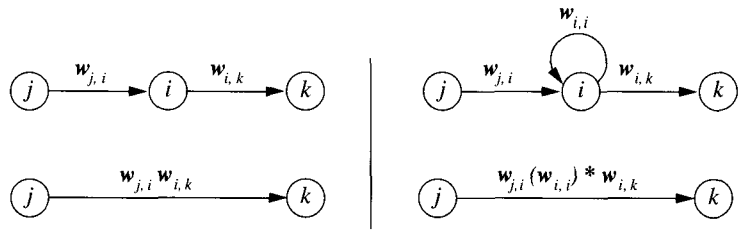
until the only nodes in G_t are the start node and the single accepting node

2.2. determine the expression accepted by G_t

end for

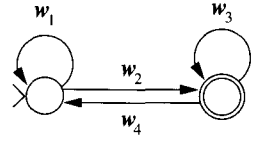
3. The regular expression accepted by G is obtained by joining the expressions for each G_t with $\cup$.

The deletion of node i is accomplished by finding all paths j, i, k of length two that have i as the intermediate node. An arc from j to k is added, bypassing the node i. If there is no arc from 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 i to k. The label for the new arc is $w_{j,i}(w_{i,i})^* w_{i,k}$. These graph transformations are illustrated below.



Step 2.2 in the algorithm may appear to be begging the question; the objective of the entire algorithm is to determine the expression accepted by G_t's. After the node deletion process is completed, the regular expression can easily be obtained from the resulting graph. The reduced graph has at most two nodes, the start node and an accepting node. If these are the same node, the reduced graph has the form

accepting w^*. A graph with distinct start and accepting nodes reduces to

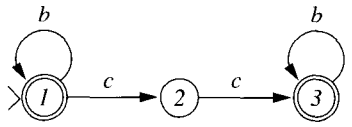


and accepts the expression $w_1{}^* w_2 (w_3 \cup w_4 (w_1)^* w_2)^*$. This expression may be simplified if any of the arcs in the graph are labeled $\emptyset$.

Example 7.2.1

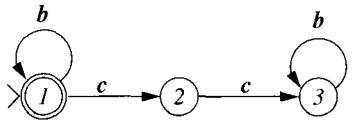
The reduction technique of Algorithm 7.2.2 is used to generate a regular expression for the language of the NFA with state diagram G.

G:

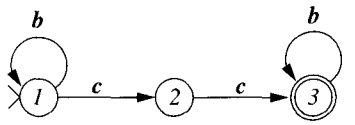


Two expression graphs, each with a single accepting node, are constructed from G.

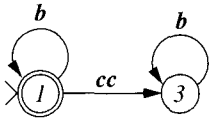
G_1:

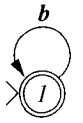


G_2:

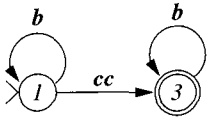


Reducing G_1 consists of deleting nodes 2 and 3 yielding





The expression accepted by G_1 is b^*. The removal of node 2 from G_2 produces



with associated expression b^*ccb^*. The expression accepted by G, built from the expressions accepted by G_1 and G_2, is $b^* \cup b^*ccb^*$. □

The results of the previous two sections yield a characterization of the regular sets originally established by Kleene. The technique presented in Section 7.1 can be used to build an NFA-λ to accept any regular set. Conversely, Algorithm 7.2.2 constructs 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 7.2.3 (Kleene)

A language L is accepted by a DFA with alphabet Σ if, and only if, L is a regular set over Σ.

7.3 Regular Grammars and Finite Automata

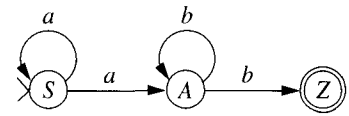
A context-free grammar is called regular (Section 3.3) if each rule is of the form $A \to aB, A \to a$, or $A \to \lambda$. A string derivable in a regular grammar contains at most one variable which, if present, occurs as the rightmost symbol. A derivation is terminated by the application of a rule of the form $A \to a$ or $A \to \lambda$. A language generated by a regular grammar is called regular.

The language a^+b^+ is generated by the grammar G and accepted by the NFA M. 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.

G: $S \to aS \mid aA$

$A \to bA \mid b$

M:



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 the generation of a terminal string in a 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 $S \to aS, S \to aA$, and $A \to bA$ 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 $A \to a$ must be incorporated into the construction of the corresponding automaton.

Theorem 7.3.1

Let G = (V, Σ, P, S) be a regular grammar. Define the NFA M = (Q, Σ, δ, S, F) as follows:

i) $Q = \begin{cases} V \cup \{Z\} & \text{where } Z \notin V, \text{ if P contains a rule } A \to a \\ V & \text{otherwise.} \end{cases}$

ii) $\delta(A, a) = B$ whenever $A \to aB \in P$
$\delta(A, a) = Z$ whenever $A \to a \in P$.

iii) $F = \begin{cases} \{A \mid A \to \lambda \in P\} \cup \{Z\} & \text{if } Z \in Q \\ \{A \mid A \to \lambda \in P\} & \text{otherwise.} \end{cases}$

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 $S \Rightarrow \lambda$, $S \stackrel{*}{\Rightarrow} wC \Rightarrow wa$, or $S \stackrel{*}{\Rightarrow} wC \Rightarrow w$ where the derivation $S \stackrel{*}{\Rightarrow} wC$ consists of the application of rules of the form $A \to aB$. If L(G) contains the null string, then S is an accepting state of M and $\lambda \in$ L(M). 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).

The derivation of a nonnull string is terminated by the application of a rule $C \to a$ or $C \to \lambda$. In a derivation of the form $S \stackrel{*}{\Rightarrow} wC \Rightarrow wa$, 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 $S \stackrel{*}{\Rightarrow} wC \Rightarrow w$ is terminated by the application of a lambda rule. Since $C \to \lambda$ 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 $S \stackrel{*}{\Rightarrow} wC$.

Conversely, let $w = ua$ be a string accepted by M. A computation accepting w has the form

$$[S, w] \stackrel{*}{\vdash} [B, \lambda], \quad \text{where } B \neq Z,$$

or

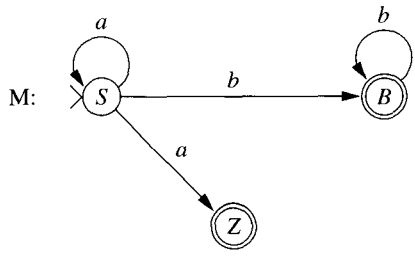
$$[S, w] \stackrel{*}{\vdash} [A, a] \vdash [Z, \lambda].$$

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 $[S, w] \stackrel{*}{\vdash} [A, a]$. The string w is obtained by terminating this derivation with the rule $A \to a$. Thus every string accepted by M is in the language of G. ■

Example 7.3.1

The grammar G generates and the NFA M accepts the language $a^*(a \cup b^+)$.

$$\text{G: } S \to aS \mid bB \mid a$$
$$B \to bB \mid \lambda$$



□

The preceding transformation can be reversed to construct a regular grammar from an NFA. The transition $\delta(A, a) = B$ produces the rule $A \to aB$. Since every transition results in a new machine state, no rules of the form $A \to a$ are produced. The rules obtained from the transitions generate derivations of the form $S \overset{*}{\Rightarrow} wC$ 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 $S \overset{*}{\Rightarrow} wC$ with the application of a rule $C \to \lambda$ generates w in G. The grammar is completed by adding lambda rules for all accepting states of the automaton.

These constructions can be applied sequentially to shift from automaton to grammar and back again. A grammar G constructed from an NFA M using the previous techniques can be transformed into an equivalent automaton M′ using the construction presented in Theorem 7.3.1:

$$\text{M} \longrightarrow \text{G} \longrightarrow \text{M}'.$$

Since G contains only rules of the form $A \to aB$ or $A \to \lambda$, 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′:

$$\text{G} \longrightarrow \text{M} \longrightarrow \text{G}'.$$

G′, the grammar 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 $Z \to \lambda$. All rules $A \to a$ are then replaced by $A \to aZ$.

Example 7.3.2

The regular grammar G′ that accepts L(M) is constructed from the automaton M from Example 7.3.1.

$$\begin{aligned} \text{G}'\text{:}\ \ S &\rightarrow aS \mid bB \mid aZ \\ B &\rightarrow bB \mid \lambda \\ Z &\rightarrow \lambda \end{aligned}$$

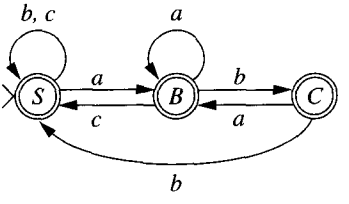
The transitions provide the S rules and the first B rule. The lambda rules are added since B and Z are accepting states. □

The two conversion techniques allow us to conclude that the languages generated by regular grammars are precisely those accepted by finite automata. It follows from Theorems 7.2.3 and 7.3.1 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 another categorization of the regular sets: the languages generated by regular grammars.

Example 7.3.3

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 7.3.1 is used to construct an NFA that accepts this language.

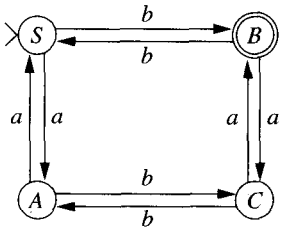
$$\begin{aligned} S &\rightarrow bS \mid cS \mid aB \mid \lambda \\ B &\rightarrow aB \mid cS \mid bC \mid \lambda \\ C &\rightarrow aB \mid bS \mid \lambda \end{aligned}$$



□

Example 7.3.4

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 6.3.4. 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

$$S \rightarrow aA \mid bB$$
$$A \rightarrow aS \mid bC$$
$$B \rightarrow bS \mid aC \mid \lambda$$
$$C \rightarrow aB \mid bA.$$

□

7.4 Closure Properties of Regular Languages

Regular languages have been defined, generated, and accepted. A language over an alphabet Σ is regular if it is

i) a regular set (expression) over Σ

ii) accepted by DFA, NFA, or NFA-λ

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 in determining 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 6.5.3 using acceptance by finite-state machines.

Theorem 7.4.1

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 Σ, then so is $\overline{L} = \Sigma^* - L$, the set containing all strings in Σ^* that are not in L. Theorem 6.3.3 uses the properties of DFAs to construct a machine that accepts $\overline{L}$ from one that accepts L. Complementation and intersection combine to establish the closure of regular languages under intersection.

Theorem 7.4.2
Let L be a regular language over Σ. The language $\overline{L}$ is regular.

Theorem 7.4.3
Let L_1 and L_2 be regular languages over Σ. The language $L_1 \cap L_2$ is regular.

Proof By DeMorgan's law

$$L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}.$$

The right-hand side of the equality is regular since it is built from L_1 and L_2 using union and complementation. ■

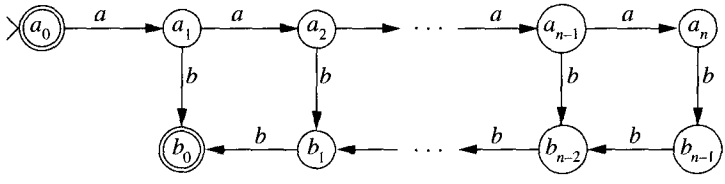
Closure properties provide additional tools for establishing the regularity of languages. The operations of complementation and intersection, as well as union, concatenation, and Kleene star, can be used to combine regular languages.

Example 7.4.1

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, $L = L_1 \cap \overline{L_2}$ is regular. □

7.5 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 b_i. The accepting state b_0 is entered when an equal number of b's are processed. We will prove that any deterministic automaton built to accept the language $L = \{a^i b^i \mid i \geq 0\}$ must have an infinite sequence of states that have the same role as the a_i's in the previous diagram.

Assume L is accepted by a DFA 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 $i \neq j, a^i b^i \in \mathrm{L}$ and $a^j b^i \notin \mathrm{L}$. Hence, $\hat{\delta}(q_0, a^i b^i) \neq \hat{\delta}(q_0, a^j b^i)$ since the former is an accepting state and the latter rejecting. Now $\hat{\delta}(q_0, a^i b^i) = \hat{\delta}(\hat{\delta}(q_0, a^i), b^i) = \hat{\delta}(a_i, b^i)$ and $\hat{\delta}(q_0, a^j b^i) = \hat{\delta}(\hat{\delta}(q_0, a^j), b^i) = \hat{\delta}(a_j, b^i)$. We conclude that $a_i \neq a_j$ since $\hat{\delta}(a_i, b^i) \neq \hat{\delta}(a_j, b^i)$.

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 $a_0, a_1, a_2, \ldots$. 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 establishes Theorem 7.5.1.

Theorem 7.5.1

The language $\{a^i b^i \mid i \geq 0\}$ is not regular.

The argument establishing Theorem 7.5.1 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 7.5.1 can be generalized to establish the nonregularity of a number of languages.

Corollary 7.5.2 (to the proof of Theorem 7.5.1)

Let L be a language over Σ. If there are strings $u_i \in \Sigma^*$ and $v_i \in \Sigma^*$, $i \geq 0$, with $u_i v_i \in \mathrm{L}$ and $u_i v_j \notin \mathrm{L}$ for $i \neq j$, then L is not a regular language.

The proof is identical to Theorem 7.5.1 with u_i replacing a^i and v_i replacing b^i.

Example 7.5.1

The set L consisting of the palindromes over $\{a, b\}$ is not regular. By Corollary 7.5.2, it is sufficient to discover two sequences of strings u_i and v_i that satisfy $u_i v_i \in \mathrm{L}$ and $u_i v_j \notin \mathrm{L}$ for all $i \neq j$. The strings

$$u_i = a^i b$$

$$v_i = a^i$$

fulfill these requirements. □

Example 7.5.2

Grammars were introduced as a formal structure for defining the syntax of languages. Corollary 7.5.2 can be used to show that regular grammars are not a powerful enough tool to define programming languages containing arithmetic or Boolean expressions in

infix form. The grammar AE (Section 4.3), which generates infix additive expressions, demonstrates the nonregularity of these languages.

$$\begin{aligned} \text{AE: } & S \rightarrow A \\ & A \rightarrow T \mid A + T \\ & T \rightarrow b \mid (A) \end{aligned}$$

Infix notation permits—in fact, requires—the nesting of parentheses. The grammar generates strings containing an equal number of left and right parentheses surrounding a correctly formed expression. 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), $((b))$, $(((b)))$, ... The strings $(^i b$ and $)^i$ satisfy the requirements of the sequences u_i and v_i of Corollary 7.5.2. Thus the language defined by the grammar AE is not regular. A similar argument can be used to show that programming languages such as Pascal, C, ADA, and ALGOL, among others, are not regular. □

Theorem 7.4.3 established a positive closure result for the operation intersection; the intersection of two regular languages produces a regular language. Theorem 7.5.3 establishes a negative result; the intersection of a regular language with a context-free language need not be regular.

Theorem 7.5.3

Let L_1 be a regular language and L_2 be a context-free language. The language $L_1 \cap L_2$ is not necessarily regular.

Proof Let $L_1 = a^*b^*$ and $L_2 = \{a^i b^i \mid i \geq 0\}$. L_2 is context-free since it is generated by the grammar $S \rightarrow aSb \mid \lambda$. The intersection of L_1 and L_2 is L_2, which is not regular by Theorem 7.5.1. ■

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.

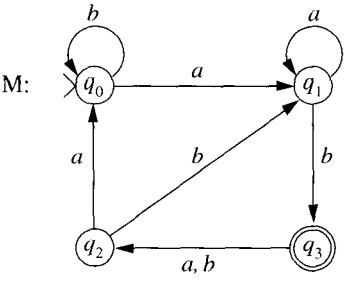
Example 7.5.3

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 7.4.2 and 7.4.3, so is $\overline{L} \cap a^*b^*$. But $\overline{L} \cap a^*b^* = \{a^i b^i \mid i \geq 0\}$, which we know is not regular. □

7.6 The Pumping Lemma for Regular Languages

The existence of nonregular sets was established in the previous section. The proof consisted of demonstrating the impossibility of constructing a DFA that accepts 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 $uv^i w$ 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 when there are 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 7.6.1

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 7.6.1 guarantees the existence of a cycle in the initial subpath. The preceding remarks are formalized in Corollary 7.6.2.

Corollary 7.6.2

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 $\mathbf{p} = \mathbf{qrs}$, the length of **qr** is less than or equal to k, and **r** is a cycle.

Theorem 7.6.3 (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 $length(z) \geq k$. Then z can be written uvw with $length(uv) \leq k$, $length(v) > 0$, 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 7.6.2, 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**. ■

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 $i \geq 0$.

Example 7.6.1

Let L = $\{z \in \{a, b\}^* \mid length(z)$ 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 $length(uv) \leq k$, $v \neq \lambda$, and $uv^iw \in$ L for all $i \geq 0$.

Consider any string z of length k^2. The pumping lemma requires a decomposition of z into substrings u, v, and w with $0 < length(v) \leq k$. This observation can be used to place an upper bound on the length of uv^2w:

$$\begin{aligned} length(uv^2w) &= length(uvw) + length(v) \\ &= k^2 + length(v) \\ &\leq k^2 + k \\ &< k^2 + 2k + 1 \\ &= (k+1)^2. \end{aligned}$$

The length of uv^2w is greater than k^2 and less than $(k+1)^2$ and therefore is not a perfect square. We have shown that there is no possible 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. □

Example 7.6.2

The language $L = \{a^i \mid i \text{ is prime}\}$ is not regular. Assume that a DFA with k states accepts L. Let n be a prime greater than k. If L is regular, 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 such a decomposition exists.

The length of $z = uv^{n+1}w$ must be prime if z is in L. But

$$\begin{aligned} length(uv^{n+1}w) &= length(uvv^nw) \\ &= length(uvw) + length(v^n) \\ &= n + n(length(v)) \\ &= n(1 + length(v)). \end{aligned}$$

Since its length is not prime, $uv^{n+1}w$ is not in L. Thus there is no division of a^n that satisfies the pumping lemma and L is not regular. □

In the preceding examples, the constraints on the length of the strings in the language 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 satisifies the conditions of the pumping lemma. We will now present another argument, this time using the pumping lemma, that establishes the nonregularity $\{a^ib^i \mid i \geq 0\}$.

Example 7.6.3

To show that $L = \{a^ib^i \mid i \geq 0\}$ is not regular we must find a string in L of appropriate length that has no pumpable string. Assume that L is regular and let k be the number specified by the pumping lemma. Let z be the string a^kb^k. Any decomposition of uvw

of z satisfying the conditions of the pumping lemma must have the form

$$\begin{array}{lll} u & v & w \\ a^i & a^j & a^{k-i-j}b^k, \end{array}$$

where $i + j \leq k$ and $j > 0$. Pumping any substring of this form produces $uv^2w = a^ia^ja^ja^{k-i-j}b^k = a^ka^jb^k$, 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. □

Example 7.6.4

The language L = $\{a^ib^jc^j \mid i > 0,\ j > 0\}$ 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 $length(uv) \leq k$, $length(v) > 0$, and $uv^iw \in$ L for all $i \geq 0$.

Consider the string $z = ab^kc^k$, 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

$$\begin{array}{lll} u & v & w \\ ab^i & b^j & b^{k-i-j}c^k \end{array}$$

where $i + j \leq k - 1$ and $j > 0$. Pumping v produces $uv^2w = ab^ib^jb^jb^{k-i-j}c^k = ab^kb^jc^k$, which is not in L.

Case 2: A decomposition of z in which $a \in v$ has the form

$$\begin{array}{lll} u & v & w \\ \lambda & ab^i & b^{k-i}c^k \end{array}$$

where $i \leq k - 1$. Pumping v produces $uv^2w = ab^iab^ib^{k-i}c^k = ab^iab^kc^k$, which is not in L.

Since z has no "pumpable" decomposition, L is not regular. □

The pumping lemma can be used to determine the size of the language accepted by a DFA. Pumping a string generates an infinite sequence of strings that are accepted by the DFA. To determine whether a regular language is finite or infinite it is necessary only to determine if it contains a pumpable string.

Theorem 7.6.4

Let M be a DFA with k states.

i) L(M) is not empty if, and only if, M accepts a string z with $length(z) < k$.

ii) L(M) has an infinite number of members if, and only if, M accepts a string z where $k \leq length(z) < 2k$.

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, $length(z) < k$.

ii) If M accepts a string z with $k \leq length(z) < 2k$, 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 $k \leq length(z) < 2k$, there is nothing left to show. Assume that $length(z) \geq 2k$. By the pumping lemma, $z = uvw, length(v) \leq k$, 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 7.6.4, 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 7.6.5

Let M be a DFA. There is a decision procedure to determine whether

i) L(M) is empty

ii) L(M) is finite

iii) L(M) is infinite.

The closure properties of regular language can be combined with Corollary 7.6.5 to develop a decision procedure that determines whether two DFAs accept the same language.

Corollary 7.6.6

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 7.4.1, 7.4.2, and 7.4.3, 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 7.6.5, there is a decision procedure to determine whether L is empty, or equivalently, whether M_1 and M_2 accept the same language. ■

7.7 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 language.

Definition 7.7.1

Let L be a language over Σ. Strings $u, v \in \Sigma^*$ are indistinguishable in L if, for every $w \in \Sigma^*$, either both uw and vw are 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 is sufficient to produce strings with different membership values in the language L.

Indistinguishability in a language L defines a binary relation $\equiv_L$ on Σ^*. It is easy to see that $\equiv_L$ is reflexive, symmetric, and transitive. These observations provide the basis for Lemma 7.7.1.

Lemma 7.7.2

For any language L, the relation $\equiv_L$ is an equivalence relation.

Example 7.7.1

Let L be the regular language $a(a \cup b)(bb)^*$. Strings aa and ab are indistinguishable since aaw and abw are in L if, and only if, w consists of an even number of b's. 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
$[\lambda]_L$	λ
$[b]_L$	$b(a \cup b)^* \cup a(a \cup b)a(a \cup b)^* \cup a(a \cup b)ba(a \cup b)^*$
$[a]_L$	a
$[aa]_L$	$a(a \cup b)(bb)^*a$
$[aab]_L$	$a(a \cup b)b(bb)^*$

□

Example 7.7.2

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. This example shows that the indistinguishability relation $\equiv_L$ may generate infinitely many equivalence classes. □

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.

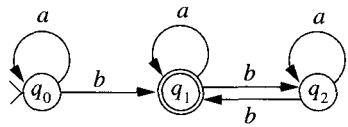
Definition 7.7.3

Let M = (Q, Σ, δ, 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 computations of M with input u and input v halt in the same state. It is easy to see that indistinguishability defined in this manner is also an equivalence relation over Σ^*. 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 7.7.3

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 Σ^* defined by the relation $\equiv_M$ are

State	Associated Equivalence Class
q_0	a^*
q_1	$a^*ba^*(ba^*ba^*)^*$
q_2	$a^*ba^*ba^*(ba^*ba^*)^*$

□

The 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 ≡ over Σ^* is said to be *right-invariant* if $u \equiv v$ implies $uw \equiv vw$ for every $w \in \Sigma^*$.

Theorem 7.7.4 (Myhill-Nerode)
The following are equivalent:

i) L is regular over Σ.

ii) There is a right-invariant equivalence relation $\equiv$ on Σ^* with finitely many equivalence classes such that L is the union of a subset of the equivalences 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 $\equiv_M$ satifies the conditions of (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 $\equiv_M$. Corresponding to each state q_i of M is an equivalence class consisting of the strings for which the computation of M halts in q_i. The language L is the union of the equivalence classes associated with the final states of M.

Condition (ii) implies condition (iii): Let $\equiv$ be an equivalence relation that satisfies (ii). We begin by showing that every equivalence class of $[u]_{\equiv}$ is a subset of $[u]_{\equiv_L}$. Let u and v be 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 same equivalence class of $\equiv$. Since L is the union of some set of equivalence classes of $\equiv$, every string in a particular 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 finitely many equivalence classes we will build a DFA M_L that accepts L. The alphabet of M_L is Σ and the states are the equivalence classes of $\equiv_L$. The start state is the equivalence class containing λ. An equivalence class is an accepting state if it contains an element $u \in L$.

For a symbol $a \in \Sigma$, define $\delta([u]_{\equiv_L}, a) = [ua]_{\equiv_L}$. We must show that the definition of the transition function is independent of the choice of a particular element from the equivalence class $[u]_{\equiv_L}$. Let u, v be two strings in $[u]_{\equiv_L}$. Since $[u]_{\equiv_L} = [v]_{\equiv_L}$, $\delta([u]_{\equiv_L}, a)$ must produce the same state as $\delta([v]_{\equiv_L}, a)$; that is, $[ua]_{\equiv_L}$ 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, $\hat{\delta}(q_0, u) = [u]_{\equiv_L}$. If u is in L, the computation $\hat{\delta}(q_0, u)$ halts in the accepting state $[u]_{\equiv_L}$. Exercise 21 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 Σ^* generated by the machine M_L. ■

Example 7.7.4

The DFA M from Example 6.7.1 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	λ	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 classes of the relation $\equiv_L$ group strings on the basis of indistinguishability of membership of extensions of the strings in L. The equivalence classes are

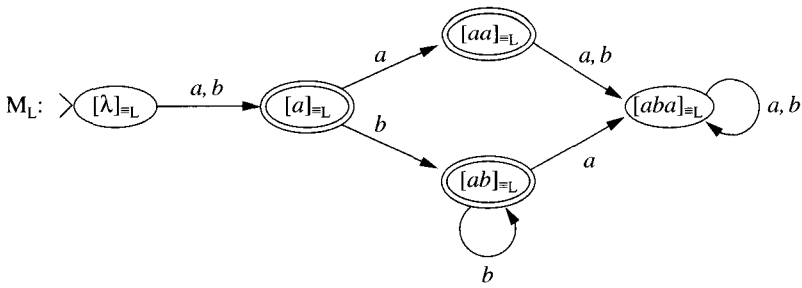
$\equiv_L$ Equivalence Classes	
$[\lambda]_{\equiv_L}$	λ
$[a]_{\equiv_L}$	$a \cup b$
$[aa]_{\equiv_L}$	$aa \cup ba$
$[ab]_{\equiv_L}$	$ab^+ \cup bb^+$
$[aba]_{\equiv_L}$	$(aa(a \cup b) \cup ab^+a \cup ba(a \cup b) \cup bb^+a)(a \cup b)^*$

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

$$
\begin{aligned}
[\lambda]_{\equiv_L} &= cl_M(q_0) \\
[a]_{\equiv_L} &= cl_M(q_1) \cup cl_M(q_4) \\
[aa]_{\equiv_L} &= cl_M(q_2) \cup cl_M(q_5) \\
[ab]_{\equiv_L} &= cl_M(q_3) \cup cl_M(q_6) \\
[aba]_{\equiv_L} &= cl_M(q_7).
\end{aligned}
$$

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 6.7.1 obtained using the minimization technique presented in Section 6.7. □

Theorem 7.7.5 shows that the DFA M_L is the minimal state DFA that accepts L.

Theorem 7.7.5

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 7.7.4.

Proof Let $M = (Q, \Sigma, \delta, q_0, F)$ be any DFA that accepts L and $\equiv_M$ the equivalence relation generated by M. By the Myhill-Nerode theorem, the equivalence classes of $\equiv_M$ are subsets of the equivalence classes of $\equiv_L$. Since the equivalence classes of both $\equiv_M$ and $\equiv_L$ partition Σ^*, $\equiv_M$ 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

$$\begin{aligned}
&\text{the number of states of } M \geq \\
&\text{the number of equivalence classes of } \equiv_M \geq \\
&\text{the number of equivalence classes of } \equiv_L = \\
&\text{the number of states of } M_L.
\end{aligned}$$

■

The statement of Theorem 7.7.5 asserts that the M_L is *the* minimal state DFA that accepts L. Exercise 27 establishes that all minimal state DFAs accepting L are identical to M_L except possibly for the names assigned to the states.

Theorems 7.7.4 and 7.7.5 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 7.7.6 shows that

the machine whose nodes are the $\equiv_L$ equivalence classes is the machine produced by the minimization algorithm in Section 6.7.

Theorem 7.7.6

Let M be a DFA that accepts L and M′ the machine obtained from M by minimization construction in Section 6.7. Then $M' = M_L$.

Proof By Theorem 7.7.5 and Exercise 27, 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 7.7.3, 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 extablishing the nonregularity of a language. A language L is not regular if the equivalence relation $\equiv_L$ has infinitely many equivalence classes.

Example 7.7.5

In Example 7.7.2, 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. □

Example 7.7.6

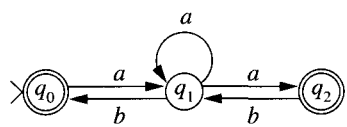
The Myhill-Nerode theorem will be used to show that the language $L = \{a^{2^i} \mid i \geq 0\}$ is not regular. To accomplish this, we show that a^{2^i} and a^{2^j} are distinguishable by L whenever $i < j$. Concatenating a^{2^i} with each of these strings produces $a^{2^i}a^{2^i} = a^{2^{i+1}} \in L$ 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 $[a^0] \equiv_L$, $[a^1] \equiv_L$, $[a^2] \equiv_L$, $[a^4] \equiv_L$, . . . of distinct equivalence classes of L. □

Exercises

1. Use the technique from Section 7.2 to build the state diagram of an NFA-λ that accepts the language $(ab)^*ba$. Compare this with the DFA constructed in Exercise 6.16 (a).
2. For each of the state diagrams in Exercise 6.34, use Algorithm 7.2.2 to construct a regular expression for the language accepted by the automaton.
3. The language of the DFA M in Example 6.3.4 consists of all strings over $\{a, b\}$ with an even number of a's and an odd number of b's. Use Algorithm 7.2.2 to construct a regular expression for L(M). Exercise 2.37 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

$$\begin{aligned} \text{G: } S &\to aS \mid bA \mid a \\ A &\to aS \mid bA \mid b. \end{aligned}$$

   a) Use Theorem 7.3.1 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 7.3.1. Prove that if $S \overset{*}{\Rightarrow} wC$ then there is a computation $[S, w] \vdash^* [C, \lambda]$ in M.
7. Let L be a regular language over $\{a, b, c\}$. Show that each of the following sets is regular.
   a) $\{w \mid w \in \text{L and } w \text{ contains an } a\}$
   b) $\{w \mid w \in \text{L or } w \text{ contains an } a\}$
   c) $\{w \mid w \notin \text{L and } w \text{ does not contain an } a\}$
8. Let L be a regular language. Show that the following languages are regular.
   a) The set $\text{P} = \{u \mid uv \in \text{L}\}$ of prefixes of L

b) The set $S = \{v \mid uv \in L\}$ of suffixes of L

c) The set $L^R = \{w^R \mid w \in L\}$ of reversals of L

d) The set $E = \{uv \mid v \in L\}$ of strings that have a suffix in L

9. Let L be a regular language containing only strings of even length. Let L' be the language $\{u \mid uv \in L \text{ and } length(u) = length(v)\}$. L' is the set of all strings that contain the first half of strings from L. Prove that L' is regular.

10. Use Corollary 7.5.2 to show that each of the following sets is not regular.

a) The set of strings over $\{a, b\}$ with the same number of a's and b's.

b) The set of palindromes of even length over $\{a, b\}$.

c) The set of strings over $\{(,)\}$ in which the parentheses are paired. Examples include $\lambda, (), ()(), (())()$.

d) The language $\{a^i(ab)^j(ca)^{2i} \mid i, j > 0\}$.

11. Use the pumping lemma to show that each of the following sets is not regular.

a) The set of palindromes over $\{a, b\}$

b) $\{a^n b^m \mid n < m\}$

c) $\{a^i b^j c^{2j} \mid i \geq 0,\ j \geq 0\}$

d) $\{ww \mid w \in \{a, b\}^*\}$

e) The set of initial sequences of the infinite string

$$abaabaaabaaaab \ldots ba^n ba^{n+1} b \ldots$$

f) The set of strings over $\{a, b\}$ in which the number of a's is a perfect cube

12. Prove that the set of nonpalindromes over $\{a, b\}$ is not a regular language.

13. 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.

14. Give examples of languages L_1 and L_2 over $\{a, b\}$ that satisfy the descriptions below.

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.

15. Let Σ_1 and Σ_2 be two alphabets. A **homomorphism** is a total function h from Σ_1^* to Σ_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 Σ_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.

16. A context-free grammar $G = (V, \Sigma, P, S)$ is called **right-linear** if each rule is the form

    i) $A \rightarrow u$

    ii) $A \rightarrow uB$,

    where $A, B \in V$, and $u \in \Sigma^*$. Show that the right-linear grammars generate precisely the regular sets.

17. 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) $A \rightarrow a$

    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.

18. A context-free grammar $G = (V, \Sigma, P, S)$ is called **left-linear** if each rule is of the form

    i) $A \rightarrow u$

    ii) $A \rightarrow Bu$,

    where $A, B \in V$, and $u \in \Sigma^*$. Show that the left-linear grammars generate precisely the regular sets.

19. Give a regular language L such that $\equiv_L$ has exactly three equivalence classes.

20. Give the $\equiv_L$ equivalence classes of the language a^+b^+.

21. 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.

22. Let $u \equiv_L v$. Prove that $ux \equiv_L vx$ for any $x \in \Sigma^*$ where Σ is the alphabet of the language L.

23. Use the Myhill-Nerode theorem to prove that the language $\{a^i \mid i$ is a perfect square $\}$ is not regular.

24. 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 7.7.4. Show that u and v are distinguishable.
25. Give the equivalence classes defined by the relation $\equiv_M$ for the DFA in Example 6.3.1.
26. Give the equivalence classes defined by the relation $\equiv_M$ for the DFA in Example 6.3.3.
27. Let M_L be the minimal state DFA that accepts a language L defined in Theorems 7.7.4 and 7.7.5. 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 7.2 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 15) 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 number of equivalence classes of a language and regularity was established in Myhill [1957] and Nerode [1958].

CHAPTER 8

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.

8.1 Pushdown Automata

Theorem 7.5.1 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 a's in the 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 push places an element on the stack and a pop removes the top element. Definition 8.1.1 formalizes the concept of a pushdown automaton. The components Q, Σ, q_0, and F of a PDA are the same as in a finite automaton.

Definition 8.1.1

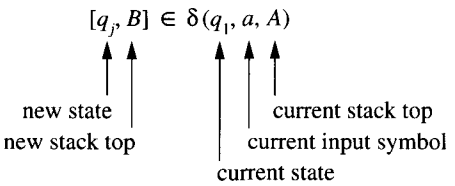
A **pushdown automaton** is a sextuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q is a finite set of states, Σ a finite set called the *input alphabet,* Γ a finite set called the *stack alphabet,* q_0 the start state, $F \subseteq Q$ a set of final states, and δ a transition function from $Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\})$ to subsets of $Q \times (\Gamma \cup \{\lambda\})$.

A PDA has two alphabets: an input alphabet Σ from which the input strings are built and a stack alphabet Γ whose elements may be stored on the stack. Elements of the stack alphabet are denoted by capital letters. A stack is represented as a string of stack elements; the element on the top of the stack is the leftmost symbol in the string. Greek letters represent strings of stack symbols. The notation $A\alpha$ represents a stack with A as the top element. An empty stack is denoted λ. The computation of a PDA begins with 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 δ 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

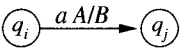
i) change the state from q_i to q_j

ii) process the symbol a (advance the tape head)

iii) remove A from the top of the stack (pop the stack)

iv) push B onto the stack.

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 replacement: A/B represents replacing A on the top of the stack by B.

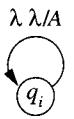
The domain of the transition function permits lambda in the input symbol and stack top positions. A lambda argument specifies that the value of the component should be neither consulted nor acted upon by the transition. The applicability of the transition is completely determined by the positions that do not contain lambda.

When a λ occurs as an argument in the stack position of the transition function, the transition is applicable whenever the current state and 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. A transition $[q_j, B] \in \delta(q_i, a, \lambda)$ will enter q_j and add B to the top of the stack whenever the machine is in state q_i scanning an a.

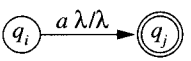
If the input position is λ, the transition does not process an input symbol. Transition (i) pops and (ii) pushes the stack symbol A without altering the state or the input.

i) $[q_i, \lambda] \in \delta(q_i, \lambda, A)$

ii) $[q_i, A] \in \delta(q_i, \lambda, \lambda)$



iii) $[q_j, \lambda] \in \delta(q_i, a, \lambda)$



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 α the stack. The notation $[q_i, w, \alpha] \vdash_M [q_j, v, \beta]$ 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, $\vdash_M^*$ 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 a's to the number of b's. The computation generated by the input string $aabb$ illustrates the actions of M.

M: $Q = \{q_0, q_1\}$

$\Sigma = \{a, b\}$

$\Gamma = \{A\}$

$F = \{q_0, q_1\}$

$\delta(q_0, a, \lambda) = \{[q_0, A]\}$

$\delta(q_0, b, A) = \{[q_1, \lambda]\}$

$\delta(q_1, b, A) = \{[q_1, \lambda]\}$

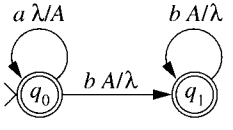
$[q_0, aabb, \lambda]$

$\vdash [q_0, abb, A]$

$\vdash [q_0, bb, AA]$

$\vdash [q_1, b, A]$

$\vdash [q_1, \lambda, \lambda]$



The computation of M processes the entire input string and halts in an accepting state with an empty stack. These conditions become our criteria for acceptance.

Definition 8.1.2

Let M = (Q, Σ, Γ, δ, q_0, F) be a PDA. A string $w \in \Sigma^*$ is **accepted** by M if there is a computation

$$[q_0, w, \lambda] \vdash^* [q_i, \lambda, \lambda]$$

where $q_i \in F$. The **language** of M, denoted L(M), is the set of strings accepted by M.

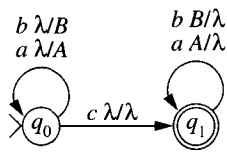
A computation that accepts a string is called *successful.* A computation that processes the entire input string and halts in a nonaccepting configuration is said to be unsuccessful. Because of the nondeterministic nature of the transition function, there may be computations that cannot complete the processing of the input string. Computations of this form are also called 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 be in the language. The existence of additional unsuccessful computations does not affect the acceptance of the string.

Example 8.1.1

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.

$$\begin{aligned} \text{M: } Q &= \{q_0, q_1\} & \delta(q_0, a, \lambda) &= \{[q_0, A]\} \\ \Sigma &= \{a, b, c\} & \delta(q_0, b, \lambda) &= \{[q_0, B]\} \\ \Gamma &= \{A, B\} & \delta(q_0, c, \lambda) &= \{[q_1, \lambda]\} \\ F &= \{q_1\} & \delta(q_1, a, A) &= \{[q_1, \lambda]\} \\ & & \delta(q_1, b, B) &= \{[q_1, \lambda]\} \end{aligned}$$



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 given below.

$$\begin{aligned} & [q_0, abcba, \lambda] \\ \vdash\ & [q_0, bcba, A] \\ \vdash\ & [q_0, cba, BA] \\ \vdash\ & [q_1, ba, BA] \\ \vdash\ & [q_1, a, A] \\ \vdash\ & [q_1, \lambda, \lambda] \end{aligned}$$

□

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 $[q_j, C] \in \delta(q_i, u, A)$, $[q_k, D] \in \delta(q_i, v, B)$ are called **compatible** if any of the following conditions are satisfied:

i) $u = v$ and $A = B$.

ii) $u = v$ and $A = \lambda$ or $B = \lambda$.

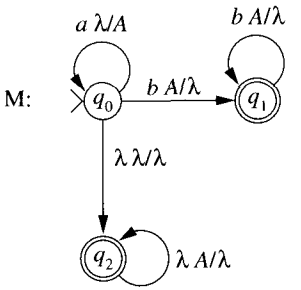
iii) $A = B$ and $u = \lambda$ or $v = \lambda$.

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 8.1.1 and the machine constructed to accept $\{a^i b^i \mid i \geq 0\}$ are deterministic.

Example 8.1.2

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.

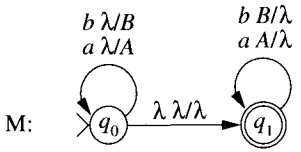


When scanning an a in state q_0, there are two transitions that are applicable. A string of the form $a^i b^i$, $i > 0$, is accepted by a computation that remains in states q_0 and q_1. If a transition to state q_2 follows the processing of the final a in a string a^i, the stack is emptied and the input 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 λ-transition from q_0 allows the machine to enter q_2 after the entire input string has been read, since a symbol is not required to process a λ-transition. This transition, which is applicable whenever the machine is in state q_0, introduces nondeterminism into the computations of M. □

Example 8.1.3

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 8.1.1, 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. □

8.2 Variations on the PDA Theme

Pushdown automata are often defined in a manner that differs slightly from Definition 8.1.1. 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

$$[q_j, \lambda] \in \delta(q_i, a, \lambda)$$
$$[q_j, \lambda] \in \delta(q_i, \lambda, A)$$
$$[q_j, A] \in \delta(q_i, \lambda, \lambda).$$

Theorem 8.2.1 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 8.2.1
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

$$[p_1, \lambda] \in \delta(q_i, a, \lambda)$$
$$\delta(p_1, \lambda, A) = \{[p_2, \lambda]\}$$
$$\delta(p_2, \lambda, \lambda) = \{[q_j, B]\}.$$

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. Removing all nonatomic 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 $[q_j, BCD] \in \delta(q_i, u, A)$ pushes BCD onto the stack with B becoming the new stack top. The apparent generalization does not increase the set of languages accepted by pushdown automaton. A PDA

containing extended transitions is called an **extended PDA**. Each extended PDA can be converted into an equivalent PDA in the sense of Definition 8.1.1.

To construct a PDA from an extended PDA, extended transitions are converted to 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 to push the elements in the correct order. The sequence of transitions

$$[p_1, D] \in \delta(q_i, u, A)$$
$$\delta(p_1, \lambda, \lambda) = \{[p_2, C]\}$$
$$\delta(p_2, \lambda, \lambda) = \{[q_j, B]\}$$

pushes the string BCD onto the stack and leaves the machine in state q_j. This produces the same result as the single extended transition $[q_j, BCD] \in \delta(q_i, u, A)$. The preceding argument yields Theorem 8.2.2.

Theorem 8.2.2

Let M be an extended PDA. Then there is a PDA M′ such that L(M′) = L(M).

Example 8.2.1

Let $L = \{a^i b^{2i} \mid i \geq 1\}$. A 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.

PDA	Atomic PDA	Extended PDA
$Q = \{q_0, q_1, q_2\}$	$Q = \{q_0, q_1, q_2, q_3, q_4\}$	$Q = \{q_0, q_1\}$
$\delta(q_0, a, \lambda) = \{[q_2, A]\}$	$\delta(q_0, a, \lambda) = \{[q_3, \lambda]\}$	$\delta(q_0, a, \lambda) = \{[q_0, AA]\}$
$\delta(q_2, \lambda, \lambda) = \{[q_0, A]\}$	$\delta(q_3, \lambda, \lambda) = \{[q_2, A]\}$	$\delta(q_0, b, A) = \{[q_1, \lambda]\}$
$\delta(q_0, b, A) = \{[q_1, \lambda]\}$	$\delta(q_2, \lambda, \lambda) = \{[q_0, A]\}$	$\delta(q_1, b, A) = \{[q_1, \lambda]\}$
$\delta(q_1, b, A) = \{[q_1, \lambda]\}$	$\delta(q_0, b, \lambda) = \{[q_4, \lambda]\}$	
	$\delta(q_4, \lambda, A) = \{[q_1, \lambda]\}$	
	$\delta(q_1, b, \lambda) = \{[q_4, \lambda]\}$	

□

By Definition 8.1.2, input 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 $[q_0, w, \lambda] \vdash^* [q_i, \lambda, \alpha]$, 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 8.2.3

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 δ' is constructed by augmenting δ with the transitions

$$\delta'(q_i, \lambda, \lambda) = \{[q_f, \lambda]\} \qquad \text{for all } q_i \in F$$

$$\delta'(q_f, \lambda, A) = \{[q_f, \lambda]\} \qquad \text{for all } A \in \Gamma.$$

Let $[q_0, w, \lambda] \overset{*}{\underset{M}{\vdash}} [q_i, \lambda, \alpha]$ be a computation accepting w by final state. In M', this computation is completed by entering the accepting state q_f and emptying the stack.

$$\begin{aligned} &[q_0, w, \lambda] \\ &\overset{*}{\underset{M}{\vdash}} [q_i, \lambda, \alpha] \\ &\underset{M'}{\vdash} [q_f, \lambda, \alpha] \\ &\overset{*}{\underset{M'}{\vdash}} [q_f, \lambda, \lambda] \end{aligned}$$

We must also guarantee that the new transitions do not cause M' to accept strings that are not in L(M). The only accepting state of M' is q_f, which can be entered from any accepting state of M. The transitions for q_f pop the stack but do not process input. Entering q_f with unprocessed input results in an unsuccessful computation. ■

A string w is said to be accepted by **empty stack** if there is a computation $[q_0, w, \lambda] \vdash [q_i, \lambda, \lambda]$. No restriction is placed on the halting state q_i. 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 8.2.4

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 $A \in \Gamma \cup \{\lambda\}$. The computations of M and M' are identical except that those of M begin in state q_0 and M' in state q_0'. Every 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 8.2.3 and 8.2.4 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 8.2.5
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 $L_E(M_2) = L$.

8.3 Pushdown Automata and Context-Free Languages

The variations of the pushdown automaton illustrate the robustness of acceptance using a stack memory. The characterization of pushdown automata as acceptors of context-free languages is established by developing a correspondence between computations in 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 $S \to \lambda$, have the form $A \to aA_1A_2 \ldots A_n$. In a leftmost derivation, the variables A_i must be processed in a left-to-right manner. Pushing $A_1A_2 \ldots A_n$ onto the stack stores the variables in the order required by the derivation.

The Greibach normal form grammar G that accepts $\{a^ib^i \mid i > 0\}$ is used to illustrate the construction of an equivalent PDA.

$$\begin{aligned} \text{G: } S &\to aAB \mid aB \\ A &\to aAB \mid aB \\ B &\to b \end{aligned}$$

The PDA has two states: a start state q_0 and an accepting state q_1. An S rule of the form $S \to aA_1A_2 \ldots A_n$ generates a transition that processes the terminal symbol a, pushes the variables $A_1A_2 \ldots A_n$ 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 transition function of the 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 obtained by processing *aaabbb* exhibits the correspondence between derivations in the Greibach normal form grammar and computations in the associated PDA.

$$
\begin{aligned}
S &\Rightarrow aAB & [q_0, aaabbb, \lambda] &\vdash [q_1, aabbb, AB] \\
&\Rightarrow aaABB & &\vdash [q_1, abbb, ABB] \\
&\Rightarrow aaaBBB & &\vdash [q_1, bbb, BBB] \\
&\Rightarrow aaabBB & &\vdash [q_1, bb, BB] \\
&\Rightarrow aaabbB & &\vdash [q_1, b, B] \\
&\Rightarrow aaabbb & &\vdash [q_1, \lambda, \lambda]
\end{aligned}
$$

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.

Theorem 8.3.1

Let L be a context-free language. Then there is a PDA that accepts L.

Proof Let G = (V, Σ, P, S) be a grammar in Greibach normal form that generates L. An extended PDA M with start state q_0 is defined by

$$
\begin{aligned}
Q_M &= \{q_0, q_1\} \\
\Sigma_M &= \Sigma \\
\Gamma_M &= V - \{S\} \\
F_M &= \{q_1\}
\end{aligned}
$$

with transitions

$$
\begin{aligned}
\delta(q_0, a, \lambda) &= \{[q_1, w] \mid S \to aw \in P\} \\
\delta(q_1, a, A) &= \{[q_1, w] \mid A \to aw \in P \text{ and } A \in V - \{S\}\} \\
\delta(q_0, \lambda, \lambda) &= \{[q_1, \lambda]\} \text{ if } S \to \lambda \in P.
\end{aligned}
$$

We first show that $L \subseteq L(M)$. Let $S \overset{*}{\Rightarrow} uw$ be a derivation with $u \in \Sigma^+$ and $w \in V^*$. We will prove that there is a computation

$$[q_0, u, \lambda] \overset{*}{\vdash} [q_1, \lambda, w]$$

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 $S \to aw$ yields the desired computation. Assume that for all strings uw generated by derivations $S \overset{n}{\Rightarrow} uw$ there is a computation

$$[q_0, u, \lambda] \overset{*}{\vdash} [q_1, \lambda, w]$$

in M.

Now let $S \overset{n+1}{\Longrightarrow} uw$ be a derivation with $u = va \in \Sigma^+$ and $w \in V^*$. This derivation can be written

$$S \overset{n}{\Rightarrow} vAw_2 \Rightarrow uw,$$

where $w = w_1w_2$ and $A \to aw_1$ 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

$$\begin{aligned}[q_0, va, \lambda] &\overset{*}{\vdash} [q_1, a, Aw_2] \\ &\vdash [q_1, \lambda, w_1w_2].\end{aligned}$$

For every string u in L of positive length, the acceptance of u is exhibited by the computation in M corresponding to the derivation $S \overset{*}{\Rightarrow} u$. If $\lambda \in$ L, then $S \to \lambda$ 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 $[q_0, u, \lambda] \overset{*}{\vdash} [q_1, \lambda, w]$ there is a corresponding derivation $S \overset{*}{\Rightarrow} uw$ in G. The proof is also by induction and is left as an exercise. ■

To complete the categorization 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. The grammar is designed so that the application of a rule corresponds to a transition in the PDA. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA. An extended PDA M′ with transition function δ' is constructed from M by augmenting δ with the transitions

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. $S \to \langle q_0, \lambda, q_j \rangle$ 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

$$\{\langle q_i, A, q_k \rangle \to x\langle q_j, B, q_k \rangle \mid q_k \in Q\}.$$

3. Each transition $[q_j, BA] \in \delta(q_i, x, A)$, where $A \in \Gamma$, generates the set of rules

$$\{\langle q_i, A, q_k \rangle \to x\langle q_j, B, q_n \rangle\langle q_n, A, q_k \rangle \mid q_k, q_n \in Q\}.$$

4. For each state $q_k \in Q$,

$$\langle q_k, \lambda, q_k \rangle \to \lambda.$$

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 8.3.1

A grammar G is constructed from the PDA M. The language of M is the set $\{a^n cb^n \mid n \geq 0\}$.

$$\begin{aligned} \text{M: } Q &= \{q_0, q_1\} & \delta(q_0, a, \lambda) &= \{[q_0, A]\} \\ \Sigma &= \{a, b, c\} & \delta(q_0, c, \lambda) &= \{[q_1, \lambda]\} \\ \Gamma &= \{A\} & \delta(q_1, b, A) &= \{[q_1, \lambda]\} \\ F &= \{q_1\} \end{aligned}$$

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 grammar G are given in Table 8.3.1, preceded by the transition from which they were constructed. □

The relationship between computations in a PDA and derivations in the associated grammar are demonstrated using the grammar and PDA of Example 8.3.1. 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.

$$\begin{aligned} & [q_0, aacbb, \lambda] && S \Rightarrow \langle q_0, \lambda, q_1 \rangle \\ \vdash\ & [q_0, acbb, A] && \Rightarrow a\langle q_0, A, q_1 \rangle \\ \vdash\ & [q_0, cbb, AA] && \Rightarrow aa\langle q_0, A, q_1 \rangle\langle q_1, A, q_1 \rangle \\ \vdash\ & [q_1, bb, AA] && \Rightarrow aac\langle q_1, A, q_1 \rangle\langle q_1, A, q_1 \rangle \\ \vdash\ & [q_1, b, A] && \Rightarrow aacb\langle q_1, \lambda, q_1 \rangle\langle q_1, A, q_1 \rangle \\ & && \Rightarrow aacb\langle q_1, A, q_1 \rangle \\ \vdash\ & [q_1, \lambda, \lambda] && \Rightarrow aacbb\langle q_1, \lambda, q_1 \rangle \\ & && \Rightarrow aacbb \end{aligned}$$

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

TABLE 8.3.1 Rules of G Constructed from Transitions

Transition	Rule
	$S \to \langle q_0, \lambda, q_1 \rangle$
$\delta(q_0, a, \lambda) = \{[q_0, A]\}$	$\langle q_0, \lambda, q_0 \rangle \to a\langle q_0, A, q_0 \rangle$
	$\langle q_0, \lambda, q_1 \rangle \to a\langle q_0, A, q_1 \rangle$
$\delta(q_0, a, A) = \{[q_0, AA]\}$	$\langle q_0, A, q_0 \rangle \to a\langle q_0, A, q_0 \rangle\langle q_0, A, q_0 \rangle$
	$\langle q_0, A, q_1 \rangle \to a\langle q_0, A, q_0 \rangle\langle q_0, A, q_1 \rangle$
	$\langle q_0, A, q_0 \rangle \to a\langle q_0, A, q_1 \rangle\langle q_1, A, q_0 \rangle$
	$\langle q_0, A, q_1 \rangle \to a\langle q_0, A, q_1 \rangle\langle q_1, A, q_1 \rangle$
$\delta(q_0, c, \lambda) = \{[q_1, \lambda]\}$	$\langle q_0, \lambda, q_0 \rangle \to c\langle q_1, \lambda, q_0 \rangle$
	$\langle q_0, \lambda, q_1 \rangle \to c\langle q_1, \lambda, q_1 \rangle$
$\delta(q_0, c, A) = \{[q_1, A]\}$	$\langle q_0, A, q_0 \rangle \to c\langle q_1, A, q_0 \rangle$
	$\langle q_0, A, q_1 \rangle \to c\langle q_1, A, q_1 \rangle$
$\delta(q_1, b, A) = \{[q_1, \lambda]\}$	$\langle q_1, A, q_0 \rangle \to b\langle q_1, \lambda, q_0 \rangle$
	$\langle q_1, A, q_1 \rangle \to b\langle q_1, \lambda, q_1 \rangle$
	$\langle q_0, \lambda, q_0 \rangle \to \lambda$
	$\langle q_1, \lambda, q_1 \rangle \to \lambda$

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 δ contains transitions that do not remove a symbol from the stack. The application of the rule $\langle q_0, A, q_1 \rangle \to c\,\langle q_1, A, q_1 \rangle$ represents a computation that processes c without removing the A from the top of the stack.

Theorem 8.3.2

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 above from the extended PDA M′. We must show that there is a derivation $S \overset{*}{\Rightarrow} w$ if, and only if, $[q_0, w, \lambda] \overset{*}{\vdash} [q_j, \lambda, \lambda]$ for some $q_j \in F$. This follows from Lemmas 8.3.3 and 8.3.4, which establish the correspondence of derivations in G to computations in M′.

Lemma 8.3.3
If $\langle q_i, A, q_j\rangle \overset{*}{\Rightarrow} w$ 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 $\langle q_i, \lambda, q_i\rangle \Rightarrow \lambda$ 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 $\langle q_i, A, q_j\rangle \overset{n}{\Rightarrow} v$. 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

$$\begin{aligned}\langle q_i, A, q_j\rangle &\Rightarrow u\langle q_k, B, q_j\rangle \\ &\overset{n}{\Rightarrow} uv = w,\end{aligned}$$

where $\langle q_i, A, q_j\rangle \to u\langle q_k, B, q_j\rangle$ 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 $\langle q_k, B, q_j\rangle \overset{n}{\Rightarrow} v$.

The rule $\langle q_i, A, q_j\rangle \to u\langle q_k, B, q_j\rangle$ 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

$$\begin{aligned}\langle q_i, A, q_j\rangle &\Rightarrow u\langle q_k, B, q_m\rangle\langle q_m, A, q_j\rangle \\ &\overset{n}{\Rightarrow} w.\end{aligned}$$

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 8.3.4
If $[q_i, w, A] \vdash^* [q_j, \lambda, \lambda]$ where $A \in \Gamma \cup \{\lambda\}$, then there is a derivation $\langle q_i, A, q_j\rangle \overset{*}{\Rightarrow} w$.

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 $\langle q_i, \lambda, q_i\rangle \to \lambda$.

Assume that every computation $[q_i, v, A] \vdash^n [q_j, \lambda, \lambda]$ has a corresponding derivation $\langle q_i, A, q_j\rangle \overset{*}{\Rightarrow} v$ in G. Consider a computation of length $n + 1$. A computation of the prescribed form beginning with a nonextended transition can be written

$$
\begin{aligned}
&[q_i, w, A] \\
&\vdash [q_k, v, B] \\
&\overset{n}{\vdash} [q_j, \lambda, \lambda],
\end{aligned}
$$

where $w = uv$ and $[q_k, B] \in \delta(q_i, u, A)$. By the inductive hypothesis there is a derivation $\langle q_k, B, q_j \rangle \overset{*}{\Rightarrow} v$. 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

$$
\begin{aligned}
\langle q_i, A, q_j \rangle &\Rightarrow u \langle q_k, B, q_j \rangle \\
&\overset{*}{\Rightarrow} uv.
\end{aligned}
$$

A computation in M′ beginning with an extended transition $[q_j, BA] \in \delta(q_i, u, A)$ has the form

$$
\begin{aligned}
&[q_i, w, A] \\
&\vdash [q_k, v, BA] \\
&\overset{*}{\vdash} [q_m, y, A] \\
&\overset{*}{\vdash} [q_j, \lambda, \lambda],
\end{aligned}
$$

where $w = uv$ and $v = xy$. The rule $\langle q_i, A, q_j \rangle \rightarrow u \langle q_k, B, q_m \rangle \langle q_m, A, q_j \rangle$ is generated by the first transition of the computation. By the inductive hypothesis, G contains derivations

$$
\begin{aligned}
\langle q_k, B, q_m \rangle &\overset{*}{\Rightarrow} x \\
\langle q_m, A, q_j \rangle &\overset{*}{\Rightarrow} y.
\end{aligned}
$$

Combining these derivations with the preceding rule produces a derivation of w from $\langle q_i, A, q_j \rangle$. ■

Proof of Theorem 8.3.2 Let w be any string in L(G) with derivation $S \Rightarrow \langle q_0, \lambda, q_j \rangle \overset{*}{\Rightarrow} w$. By Lemma 8.3.3, there is a computation $[q_0, w, \lambda] \overset{*}{\underset{M}{\vdash}} [q_j, \lambda, \lambda]$ exhibiting the acceptance of w by M′.

Conversely, if $w \in L(M) = L(M')$ then there is a computation $[q_0, w, \lambda] \overset{*}{\vdash} [q_j, \lambda, \lambda]$ that accepts w. Lemma 8.3.4 establishes the existence of a corresponding derivation $\langle q_0, \lambda, q_j \rangle \overset{*}{\Rightarrow} w$ 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. ■

8.4 The Pumping Lemma for Context-Free Languages

The pumping lemma establishes a periodicity property for strings in context-free languages. Recursive derivations in a context-free grammar have the form $A \overset{*}{\Rightarrow} uAv$. Derivation trees are used to establish conditions that ensure the presence of recursive subderivations in derivations of sufficiently long strings. Throughout this section the grammars are

assumed to be in Chomsky normal form. With this assumption, the derivation of every string in the language can be represented by a binary derivation tree.

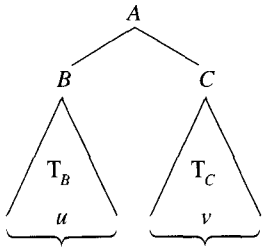
Lemma 8.4.1

Let G be a context-free grammar in Chomsky normal form and $A \stackrel{*}{\Rightarrow} w$ 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 tree. Since G is in Chomsky normal form, a derivation tree of depth one 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 as required.

Assume that the property holds for all derivation trees of depth n or less. Let $A \stackrel{*}{\Rightarrow} w$ be a derivation with tree T of depth $n + 1$. Since the grammar is in Chomsky normal form, the derivation can be written $A \Rightarrow BC \stackrel{*}{\Rightarrow} uv$ where $B \stackrel{*}{\Rightarrow} u$, $C \stackrel{*}{\Rightarrow} v$, and $w = uv$. The derivation tree of $A \stackrel{*}{\Rightarrow} w$ is constructed from T_B and T_C, the derivation trees of $B \stackrel{*}{\Rightarrow} u$ and $C \stackrel{*}{\Rightarrow} v$.



The trees T_B and T_C both have depth n or less. By the inductive hypothesis, $length(u) \leq 2^{n-1}$ and $length(v) \leq 2^{n-1}$. Therefore, $length(w) = length(uv) \leq 2^n$. ■

Corollary 8.4.2

Let G = (V, Σ, P, S) be a context-free grammar in Chomsky normal form and $S \stackrel{*}{\Rightarrow} w$ a derivation of $w \in L(G)$. If $length(w) \geq 2^n$, then the derivation tree has depth at least $n + 1$.

Theorem 8.4.3 (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 $length(z) > k$ can be written $z = uvwxy$ where

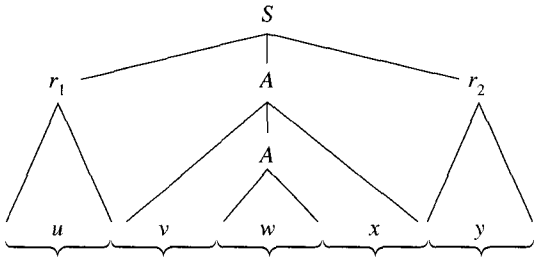
i) $length(vwx) \leq k$

ii) $length(v) + length(x) > 0$

iii) $uv^i wx^i y \in L$, for $i \geq 0$.

Proof Let $G = (V, \Sigma, P, S)$ be a Chomsky normal form grammar that generates L and let $k = 2^n$ where $n = card(V)$. 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 $S \stackrel{*}{\Rightarrow} z$ a derivation in G. By the corollary, there is a path of length at least $n + 1 = card(V) + 1$ in the derivation tree of $S \stackrel{*}{\Rightarrow} z$. 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 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.

The derivation tree can be divided into subtrees



where the nodes labeled by the variable A indicated in the diagram are the final two occurrences of A in the path **p**.

The derivation of z consists of the subderivations

1. $S \stackrel{*}{\Rightarrow} r_1 A r_2$
2. $r_1 \stackrel{*}{\Rightarrow} u$
3. $A \stackrel{+}{\Rightarrow} vAx$
4. $A \stackrel{*}{\Rightarrow} w$
5. $r_2 \stackrel{*}{\Rightarrow} y$.

Subderivation 3 may be omitted or be repeated any number of times before applying subderivation 4. The resulting derivations generate the strings $uv^i wx^i y \in L(G) = L$.

We now show that conditions (ii) and (iii) in the pumping lemma are satisfied by this decomposition. The subderivation $A \stackrel{+}{\Rightarrow} vAx$ 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

$$\begin{aligned} A &\Rightarrow BC \\ &\overset{*}{\Rightarrow} vAyC \\ &\overset{*}{\Rightarrow} vAyz \\ &= vAx. \end{aligned}$$

The string z 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 of **p** from the first occurrence of the variable A in the diagram to a leaf must be of length at most $n + 2$. Since this is the longest path in the subtree with root A, the derivation tree generated by the derivation $A \overset{*}{\Rightarrow} vwx$ has depth at most $n + 1$. It follows from Lemma 8.4.1 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.

Example 8.4.1

The language $L = \{a^i b^i c^i \mid i \geq 0\}$ is not context-free. Assume L is context-free. By Theorem 8.4.1, the string $w = a^k b^k c^k$, 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. □

Example 8.4.2

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) $vwx \in \boldsymbol{a}^*$ or $vwx \in \boldsymbol{b}^*$

ii) $vwx \in \boldsymbol{a}^*\boldsymbol{b}^*$ or $vwx \in \boldsymbol{b}^*\boldsymbol{a}^*$.

By an argument similar to that in Example 8.4.1, the substrings v and x 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. □

Example 8.4.3

The language $L = \{w \in a^* \mid length(w) \text{ is prime}\}$ is not context-free. Assume L is context-free and n a prime greater than k, the constant of Theorem 8.4.3. The string a^n must have a decomposition $uvwxy$ that satisfies the conditions of the pumping lemma. Let $m = length(u) + length(w) + length(y)$. The length of any string uv^iwx^iy is $m + i(n - m)$.

In particular, $length(wv^{n+1}wx^{n+1}y) = m + (n + 1)(n - m) = n(n - m + 1)$. Both of the terms in the preceding product are natural numbers greater than one. 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. □

8.5 Closure Properties of Context-Free Languages

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 8.5.1

The set of context-free languages is closed under the operations union, concatenation, and Kleene star.

Proof Let L_1 and L_2 be two 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.

i) Union: Define $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\}, S)$. A string w is in L(G) if, and only if, there is a derivation $S \Rightarrow S_i \overset{*}{\underset{G_i}{\Rightarrow}} w$ for $i = 1$ or 2. Thus w is in L_1 or L_2. On the other hand, any derivation $S_i \overset{*}{\underset{G_i}{\Rightarrow}} w$ can be initialized with the rule $S \to S_i$ to generate w in G.

ii) Concatenation: Define $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1S_2\}, S)$. The start symbol initiates derivations in both G_1 and G_2. A leftmost derivation of a terminal string in G has the form $S \Rightarrow S_1S_2 \overset{*}{\Rightarrow} uS_2 \overset{*}{\Rightarrow} uv$, 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 $L(G) \subseteq L_1L_2$. The

opposite inclusion is established by observing that every string w in L_1L_2 can be written uv with $u \in L_1$ and $v \in L_2$. The derivations $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} u$ and $S_2 \overset{*}{\underset{G_2}{\Rightarrow}} v$ along with the S rule of G generate w in G.

iii) Kleene star: Define $G = (V_1, \Sigma_1, P_1 \cup \{S \rightarrow S_1S \mid \lambda\}, S)$. 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 8.5.1 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 8.5.2

The set of context-free languages is not closed under intersection or complementation.

Proof

i) Intersection: Let $L_1 = \{a^ib^ic^j \mid i, j \geq 0\}$ and $L_2 = \{a^jb^ic^i \mid i, j \geq 0\}$. L_1 and L_2 are both context-free, since they are generated by G_1 and G_2, respectively.

$$\begin{array}{lll} G_1\text{:} & S \rightarrow BC & \qquad G_2\text{:} \quad S \rightarrow AB \\ & B \rightarrow aBb \mid \lambda & \qquad\quad A \rightarrow aA \mid \lambda \\ & C \rightarrow cC \mid \lambda & \qquad\quad B \rightarrow bBc \mid \lambda \end{array}$$

The intersection of L_1 and L_2 is the set $\{a^ib^ic^i \mid i \geq 0\}$, which, by Example 8.4.1, is not context-free.

ii) 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 8.5.1, 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). ■

Theorem 7.5.3 demonstrated that the intersection of a regular and context-free language need not be regular. The correspondence between languages and 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 8.5.3

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 $M = (Q_M, \Sigma_M, \Gamma, \delta_M, p_0, F_M)$ 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 while 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

$$[[p_0, q_0], w, \lambda] \vdash^* [[p_i, q_j], \lambda, \lambda],$$

where p_i and q_j are final states of M and N, respectively.

The inclusion $L(N) \cap L(M) \subseteq L(M')$ is established by showing that there is a computation

$$[[p_0, q_0], w, \lambda] \overset{*}{\underset{M'}{\vdash}} [[p_i, q_j], u, \alpha]$$

whenever

$$[p_0, w, \lambda] \overset{*}{\underset{M}{\vdash}} [p_i, u, \alpha] \quad \text{and} \quad [q_0, w] \overset{*}{\underset{N}{\vdash}} [q_j, u]$$

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

$$[p_0, w, \lambda] \overset{n+1}{\underset{M}{\vdash}} [p_i, u, \alpha] \qquad \text{and} \qquad [q_0, w] \overset{*}{\underset{N}{\vdash}} [q_j, u]$$

be computations in the PDA and DFA, respectively. The computation in M can be written

$$
\begin{aligned}
&[p_0, w, \lambda] \\
&\overset{\mu}{\vdash} [p_k, v, \beta] \\
&\vdash [p_i, u, \alpha],
\end{aligned}
$$

where either $v = u$ or $v = au$. To show that there is a computation $[[p_0, q_0], w, \lambda] \overset{*}{\underset{M}{\vdash}} [[p_i, q_j], u, \alpha]$, 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

$$
\begin{aligned}
[[p_0, q_0], w, \lambda] &\overset{\mu}{\underset{M'}{\vdash}} [[p_k, q_j], v, \beta] \\
&\underset{M'}{\vdash} [[p_i, q_j], v, \alpha]
\end{aligned}
$$

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

$$
\begin{aligned}
&[q_0, w] \\
&\overset{*}{\underset{N}{\vdash}} [q_m, v] \\
&\underset{N}{\vdash} [q_j, u],
\end{aligned}
$$

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

$$
\begin{aligned}
[[p_0, q_0], w, \lambda] &\overset{\mu}{\underset{M'}{\vdash}} [[p_k, q_m], v, \beta] \\
&\underset{M'}{\vdash} [[p_i, q_j], u, \alpha].
\end{aligned}
$$

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. ■

Example 8.5.1

Let L be the language $\{ww \mid w \in \{a, b\}^*\}$. L is not context-free but $\overline{L}$ is.

i) Assume L is context-free. Then, by Theorem 8.5.3,

$$L \cap a^*b^*a^*b^* = \{a^i b^j a^i b^j \mid i, j \geq 0\}$$

is context-free. However, this language was shown not to be context-free in Example 8.4.2, contradicting our assumption.

ii) To show that $\overline{L}$ is context-free, we construct two context-free grammars G_1 and G_2 with $L(G_1) \cup L(G_2) = \overline{L}$.

$$G_1\colon\ S \to aA \mid bA \mid a \mid b \qquad G_2\colon\ S \to AB \mid BA$$
$$A \to aS \mid bS \qquad\qquad A \to ZAZ \mid a$$
$$B \to ZBZ \mid b$$
$$Z \to a \mid b$$

The grammar G_1 generates the strings of odd length over $\{a, b\}$, all of which are in $\overline{L}$. G_2 generates the set of even length string in $\overline{L}$. 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 $length(u_1) = length(u_2)$ and $length(v_1) = length(v_2)$. Since the u's and v's are arbitrary strings in Σ^*, this characterization can be rewritten u_1xpqyv_2, where $length(p) = length(u_1)$ and $length(q) = length(v_2)$. The recursive variables of G_2 generate precisely this set of strings. □

8.6 A Two-Stack Automaton

Finite automata accept regular languages. Pushdown automata accept context-free languages. The increase in the set of languages accepted is due to the addition of a stack memory. Are two stacks, like two heads, better than one? In this section the notion of pushdown automata is extended to include machines with two stacks.

Definition 8.6.1

A **two-stack PDA** is structure (Q, Σ, Γ, δ, q_0, F). Q, Σ, Γ, q_0, and F are the same as in a one-stack PDA. The transition function maps $Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\})$ to subsets of $Q \times (\Gamma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\})$.

A transition consults the state, the input symbol, and both stack tops. The action of a transition may alter any or all of these components. Strings are accepted if a computation completely reads the input and halts in a final state with both stacks empty. Unlike the alterations to PDAs proposed in Section 8.2, an additional stack enlarges the set of languages accepted. The interplay between the stacks allows the machine to retain a copy of the stack while using the other stack to process the input.

Example 8.6.1

The two-stack PDA defined below accepts the language $L = \{a^i b^i c^i \mid i \geq 0\}$. The first stack is used to match the a's and b's and the second the b's and c's.

$$
\begin{aligned}
&Q = \{q_0, q_1, q_2\} \qquad && \delta(q_0, \lambda, \lambda, \lambda) = \{[q_2, \lambda, \lambda]\} \\
&\Sigma = \{a, b, c\} && \delta(q_0, a, \lambda, \lambda) = \{[q_0, A, \lambda]\} \\
&\Gamma = \{A\} && \delta(q_0, b, A, \lambda) = \{[q_1, \lambda, A]\} \\
&F = \{q_2\} && \delta(q_1, b, A, \lambda) = \{[q_1, \lambda, A]\} \\
& && \delta(q_1, c, \lambda, A) = \{[q_2, \lambda, \lambda]\} \\
& && \delta(q_2, c, \lambda, A) = \{[q_2, \lambda, \lambda]\}
\end{aligned}
$$

The computation that accepts $aabbcc$

$$
\begin{aligned}
& [q_0, aabbcc, \lambda, \lambda] \\
\vdash\ & [q_0, abbcc, A, \lambda] \\
\vdash\ & [q_0, bbcc, AA, \lambda] \\
\vdash\ & [q_1, bcc, A, A] \\
\vdash\ & [q_1, cc, \lambda, AA] \\
\vdash\ & [q_2, c, \lambda, A] \\
\vdash\ & [q_2, \lambda, \lambda, \lambda]
\end{aligned}
$$

illustrates the interplay between the two stacks. □

Clearly, every context-free language is accepted by a two-stack automaton. The acceptance of $\{a^i b^i c^i \mid i \geq 0\}$ by a two-stack automaton shows that the context-free languages are a proper subset of the languages accepted by two-stack PDAs.

Finite automata recognize the strings a^i, pushdown automata $a^i b^i$, and two-stack pushdown automata $a^i b^i c^i$. Is yet another stack necessary to accept $a^i b^i c^i d^i$? Example 8.6.2 illustrates a technique by which two-stack automata can be used to accept strings consisting of any number of substrings of equal length.

Example 8.6.2

The two-stack PDA M accepts the language $L = \{a^i b^i c^i d^i \mid i \geq 0\}$. The computations of M process the strings of L in the following manner:

i) Processing a pushes A onto stack 1.
ii) Processing b pops A and pushes B onto stack 2.
iii) Processing c pops B and pushes C onto stack 1.
iv) Processing d pops C.

$$
\begin{aligned}
\text{M: } Q &= \{q_0, q_1, q_2, q_3\} & \delta(q_0, \lambda, \lambda, \lambda) &= \{[q_3, \lambda, \lambda]\} \\
\Sigma &= \{a, b, c, d\} & \delta(q_0, a, \lambda, \lambda) &= \{[q_0, A, \lambda]\} \\
\Gamma &= \{A, B, C\} & \delta(q_0, b, A, \lambda) &= \{[q_1, \lambda, B]\} \\
F &= \{q_3\} & \delta(q_1, b, A, \lambda) &= \{[q_1, \lambda, B]\} \\
& & \delta(q_1, c, \lambda, B) &= \{[q_2, C, \lambda]\} \\
& & \delta(q_2, c, \lambda, B) &= \{[q_2, C, \lambda]\} \\
& & \delta(q_2, d, C, \lambda) &= \{[q_3, \lambda, \lambda]\} \\
& & \delta(q_3, d, C, \lambda) &= \{[q_3, \lambda, \lambda]\}
\end{aligned}
$$

□

The languages accepted by two-stack automata include, but are not limited to, the context-free languages. Techniques developed in Chapter 9 will allow us to categorize the languages accepted by two-stack PDAs.

Exercises

1. Let M be the PDA

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2\} & \delta(q_0, a, \lambda) &= \{[q_0, A]\} \\
\Sigma &= \{a, b\} & \delta(q_0, \lambda, \lambda) &= \{[q_1, \lambda]\} \\
\Gamma &= \{A\} & \delta(q_0, b, A) &= \{[q_2, \lambda]\} \\
F &= \{q_1, q_2\} & \delta(q_1, \lambda, A) &= \{[q_1, \lambda]\} \\
& & \delta(q_2, b, A) &= \{[q_2, \lambda]\} \\
& & \delta(q_2, \lambda, A) &= \{[q_2, \lambda]\}
\end{aligned}
$$

   a) Describe the language accepted by M.
   b) Give the state diagram of M.
   c) Trace all computations of the strings aab, abb, aba in M.
   d) Show that $aabb$, $aaab \in$ L(M).

2. Let M be the PDA in Example 8.1.3.
   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) $\{a^i b^i \mid 0 \leq i \leq j\}$

b) $\{a^i c^j b^i \mid i, j \geq 0\}$

c) $\{a^i b^j c^k \mid i + k = j\}$

d) $\{w \mid w \in \{a, b\}^*$ and w has the same number of a's and b's$\}$

e) $\{w \mid w \in \{a, b\}^*$ and w has twice as many a's as b's$\}$

f) $\{a^i b^i \mid i \geq 0\} \cup \boldsymbol{a}^* \cup \boldsymbol{b}^*$

g) $\{a^i b^j c^k \mid i = j$ or $j = k\}$

h) $\{a^i b^j \mid i \neq j\}$

i) $\{a^i b^j \mid 0 \leq i \leq j \leq 2i\}$

j) $\{a^{i+j} b^i c^j \mid i, j > 0\}$

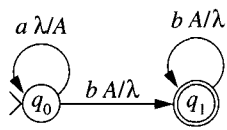
k) the set of palindromes over $\{a, b\}$

4. Construct a PDA with only two stack elements that accepts the language

$$\{w d w^R \mid w \in \{a, b, c\}^*\}.$$

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 $\{w \in \{a, b\}^* \mid w$ has a prefix containing more b's than a'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, Σ, Γ, δ, 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, Σ, Γ, δ, 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 8.3.1 to construct a PDA that accepts the language of the Greibach normal form grammar

$$S \rightarrow aABA \mid aBB$$
$$A \rightarrow bA \mid b$$
$$B \rightarrow cB \mid c.$$

13. Let G be a grammar in Greibach normal form and M the PDA constructed from G. Prove that if $[q_0, u, \lambda] \vdash^* [q_1, \lambda, w]$ i M, then there is a derivation $S \stackrel{*}{\Rightarrow} uw$ in G. This completes the proof of Theorem 8.3.1.

14. Let M be the PDA

$$Q = \{q_0, q_1, q_2\} \qquad \delta(q_0, a, \lambda) = \{[q_0, A]\}$$
$$\Sigma = \{a, b\} \qquad \delta(q_0, b, A) = \{[q_1, \lambda]\}$$
$$\Gamma = \{A\} \qquad \delta(q_1, b, \lambda) = \{[q_2, \lambda]\}$$
$$F = \{q_2\} \qquad \delta(q_2, b, A) = \{[q_1, \lambda]\}.$$

a) Describe the language accepted by M.

b) Using the technique from Theorem 8.3.2, build a context-free grammar G that generates L(M).

c) Trace the computation of *aabbbb* in M.

d) Give the derivation of *aabbbb* in G.

15. Let M be the PDA in Example 8.1.1.

a) Trace the computation in M that accepts *bbcbb*.

b) Use the technique from Theorem 8.3.2 to construct a grammar G that accepts L(M).

c) Give the derivation of *bbcbb* in G.

16. Theorem 8.3.2 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) $\{a^i b^j c^k \mid 0 < i < j < k\}$
    e) $\{ww^R w \mid w \in \{a, b\}^*\}$
    f) The set of finite-length prefixes of the infinite string
    $$abaabaaabaaaab \ldots ba^n ba^{n+1} b \ldots$$
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, Σ, P, S) is called **linear** if every rule has the form
    $$A \rightarrow u$$
    $$A \rightarrow uBv$$
    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 $length(z) \geq k$, z can be written $z = uvwxy$ with

i) $length(uvxy) \leq k$

ii) $length(vx) > 0$

iii) $uv^i wx^i y \in L$, 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 8.5.3 to construct a PDA M′ that accepts L(N) ∩ L(M).
    d) Trace the computations that accept $aaab$ in N, M, and M′.

24. Let G = (V, Σ, P, S) be a context-free grammar. Define an extended PDA M as follows:

$$Q = \{q_0\}$$
$$\Sigma = \Sigma_G$$
$$\Gamma = \Sigma_G \cup V$$
$$F = \{q_0\}$$
$$\delta(q_0, \lambda, \lambda) = \{[q_0, S]\}$$
$$\delta(q_0, \lambda, A) = \{[q_0, w] \mid A \rightarrow w \in P\}$$
$$\delta(q_0, a, a) = \{[q_0, \lambda] \mid a \in \Sigma\}$$

    Prove that L(M) = L(G).

25. Complete the proof of Theorem 8.5.3.

26. Prove that the set of context-free languages is closed under reversal.

27. Let L be a context-free language over Σ and $a \in \Sigma$. Define $er_a(L)$ to be the set obtained by removing all occurrences of a from strings of L. $er_a(L)$ is the language L with a erased. For example, if $abab$, $bacb$, $aa \in L$, 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 7.15. Let L be a context-free language over Σ and $h : \Sigma^* \rightarrow \Sigma^*$ 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 Σ. Prove that $\{w \mid h(w) \in L\}$ is context-free. In other words, the set 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) $\{(ab)^i (bc)^i (ca)^i \mid i \geq 0\}$
31. Construct two-stack PDAs that accept the following languages.
    a) $\{a^i b^{2i} a^i \mid i \geq 0\}$
    b) $\{a^i b^j a^i b^j \mid i, j \geq 0\}$
    c) $\{ww \mid w \in \{a, b\}^*\}$

Bibliographic Notes

Pushdown automata were introduced in Oettinger [1961]. The relationship between context-free languages and pushdown automata was discovered by Chomsky [1962], Evey [1963], and Schutzenberger [1963]. Closure properties for context-free languages presented in Section 8.5 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.

CHAPTER 9

Turing Machines

The Turing machine exhibits many of the features commonly associated with a modern computer. This is no accident; the Turing machine predated the stored-program computer and provided a model for its design and development. 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 Turing machine is another step in the development of finite-state computing machines. In a sense to be made precise in Chapters 11 and 13, this class of machines represents the ultimate achievement in the design of abstract computing devices.

9.1 The Standard Turing Machine

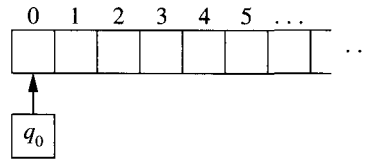
A Turing machine is a finite-state machine in which a transition prints a symbol on the tape. The tape head may move in either direction, allowing the machine to read and manipulate the input as many times as desired. The structure of a Turing machine is similar to that of a finite automaton, with the transition function incorporating these additional features.

Definition 9.1.1

A **Turing machine** is a quintuple $M = (Q, \Sigma, \Gamma, \delta, q_0)$ where Q is a finite set of states, Γ is a finite set called the *tape alphabet*, Γ contains a special symbol B that represents

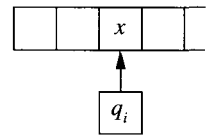
a blank, Σ is a subset of $\Gamma - \{B\}$ called the *input alphabet,* δ is a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$ called the *transition function,* and $q_0 \in Q$ is a distinguished state called the *start state.*

The tape of a Turing machine extends indefinitely in one direction. The tape positions are numbered by the natural numbers with the leftmost position numbered zero.

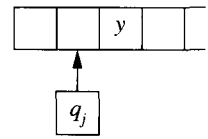


A computation begins with the machine in state q_0 and the tape head scanning the leftmost position. The input, a string from Σ^*, is written on the tape beginning at position one. Position zero and the remainder of the tape are assumed to be blank. The tape alphabet provides additional symbols that 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 Turing machine **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.

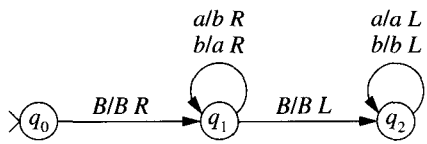
Turing machines are designed to perform computations on strings from the input alphabet. A computation begins with the tape head scanning the leftmost tape square and the input string beginning at position one. All tape squares to the right of the input string are assumed to be blank. The Turing machine defined in Definition 9.1.1, with initial conditions as described above, is referred to as the **standard Turing machine**.

Example 9.1.1

The transition function of a standard Turing machine with input alphabet $\{a, b\}$ is given below. 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.

δ	B	a	b
q_0	q_1, B, R		
q_1	q_2, B, L	q_1, b, R	q_1, a, R
q_2		q_2, a, L	q_2, b, L

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 above. □

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_ivB, 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_ivB 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 $uq_ivB \vdash_M xq_jyB$ indicates that the configuration xq_jyB is obtained from uq_ivB by a single transition of M. Following the standard conventions, $uq_ivB \vdash_M^* xq_jyB$ signifies that xq_jyB can be obtained from uq_ivB by a finite number,

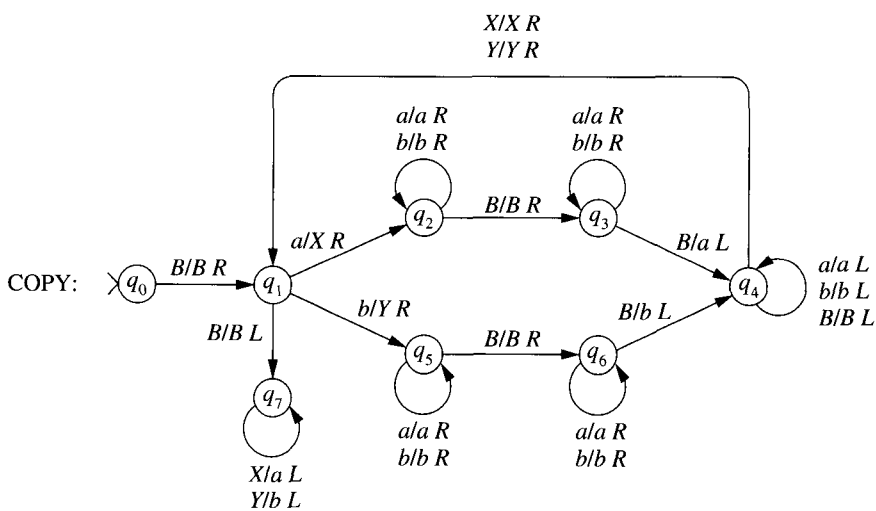
possibly zero, of transitions. The reference to the machine is omitted when there is no possible ambiguity.

The Turing machine in Example 9.1.1 interchanges the a's and b's in the input string. Tracing the computation generated by the input string $abab$ yields

$$
\begin{aligned}
& q_0 Bababa B \\
\vdash\ & B q_1 ababa B \\
\vdash\ & B b q_1 baba B \\
\vdash\ & B ba q_1 aba B \\
\vdash\ & B bab q_1 ba B \\
\vdash\ & B baba q_1 a B \\
\vdash\ & B babq_2 aa B \\
\vdash\ & B baq_2 baa B \\
\vdash\ & B bq_2 aba a B \\
\vdash\ & B q_2 baba a B \\
\vdash\ & q_2 Bbaba a B.
\end{aligned}
$$

Example 9.1.2

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 $BuBuB$.



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 X's and Y's are returned to a's and b's in state q_7.

□

9.2 Turing Machines as Language Acceptors

Turing machines have been introduced as a paradigm for effective computation. A Turing machine computation consists of a sequence of elementary operations. 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 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 9.2.1

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 final state of the machine. The language of M, L(M), is the set of all strings accepted by M.

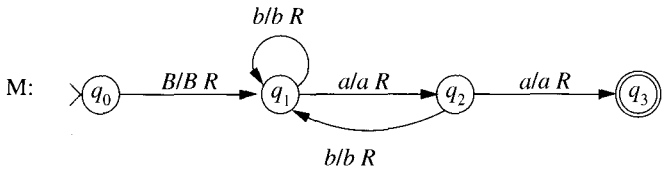
A language accepted by a Turing machine is called a **recursively enumerable language**. The ability of a Turing machine to move in both directions and process blanks introduces the possibility that the machine may not halt for a particular input. A language that is accepted by a Turing machine that halts for all input strings is said to be **recursive**. 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 while others may not. The existence of one Turing machine that halts for all inputs is sufficient to show that the language is recursive.

Membership in a recursive language is decidable; the computations of a Turing machine that halts for all inputs provide a decision procedure for determining membership. The state of the machine in which the computation terminates indicates whether the input string is in the language. The terms *recursive* and *recursively enumerable* have their origin in the functional interpretation of Turing computability that will be presented in Chapter 13.

Example 9.2.1

The Turing machine



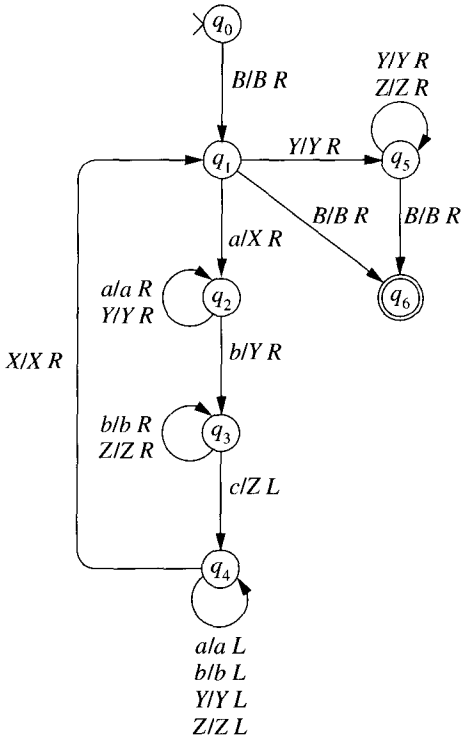
accepts the language $(a \cup b)^*aa(a \cup b)^*$. The computation

$$\begin{aligned} & q_0Baabb B \\ \vdash\ & Bq_1aabb B \\ \vdash\ & Baq_2abb B \\ \vdash\ & Baaq_3bb B \end{aligned}$$

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. □

Example 9.2.2

The language $\{a^ib^ic^i \mid i \geq 0\}$ 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. □

9.3 Alternative Acceptance Criteria

The acceptance of a string in the Turing machine defined in Definition 9.2.1 is based upon the state of the machine when the computation halts. Alternative approaches to defining the language of a Turing machine 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 causes the Turing machine to halt. 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 9.3.1

Let $M = (Q, \Sigma, \Gamma, \delta, q_0)$ be a Turing machine that **accepts by halting**. A string $u \in \Sigma^*$ is accepted by halting if the computation of M with input u halts (normally).

Theorem 9.3.2

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 $M' = (Q, \Sigma, \Gamma, \delta, q_0, Q)$, 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 $M' = (Q \cup \{q_f\}, \Sigma, \Gamma, \delta', q_0)$ that accepts by halting as follows:

i) If $\delta(q_i, x)$ is defined, then $\delta'(q_i, x) = \delta(q_i, x)$.

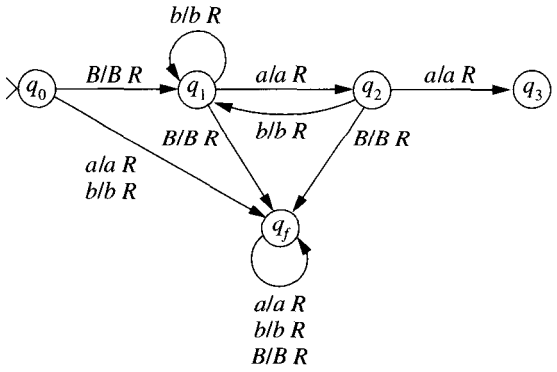
ii) For each state $q_i \in Q - F$, if $\delta(q_i, x)$ is undefined, then $\delta'(q_i, x) = [q_f, x, R]$.

iii) For each $x \in \Gamma$, $\delta'(q_f, x) = [q_f, x, R]$.

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_f. Upon entering q_f, 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 9.3.1

The Turing machine from Example 9.2.1 is altered to accept $(a \cup b)^*aa(a \cup b)^*$ by halting. The machine below is constructed as specified by Theorem 9.3.2. A computation enters q_f only 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_f and from q_f to q_f labeled $a/a\ R$ and $b/b\ R$ also accepts $(a \cup b)^*aa(a \cup b)^*$ by halting. □

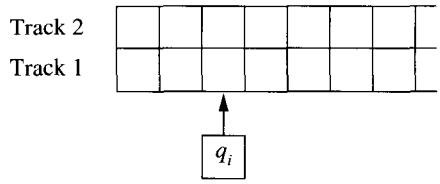
In Exercise 7 a type of acceptance, referred to as **acceptance by entering**, is introduced that uses final states but does not require the accepting computations to terminate. A string is accepted if the computation ever enters a final state; after entering a final state, the remainder of the computation 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 9.2.1. 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.

9.4 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 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 9.4.1

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 function

$$\delta'(q_i, [x, y]) = \delta(q_i, [x, y])$$

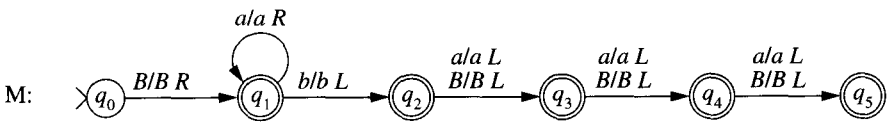
accepts L(M). ■

9.5 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.

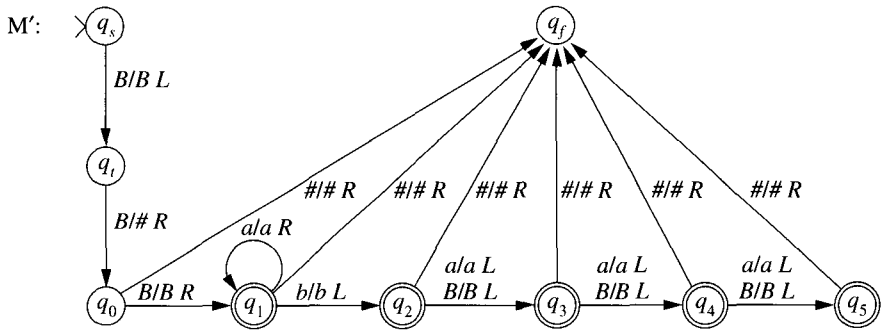
A machine with a two-way tape can be constructed to simulate the actions of a standard machine by placing a special symbol on the tape to represent the left-hand boundary of the one-way tape. The symbol #, which is assumed not to be an element of the tape alphabet of the standard machine, is used to 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 accepts strings over $\{a, b\}$ in which the first b, if present, is preceded by at least three a's.

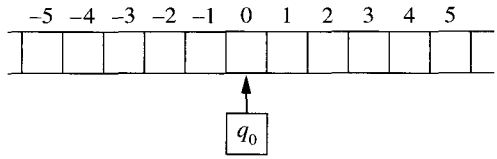


All the states of M other than q_0 are accepting. When the first b is encountered, the tape head moves four positions to the left, if possible. Acceptance is completely determined by the boundary of the tape. A string is rejected by M whenever the tape head attempts to cross the left-hand boundary. All computations that remain within the bounds of the tape accept the input.

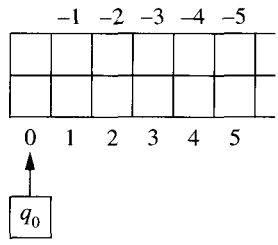
The transitions from states q_s and q_t insert the simulated endmarker to the left of the initial position of the tape head of M′, the two-way machine that accepts L(M). After writing the simulated boundary, the computation enters a copy of the one-way machine M. The failure state q_f is entered in M′ 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 9.4.1, 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 two-track tape, we construct a Turing machine M' with 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. The symbols U (up) and D (down) are introduced to designate which of the two tracks should be used to determine the transition. This information is maintained by the states of the two-track machine.

The components of M′ are constructed from those of M and the symbols U and D.

$$\mathrm{Q}' = (\mathrm{Q} \cup \{q_s, q_t\}) \times \{U, D\}$$

$$\Sigma' = \Sigma$$

$$\Gamma' = \Gamma \cup \{\#\}$$

$$\mathrm{F}' = \{[q_i, U], [q_i, D] \mid q_i \in \mathrm{F}\}.$$

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. 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 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.

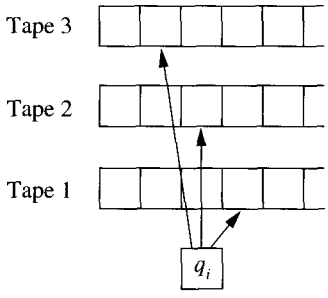
The remaining transitions simulate the transitions of M from position zero on the two-way tape. Regardless of the U or D in the state, transitions from position zero are determined by the tape symbol on track 1. When the track is specified by D, the transition is defined by schema 5 or 6. Transitions 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 9.5.1
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.

9.6 Multitape Machines

A k-tape machine consists of 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_i 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.

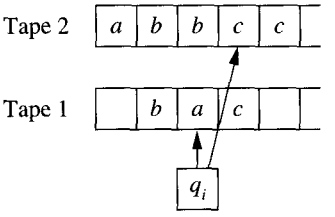
The input to a multitape machine is placed in the standard position on tape 1. All the other tapes are assumed to be blank. The tape heads originally scan the leftmost position of each tape. Any tape head attempting to move to the left of the boundary of its tape terminates the computation abnormally.

A standard Turing machine is a multitape Turing machine with a single tape. Consequently, every recursively enumerable language is accepted by a multitape machine. A computation in a two-tape machine can be simulated by a computation in 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.

Theorem 9.6.1

A language L is accepted by a multitape Turing machine if, and only if, it is accepted by a standard Turing machine.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a two-tape machine. 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.

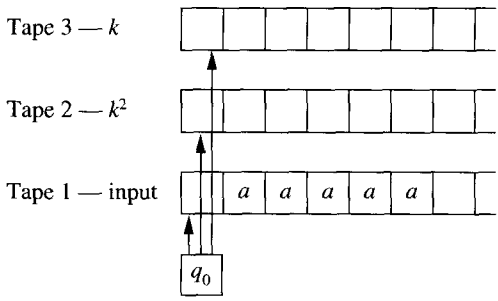
Let $\delta(q_i, x_1, x_2) = [q_j;\ y_1, d_1;\ y_2, d_2]$ be a two-tape transition. M′ begins the simulation of a transition of M in the state $[f1, q_i, U, U, U, U, U, U]$. The following five actions simulate a transition of M in the multitrack machine.

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 $[p1, q_j, x_1, x_2, y_1, y_2, d_1, d_2]$. 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. The simulation cycle terminates by returning the tape head to the initial position.

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 a final state of the multitrack machine M′ whenever q_i is a final state of M.

Example 9.6.1

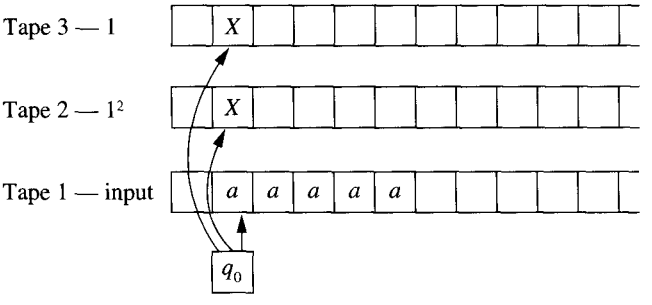
The set $\{a^k \mid k \text{ is a perfect square}\}$ is a recursively enumerable language. The design of a three-tape machine that accepts this language is presented. Tape 1 contains the input string. The input is compared with a string of X's on tape 2 whose length is a perfect square. Tape 3 holds a string whose length is the square root of the string on tape 2. The initial configuration for a computation with input *aaaaa* is



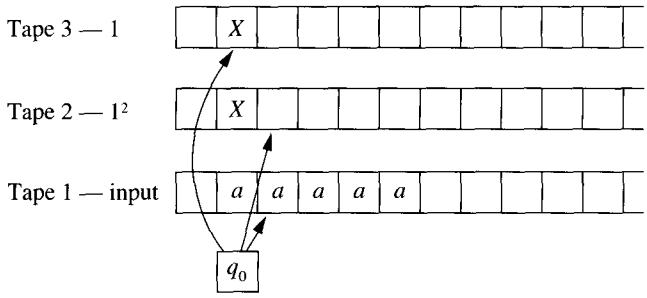
The values of k and k^2 are incremented until the length of the string on tape 2 is greater than or equal to the length of the input. A machine to perform these comparisons consists of the following actions:

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 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. 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 concatenated 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. Steps 2 through 4 are repeated.

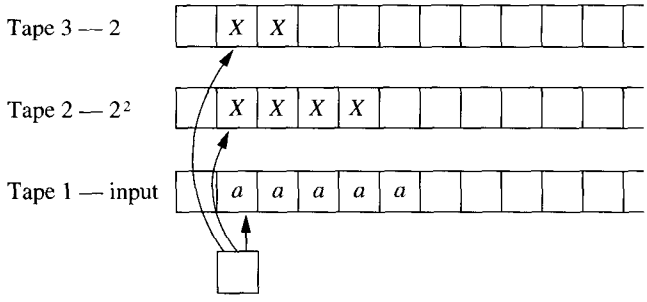
Tracing the computation for the input string *aaaaa*, step 1 produces the configuration



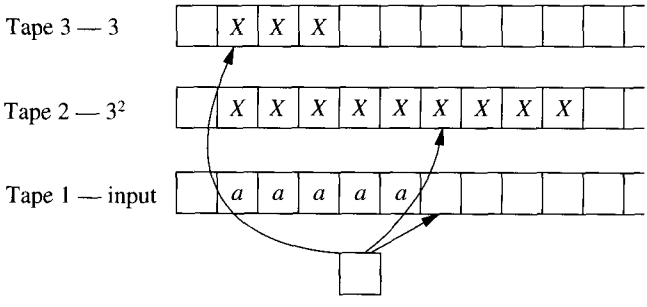
The simultaneous left-to-right movement of tape heads 1 and 2 halts when tape head 2 scans the blank in position two.



Part (c) of step 3 reformats tapes 2 and 3 so that the input string can be compared with the next perfect square.



Another iteration of step 2 halts and rejects the input.



The machine outlined above is defined by the following transitions:

$\delta(q_0, B, B, B) = [q_1;\ B, R;\ B, R;\ B, R]$ (initialize the tape)
$\delta(q_1, a, B, B) = [q_2;\ a, S;\ X, S;\ X, S]$

$\delta(q_2, a, X, X) = [q_2;\ a, R;\ X, R;\ X, S]$ (compare strings on tapes 1 and 2
$\delta(q_2, B, B, X) = [q_3;\ B, S;\ B, S;\ X, S]$ (accept)
$\delta(q_2, a, B, X) = [q_4;\ a, S;\ X, R;\ X, S]$

$\delta(q_4, a, B, X) = [q_5;\ a, S;\ X, R;\ X, S]$ (rewrite tapes 2 and 3)
$\delta(q_4, a, B, B) = [q_6;\ a, L;\ B, L;\ X, L]$
$\delta(q_5, a, B, X) = [q_4;\ a, S;\ X, R;\ X, R]$

$\delta(q_6, a, X, X) = [q_6;\ a, L;\ X, L;\ X, L]$ (reposition tape heads)
$\delta(q_6, a, X, B) = [q_6;\ a, L;\ X, L;\ B, S]$
$\delta(q_6, a, B, B) = [q_6;\ a, L;\ B, S;\ B, S]$
$\delta(q_6, B, X, B) = [q_6;\ B, S;\ X, L;\ B, S]$
$\delta(q_6, B, B, B) = [q_2;\ B, R;\ B, R;\ B, R]$. (repeat comparison cycle)

The accepting states are q_1 and q_3. The null string is accepted in q_1. Strings a^k, where k is a perfect square greater than zero, are accepted in q_3.

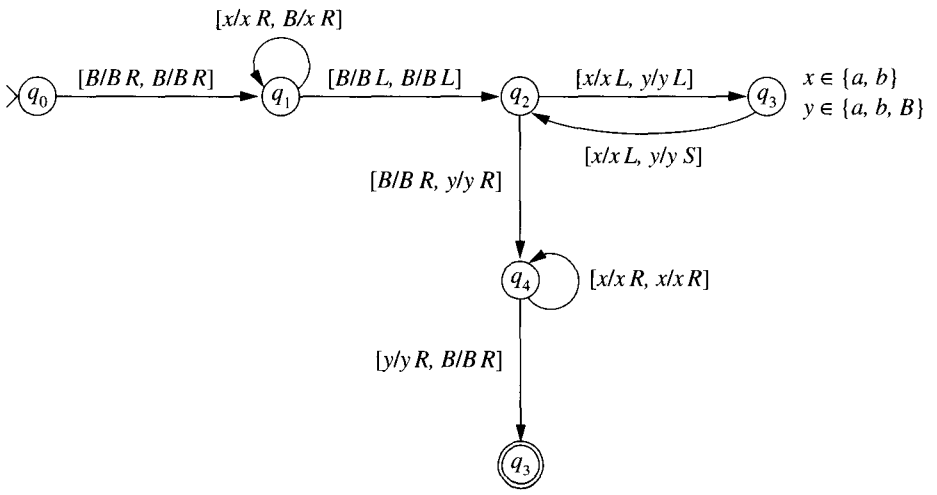
Since the machine designed above halts for all input strings, we have shown that the language $\{a^k \mid k \text{ is a perfect square}\}$ is not only recursively enumerable but also recursive.

□

Example 9.6.2

A multitape machine can also be represented by a state diagram. The arcs in the diagram contain the information for each tape. The two-tape Turing machine, whose state diagram follows, 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.



□

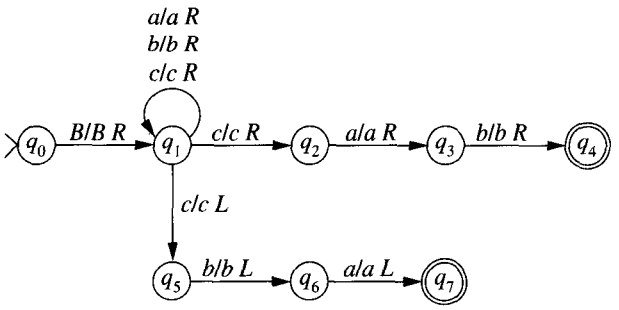
9.7 Nondeterministic Turing Machines

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 partial 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 a computation that terminates in an accepting state. As usual, the language of a machine is the set of strings accepted by the machine.

Example 9.7.1

The nondeterministic Turing machine given below 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. □

A nondeterministic Turing machine may produce several 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 $(m_1, m_2, \ldots, m_k)$ of values from 1 to n defines a computation in the nondeterministic machine. The computation associated with this sequence consists of k or fewer transitions. The jth transition is determined by the state, tape symbol scanned and m_j, the jth number in the sequence. Assume the $j-1$st 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.

The transitions of the nondeterministic machine in Example 9.7.1 can be ordered as shown in Table 9.7.1. 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

$q_0BacabB$ 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	
$\vdash Bacabq_1B$	$\vdash Bacabq_4B$	

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

TABLE 9.7.1 Ordering of Transitions

State	Symbol	Transition	State	Symbol	Transition
q_0	B	1 q_1, B, R	q_2	a	1 q_3, a, R
		2 q_1, B, R			2 q_3, a, R
		3 q_1, B, R			3 q_3, a, R
q_1	a	1 q_1, a, R	q_3	b	1 q_4, b, R
		2 q_1, a, R			2 q_4, b, R
		3 q_1, a, R			3 q_4, b, R
q_1	b	1 q_1, b, R	q_5	b	1 q_6, b, L
		2 q_1, b, R			2 q_6, b, L
		3 q_1, b, R			3 q_6, b, L
q_1	c	1 q_1, c, R	q_6	a	1 q_7, a, L
		2 q_2, c, R			2 q_7, a, L
		3 q_5, c, L			3 q_7, a, L

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.

The machine constructed in Example 9.7.1 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 23 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. Let $M = (Q, \Sigma, \Gamma, \delta, q_0)$ be a nondeterministic machine that accepts strings by halting. 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 computations of M are simulated in M'. The correspondence between a sequence $(m_1, \ldots, m_k)$ and a computation of M' ensures that all possible computations of length k are examined. A computation in M' consists of the actions:

1. A sequence of integers from 1 to n is written on tape 3.
2. The input string on tape 1 is copied to the standard position on tape 2.

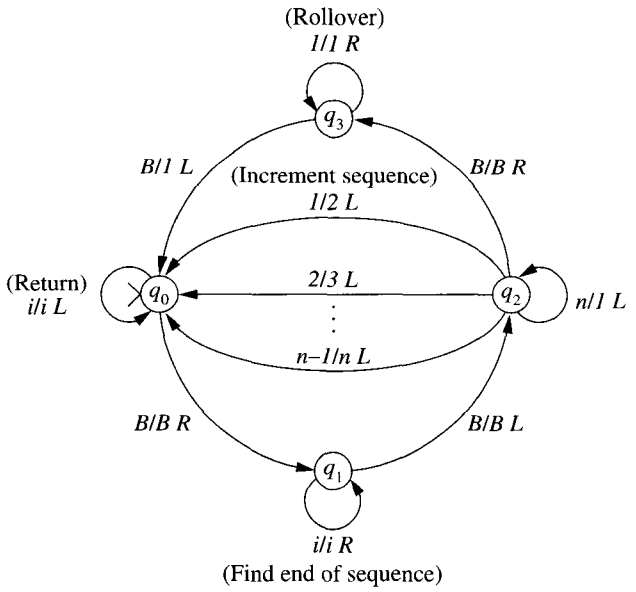


FIGURE 9.1 Turing machine generating $\{1, 2, \ldots, n\}^*$.

3. The computation of M defined by the sequence on tape 3 is simulated on tape 2.
4. If the simulation halts, the computation of M′ halts and input is accepted.
5. A new sequence is generated on tape 3 and steps 2 through 5 are repeated.

The simulation is guided by the sequence of values on tape 3. The deterministic Turing machine in Figure 9.1 generates all sequences of integers from 1 to n. Sequences of length one are generated in numeric order, followed by sequences of length two, 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 $1/1, 2/2, \ldots, n/n$.

Using this exhaustive generation of numeric sequences, we now construct a deterministic three-tape machine M′ that accepts L(M). The machine M′ is constructed by interweaving the generation of the sequences on tape 3 with the simulation 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 Σ and Γ be the input and tape alphabets of M. The alphabets of M′ are

$$\Sigma_{M'} = \Sigma$$
$$\Gamma_{M'} = \{x, \#x \mid x \in \Gamma\} \cup \{1, \ldots, n\}.$$

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 3. These transitions are obtained from the machine in Figure 9.1. The tape heads reading tapes 1 and 2 remain stationary during this operation.

$$\delta(q_{s,0}, B, B, B) = [q_{s,1};\ B, S;\ B, S;\ B, R]$$
$$\delta(q_{s,1}, B, B, t) = [q_{s,1};\ B, S;\ B, S;\ i, R]\ t = 1, \ldots, n$$
$$\delta(q_{s,1}, B, B, B) = [q_{s,2};\ B, S;\ B, S;\ B, L]$$
$$\delta(q_{s,2}, B, B, n) = [q_{s,2};\ B, S;\ B, S;\ 1, L]$$
$$\delta(q_{s,2}, B, B, t-1) = [q_{s,4};\ B, S;\ B, S;\ t, L]\ t = 1, \ldots, n-1$$
$$\delta(q_{s,2}, B, B, B) = [q_{s,3};\ B, S;\ B, S;\ B, R]$$
$$\delta(q_{s,3}, B, B, 1) = [q_{s,3};\ B, S;\ B, S;\ 1, R]$$
$$\delta(q_{s,3}, B, B, B) = [q_{s,4};\ B, S;\ B, S;\ 1, L]$$
$$\delta(q_{s,4}, B, B, t) = [q_{s,4};\ B, S;\ B, S;\ t, L]\ t = 1, \ldots, n$$
$$\delta(q_{s,4}, B, B, B) = [q_{c,0};\ B, S;\ B, S;\ B, S]$$

The next step is to make a copy of the input on tape 2. The symbol $\#B$ is written in position 0 to designate the left boundary of the tape.

$$\delta(q_{c,0}, B, B, B) = [q_{c,1};\ B, R;\ \#B, R;\ B, S]$$
$$\delta(q_{c,1}, x, B, B) = [q_{c,1};\ x, R;\ x, R;\ B, S] \quad \text{for all } x \in \Gamma - \{B\}$$
$$\delta(q_{c,1}, B, B, B) = [q_{c,2};\ B, L;\ B, L;\ B, S]$$
$$\delta(q_{c,2}, x, x, B) = [q_{c,2};\ x, L;\ x, L;\ B, S] \quad \text{for all } x \in \Gamma$$
$$\delta(q_{c,2}, B, \#B, B) = [q_0;\ B, S;\ \#B, S;\ B, R]$$

The transitions that simulate the computation of M on tape 2 of M′ are obtained directly from the transitions of M.

$$\delta(q_i, B, x, t) = [q_j;\ B, S;\ y, d; t, R] \quad \text{where } \delta(q_i, x) = [q_j, y, d] \text{ is}$$
$$\delta(q_i, B, \#x, t) = [q_j;\ B, S; \#y, d;\ t, R] \quad \text{a transition of M numbered } t$$

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.

$$\delta(q_i, B, x, B) = [q_{e,0};\ B, S;\ x, S;\ B, S] \quad \text{for all } x \in \Gamma$$
$$\delta(q_i, B, \#x, B) = [q_{e,0};\ B, S;\ \#x, S;\ B, S] \quad \text{for all } x \in \Gamma$$
$$\delta(q_{e,0}, B, x, B) = [q_{e,0};\ B, S;\ x, L;\ B, S] \quad \text{for all } x \in \Gamma$$
$$\delta(q_{e,0}, B, \#x, B) = [q_{e,1};\ B, S;\ B, S;\ B, L] \quad \text{for all } x \in \Gamma$$
$$\delta(q_{e,1}, B, B, t) = [q_{e,1};\ B, S;\ B, S;\ t, L] \quad t = 1, \ldots, n$$
$$\delta(q_{e,1}, B, B, B) = [q_{e,2};\ B, S;\ B, R;\ B, R]$$
$$\delta(q_{e,2}, B, x, i) = [q_{e,2};\ B, S;\ B, R;\ i, R] \quad \text{for all } x \in \Gamma \text{ and } i = 1, \ldots, n$$
$$\delta(q_{e,2}, B, B, B) = [q_{e,3};\ B, S;\ B, L;\ B, L]$$
$$\delta(q_{e,3}, B, B, t) = [q_{e,3};\ B, S;\ B, L;\ t, L] \quad t = 1, \ldots, n$$
$$\delta(q_{e,3}, B, B, B) = [q_{s,0};\ B, S;\ B, S;\ B, S]$$

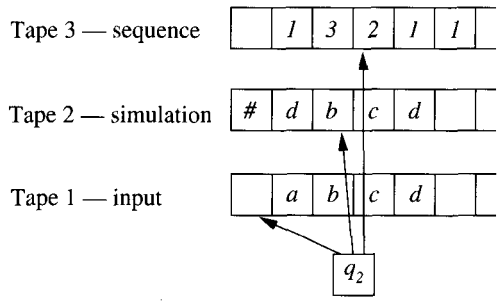
When a blank is read on tape 3, the entire segment of the tape that may have been accessed during the simulated computation has been erased. M′ then returns the tape heads to their initial position and enters $q_{s,0}$ to generate the next sequence and continue the simulation of computations.

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.

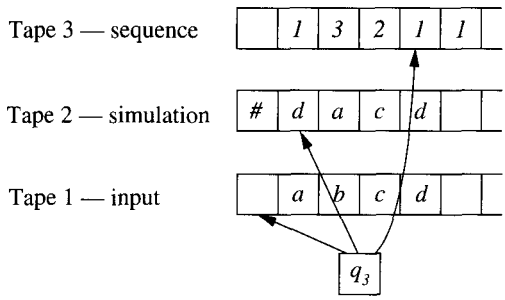
The actions the determinstic machine constructed following the preceding strategy are illustrated by examining the following sequence of machine configurations. Consider a nondeterministic machine M in which the computation defined by the sequence (1, 3, 2, 1, 1) and input string *abcd* is

$$\begin{aligned} & q_0 B a b c d B \ 1 \\ \vdash\ & B q_1 a b c d B \ 3 \\ \vdash\ & B d q_2 b c d B \ 2 \\ \vdash\ & B q_3 d a c d B. \end{aligned}$$

The sequence (1, 3, 2, 1, 1) that defines the 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 2 from state q_2 with M scanning b causes the machine to print a, enter state q_3, and move to the left. This transition is simulated in M′ by the transition $\delta'(q_2, B, b, 2) = [q_3;\ B, S;\ a, L;\ 2, R]$. The transition of M′ alters tape 2 as prescribed by the transition of M and moves the head on tape 3 to indicate 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.

9.8 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 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.

Since the machines in this section are designed for two distinct purposes, a machine that accepts a language will be denoted M while an enumerating machine will be denoted E.

Definition 9.8.1

A k-tape Turing machine $E = (Q, \Sigma, \Gamma, \delta, q_0)$ enumerates a language L if

i) the computation begins with all tapes blank

ii) with each transition, the tape head on tape 1 (the output tape) remains stationary or moves to the right

iii) at any point in the compuation, the nonblank portion of tape 1 has the form

$$B\#u_1\#u_2\#\ldots\#u_k\# \quad \text{or} \quad B\#u_1\#u_2\#\ldots\#u_k\#v,$$

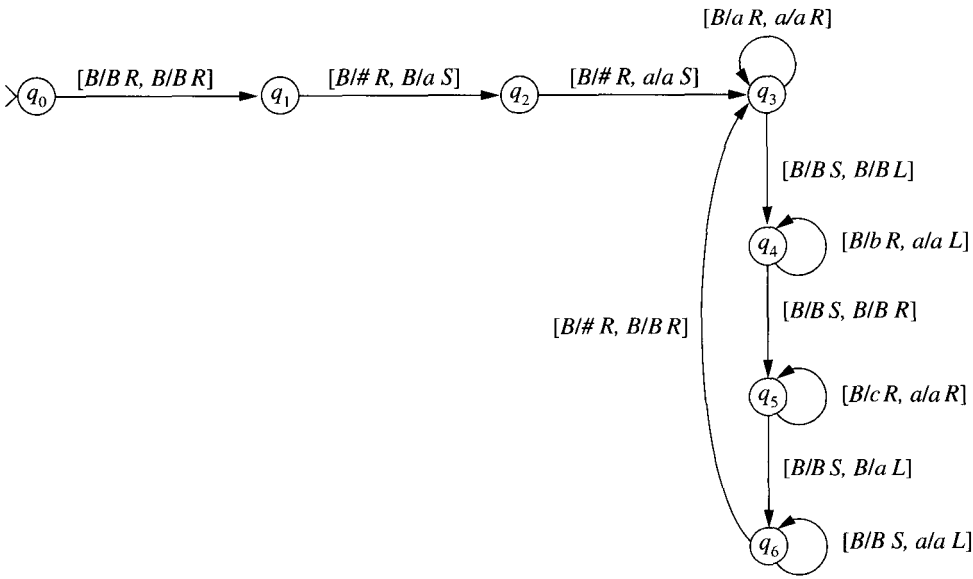
where $u_i \in L$ and $v \in \Sigma^*$

iv) 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 9.8.1

The machine E enumerates the language $L = \{a^i b^i c^i \mid i \geq 0\}$. A Turing machine accepting this language was given in Example 9.2.2.



The computation of E begins by writing ## on the output tape, indicating that $\lambda \in L$. Simultaneously, an a is written in position 1 of tape 2, with the head returning to tape position 0. At this point, E enters the nonterminating loop described below.

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 1.
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. □

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 9.8.2 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 9.8.2

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′ are

1. The computation begins by simulating the actions 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. ■

Theorem 9.8.2 justifies the selection of the term *enumerate* to describe this type of computation. The computation sequentially and exhaustively lists the strings in the language. 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, etc. 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 9.8.3

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

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. Repeat steps 1 through 4, until all strings in Σ^* have been tested.

The generate-and-test approach requires the ability to generate the entire set of strings over Σ 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 Σ^*. The lexicographical ordering of the set of strings over a nonempty alphabet Σ defines a one-to-one correspondence between the natural numbers and the strings in Σ^*.

Definition 9.8.4

Let $\Sigma = \{a_1, \ldots, a_n\}$ be an alphabet. The lexicographical ordering lo of Σ^* is defined recursively as follows:

i) Basis: $lo(\lambda) = 0, lo(a_i) = i$ for $i = 1, 2, \ldots, n$.

ii) Recursive step: $lo(a_i u) = lo(u) + i \cdot n^{\text{length}(u)}$.

The values assigned by the function lo define a total ordering on the set Σ^*. 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 9.8.2

Let $\Sigma = \{a, b, c\}$ and let a, b, and c be assigned the values 1, 2, and 3, respectively. The lexicographical ordering produces

$$\begin{array}{lllll} lo(\lambda) = 0 & lo(a) = 1 & lo(aa) = 4 & lo(ba) = 7 & lo(ca) = 10 \quad lo(aaa) = 13 \\ & lo(b) = 2 & lo(ab) = 5 & lo(bb) = 8 & lo(cb) = 11 \quad lo(aab) = 14 \\ & lo(c) = 3 & lo(ac) = 6 & lo(bc) = 9 & lo(cc) = 12 \quad lo(aac) = 15. \end{array}$$ □

Lemma 9.8.5

For any alphabet Σ, there is a Turing machine E_{Σ^*} that enumerates Σ^* 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 $u_0 = \lambda, u_1, u_2, u_3, \ldots$ of the strings of Σ^*. A two-dimensional table is constructed whose columns are labeled by the strings of Σ^* and rows by the natural numbers.

.	.	.	.	
.	.	.	.	
.	.	.	.	
3	$[\lambda, 3]$	$[u_1, 3]$	$[u_2, 3]$	. . .
2	$[\lambda, 2]$	$[u_1, 2]$	$[u_2, 2]$	. . .
1	$[\lambda, 1]$	$[u_1, 1]$	$[u_2, 1]$	. . .
0	$[\lambda, 0]$	$[u_1, 0]$	$[u_2, 0]$	. . .
	λ	u_1	u_2	. . .

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 29).

The machine E built to enumerate L interleaves the enumeration of the ordered pairs with the computations of M. The computation of E consists of the loop

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. Repeat steps 1 through 4.

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 Σ^* is examined.

Combining the preceding argument with Lemma 9.8.3 yields

Theorem 9.8.6
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 9.8.7 shows that recursive languages may be characterized as the family of languages whose elements can be enumerated in order.

Theorem 9.8.7
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 Σ. 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 a machine E_{Σ^*} that enumerates Σ^* in lexicographical order. The machine E is a hybrid, interleaving the computations of M and E_{Σ^*}. The computation of E consists of the loop

1. Machine E_{Σ^*} is run, producing a string $u \in \Sigma^*$.
2. Machine 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, steps 1 through 4, is repeated.

Since M halts for all inputs, E cannot enter a nonterminating computation in step 2.

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 9.8.2 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

δ	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

δ	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.

a) Move the input one space to the right. Input configuration q_0BuB, result q_fBBuB.

b) Concatenate a copy of the reversed input string to the input. Input configuration q_0BuB, result q_fBuu^RB.

c) Insert a blank between each of the input symbols. For example, input configuration q_0BabaB, result $q_fBaBbBaB$.

d) Erase the b's from the input. For example, input configuration $q_0BbabaababB$, result $q_fBaaaaB$.

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 Turing machines with input alphabet $\{a, b\}$ that accept the following languages by final state.

a) $\{a^ib^i \mid i \geq 0,\ j \geq i\}$

b) $\{a^ib^ja^ib^j \mid i,\ j > 0\}$

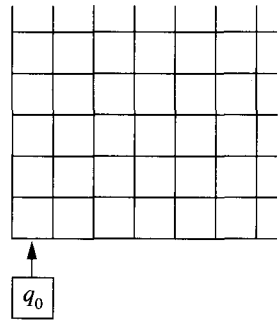
c) Strings with the same number of a's and b's

6. Modify your solution to Exercise 5(a) to obtain a Turing machine that accepts the language $\{a^ib^i \mid i \geq 0,\ j \geq i\}$ 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. 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.

9. A two-dimensional Turing machine is one in which the tape consists of a two-dimensional array of tape squares.



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.

10. 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 $3length(u) + 4$ transitions.
11. 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

$$ab^{n_1}ab^{n_2}\ldots ab^{n_k}, k > 0,$$

    where $n_1 < n_2 < \cdots < n_k$.
12. The transitions of a 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.
13. 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.
14. 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

$$\delta(q_i, xy) = [q_j, z, d] \qquad x, y, z \in \Gamma;\ d \in \{L, R\}.$$

    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.

15. Prove that every recursively enumerable language is accepted by a Turing machine with a single accepting state.
16. 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) $length(u) \geq 3$.
    ii) u contains the same number of a's and b's.
17. Construct a two-tape nondeterministic Turing machine that accepts the strings of odd length over $\{a, b, c\}$ with a c in the middle position. Every computation with input w should halt after at most $length(w) + 2$ transitions.
18. 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 $2length(w) + 2$ transitions. Using the deterministic machine from Example 9.6.2 that accepts L, what is the maximum number of transitions required for a computation with an input of length n?
19. A machine that generates all sequences consisting of integers from 1 to n was given in Figure 9.1. 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.
20. Construct a Turing machine that generates the set $L = \{a^i \mid i \text{ is divisible by } 3\}$. L is generated by a machine M under the following conditions:
    i) After the first transition, whenever the machine is in state q_0 scanning the left-most square, an element of L is on the tape.
    ii) All elements of L are eventually generated.
21. Construct a Turing machine that generates the set $\{a^i b^i \mid i \geq 0\}$.
22. Let L be a language accepted by a nondeterministic Turing machine in which every computation terminates. Prove that L is recursive.
23. Prove the equivalent of Theorem 9.3.2 for nondeterministic Turing machines.
24. Prove that every finite language is recursive.
25. Prove that a language L is recursive if, and only if, L and $\overline{L}$ are recursively enumerable.
26. Prove that the recursive languages are closed under union, intersection, and complement.
27. Build a Turing machine that enumerates the set of even length strings over $\{a\}$.
28. Build a Turing machine E_{Σ^*} that enumerates Σ^* where $\Sigma = \{0, 1\}$. *Note:* This machine may be thought of as enumerating all finite length bit strings.
29. Build a machine that enumerates the ordered pairs $\mathbf{N} \times \mathbf{N}$. 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.

30. In Theorem 9.8.7, 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?
31. Prove that the two-stack automata introduced in Section 8.6 accept precisely the recursively enumerable languages.
32. Define a nondeterministic two-track Turing machine. Prove that these machines accept precisely the recursively enumerable languages.
33. Prove that every context-free language is recursive. *Hint:* Construct a nondeterministic two-tape Turing machine that simulates the computation of a pushdown automaton.

Bibliographic Notes

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 capabilities and limitations of Turing machines as language acceptors are examined in Chapter 11. The use of Turing machines for the computation of functions is presented in Chapters 12 and 13. The books by Kleene [1952], Minsky [1967], Brainerd and Landweber [1974], and Hennie [1977] give an introduction to computability and Turing machines.

CHAPTER 10

The Chomsky Hierarchy

Phrase-structure grammars provide a formal system for generating strings over an alphabet. The productions, or rules, of a grammar specify permissible string transformations. Families of grammars are categorized by the form of the productions. The regular and context-free grammars introduced in Chapter 3 are two important families of phrase-structure grammars. In this chapter, two additional families of grammars, unrestricted grammars and context-sensitive grammars, are presented. These four families make up the Chomsky hierarchy, named after Noam Chomsky, who proposed them as syntactic models of natural language.

Automata were designed to mechanically recognize regular and context-free languages. The relationship between grammatical generation and mechanical acceptance is extended to the new families of grammars. Turing machines are shown to accept the languages generated by unrestricted grammars. A class of machines, the linear-bounded automata, obtained by limiting the memory available to a Turing machine, accepts the languages generated by context-sensitive grammars.

10.1 Unrestricted Grammars

The unrestricted grammars are the largest class of phrase-structure grammars. A production $u \rightarrow v$ indicates that an occurrence of a substring u in a string may be replaced with the string v. A derivation is a sequence of permissible replacements. The only constraint

on a production of an unrestricted grammar is that the left-hand side not be null. These general string rewriting systems are also called *type 0 grammars.*

Definition 10.1.1

An **unrestricted 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 set of productions, and S is a distinguished element of V. A production of an unrestricted grammar has the form $u \rightarrow v$, where $u \in (V \cup \Sigma)^+$ and $v \in (V \cup \Sigma)^*$. The sets V and Σ are assumed to be disjoint.

The previously defined families of grammars are subsets of the more general class of unrestricted grammars. A context-free grammar is a phrase-structure grammar in which the left-hand side of every rule is a single variable. The productions of a regular grammar are required to have the form

i) $A \rightarrow aB$

ii) $A \rightarrow a$

iii) $A \rightarrow \lambda$,

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

The notational conventions introduced in Chapter 3 for context-free grammars are used for arbitrary phrase-structure grammars. The application of the rule $u \rightarrow v$ to the string xuy, written $xuy \Rightarrow xvy$, produces the string xvy. A string q is **derivable** from p, $p \overset{*}{\Rightarrow} q$, if there is a sequence of rule applications that transforms p to q. The **language** of G, denoted L(G), is the set of terminal strings derivable from the start symbol S. Symbolically, $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$.

Example 10.1.1

The unrestricted grammar

$$\begin{aligned} V &= \{S, A, C\} \\ \Sigma &= \{a, b, c\} \\ S &\rightarrow aAbc \mid \lambda \\ A &\rightarrow aAbC \mid \lambda \\ Cb &\rightarrow bC \\ Cc &\rightarrow cc \end{aligned}$$

with start symbol S generates the language $\{a^i b^i c^i \mid i \geq 0\}$. The string $a^i b^i c^i$, $i > 0$, is generated by a derivation that begins

$$\begin{aligned} S &\Longrightarrow aAbc \\ &\overset{i-1}{\Longrightarrow} a^i A(bC)^{i-1}bc \\ &\Longrightarrow a^i (bC)^{i-1}bc. \end{aligned}$$

The rule $Cb \rightarrow bC$ allows the final C to pass through the b's that separate it from the c's at the end of the string. Upon reaching the leftmost c, the variable C is replaced with c. This process is repeated until each occurrence of the variable C is moved to the right of all the b's and transformed into a c. □

Example 10.1.2

The unrestricted grammar with terminal alphabet $\{a, b, [,]\}$ defined by the productions

$$\begin{aligned} S &\rightarrow aT[a] \mid bT[b] \mid [\,] \\ T[&\rightarrow aT[A \mid bT[B \mid [\\ Aa &\rightarrow aA \\ Ab &\rightarrow bA \\ Ba &\rightarrow aB \\ Bb &\rightarrow bB \\ A] &\rightarrow a] \\ B] &\rightarrow b] \end{aligned}$$

generates the language $\{u[u] \mid u \in \{a, b\}^*\}$.

The addition of an a or b to the beginning of the string is accompanied by the generation of the variable A or B. Using the rules that interchange the position of a variable and a terminal, the derivation progresses by passing the variable through the copy of the string enclosed in the brackets. When the variable is adjacent to the symbol], the appropriate terminal is added to the second string. The entire process may be repeated to generate additional terminal symbols or be terminated by the application of the rule $T[\rightarrow [$. The derivation

$$\begin{aligned} S &\Rightarrow aT[a] \\ &\Rightarrow aaT[Aa] \\ &\Rightarrow aaT[aA] \\ &\Rightarrow aaT[aa] \\ &\Rightarrow aabT[Baa] \\ &\Rightarrow aabT[aBa] \\ &\Rightarrow aabT[aaB] \\ &\Rightarrow aabT[aab] \\ &\Rightarrow aab[aab] \end{aligned}$$

exhibits the roles of the variables in a derivation. □

In the preceding grammars, the left-hand side of each rule contained a variable. This is not required by the definition of unrestricted grammar. However, the imposition of the

restriction that the left-hand side of a rule contain a variable does not reduce the set of languages that can be generated (Exercise 3).

Theorem 10.1.2

Let G = (V, Σ, P, S) be an unrestricted grammar. Then L(G) is a recursively enumerable language.

Proof We will sketch the design of a three-tape nondeterministic Turing machine M that accepts L(G). Tape 1 holds an input string p from Σ^*. We will design M so that its computations simulate derivations of the grammar G. A representation of the rules of G is written on tape 2. A rule $u \rightarrow v$ is represented by the string $u\#v$, where # is a tape symbol reserved for this purpose. Rules are separated by two consecutive #'s. The derivations of G are simulated on tape 3.

A computation of the machine M that accepts L(G) consists of the following actions:

1. S is written on position one of tape 3.
2. The rules of G are written on tape 2.
3. A rule $u\#v$ is chosen from tape 2.
4. An instance of the string u is chosen on tape 3, if one exists. Otherwise, the computation halts in a rejecting state.
5. The string u is replaced by v on tape 3.
6. If the strings on tape 3 and tape 1 match, the computation halts in an accepting state.
7. To apply another rule, steps 3 through 7 are repeated.

Since the length of u and v may differ, the simulation of a rule application $xuy \Rightarrow xvy$ may require shifting the position of the string y.

For any string $p \in$ L(G), there is a sequence of rule applications that derives p. This derivation will be examined by one of the nondeterministic computations of the machine M. Conversely, the actions of M on tape 3 generate precisely the strings derivable from S in G. The only strings accepted by M are terminal strings in L(G). Thus, L(M) = L(G). ■

Example 10.1.3

The language L = $\{a^i b^i c^i \mid i \geq 0\}$ is generated by the rules

$$S \rightarrow aAbc \mid \lambda$$
$$A \rightarrow aAbC \mid \lambda$$
$$Cb \rightarrow bC$$
$$Cc \rightarrow cc.$$

Computations of the machine that accepts L simulate derivations of the grammar. The rules of the grammar are represented on tape 2 by

$$BS\#aAbc\#\#S\#\#\#A\#aAbC\#\#A\#\#\#Cb\#bC\#\#Cc\#ccB.$$

The rule $S \rightarrow \lambda$ is represented by the string $S\#\#\#$. The first # separates the left-hand side of the rule from the right-hand side. The right-hand side of the rule, the null string in this case, is followed by the string ##. □

Theorem 10.1.3

Let L be a recursively enumerable language. Then there is an unrestricted grammar G with L(G) = L.

Proof Since L is recursively enumerable, it is accepted by a deterministic Turing machine M = (Q, Σ, Γ, δ, q_0, F). An unrestricted grammar G = (V, Σ, P, S) is designed whose derivations simulate the computations of M. Using the representation of a Turing machine configuration as a string that was introduced in Section 9.1, the effect of a Turing machine transition $\delta(q_i, x) = [q_j, y, R]$ on the configuration uq_ixvB can be represented by the string transformation $uq_ixvB \Rightarrow uyq_jvB$.

The derivation of a terminal string in G consists of three distinct subderivations:

i) The generation of a string $u[q_0Bu]$ where $u \in \Sigma^*$

ii) The simulation of a computation of M on the string $[q_0Bu]$

iii) If M accepts u, the removal of the simulation substring

The grammar G contains a variable A_i for each terminal symbol $a_i \in \Sigma$. These variables, along with $S, T, [$, and], are used in the generation of the strings $u[q_0Bu]$. The simulation of a computation uses variables corresponding to the states of M. The variables E_R and E_L are used in the third phase of a derivation. The set of terminals of G is the input alphabet of M.

$$\Sigma = \{a_1, a_2, \ldots, a_n\}$$
$$V = \{S, T, E_R, E_L, [,], A_1, A_2, \ldots, A_n\} \cup Q$$

The rules for each of the three parts of a derivation are given separately. A derivation begins by generating $u[q_0Bu]$, where u is an arbitrary string in Σ^*. The strategy used for generating strings of this form was presented in Example 10.1.2.

1. $S \rightarrow a_iT[a_i] \mid [q_0B]$ for $1 \leq i \leq n$
2. $T[\rightarrow a_iT[A_i \mid [q_0B$ for $i \leq i \leq n$
3. $A_ia_j \rightarrow a_jA_i$ for $1 \leq i, j \leq n$
4. $A_i] \rightarrow a_i]$ for $1 \leq i \leq n$

The computation of the Turing machine with input u is simulated on the string $[q_0Bu]$. The rules are obtained by rewriting of the transitions of M as string transformations.

5. $q_ixy \rightarrow zq_jy$ whenever $\delta(q_i, x) = [q_j, z, R]$ and $y \in \Gamma$
6. $q_ix] \rightarrow zq_jB]$ whenever $\delta(q_i, x) = [q_j, z, R]$
7. $yq_ix \rightarrow q_jyz$ whenever $\delta(q_i, x) = [q_j, z, L]$ and $y \in \Gamma$

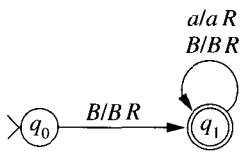
If the computation of M halts in an accepting state, the derivation erases the string within the brackets. The variable E_R erases the string to the right of the halting position of the tape head. Upon reaching the endmarker], the variable E_L (erase left) is generated.

8. $q_i x \rightarrow E_R$ whenever $\delta(q_i, x)$ is undefined and $q_i \in \mathrm{F}$
9. $E_R x \rightarrow E_R$ for $x \in \Gamma$
10. $E_R] \rightarrow E_L$
11. $x E_L \rightarrow E_L$ for $x \in \Gamma$
12. $[E_L \rightarrow \lambda$

The derivation that begins by generating $u[q_0 B u]$ terminates with u whenever $u \in \mathrm{L(M)}$. If $u \notin \mathrm{L(M)}$, the brackets enclosing the simulation of the computation are never erased and the derivation does not produce a terminal string. ■

Example 10.1.4

The construction of a grammar that generates the language accepted by a Turing machine is demonstrated using the machine



that accepts $a^*b(a \cup b)^*$. When the first b is encountered, M halts and accepts in state q_1.

The variables and terminals of G are

$$\Sigma = \{a, b\}$$

$$\mathrm{V} = \{S, T, E_R, E_L, [,], A, X\} \cup \{q_0, q_1\}.$$

The rules are given in three sets.

Input generating rules:

$$S \rightarrow aT[a] \mid bT[b] \mid [q_0 B]$$

$$T[\rightarrow aT[A \mid bT[X \mid [q_0 B$$

$$Aa \rightarrow aA$$

$$Ab \rightarrow bA$$

$$A] \rightarrow a]$$

$$Xa \rightarrow aX$$

$$Xb \rightarrow bX$$

$$X] \rightarrow b]$$

Simulation rules:

Transition	Rules
$\delta(q_0, B) = [q_1, B, R]$	$q_0Ba \rightarrow Bq_1a$
	$q_0Bb \rightarrow Bq_1b$
	$q_0BB \rightarrow Bq_1B$
	$q_0B] \rightarrow Bq_1B]$
$\delta(q_1, a) = [q_1, a, R]$	$q_1aa \rightarrow aq_1a$
	$q_1ab \rightarrow aq_1b$
	$q_1aB \rightarrow aq_1B$
	$q_1a] \rightarrow aq_1B]$
$\delta(q_1, B) = [q_1, B, R]$	$q_1Ba \rightarrow Bq_1a$
	$q_1Bb \rightarrow Bq_1b$
	$q_1BB \rightarrow Bq_1B$
	$q_1B] \rightarrow Bq_1B]$

Erasure rules:

$$
\begin{aligned}
q_1b &\rightarrow E_R \\
E_Ra &\rightarrow E_R \qquad aE_L \rightarrow E_L \\
E_Rb &\rightarrow E_R \qquad bE_L \rightarrow E_L \\
E_RB &\rightarrow E_R \qquad BE_L \rightarrow E_L \\
E_R] &\rightarrow E_L \qquad [E_L \rightarrow \lambda
\end{aligned}
$$

The computation that accepts the string ab in M and the corresponding derivation in the grammar G that accepts ab are

$$
\begin{aligned}
q_0Babb \qquad & S \Rightarrow aT[a] \\
\vdash Bq_1abB \qquad & \Rightarrow abT[Xa] \\
\vdash Baq_1bB \qquad & \Rightarrow ab[q_0BXa] \\
& \Rightarrow ab[q_0BaX] \\
& \Rightarrow ab[q_0Bab] \\
& \Rightarrow ab[Bq_1ab] \\
& \Rightarrow ab[Baq_1b] \\
& \Rightarrow ab[BaE_R] \\
& \Rightarrow ab[BaE_L \\
& \Rightarrow ab[BE_L \\
& \Rightarrow ab[E_L \\
& \Rightarrow ab
\end{aligned}
$$

□

Properties of unrestricted grammars can be used to establish closure results for recursively enumerable languages. The proofs, similar to those presented in Theorem 8.5.1 for context-free languages, are left as exercises.

Theorem 10.1.4

The set of recursively enumerable languages is closed under union, concatenation, and Kleene star.

10.2 Context-Sensitive Grammars

The context-sensitive grammars represent an intermediate step between the context-free and the unrestricted grammars. No restrictions are placed on the left-hand side of a production, but the length of the right-hand side is required to be at least that of the left.

Definition 10.2.1

A phrase-structure grammar $G = (V, \Sigma, P, S)$ is called **context-sensitive** if each production has the form $u \rightarrow v$, where $u \in (V \cup \Sigma)^+$, $v \in (V \cup \Sigma)^+$, and $length(u) \leq length(v)$.

A rule that satisfies the conditions of Definition 10.2.1 is called **monotonic.** Context-sensitive grammars are also called *monotonic or noncontracting,* since the length of the derived string remains the same or increases with each rule application. The language generated by a context-sensitive grammar is called, not surprisingly, a *context-sensitive language.*

Context-sensitive grammars were originally defined as phrase-structure grammars in which each rule has the form $uAv \rightarrow uwv$, where $A \in V, w \in (V \cup \Sigma)^+$ and $u, v \in (V \cup \Sigma)^*$. The preceding rule indicates that the variable A can be replaced by w only when it appears in the context of being preceded by u and followed by v. Clearly, every grammar defined in this manner is monotonic. On the other hand, a transformation defined by a monotonic rule can be generated by a set of rules of the form $uAv \rightarrow uwv$ (Exercises 10 and 11).

The monotonic property of the rules guarantees that the null string is not an element of a context-sensitive language. Removing the rule $S \rightarrow \lambda$ from the grammar in Example 10.1.1, we obtain the unrestricted grammar

$$\begin{aligned} S &\rightarrow aAbc \\ A &\rightarrow aAbC \mid \lambda \\ Cb &\rightarrow bC \\ Cc &\rightarrow cc \end{aligned}$$

that generates the language $\{a^i b^i c^i \mid i > 0\}$. The lambda rule violates the monotonicity property of context-sensitive rules. Replacing the S and A rules with

$$S \rightarrow aAbc \mid abc$$
$$A \rightarrow aAbC \mid abC$$

produces an equivalent context-sensitive grammar.

A nondeterministic Turing machine, similar to the machine in Theorem 10.1.2, is designed to accept a context-sensitive language. The noncontracting nature of the rules permits the length of the input string to be used to terminate the simulation of an unsuccessful derivation. When the length of the derived string surpasses that of the input, the computation halts and rejects the string.

Theorem 10.2.2

Every context-sensitive language is recursive.

Proof Following the approach developed in Theorem 10.1.2, derivations of the context-sensitive grammar are simulated on a three-tape nondeterministic Turing machine M. The entire derivation, rather than just the result, is recorded on tape 3. When a rule $u \rightarrow v$ is applied to the string xuy on tape 3, the string xvy is written on the tape following $xuy\#$. The symbol # is used to separate the derived strings.

A computation of M with input string p performs the following sequence of actions:

1. $S\#$ is written beginning at position one of tape 3.
2. The rules of G are written on tape 2.
3. A rule $u\#v$ is chosen from tape 2.
4. Let $q\#$ be the most recent string written on tape 3:
   a) An instance of the string u in q is chosen, if one exists. In this case, q can be written xuy.
   b) Otherwise, the computation halts in a rejecting state.
5. $xvy\#$ is written on tape 3 immediately following $q\#$.
6. a) If $xvy = p$, the computation halts in an accepting state.
   b) If xvy occurs at another position on tape 3, the computation halts in a rejecting state.
   c) If $length(xvy) > length(p)$, the computation halts in a rejecting state.
7. Steps 3 through 7 are repeated.

There are only a finite number of strings in $(V \cup \Sigma)^*$ with length less than or equal to $length(p)$. This implies that every derivation eventually halts, enters a cycle, or derives a string of length greater than $length(p)$. A computation halts at step 4 when the rule that has been selected cannot be applied to the current string. Cyclic derivations, $S \overset{*}{\Rightarrow} w \overset{+}{\Rightarrow} w$, are terminated in step 6(b). The length bound is used in step 6(c) to terminate all other unsuccessful derivations.

Every string in L(G) is generated by a noncyclic derivation. The simulation of such a derivation causes M to accept the string. Since every computation of M halts, L(G) is recursive (Exercise 9.22). ■

10.3 Linear-Bounded Automata

We have examined several alterations to the standard Turing machine that do not alter the set of languages accepted by the machines. Restricting the amount of the tape available for a computation decreases the capabilities of a Turing machine computation. A linear-bounded automaton is a Turing machine in which the amount of available tape is determined by the length of the input string. The input alphabet contains two symbols, ⟨ and ⟩, that designate the left and right boundaries of the tape.

Definition 10.3.1

A **linear-bounded automaton** (LBA) is a structure $M = (Q, \Sigma, \Gamma, \delta, q_0, \langle, \rangle, F)$, where Q, Σ, Γ, δ, q_0, and F are the same as for a nondeterministic Turing machine. The symbols ⟨ and ⟩ are distinguished elements of Σ.

The initial configuration of a computation is $q_0\langle w\rangle$, requiring $length(w) + 2$ tape positions. The endmarkers ⟨ and ⟩ are written on the tape but not considered part of the input. A computation remains within the boundaries specified by ⟨ and ⟩. The endmarkers may be read by the machine but cannot be erased. Transitions scanning ⟨ must designate a move to the right and those reading ⟩ to the left. A string $w \in (\Sigma - \{\langle, \rangle\})^*$ is accepted by an LBA if a computation with input $\langle w\rangle$ halts in an accepting state.

We will show that every context-sensitive language is accepted by a linear-bounded automaton. An LBA is constructed to simulate the derivations of the context-sensitive grammar. The Turing machine constructed to simulate the derivations of an unrestricted grammar begins by writing the rules of the grammar on one of the tapes. The restriction on the amount of tape available to an LBA prohibits this approach. Instead, states and transitions of the LBA are used to encode the rules.

Consider a context-sensitive grammar with variables $\{S, A\}$ and terminal alphabet $\{a\}$. The application of a rule $Sa \rightarrow aAS$ can be simulated by a sequence of transitions in an LBA (Figure 10.1). The first two transitions verify that the string on the tape beginning at the position of the tape head matches the left-hand side of the rule.

The application of the rule generates a string transformation $uSav \Rightarrow uaASv$. Before Sa is replaced with aAS, the string v is traversed to determine whether the derived string fits on the segment of the tape available to the computation. If the ⟩ is read, the computation terminates. Otherwise, the string v is shifted one position to the right and Sa is replaced by aAS.

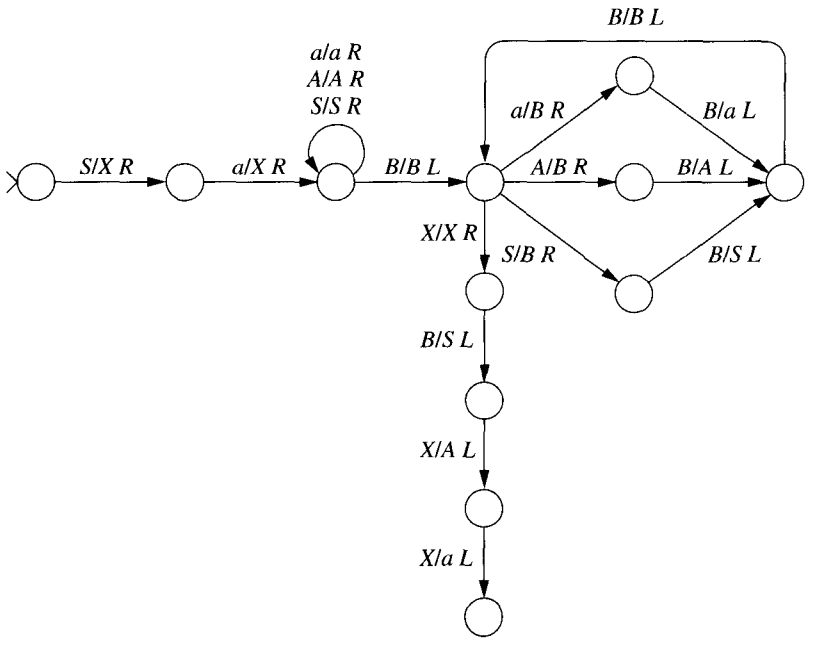


FIGURE 10.1 LBA simulation of application of $Sa \rightarrow aAs$.

Theorem 10.3.2
Let L be a context-sensitive language. Then there is a linear-bounded automaton M with L(M) = L.

Proof Since L is a context-sensitive language, L = L(G) for some context-sensitive grammar G = (V, Σ, P, S). An LBA M with a two-track tape is constructed to simulate the derivations of G. The first track contains the input, including the endmarkers. The second track holds the string generated by the simulated derivation.

Each rule of G is encoded in a submachine of M. A computation of M with input $\langle p \rangle$ consists of the following sequence of actions:

1. S is written on track 2 in position one.
2. The tape head is moved into a position in which it scans a symbol of the string on track 2.
3. A rule $u \rightarrow v$ is nondeterministically selected, and the computation attempts to apply the rule.
4. a) If a substring on track 2 beginning at the position of the tape head does not match u, the computation halts in a rejecting state.

b) If the tape head is scanning u but the string obtained by replacing u by v is greater than $length(p)$, then the computation halts in a rejecting state.

c) Otherwise, u is replaced by v on track 2.

5. If track 2 contains the string p, the computation halts in an accepting state.
6. Steps 2 through 6 are repeated.

Every string in L is generated by a derivation of G. The simulation of the derivation causes M to accept the string. Conversely, a computation of M with input $\langle p \rangle$ that halts in an accepting state consists of a sequence of string transformations generated by steps 2 and 3. These transformations define a derivation of p in G. ■

Theorem 10.3.3

Let L be a language accepted by a linear-bounded automaton. Then $L - \{\lambda\}$ is a context-sensitive language.

Proof Let $M = (Q, \Sigma_M, \Gamma, \delta, q_0, \langle, \rangle, F)$ be an LBA that accepts L. A context-sensitive grammar G is designed to generate L(M). Employing the approach presented in Theorem 10.1.3, a computation of M that accepts the input string p is simulated by a derivation of p in G. The techniques used to construct an unrestricted grammar that simulates a Turing machine computation cannot be employed since the rules that erase the simulation do not satisfy the monotonicity restrictions of a context-sensitive grammar. The inability to erase symbols in a derivation of context-sensitive grammar restricts the length of a derived string to that of the input. The simulation is accomplished by using composite objects as variables.

The terminal alphabet of G is obtained from the input alphabet of M by deleting the endmarkers. Ordered pairs are used as variables. The first component of an ordered pair is a terminal symbol. The second is a string consisting of a combination of a tape symbol and possibly a state and endmarker(s).

$$\Sigma_G = \Sigma_M - \{\langle, \rangle\} = \{a_i, a_2, \ldots, a_n\}$$

$$\begin{aligned} V = \{&S, A, [a_i, x], [a_i, \langle x], [a_i, x\rangle], [a_i, \langle x\rangle], [a_i, q_k x], \\ &[a_i, q_k\langle x], [a_i, \langle q_k x], [a_i, q_k x\rangle], [a_i, x q_k\rangle], [a_i, q_k\langle x\rangle], \\ &[a_i, \langle q_k x\rangle], [a_i, \langle x q_k\rangle]\}, \end{aligned}$$

where $a_i \in \Sigma_G, x \in \Gamma$, and $q_k \in Q$.

The S and A rules generate ordered pairs whose components represent the input string and the initial configuration of a computation of M.

1. $S \rightarrow [a_i, q_0\langle a_i]A$
   $\rightarrow [a_i, q_0\langle a_i\rangle]$
   for every $a_i \in \Sigma_G$
2. $A \rightarrow [a_i, a_i]A$
   $\rightarrow [a_i, a_i\rangle]$
   for every $a_i \in \Sigma_G$

Derivations using the S and A rules generate sequences of ordered pairs of the form

$$[a_i, q_0\langle a_i\rangle]$$
$$[a_{i_1}, q_0\langle a_{i_1}][a_{i_2}, a_{i_2}] \ldots [a_{i_n}, a_{i_n}\rangle].$$

The string obtained by concatenating the elements in the first components of the ordered pairs, $a_{i_1}a_{i_2}\ldots a_{i_n}$, represents the input string to a computation of M. The second components produce $q_0\langle a_{i_1}a_{i_2}\ldots a_{i_n}\rangle$, the initial configuration of the LBA.

The rules that simulate a computation are obtained by rewriting the transitions of M as transformations that alter the second components of the ordered pairs. Note that the second components do not produce the string $q_0\langle\,\rangle$; the computation with the null string as input is not simulated by the grammar. The techniques presented in Theorem 10.1.3 can be modified to produce the rules needed to simulate the computations of M. The details are left as an exercise.

Upon the completion of a successful computation, the derivation must generate the original input string. When an accepting configuration is generated, the variable with the accepting state in the second component of the ordered pair is transformed into the terminal symbol contained in the first component.

3. $[a_i, q_k\langle x] \rightarrow a_i$
   $[a_i, q_k\langle x\rangle] \rightarrow a_i$
   whenever $\delta(q_k, \langle) = \emptyset$ and $q_k \in \mathrm{F}$

   $[a_i, xq_k\rangle] \rightarrow a_i$
   $[a_i, \langle xq_k\rangle] \rightarrow a_i$
   whenever $\delta(q_k, \rangle) = \emptyset$ and $q_k \in \mathrm{F}$

   $[a_i, q_kx] \rightarrow a_i$
   $[a_i, q_kx\rangle] \rightarrow a_i$
   $[a_i, \langle q_kx] \rightarrow a_i$
   $[a_i, \langle q_kx\rangle] \rightarrow a_i$
   whenever $\delta(q_k, x) = \emptyset$ and $q_k \in \mathrm{F}$

The derivation is completed by transforming the remaining variables to the terminal contained in the first component.

4. $[a_i, u]a_j \rightarrow a_ia_j$
   $a_j[a_i, u] \rightarrow a_ja_i$
   for every $a_j \in \Sigma_\mathrm{G}$ and $[a_i, u] \in \mathrm{V}$ ■

10.4 The Chomsky Hierarchy

Chomsky numbered the four families of grammars (and languages) that make up the hierarchy. Unrestricted, context-sensitive, context-free, and regular grammars are referred to

as type 0, type 1, type 2, and type 3 grammars, respectively. The restrictions placed on the rules increase with the number of the grammar. The nesting of the families of grammars of the Chomsky hierarchy induces a nesting of the corresponding languages. Every context-free language containing the null string is generated by a context-free grammar in which $S \rightarrow \lambda$ is the only lambda rule (Theorem 5.1.5). Removing this single lambda rule produces a context-sensitive grammar that generates $L - \{\lambda\}$. Thus, the language $L - \{\lambda\}$ is context-sensitive whenever L is context-free. Ignoring the complications presented by the null string in context-sensitive languages, every type i language is also type $(i - 1)$.

The preceding inclusions are proper. The set $\{a^i b^i \mid i \geq 0\}$ is context-free but not regular (Theorem 7.5.1). Similarly, $\{a^i b^i c^i \mid i > 0\}$ is context-sensitive but not context-free (Example 8.4.1). In Chapter 11, the language L_H is shown to be recursively enumerable but not recursive. Combining this result with Theorem 10.2.2 establishes the proper inclusion of context-sensitive languages in the set of recursively enumerable languages.

Each class of languages in the Chomsky hierarchy has been characterized as the languages generated by a family of grammars and accepted by a type of machine. The relationships developed between generation and recognition are summarized in the following table.

Grammars	Languages	Accepting Machines
Type 0 grammars, phrase-structure grammars, unrestricted grammars	Recursively enumerable languages	Turing machine, nondeterministic Turing machine
Type 1 grammars, context-sensitive grammars, monotonic grammars	Context-sensitive languages	Linear-bounded automata
Type 2 grammars, context-free grammars	Context-free languages	Pushdown automata
Type 3 grammars, regular grammars, left-linear grammars, right-linear grammars	Regular languages	Deterministic finite automata, nondeterministic finite automata

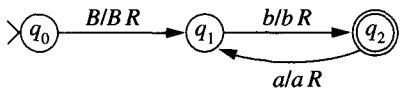
Exercises

1. Design unrestricted grammars to generate the following languages:
   a) $\{a^i b^j a^i b^j \mid i, j \geq 0\}$
   b) $\{a^i b^i c^i d^i \mid i \geq 0\}$
   c) $\{www \mid w \in \{a, b\}^*\}$
2. Prove that every terminal string generated by the grammar

$$S \rightarrow aAbc \mid \lambda$$
$$A \rightarrow aAbC \mid \lambda$$
$$Cb \rightarrow bC$$
$$Cc \rightarrow cc$$

has the form $a^i b^i c^i$ for some $i \geq 0$.

3. Prove that every recursively enumerable language is generated by a grammar in which each rule has the form $u \rightarrow v$ where $u \in V^+$ and $v \in (V \cup \Sigma)^*$.
4. Prove that the recursively enumerable languages are closed under the following operations:
   a) union
   b) intersection
   c) concatenation
   d) Kleene star
   e) homomorphic images
5. Let M be the Turing machine



   a) Give a regular expression for L(M).
   b) Using the techniques from Theorem 10.1.3, give the rules of an unrestricted grammar G that accepts L(M).
   c) Trace the computation of M when run with input *bab* and give the corresponding derivation in G.
6. Let G be the monotonic grammar

$$\text{G:} \quad S \rightarrow SBA \mid a$$
$$BA \rightarrow AB$$
$$aA \rightarrow aaB$$
$$B \rightarrow b.$$

   a) Give a derivation of *aaabb*.
   b) What is L(G)?
   c) Construct a context-free grammar that generates L(G).
7. Let L be the language $\{a^i b^{2i} a^i \mid i > 0\}$.

a) Use the pumping lemma for context-free languages to show that L is not context-free.

b) Construct a context-sensitive grammar G that generates L.

c) Give the derivation of *aabbbbaa* in G.

d) Construct an LBA M that accepts L.

e) Trace the computation of M with input *aabbbbaa*.

8. Let $L = \{a^i b^j c^k \mid 0 < i \leq j \leq k\}$.

a) L is not context-free. Can this be proven using the pumping lemma for context-free languages? If so, do so. If not, show that the pumping lemma is incapable of establishing that L is not context-free.

b) Give a context-sensitive grammar that generates L.

9. Let M be an LBA with alphabet Σ. Outline a general approach to construct monotonic rules that simulate the computation of M. The rules of the grammar should consist of variables in the set

$$\{[a_i, x], [a_i, \langle x], [a_i, x\rangle], [a_i, \langle x\rangle], [a_i, q_k x], [a_i, q_k\langle x], [a_i, \langle q_k x], [a_i, q_k x\rangle],$$
$$[a_i, x q_k\rangle], [a_i, q_k\langle x\rangle], [a_i, \langle q_k x\rangle], [a_i, \langle x q_k\rangle]\},$$

where $a_i \in \Sigma$, $x \in \Gamma$, and $q_i \in Q$. This completes the construction of the grammar in Theorem 10.3.3.

10. Let $u \rightarrow v$ be a monotonic rule. Construct a sequence of monotonic rules, each of whose right-hand side has length two or less, that defines the same transformation as $u \rightarrow v$.

11. Construct a sequence of context-sensitive rules $uAv \rightarrow uwv$ that define the same transformation as the monotonic rule $AB \rightarrow CD$. *Hint:* A sequence of three rules, each of whose left-hand side and right-hand side is of length two, suffices.

12. Use the results from Exercises 10 and 11 to prove that every context-sensitive language is generated by a grammar in which each rule has the form $uAv \rightarrow uwv$, where $w \in (V \cup \Sigma)^+$ and $u, v \in (V \cup \Sigma)^*$.

13. Let T be a full binary tree. A path through T is a sequence of left-down (L), right-down (R), or up (U) moves. Thus paths may be identified with strings over $\Sigma = \{L, R, U\}$. Consider the language $L = \{w \in \Sigma^* \mid w$ describes a path from the root back to the root$\}$. For example, λ, LU, $LRUULU \in L$ and U, $LRU \notin L$. Establish L's place in the Chomsky hierarchy.

14. Prove that the context-sensitive languages are not closed under arbitrary homomorphisms. A homomorphism is λ-free if $h(u) = \lambda$ implies $u = \lambda$. Prove that the context-sensitive grammars are closed under λ-free homomorphisms.

15. Let L be a recursively enumerable language over Σ and c a terminal symbol not in Σ. Show that there is a context-sensitive language L′ over $\Sigma \cup \{c\}$ such that, for every $w \in \Sigma^*$, $w \in L$ if, and only if, $wc^i \in L'$ for some $i \geq 0$.

16. Prove that every recursively enumerable language is the homomorphic image of a context-sensitive language. *Hint:* Use Exercise 15.
17. A grammar is said to be context-sensitive with erasing if every rule has the form $uAv \rightarrow uvw$, where $A \in \mathrm{V}$ and $u, v, w \in (\mathrm{V} \cup \Sigma)^*$. Prove that this family of grammars generates the recursively enumerable languages.
18. A linear-bounded automaton is deterministic if at most one transition is specified for each state and tape symbol. Prove that every context-free language is accepted by a deterministic LBA.
19. Let L be a context-sensitive language that is accepted by a deterministic LBA. Prove that $\overline{\mathrm{L}}$ is context-sensitive. Note that a computation in an arbitrary deterministic LBA need not halt.

Bibliographic Notes

The Chomsky hierarchy was introduced by Chomsky [1959]. This paper includes the proof that the unrestricted grammars generate precisely recursively enumerable languages. Linear-bounded automata were presented in Myhill [1960]. The relationship between linear-bounded automata and context-sensitive languages was developed by Landweber [1963] and Kuroda [1964]. Solutions to Exercises 10, 11, and 12, which exhibit the relationship between monotonic and context-sensitive grammars, can be found in Kuroda [1964].

PART IV

Decidability and Computability

The Turing machine represents the culmination of a series of increasingly powerful abstract computing devices. 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 formulation of this thesis for decision problems and its implications for computability are discussed in Chapter 11.

With the acceptance of the Church-Turing thesis, it becomes obvious that the extent of algorithmic problem solving can be identified with the capabilities of Turing machines. Consequently, to prove a problem to be unsolvable it suffices to show that there is no Turing machine that solves the problem. The first several problems that we show to have no algorithmic solution are concerned with properties of computations, but later problems exhibit the undecidability of questions on derivability using the rules of a grammar, search procedures, and properties of context-free languages.

Arguments establishing both the decidability and undecidability of problems utilize the ability of one Turing machine to simulate the computations of another. To accomplish this, an encoding of the transitions of the machine to be simulated are provided as input. This is the same philosophy as that employed by a stored-program computer, where the instructions of the program to be executed are loaded into the memory of the computer for execution.

The capabilities of Turing machines are extended by defining the result of a computation by the configuration of the tape when the computation terminates. This generalization permits Turing machines to compute functions and perform other standard computational tasks. The extended Church-Turing thesis asserts that any algorithmic process can be performed by an appropriately designed Turing machine. As supporting evidence, we show that a computation defined using a typical programming language can be simulated by a Turing machine.

The examination of the properties of computability is completed with a characterization of the set of functions that can be algorithmically computed. This provides an answer to the question concerning the capabilities of algorithmic computation that motivated our excursion into computability theory.

CHAPTER 11

Decidability

A decision problem consists of a set of questions whose answers are either yes or no. A solution to a decision problem is an effective procedure that determines the answer for each question in the set. A decision problem is undecidable if there is no algorithm that solves the problem. The ability of Turing machines to return affirmative and negative responses makes them an appropriate mathematical system for constructing solutions to decision problems. The Church-Turing thesis asserts that a Turing machine can be designed to solve any decision problem that is solvable by any effective procedure. Consequently, to establish that a problem is unsolvable it suffices to show that there is no Turing machine solution. Techniques are developed to establish the undecidability of several important questions concerning the capabilities of algorithmic computation.

Throughout the first four sections of this chapter we will consider Turing machines with input alphabet $\{0, 1\}$ and tape alphabet $\{0, 1, B\}$. The restriction on the alphabets imposes no limitation on the computational capabilities of Turing machines, since the computation of an arbitrary Turing machine M can be simulated on a machine with these restricted alphabets. The simulation requires encoding the symbols of M as strings over $\{0, 1\}$. This is precisely the approach employed by digital computers, which use the ASCII (American Standard Code for Information Interchange) or EBCDIC (Extended Binary Coded Decimal Interchange Code) encodings to represent characters as binary strings.

11.1 Decision Problems

A **decision problem P** is a set of questions, each of which has a yes or no answer. The single question "Is 8 a perfect square?" is an example of the type of question under consideration in a decision problem. A decision problem usually consists of an infinite number of related questions. For example, the problem $\mathbf{P}_{SQ}$ of determining whether an arbitrary natural number is a perfect square consists of the following questions:

$\mathbf{p}_0$: Is 0 a perfect square?

$\mathbf{p}_1$: Is 1 a perfect square?

$\mathbf{p}_2$: Is 2 a perfect square?

$\vdots$

A solution to a decision problem **P** is an algorithm that determines the appropriate answer to every question $\mathbf{p} \in \mathbf{P}$.

Since a solution to a decision problem is an algorithm, a review of our intuitive notion of algorithmic computation may be beneficial. We have not defined—and probably cannot precisely define—*algorithm*. This notion falls into the category of "I can't describe it but I know one when I see one." We can, however, list several properties that seem fundamental to the concept of algorithm. An algorithm that solves a decision problem should be

- Complete: It produces an answer, either positive or negative, to each question in the problem domain.
- Mechanistic: It consists of a finite sequence of instructions, each of which can be carried out without requiring insight, ingenuity, or guesswork.
- Deterministic: When presented with identical input, it always produces the same result.

A procedure that satisfies the preceding properties is often called *effective*.

The computations of a standard Turing machine are clearly mechanistic and deterministic. A Turing machine that halts for every input string is also complete. Because of the intuitive effectiveness of their computations, Turing machines provide a formal framework that can be used to construct solutions to decision problems. A problem is answered affirmatively if the input is accepted by a Turing machine and negatively if it is rejected.

Recall the newspaper vending machine described at the beginning of Chapter 6. Thirty cents in nickels, dimes, and quarters is required to open the latch. If more than 30 cents is inserted, the machine keeps the entire amount. Consider the problem of a miser who wants to buy a newspaper but refuses to pay more than the minimum. A solution to this problem is a procedure that determines whether a set of coins contains a combination that totals exactly 30 cents.

The transformation of a decision problem from its natural domain to an equivalent problem that can be answered by a Turing machine is known as constructing a representation of the problem. To solve the correct-change problem with a Turing machine, the problem must be formulated as a question of accepting strings. The miser's change can be represented as an element of $\{n, d, q\}^*$ where n, d, and q designate a nickel, a dime, and a quarter, respectively. Note that a representation is not unique; there are six strings that represent the set of coins consisting of a nickel, a dime, and a quarter.

The Turing machine in Figure 11.1 solves the correct-change problem. The sequences in the start state represent the five distinct combinations of coins that provide an affirmative answer to the question. For example, the solution consisting of one dime and four nickels is represented by $(d, 4n)$. The sequences in the state entered as the result of a transition specify the coins that, when combined with the previously processed coins, produce a combination that totals exactly 30 cents. The input $qqdnd$ is accepted since the set of coins represented by this string contains a quarter and a nickel, which total precisely 30 cents.

We have chosen the standard model of the Turing machine as a formal system in which solutions to decision problems are formulated. The completeness property of effective computation requires the computation of the machine to terminate for every input string. Thus the language accepted by a Turing machine that solves a decision problem is recursive. Conversely, every deterministic Turing machine M that accepts a recursive language can be considered a solution to a decision problem. The machine M solves the problem consisting of questions of the form "Is the string w in L(M)?" for every string $w \in \Sigma^*$.

The duality between solvable decision problems and recursive languages can be exploited to broaden the techniques available for establishing the decidability of a decision problem. A problem is **decidable** if it has a representation in which the set of accepted input strings form a recursive language. Since computations of deterministic multitrack and multitape machines can be simulated on a standard Turing machine, solutions using these machines also establish the decidability of a problem.

Example 11.1.1

The decision problem $\mathbf{P}_{SQ}$ is decidable. The three-tape Turing machine from Example 9.6.1 solves $\mathbf{P}_{SQ}$. □

Determinism is one of the fundamental properties of algorithms. However, it is often much easier to design a nondeterministic Turing machine to solve a decision problem. In Section 9.7 it was shown that every language accepted by a nondeterministic Turing machine is also accepted by a deterministic one. A solution to a decision problem requires more than a machine that accepts the appropriate strings; it also demands that all computations terminate. A nondeterministic machine in which every computation terminates can be used to establish the existence of a decision procedure. The languages of such machines are recursive (Exercise 9.22), ensuring the existence of a deterministic solution.

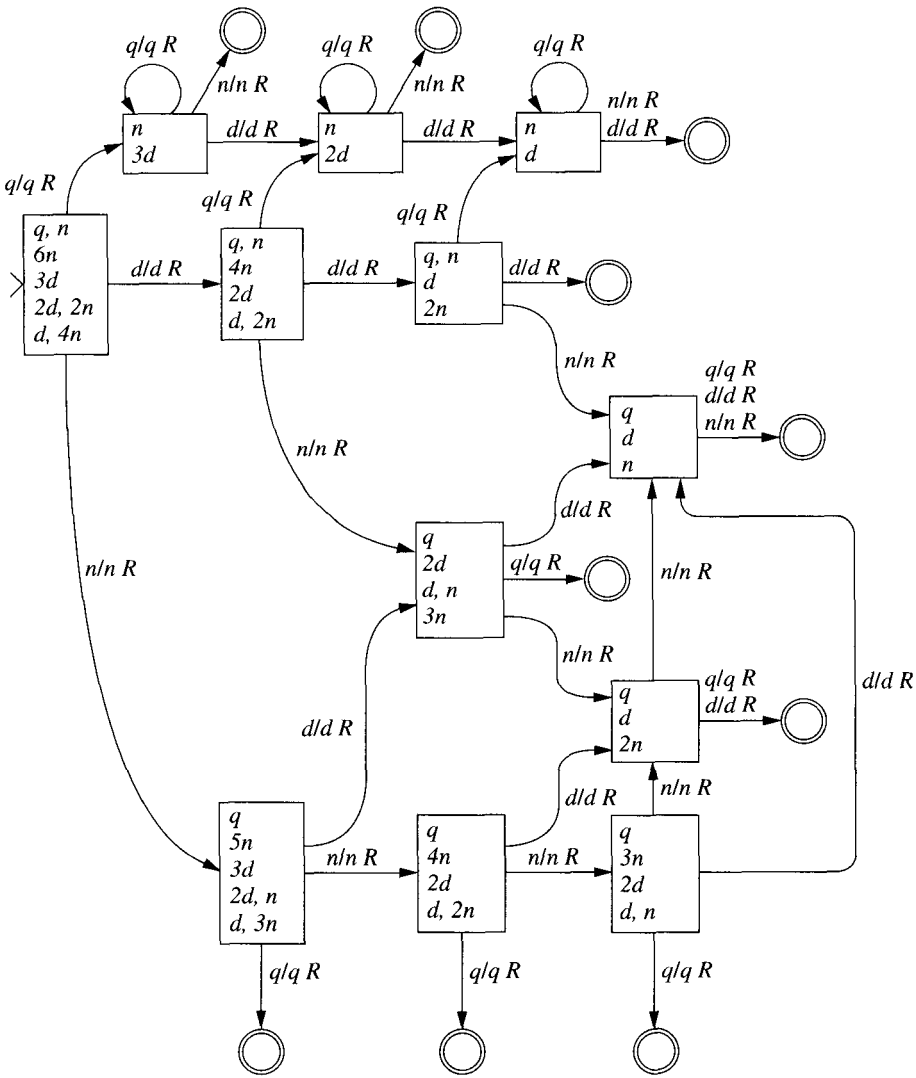


FIGURE 11.1 Solution to the correct-change problem.

Example 11.1.2

The problem of determining whether there is a path from a node v_i to a node v_j in a directed graph is decidable. A directed graph consists of a set of nodes $N = \{v_1, \ldots, v_n\}$ and arcs $A \subseteq N \times N$. To represent the graph as strings over $\{0, 1\}$, node v_k is encoded by

1^k. An arc $[v_s, v_t]$ is represented by the string $en(v_s)0en(v_t)$, where $en(v_s)$ and $en(v_t)$ are the encodings of nodes v_s and v_t. The string *00* is used to separate arcs.

The input to the machine consists of a representation of the graph followed by the encoding of nodes v_i and v_j. Three *0*'s separate $en(v_i)$ and $en(v_j)$ from the representation of the graph. The directed graph

$$N = \{v_1, v_2, v_3\}$$
$$A = \{[v_1, v_2], [v_1, v_1], [v_2, v_3], [v_3, v_2]\}$$

is represented by the string *10110010100110111001111011*. A computation to determine whether there is a path from v_3 to v_1 in this graph begins with the input *10110010100110 11100111011000111101*.

A nondeterministic two-tape Turing machine M is designed to solve the path problem. The actions of M are summarized below.

1. The input is checked to determine if its format is that of a representation of a directed graph followed by the encoding of two nodes. If not, M halts and rejects the string.
2. The input is now assumed to have form $R000en(v_i)0en(v_j)$, where R is the representation of a directed graph. If $i = j$, M halts in an accepting state.
3. The encoding of node v_i followed by *0* is written on tape 2 in the leftmost position.
4. Let v_s be the rightmost node encoded on tape 2. An arc from v_s to v_t, where v_t is not already on tape 2, is nondeterministically chosen from R. If no such arc exists, M halts in a rejecting state.
5. If $t = j$, M halts in an accepting state. Otherwise, $en(v_t)0$ is written at the end of the string on tape 2 and steps 4 and 5 are repeated.

Steps 4 and 5 generate paths beginning with node v_i on tape 2. Every computation of M terminates after at most n iterations, where n is the number of nodes in the graph, since step 4 guarantees that only noncyclic paths are examined. It follows that L(M) is recursive and the problem is decidable. □

11.2 The Church-Turing Thesis

The concept of an abstract machine was introduced to provide a formalization of algorithmic computation. Turing machines have been used to accept languages and solve decision problems. These computations are restricted to returning a yes or no answer. The

Church-Turing thesis asserts that every solvable decision problem can be transformed into an equivalent Turing machine problem.

By defining the result of a computation by the symbols on the tape when the machine halts, Turing machines can be used to compute functions. Chapters 12 and 13 examine capabilities and limitations of evaluating functions using Turing machine computations. A more general and concise form of the Church-Turing thesis in terms of effectively computable functions is presented in Chapter 13. Since our attention has been focused on yes/no problems, the Church-Turing thesis is presented and its implications are discussed for this class of problems.

A solution to a decision problem requires the computation to return an answer for every instance of the problem. Relaxing this restriction, we obtain the notion of a partial solution. A partial solution to a decision problem **P** is a not necessarily complete effective procedure that returns an affirmative response for every $\mathbf{p} \in \mathbf{P}$ whose answer is yes. If the answer to **p** is negative, however, the procedure may return no or fail to produce an answer.

Just as a solution to a decision problem can be formulated as a question of membership in a recursive language, a partial solution to a decision problem is equivalent to the question of membership in a recursively enumerable language.

The Church-Turing thesis for decision problems There is an effective procedure to solve a decision problem if, and only if, there is a Turing machine that halts for all input strings and solves the problem.

The extended Church-Turing thesis for decision problems A decision problem **P** is partially solvable if, and only if, there is a Turing machine that accepts precisely the elements of **P** whose answer is yes.

To appreciate the content of the Church-Turing thesis, it is necessary to understand the nature of the assertion. The Church-Turing thesis is not a mathematical theorem; it cannot be proved. This would require a formal definition of the intuitive notion of an effective procedure. The claim could, however, be disproved. This could be accomplished by discovering an effective procedure that cannot be computed by a Turing machine. There is an impressive pool of evidence that suggests that such a procedure will not be found. More about that later.

The Church-Turing thesis may be considered to provide a definition of algorithmic computation. This is an extremely limiting viewpoint. There are many systems that satisfy our intuitive notion of an effective algorithm, for example, the machines designed by Post [1936], recursive functions [Kleene, 1936], the lambda calculus of Church [1941], and, more recently, programming languages for digital computers. These are but a few of the systems 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 all known effective procedures have been able to be transformed into equivalent Turing machines.

The robustness of the standard Turing machine offers additional support for the Church-Turing thesis. Adding multiple tracks, multiple tapes, and nondeterministic computation does not increase the set of recognizable languages. Other approaches to computation, developed independently of the formalization of Turing machines, have been shown to recognize the same languages. In Chapter 10 we demonstrated that the recursively enumerable languages are precisely those generated by unrestricted grammars.

A proof by the Church-Turing thesis is a shortcut often taken in establishing the existence of a decision algorithm. Rather than constructing a Turing machine solution to a decision problem, we describe an intuitively effective procedure that solves the problem. The Church-Turing thesis guarantees that a Turing machine can be designed to solve the problem. We have tacitly been using the Church-Turing thesis in this manner throughout the presentation of Turing computability. For complicated machines, we simply gave an effective description of the actions of a computation of the machine. We assumed that the complete machine could then be explicitly constructed, if desired.

The Church-Turing thesis asserts that a decision problem **P** has a solution if, and only if, there is a Turing machine that determines the answer for every $\mathbf{p} \in \mathbf{P}$. If no such Turing machine exists, the problem is said to be **undecidable**. A Turing machine computation is not encumbered by the restrictions that are inherent in any "real" computing device. The existence of a Turing machine solution to a decision problem depends entirely on the nature of the problem itself and not on the availability of memory or central processor time. The universality of Turing machine computations also has consequences for undecidability. If a problem cannot be solved by a Turing machine, it clearly cannot be solved by a resource-limited machine. The remainder of this chapter is dedicated to demonstrating the undecidability of several important problems from computability theory and formal language theory.

11.3 The Halting Problem for Turing Machines

The most famous of the undecidable problems is concerned with the properties of Turing machines themselves. The halting problem may be formulated as follows: Given an arbitrary Turing machine M with input alphabet Σ and a string $w \in \Sigma^*$, will the computation of M with input w halt? We will show that there is no algorithm that solves the halting problem. The undecidability of the halting problem is one of the fundamental results in the theory of computer science.

It is important to understand the statement of the problem. We may be able to determine that a particular Turing machine will halt for a given string. In fact, the exact set of strings for which a particular Turing machine halts may be known. For example, the machine in Example 9.3.1 halts for all and only the strings containing the substring aa. A solution to the halting problem, however, requires a general algorithm that answers the halting question for every possible combination of Turing machine and input string.

A solution to the halting problem requires the Turing machine M and the string w to be represented as an input string. Recall that, because of the ability to encode arbitrary symbols as strings over $\{0, 1\}$, we are limiting our analysis to Turing machines that have input alphabet $\{0, 1\}$ and tape alphabet $\{0, 1, B\}$. The states of a Turing machine are assumed to be named $\{q_0, q_1, \ldots, q_n\}$, with q_0 the start state.

A Turing machine is completely defined by its transition function. A transition of a deterministic Turing machine has the form $\delta(q_i, x) = [q_j, y, d]$, where q_i, $q_j \in Q$; x, $y \in \Gamma$; and $d \in \{L, R\}$. We encode the elements of M using strings of 1's:

Symbol	Encoding
0	1
1	11
B	111
q_0	1
q_1	11
$\vdots$	$\vdots$
q_n	1^{n+1}
L	1
R	11

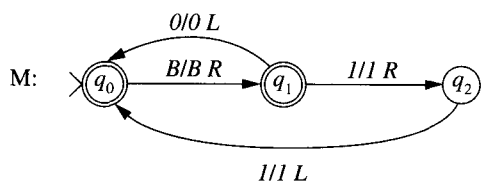
Let $en(z)$ denote the encoding of a symbol z. A transition $\delta(q_i, x) = [q_j, y, d]$ is encoded by the string

$$en(q_i)0en(x)0en(q_j)0en(y)0en(d).$$

The 0's separate the components of the transition. A representation of the machine is constructed from the encoded transitions. Two consecutive 0's are used to separate transitions. The beginning and end of the representation are designated by three 0's.

Example 11.3.1

The Turing machine



accepts the null string and strings that begin with *11*. The computation of M does not terminate for any input string beginning with *0*. The encoded transitions of M are given in the table below.

Transition	Encoding
$\delta(q_0, B) = [q_1, B, R]$	*101110110111011*
$\delta(q_1, 0) = [q_0, 0, L]$	*1101010101*
$\delta(q_1, 1) = [q_2, 1, R]$	*110110111011011*
$\delta(q_2, 1) = [q_0, 1, L]$	*1110110101101*

The machine M is represented by the string

000101110110111011001101010101001101101110110110011101101011010000. □

A Turing machine can be constructed to determine whether an arbitrary string $u \in \{0, 1\}^*$ is the encoding of a deterministic Turing machine. The computation examines u to see if it consists of a prefix *000* followed by a finite sequence of encoded transitions separated by *00*'s followed by *000*. A string that satisfies these conditions is the representation of some Turing machine M. The machine M is deterministic if the combination of the state and input symbol in every encoded transition is distinct.

Utilizing the encoding described above, the representation of a Turing machine with input alphabet $\{0, 1\}$ is itself a string over $\{0, 1\}$. The proof of Theorem 11.3.1 does not depend upon the features of this particular encoding. The argument is valid for any representation that encodes a Turing machine as a string over its input alphabet. The representation of the machine M is denoted $R(\mathrm{M})$.

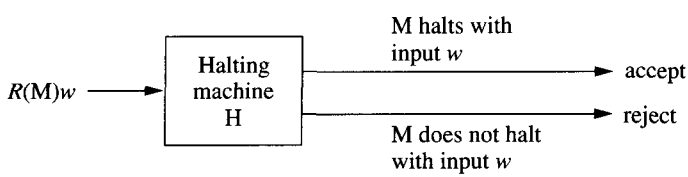
Theorem 11.3.1
The halting problem for Turing machines is undecidable.

Proof The proof is by contradiction. Assume that there is a Turing machine H that solves the halting problem. A string is accepted by H if

i) the input consists of the representation of a Turing machine M followed by a string w

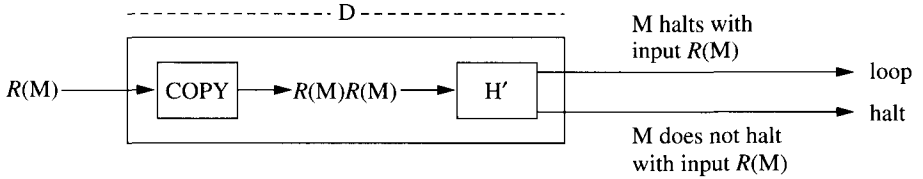
ii) the computation of M with input w halts.

If either of these conditions is not satisfied, H rejects the input. The operation of the machine H is depicted by the diagram

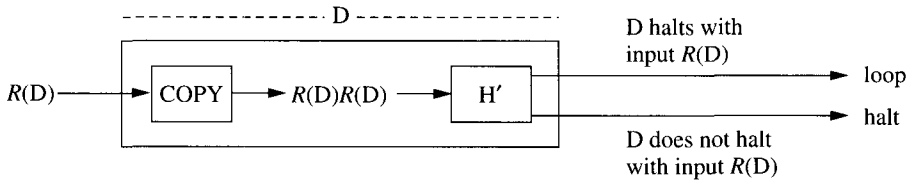


The machine H is modified to construct a Turing machine H′. The computations of H′ are the same as H except H′ loops indefinitely whenever H terminates in an accepting state, that is, whenever M halts on input w. The transition function of H′ is constructed from that of H by adding transitions that cause H′ to move indefinitely to the right upon entering an accepting configuration of H.

H′ is combined with a copy machine to construct another Turing machine D. The input to D is a Turing machine representation $R(M)$. A computation of D begins by creating the string $R(M)R(M)$ from the input $R(M)$. The computation continues by running H′ on $R(M)R(M)$.



The input to the machine D may be the representation of any Turing machine with alphabet $\{0, 1, B\}$. In particular, D is such a machine. Consider a computation of D with input $R(D)$. Rewriting the previous diagram with M replaced by D and $R(M)$ by $R(D)$, we get



Examining the preceding computation, we see that D halts with input $R(D)$ if, and only if, D does not halt with input $R(D)$. This is obviously a contradiction. However, the machine D can be constructed directly from a machine H that solves the halting problem. The assumption that the halting problem is decidable produces the preceding contradiction. Therefore, we conclude that the halting problem is undecidable. ■

Corollary 11.3.2

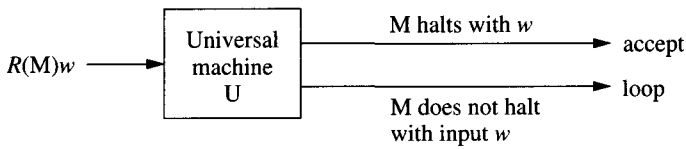
The language $L_H = \{R(M)w \mid R(M)$ is the representation of a Turing machine M and M halts with input $w\}$ over $\{0, 1\}^*$ is not recursive.

A similar argument can be used to establish the undecidability of the halting problem for Turing machines with arbitrary alphabets. The essential feature of this approach is the ability to encode the transitions of a Turing machine as a string over its own input alphabet. Two symbols are sufficient to construct such an encoding.

11.4 A Universal Machine

The halting problem provides a negative result concerning the ability to determine the outcome of a Turing machine computation from a description of the machine. The undecidability of the halting problem establishes that the language L_H, consisting of all strings $R(M)w$ for which machine M halts with input w, is not recursive. We will exhibit a single machine U that accepts the input $R(M)w$ whenever the computation of M halts with input w. Note that this machine does not solve the halting problem since U is not guaranteed to reject strings that are not in L_H. In fact, a computation of U with input $R(M)w$ will continue indefinitely whenever M does not halt with input w.

The Turing machines in this section are assumed to be deterministic with tape alphabet $\{0, 1, B\}$. The machine U is called a **universal Turing machine** since the outcome of the computation of any machine M with input w can be obtained by the computation of U with input $R(M)w$. The universal machine alone is sufficient to obtain the results of the computations of the entire family of Turing machines.



Theorem 11.4.1
The language L_H is recursively enumerable.

Proof A deterministic three-tape machine U is designed to accept L_H by halting. A computation of U begins with the input on tape 1. The encoding scheme presented in Section 11.3 is used to represent the input Turing machine. If the input string has the form $R(M)w$, the computation of M with input w is simulated on tape 3. The universal machine uses the information encoded in the representation $R(M)$ to simulate the transitions of M. A computation of U consists of the following actions:

1. If the input string does not have the form $R(M)w$ for a deterministic Turing machine M and string w, U moves indefinitely to the right.
2. The string w is written on tape 3 beginning at position one. The tape head is then repositioned at the leftmost square of the tape. The configuration of tape 3 is the initial configuration of a computation of M with input w.
3. A single *1*, the encoding of state q_0, is written on tape 2.
4. A transition of M is simulated on tape 3. The transition of M is determined by the symbol scanned on tape 3 and the state encoded on tape 2. Let x be the symbol from tape 3 and q_i the state encoded on tape 2.

a) Tape 1 is scanned for a transition whose first two components match $en(q_i)$ and $en(x)$. If there is no such transition, U halts accepting the input.

b) Assume tape 1 contains the encoded transition $en(q_i)0en(x)0en(q_j)0en(y)0en(d)$. Then

i) $en(q_i)$ is replaced by $en(q_j)$ on tape 2.

ii) The symbol y is written on tape 3.

iii) The tape head of tape 3 is moved in the direction specified by d.

5. The next transition of M is simulated by repeating steps 4 and 5.

The simulations of the universal machine U accept the strings in L_H. The computations of U loop indefinitely for strings in $\{0, 1\}^* - L_H$. Since $L_H = L(U)$, L_H is recursively enumerable. ■

Corollary 11.4.2

The recursive languages are a proper subset of the recursively enumerable languages.

Proof The acceptance of L_H by the universal machine demonstrates that L_H is recursively enumerable while Corollary 11.3.2 established that L_H is not recursive. ■

In Exercise 9.25 it was shown that a language L is recursive if both L and $\overline{L}$ are recursively enumerable. Combining this with Theorem 11.4.1 and Corollary 11.3.2 yields

Corollary 11.4.3

The language $\overline{L_H}$ is not recursively enumerable.

The computation of the universal machine U with input $R(M)$ and w simulates the computation M with input w. The ability to obtain the results of one machine via the computations of another facilitates the design of complicated Turing machines. When we say that a Turing machine M′ "runs machine M with input w," we mean that M′ is supplied with input $R(M)w$ and simulates the computation of M.

11.5 Reducibility

A decision problem **P** is many-to-one reducible to a problem **P′** if there is a Turing machine that takes any problem $\mathbf{p}_i \in \mathbf{P}$ as input and produces an associated problem $\mathbf{p}'_i \in \mathbf{P}'$, where the answer to the original problem $\mathbf{p}_i$ can be obtained from the answer to $\mathbf{p}'_i$. As the name implies, the mapping from **P** to **P′** need not be one-to-one: Multiple problems in **P** may be mapped to the same problem in **P′**.

If a decision problem **P′** is decidable and **P** is reducible to **P′**, then **P** is also decidable. A solution to **P** can be obtained by combining the reduction with the algorithm that solves **P′** (Figure 11.2).

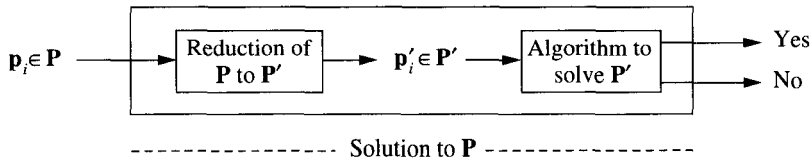


FIGURE 11.2 Solution of p_i using reduction to p'_i.

Reduction is a technique commonly employed in problem solving. When faced with a new problem, we often try to tranform it into a problem that has previously been solved. This is precisely the strategy employed in the reduction of decision problems.

Example 11.5.1

Consider the problem **P** of accepting strings in the language $L = \{uu \mid u = a^i b^i c^i \text{ for some } i \geq 0\}$. The machine M in Example 9.2.2 accepts the language $\{a^i b^i c^i \mid i \geq 0\}$. We will reduce the problem **P** to that of recognizing a single instance of $a^i b^i c^i$. The original problem can then be solved using the reduction and the machine M. The reduction is obtained as follows:

1. The input string w is copied.
2. The copy of w is used to determine whether $w = uu$ for some string $u \in \{a, b, c\}^*$.
3. If $w \neq uu$, then the tape is erased and a single a is written in the input position.
4. If $w = uu$, then the tape is erased, leaving u in the input position.

If the input string w has the form uu, then $w \in L$ if, and only if, $u = a^i b^i c^i$ for some i. The machine M has been designed to answer precisely this question. On the other hand, if $w \neq uu$, the reduction produces the string a. This string is subsequently rejected by M, indicating that the input $w \notin L$. □

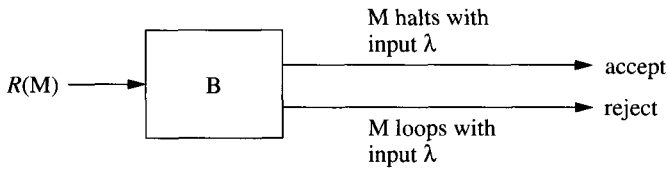
Reduction has important implications for undecidability as well as decidability. Assume that **P** is undecidable and that **P** is reducible to a problem **P′**. Then **P′** is also undecidable. If **P′** were decidable, the algorithm that solves **P′** could be used to construct a decision procedure for **P**.

The **blank tape problem** is the problem of deciding whether a Turing machine halts when a computation is initiated with a blank tape. The blank tape problem is a special case of the halting problem since it is concerned only with the question of halting when the input is the null string. We will show that the halting problem is reducible to the blank tape problem.

Theorem 11.5.1

There is no algorithm that determines whether an arbitrary Turing machine halts when a computation is initiated with a blank tape.

Proof Assume that there is a machine B that solves the blank tape problem. Such a machine can be represented

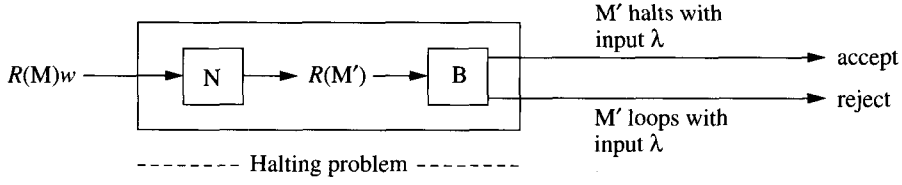


The reduction of the halting problem to the blank tape problem is accomplished by a machine N. The input to N is the representation of a Turing machine M followed by an input string w. The result of a computation of N is the representation of a machine M′ that

1. writes w on a blank tape
2. returns the tape head to the initial position with the machine in the initial state of M
3. runs M.

$R(\text{M}')$ is obtained by adding encoded transitions to $R(\text{M})$ and suitably renaming the start state of M. The machine M′ has been constructed so that it halts when run with a blank tape if, and only if, M halts with input w.

A new machine is constructed by adding N as a preprocessor to B. Sequentially running the machines N and B produces the composite machine

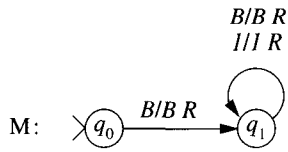


Tracing a computation, we see that the composite machine solves the halting problem. Since the preprocessor N reduces the halting problem to the blank tape problem, the blank tape problem is undecidable. ■

The input to the halting problem is the representation of a Turing machine followed by an input string. The preprocessor in the reduction modified this to construct the representation of a Turing machine that provides the input to the blank tape machine B. Example 11.5.2 details the actions of the preprocessor N.

Example 11.5.2

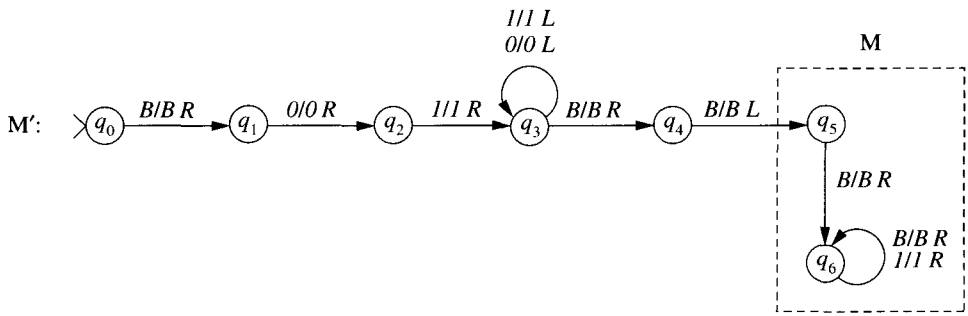
Let M be the Turing machine



that halts whenever the input string contains *0*. The encoding $R(\mathrm{M})$ of M is

$$0001011101101110110011011101101110110011011011011000.$$

With input $R(\mathrm{M})01$, the preprocessor N constructs the encoding of the Turing machine M′



When run with a blank tape, the first five states of M′ are used to write *01* in the input position. A copy of the machine M is then run with tape *B01B*. It is clear from the construction that M halts with input *01* if, and only if, M′ halts when run with a blank tape. □

The relationship between Turing machines and unrestricted grammars developed in Section 10.1 can be used to transfer the undecidability results from the domain of machines to the domain of grammars. Consider the problem of deciding whether a string w is generated by a grammar G. We can reduce the halting problem to the question of generation in an unrestricted grammar. Let M be a Turing machine and w an input string for M. First construct a Turing machine M′ that accepts every string for which M halts. This is accomplished by making every state of M an accepting state in M′. In M′, halting and accepting are synonymous.

Using Theorem 10.1.3, we can construct a grammar $\mathrm{G_{M'}}$ with $\mathrm{L(G_{M'})} = \mathrm{L(M')}$. An algorithm that decides whether $w \in \mathrm{L(G_{M'})}$ also determines whether the computation of M′ (and M) halts. Thus no such algorithm is possible.

11.6 Rice's Theorem

In the preceding section we determined that it was impossible to construct an algorithm to answer certain questions about a computation of an arbitrary Turing machine. The first example of this was the halting problem, which posed the question "Will a Turing machine M halt when run with input w?" Problem reduction allowed us to establish that there is no algorithm that answers the question "Will a Turing machine M halt when run with a blank tape?" In each of these problems, the input contained a Turing machine (or, more precisely, the representation of a Turing machine) and the decision problem was concerned with determining the result of the computation of the machine with a particular input.

Rather than asking about the computation of a Turing machine with a particular input string, we will now focus on determining whether the language accepted by an arbitrary Turing machine satisfies a prescribed property. For example, we might be interested in the existence of an algorithm that, when given a Turing machine M as input, produces an answer to

i) Is λ in L(M)?

ii) Is $L(M) = \emptyset$?

iii) Is L(M) a regular language?

iv) Is $L(M) = \Sigma^*$?

The ability to encode Turing machines permits us to transform the above questions into questions about membership in a language. As illustrated in Section 11.3, a Turing machine may be represented as a string over $\Sigma = \{0, 1\}$. Employing such an encoding, a set of Turing machines defines a language over Σ. The previous questions can be reformulated as questions of membership in the languages

i) $L_\lambda = \{R(M) \mid \lambda \in L(M)\}$

ii) $L_\emptyset = \{R(M) \mid L(M) = \emptyset\}$

iii) $L_{reg} = \{R(M) \mid L(M)$ is regular$\}$

iv) $L_{\Sigma^*} = \{R(M) \mid L(M) = \Sigma^*\}$.

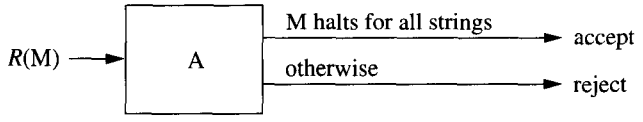
With the characterization of a set of Turing machines as a language over $\{0, 1\}$, the question of whether the set of strings accepted by a Turing machine M satisfies a property can be posed as a question of membership $R(M)$ in the appropriate language. Throughout this section, all Turing machines accept by halting.

Example 11.6.1

Problem reduction is used to show that membership in L_{Σ^*} is undecidable. In terms of Turing machines, the undecidability of membership in L_{Σ^*} indicates that it is impossible to construct an algorithm capable of determining whether an arbitrary Turing machine

accepts (halts for) every string. The proof provides another example of the utility of problem reduction in establishing undecidability.

Assume that there is a Turing machine A that decides membership in L_{Σ^*}. The input to such a machine is a string $v \in \{0, 1\}^*$. The input is accepted if $v = R(\mathrm{M})$ for some Turing machine M that halts for every input string. The output is rejected if either v is not the representation of a Turing machine or it is the representation of a machine that does not halt for some input string. The computation of machine A can be depicted by



Problem reduction is used to create a solution to the halting problem from the machine A.

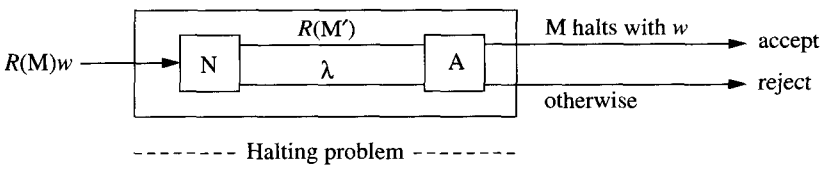
A solution to the halting problem accepts strings of the form $R(\mathrm{M})w$, where M is a machine that halts when run with input w, and rejects all other input strings. The reduction is accomplished by a machine N. The first action of N is to determine whether the input string has the expected format of the representation of some Turing machine M followed by a string w. If the input does not have this form, N erases the input, leaving the tape blank.

When the input has the form $R(\mathrm{M})w$, the computation of N constructs the encoding of a machine M′ that, when run with any input string y,

1. erases y from the tape
2. writes w on the tape
3. runs M on w.

$R(\mathrm{M}')$ is obtained from $R(\mathrm{M})$ by adding the encoding of two sets of transitions: one set that erases the input that is initially on the tape and another set that then writes the w in the input position. The machine M′ has been constructed to completely ignore its input. Every computation of M′ halts if, and only if, the computation of M with input w halts.

The machine consisting of the combination of N and A



provides a solution to the halting problem. Tracing the sequential operation of the machines, the input is accepted if, and only if, it is representation of a Turing machine M that halts when run with w.

Since the halting problem is undecidable and the reduction machine N is constructable, we must conclude that there is no machine A that decides membership in L_{Σ^*}.

□

The technique used in the preceding example can be generalized to show that many languages consisting of representations of Turing machines are not recursive. A property $\mathfrak{P}$ of recursively enumerable languages describes a condition that a recursively enumerable language may satisfy. For example, $\mathfrak{P}$ may be "The language contains the null string," "The language is the empty set," "The language is regular," or "The language contains all strings." The language of a property $\mathfrak{P}$ is defined by $L_{\mathfrak{P}} = \{R(M) \mid L(M) \text{ satisfies } \mathfrak{P}\}$. Thus $L_{\emptyset}$, the language associated with the property "The language is the empty set," consists of the representations of all Turing machines that do not accept any strings. A property $\mathfrak{P}$ of recursively enumerable languages is called *trivial* if there are no languages that satisfy $\mathfrak{P}$ or every recursively enumerable language satisfies $\mathfrak{P}$.

Theorem 11.6.1 (Rice's Theorem)
If $\mathfrak{P}$ is a nontrivial property of recursively enumerable languages, then $L_{\mathfrak{P}}$ is not recursive.

Proof Let $\mathfrak{P}$ be a nontrivial property that is not satisfied by the empty language. We will show that $L_{\mathfrak{P}} = \{R(M) \mid L(M) \text{ satisfies } \mathfrak{P}\}$ is not recursive.

Since $L_{\mathfrak{P}}$ is nontrivial, there is at least one language $L \in L_{\mathfrak{P}}$. Moreover, L is not $\emptyset$ by the assumption that the empty language does not satisfy $\mathfrak{P}$. Let M_L be a Turing machine that accepts L.

Reducibility to the halting problem will be used to show that $L_{\mathfrak{P}}$ is not recursive. As in Example 11.6.1, a preprocessor N will be designed to transform input $R(M)w$ into the encoding of a machine M′. The actions of M′ when run with input y are

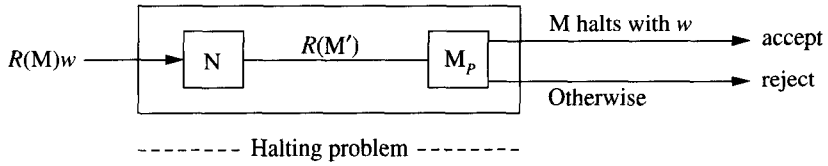
1. M′ writes w to the right of y, producing $ByBwB$
2. M′ runs M on w
3. if M halts when run with w, then M_L is run with input y.

The role of the machine M and the string w is that of a gatekeeper. The processing of the input string y by M_L is allowed only if M halts with input w.

If the computation of M halts when run with w, then M_L is allowed to process input y. In this case, the result of a computation of M′ with any input string y is exactly that of the computation of M_L with y. Consequently, $L(M') = L(M_L) = L$. If the computation of M does not halt when run with w, then M′ never halts regardless of the input string y. Thus no string is accepted by M′ and $L(M') = \emptyset$.

The machine M′ accepts $\emptyset$ when M does not halt with input w and M′ accepts L when M halts with w. Since L satisfies $\mathfrak{P}$ and $\emptyset$ does not, L(M′) satisfies $\mathfrak{P}$ if, and only if, M halts when run with input w.

Now assume that $L_{\mathfrak{P}}$ is recursive. Then there is a machine $M_{\mathfrak{P}}$ that decides membership in $L_{\mathfrak{P}}$. The machines N and $M_{\mathfrak{P}}$ combine to produce a solution to the halting problem.



Consequently, the property $\mathfrak{P}$ is not decidable.

Originally, we assumed that $\mathfrak{P}$ was not satisfied by the empty set. If $\emptyset \in L_{\mathfrak{P}}$, the preceding argument can be used to show that $\overline{L_{\mathfrak{P}}}$ is not recursive. It follows from Exercise 9.26 that $L_{\mathfrak{P}}$ must also be nonrecursive. ■

11.7 An Unsolvable Word Problem

A semi-Thue system, named after its originator, Norwegian mathematician Axel Thue, is a special type of grammar consisting of a single alphabet Σ and a set P of rules. A rule has the form $u \to v$, where $u \in \Sigma^+$ and $v \in \Sigma^*$. There is no division of the symbols into variables and terminals nor is there a designated start symbol. As before, $u \overset{*}{\Rightarrow} v$ signifies that v is derivable from u by a finite number of rule applications. The word problem for semi-Thue systems is the problem of determining, for an arbitrary semi-Thue system $S = (\Sigma, P)$ and strings $u, v \in \Sigma^*$, whether v is derivable from u in S.

We will show that the halting problem is reducible to the word problem for semi-Thue systems. The reduction is obtained by developing a relationship between Turing machine computations and derivations in appropriately designed semi-Thue systems.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a deterministic Turing machine. Using a modification of the construction presented in Theorem 10.1.3, we can construct a semi-Thue system $S_M = (\Sigma_M, P_M)$ whose derivations simulate the computations of M. The alphabet of S_M is the set $Q \cup \Gamma \cup \{[,], q_f, q_R, q_L\}$. The set P_M of rules of S_M is defined by

1. $q_i xy \to zq_j y$ whenever $\delta(q_i, x) = [q_j, z, R]$ and $y \in \Gamma$
2. $q_i x] \to zq_j B]$ whenever $\delta(q_i, x) = [q_j, z, R]$
3. $yq_i x \to q_j yz$ whenever $\delta(q_i, x) = [q_j, z, L]$ and $y \in \Gamma$
4. $q_i x \to q_R$ if $\delta(q_i, x)$ is undefined
5. $q_R x \to q_R$ for $x \in \Gamma$
6. $q_R] \to q_L]$
7. $xq_L \to q_L$ for $x \in \Gamma$
8. $[q_L \to [q_f$.

The rules that generate the string $[q_0 Bw]$ in Theorem 10.1.3 are omitted since the word problem for a semi-Thue system is concerned with derivability of a string v from another string u, not from a distinguished start symbol. The erasing rules (5 through 8)

have been modified to generate the string $[q_f]$ whenever the computation of M with input w halts.

The simulation of a computation of M in S_M manipulates strings of the form $[uqv]$ with $u, v \in \Gamma^*$, and $q \in Q \cup \{q_f, q_R, q_L\}$. Lemma 11.7.1 lists several important properties of derivations of S_M that simulate a computation of M.

Lemma 11.7.1

Let M be a deterministic Turing machine, S_M be the semi-Thue system constructed from M, and $w = [uqv]$ be a string with $u, v \in \Gamma^*$, and $q \in Q \cup \{q_f, q_R, q_L\}$.

i) There is at most one string z such that $w \underset{S_M}{\Rightarrow} z$.

ii) If there is such a z, then z also has the form $[u'q'v']$ with u', $v' \in \Gamma^*$, and $q' \in Q \cup \{q_f, q_R, q_L\}$.

Proof The application of a rule replaces one instance of an element of $Q \cup \{q_f, q_R, q_L\}$ with another. The determinism of M guarantees that there is at most one rule in P_M that can be applied to $[uqv]$ whenever $q \in Q$. If $q = q_R$ there is a unique rule that can be applied to $[uq_Rv]$. This rule is determined by the first symbol in the string $v]$. Similarly, there is only one rule that can be applied to $[uq_L]$. Finally, there are no rules in P_M that can be applied to a string containing q_f.

Condition (ii) follows immediately from the form of the rules of P_M. ■

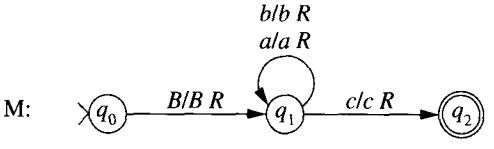
A computation of M that halts with input w produces a derivation $[q_0BwB] \underset{S_M}{\overset{*}{\Rightarrow}} [uq_Rv]$. The erasure rules transform this string to $[q_f]$. These properties are combined to yield Lemma 11.7.2

Lemma 11.7.2

A deterministic Turing machine M halts with input w if, and only if, $[q_0BwB] \underset{S_M}{\overset{*}{\Rightarrow}} [q_f]$.

Example 11.7.1

The language of the Turing machine



is $(a \cup b)^*c(a \cup b \cup c)^*$. The computation that accepts ac is given with the corresponding derivation of $[q_f]$ from $[q_0BacB]$ in the semi-Thue system S_M.

$$
\begin{aligned}
q_0BacB \quad & [q_0BacB] \\
\vdash Bq_1acB \quad & \Rightarrow [Bq_1acB] \\
\vdash Baq_1cB \quad & \Rightarrow [Baq_1cB] \\
\vdash Bacq_2B \quad & \Rightarrow [Bacq_2B] \\
& \Rightarrow [Bacq_R] \\
& \Rightarrow [Bacq_L] \\
& \Rightarrow [Baq_L] \\
& \Rightarrow [Bq_L] \\
& \Rightarrow [q_L] \\
& \Rightarrow [q_f]
\end{aligned}
$$

□

Theorem 11.7.3
The word problem for semi-Thue systems is undecidable.

Proof The preceding lemmas sketch the reduction of the halting problem to the word problem for semi-Thue systems. For a Turing machine M and corresponding semi-Thue system S_M, the computation of M with input w halting is equivalent to the derivability of $[q_f]$ from $[q_0BwB]$ in S_M. An algorithm that solves the word problem could also be used to solve the halting problem. ■

By Theorem 11.7.3, there is no algorithm that solves the word problem for an arbitrary semi-Thue system $S = (\Sigma, P)$ and pair of strings in Σ^*. The relationship between the computations of a Turing machine M and derivations of S_M developed in Lemma 11.7.2 can be used to prove that there are particular semi-Thue systems whose word problems are undecidable.

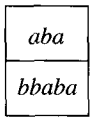
Theorem 11.7.4
Let M be a deterministic Turing machine that accepts a nonrecursive language. The word problem for the semi-Thue system S_M is undecidable.

Proof Since M recognizes a nonrecursive language, the halting problem for M is undecidable (Exercise 11). The correspondence between computations of M and derivations of S_M yields the undecidability of the word problem for this system. ■

11.8 The Post Correspondence Problem

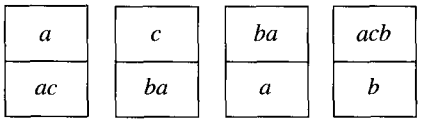
The undecidable problems presented in the preceding sections have been concerned with the properties of Turing machines or mathematical systems that simulate Turing machines. The Post correspondence problem is a combinatorial question that can be described as a

simple game of manipulating dominoes. A domino consists of two strings from a fixed alphabet, one on the top half of the domino and the other on the bottom.

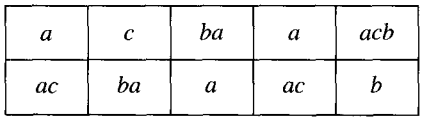


The game begins when one of the dominoes is placed on a table. Another domino is then placed to the immediate right of the domino on the table. This process is repeated, constructing a sequence of adjacent dominoes. A Post correspondence system can be thought of as defining a finite set of domino types. We assume that there is an unlimited number of dominoes of each type; playing a domino does not limit the number of future moves.

A string is obtained by concatenating the strings in the top halves of a sequence of dominoes. We refer to this as the top string. Similarly, a sequence of dominoes defines a bottom string. The game is successfully completed by constructing a finite sequence of dominoes in which the top and bottom strings are identical. Consider the Post correspondence system defined by dominoes



The sequence



is a solution to this Post correspondence system.

Formally, a **Post correspondence system** consists of an alphabet Σ and a finite set of ordered pairs $[u_i, v_i]$, $i = 1, 2, \ldots, n$, where $u_i, v_i \in \Sigma^+$. A solution to a Post correspondence system is a sequence $i_1, i_2, \ldots, i_k$ such that

$$u_{i_1} u_{i_2} \ldots u_{i_k} = v_{i_1} v_{i_2} \ldots v_{i_k}.$$

The problem of determining whether a Post correspondence system has a solution is the **Post correspondence problem**.

Example 11.8.1

The Post correspondence system with alphabet $\{a, b\}$ and ordered pairs $[aaa, aa]$, $[baa, abaaa]$ has a solution

aaa	*baa*	*aaa*
aa	*abaaa*	*aa*

□

Example 11.8.2

Consider the Post correspondence system with alphabet $\{a, b\}$ and ordered pairs $[ab, aba]$, $[bba, aa]$, $[aba, bab]$. A solution must begin with the domino

ab
aba

since this is the only domino in which prefixes on the top and bottom agree. The string in the top half of the next domino must begin with a. There are two possibilities:

ab	*ab*
aba	*aba*

(a)

ab	*aba*
aba	*bab*

(b)

The fourth elements of the strings in (a) do not match. The only possible way of constructing a solution is to extend (b). Employing the same reasoning as before, we see that the first element in the top of the next domino must be b. This lone possibility produces

ab	*aba*	*bba*
aba	*bab*	*aa*

which cannot be the initial subsequence of a solution since the seventh elements in the top and bottom differ. We have shown that there is no way of "playing the dominoes" in which the top and bottom strings are identical. Hence, this Post correspondence system has no solution. □

Theorem 11.8.1

There is no algorithm that determines whether an arbitrary Post correspondence system has a solution.

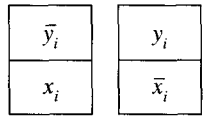
Proof Let $S = (\Sigma, P)$ be a semi-Thue system with alphabet $\{0, 1\}$ whose word problem is unsolvable. The existence of such a system is assured by Corollary 11.4.2 and Theorem 11.7.4.

For each pair of strings $u, v \in \Sigma^*$, we will construct a Post correspondence system $C_{u,v}$ that has a solution if, and only if, $u \overset{*}{\underset{S}{\Rightarrow}} v$. Since the latter problem is undecidable, there can be no general algorithm that solves the Post correspondence problem.

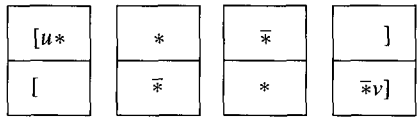
We begin by augmenting the set of productions of S with the rules $0 \to 0$ and $1 \to 1$. Derivations in the resulting system are identical to those in S except for the possible addition of rule applications that do not transform the string. The application of such a rule, however, guarantees that whenever $u \overset{*}{\underset{S}{\Rightarrow}} v$, v may be obtained from u by a derivation of even length. By abuse of notation, the augmented system is also denoted S.

Now let u and v be strings over $\{0, 1\}^*$. A Post correspondence system $C_{u,v}$ is constructed from u, v, and S. The alphabet of $C_{u,v}$ consists of 0, $\overline{0}$, 1, $\overline{1}$, [,], $*$, and $\overline{*}$. A string w consisting entirely of "barred" symbols is denoted $\overline{w}$.

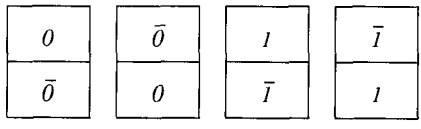
Each production $x_i \to y_i$, $i = 1, 2, \ldots, n$, of S (including $0 \to 0$ and $1 \to 1$) defines two dominoes



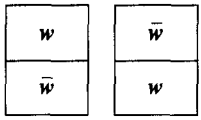
The system is completed by the dominoes



The dominoes



can be combined to form sequences dominoes that spell



for any string $w \in \{0, 1\}^*$. We will feel free to use these composite dominoes when constructing a solution to a Post correspondence system $C_{u,v}$.

First we show that $C_{u,v}$ has a solution whenever $u \overset{*}{\underset{S}{\Rightarrow}} v$. Let

$$u = u_0 \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$$

be a derivation of even length. The rules $0 \to 0$ and $1 \to 1$ ensure that there is derivation of even length whenever v is derivable from u. The ith step of the derivation can be written

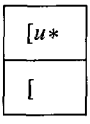
$$u_{i-1} = p_{i-1}x_{j_{i-1}}q_{i-1} \Rightarrow p_{i-1}y_{j_{i-1}}q_{i-1} = u_i,$$

where u_i is obtained from u_{i-1} by an application of the rule $x_{j_{i-1}} \to y_{j_{i-1}}$. The string

$$[u_0 * \bar{u}_1 \bar{*} u_2 * \cdots * \bar{u}_{k-1} \bar{*} u_k]$$

is a solution to $C_{u,v}$. This solution can be constructed as follows:

1. Initially play



2. To obtain a match, dominoes spelling the string $u = u_0$ on the bottom are played, producing

$[u*$	$\bar{p}_0$	$\bar{y}_{j_0}$	$\bar{q}_0$	$\bar{*}$
$[$	p_0	x_{j_0}	q_0	$*$

   The dominoes spelling p_0 and q_0 are composite dominoes. The middle domino is generated by the rule $x_{j_0} \to y_{j_0}$.
3. Since $p_0 y_{j_0} q_0 = u_1$, the top string can be written $[u_0 * \bar{u}_1$ and the bottom $[u_0$. Repeating the strategy employed above, dominoes must be played to spell $\bar{u}_1$ on the bottom,

$[u*$	$\bar{p}_0$	$\bar{y}_{j_0}$	$\bar{q}_0$	$\bar{*}$	p_1	y_{j_1}	q_1	$*$
$[$	p_0	x_{j_0}	q_0	$*$	$\bar{p}_1$	$\bar{x}_{j_1}$	$\bar{q}_1$	$\bar{*}$

producing $[u_0 * \bar{u}_1\bar{*}u_2*$ on the top.

4. This process is continued for steps 2, 3, . . . , $k - 1$ of the derivation, producing

$[u*$	$\bar{p}_0$	$\bar{y}_{j_0}$	$\bar{q}_0$	$\bar{*}$	p_1	y_{j_1}	q_1	$*$	. . .	p_{k-1}	$y_{j_{k-1}}$	q_{k-1}
$[$	p_0	x_{j_0}	q_0	$*$	$\bar{p}_1$	$\bar{x}_{j_1}$	$\bar{q}_1$	$\bar{*}$	. . .	$\bar{p}_{k-1}$	$\bar{x}_{j_{k-1}}$	$\bar{q}_{k-1}$

5. Completing the sequence with the domino

$]$
$\bar{*}v]$

produces the string $[u_0 * \bar{u}_1\bar{*}u_2 * \cdots * \bar{u}_{k-1}\bar{*}u_k]$ in both the top and the bottom, solving the correspondence system.

We will now show that a derivation $u \overset{*}{\Rightarrow} w$ can be constructed from a solution to the Post correspondence system $C_{u,v}$. A solution to $C_{u,v}$ must begin with

$[u*$
$[$

since this is the only domino whose strings begin with the same symbol. By the same argument, a solution must end with

$]$
$\bar{*}v]$

Thus the string spelled by a solution has the form $[u * w\bar{*}v]$. If w contains], then the solution can be written $[u * x\bar{*}v]y\bar{*}v]$. Since] occurs in only one domino and is the rightmost symbol on both the top and the bottom of that domino, the string $[u * x\bar{*}v]$ is also a solution of $C_{u,v}$.

In light of the previous observation, let $[u * \cdots *\overline{*}v]$ be a string that is a solution of the Post correspondence system $C_{u,v}$ in which] occurs only as the rightmost symbol. The information provided by the dominoes at the ends of a solution determines the structure of the entire solution. The solution begins with

$[u*$
$[$

A sequence of dominoes that spell u on the bottom must be played in order to match the string already generated on the top. Let $u = x_{i_1}x_{i_2}\ldots x_{i_k}$ be bottom strings in the dominoes that spell u in the solution. Then the solution has the form

$[u*$	$\overline{y}_{i_1}$	$\overline{y}_{i_2}$	$\overline{y}_{i_3}$	$\cdots$	$\overline{y}_{i_k}$	$\overline{*}$
$[$	x_{i_1}	x_{i_2}	x_{i_3}	$\cdots$	x_{i_k}	$*$

Since each domino represents a derivation $x_{i_j} \Rightarrow y_{i_j}$, we combine these to obtain the derivation $u \overset{*}{\Rightarrow} u_1$, where $u_1 = y_{i_1}y_{i_2}\ldots y_{i_k}$. The prefix of the top string of the dominoes that make up the solution has the form $[u * \overline{u}_1\overline{*}$, and the prefix of the bottom string is $[u*$. Repeating this process, we see that a solution defines a sequence of strings

$$[u * \overline{u}_1\overline{*}u_2 * \ldots \overline{*}v]$$
$$[u * \overline{u}_1\overline{*}u_2 * \overline{u}_3\overline{*}\ldots \overline{*}v]$$
$$[u * \overline{u}_1\overline{*}u_2 * \overline{u}_3\overline{*}u_4 * \ldots \overline{*}v]$$
$$\vdots$$
$$[u * \overline{u}_1\overline{*}u_2 * \overline{u}_3\overline{*}u_4 * \ldots \overline{u}_{k-1}\overline{*}v],$$

where $u_i \overset{*}{\Rightarrow} u_{i+1}$ with $u_0 = u$ and $u_k = v$. Combining these produces a derivation $u \overset{*}{\Rightarrow} v$.

The preceding two arguments constitute a reduction of the word problem for the semi-Thue system S to the Post correspondence problem. It follows that the Post correspondence problem is undecidable. ■

11.9 Undecidable Problems in Context-Free Grammars

Context-free grammars provide an important tool for defining the syntax of programming languages. The undecidability of the Post correspondence problem can be used to establish

the undecidability of several important questions concerning the languages generated by context-free grammars.

Let $C = (\Sigma_C, \{[u_1, v_1], [u_2, v_2], \ldots, [u_n, v_n]\})$ be a Post correspondence system. Two context-free grammars G_U and G_V are constructed from the ordered pairs of C.

$$
\begin{aligned}
G_U\colon\ & V_U = \{S_U\} \\
& \Sigma_U = \Sigma_C \cup \{1, 2, \ldots, n\} \\
& P_U = \{S_U \to u_i S_U i,\ S_U \to u_i i \mid i = 1, 2, \ldots, n\} \\
G_V\colon\ & V_V = \{S_V\} \\
& \Sigma_V = \Sigma_C \cup \{1, 2, \ldots, n\} \\
& P_V = \{S_V \to v_i S_V i,\ S_V \to v_i i \mid i = 1, 2, \ldots, n\}
\end{aligned}
$$

Determining whether a Post correspondence system C has a solution reduces to deciding the answers to certain questions concerning derivability in corresponding grammars G_U and G_V. The grammar G_U generates the strings that can appear in the top half of a sequence of dominoes. The digits in the rule record the sequence of dominoes that generate the string (in reverse order). Similarly, G_V generates the strings that can be obtained from the bottom half of a sequence of dominoes.

The Post correspondence system C has a solution if there is a sequence $i_1 i_2 \ldots i_{k-1} i_k$ such that

$$u_{i_1} u_{i_2} \ldots u_{i_{k-1}} u_{i_k} = v_{i_1} v_{i_2} \ldots v_{i_{k-1}} v_{i_k}.$$

In this case, G_U and G_V contain derivations

$$S_U \underset{G_U}{\overset{*}{\Rightarrow}} u_{i_1} u_{i_2} \ldots u_{i_{k-1}} u_{i_k} i_k i_{k-1} \ldots i_2 i_1$$

$$S_V \underset{G_V}{\overset{*}{\Rightarrow}} v_{i_1} v_{i_2} \ldots v_{i_{k-1}} v_{i_k} i_k i_{k-1} \ldots i_2 i_1,$$

where $u_{i_1} u_{i_2} \ldots u_{i_{k-1}} u_{i_k} i_k i_{k-1} \ldots i_2 i_1 = v_{i_1} v_{i_2} \ldots v_{i_{k-1}} v_{i_k} i_k i_{k-1} \ldots i_2 i_1$. Hence, the intersection of $L(G_U)$ and $L(G_V)$ is not empty.

Conversely, assume that $w \in L(G_U) \cap L(G_V)$. Then w consists of a string $w' \in \Sigma_C^+$ followed by a sequence $i_k i_{k-1} \ldots i_2 i_1$. The string $w' = u_{i_1} u_{i_2} \ldots u_{i_{k-1}} u_{i_k} = v_{i_1} v_{i_2} \ldots v_{i_{k-1}} v_{i_k}$ is a solution to C.

Theorem 11.9.1

There is no algorithm that determines whether the languages of two context-free grammars are disjoint.

Proof Assume there is such an algorithm. Then the Post correspondence problem could be solved as follows:

1. For an arbitrary Post correspondence system C, construct the grammars G_U and G_V from the ordered pairs of C.

2. Use the algorithm to determine whether $L(G_U)$ and $L(G_V)$ are disjoint.
3. C has a solution if, and only if, $L(G_U) \cap L(G_V)$ is nonempty.

Step 1 reduces the Post correspondence problem to the problem of determining whether two context-free languages are disjoint. Since the Post correspondence problem has already been shown to be undecidable, we conclude that the question of the intersection of context-free languages is also undecidable. ■

Example 11.9.1

The grammars G_U and G_V are constructed from the Post correspondence system $[aaa, aa]$, $[baa, abaaaa]$ from Example 11.8.1.

$$G_U: \; S_U \to aaaS_U1 \mid aaa1 \qquad G_V: \; S_V \to aaS_V1 \mid aa1$$
$$\to baaS_U2 \mid baa2 \qquad\qquad \to abaaaS_V2 \mid abaaa2$$

Derivations that exhibit the solution to the correspondence problem are

$$S_U \Rightarrow aaaS_U1 \qquad S_V \Rightarrow aaS_V1$$
$$\Rightarrow aaabaaS_U21 \qquad\qquad \Rightarrow aaabaaaS_V21$$
$$\Rightarrow aaabaaaaa121 \qquad\qquad \Rightarrow aaabaaaaa121.$$ □

The set of context-free languages is not closed under complementation. However, for an arbitrary Post correspondence system C, the languages $\overline{L(G_U)}$ and $\overline{L(G_V)}$ are context-free. The task of constructing context-free grammars that generate these languages is left as an exercise.

Theorem 11.9.2

There is no algorithm that determines whether the language of a context-free grammar $G = (V, \Sigma, P, S)$ is Σ^*.

Proof First, note that $L = \Sigma^*$ is equivalent to $\overline{L} = \emptyset$. We use this observation and show that there is no algorithm that determines whether $\overline{L(G)}$ is empty.

Again, let C be a Post correspondence system. Since $\overline{L(G_U)}$ and $\overline{L(G_V)}$ are context-free, so is $L = \overline{L(G_U)} \cup \overline{L(G_V)}$. Now, by DeMorgan's law, $\overline{L} = L(G_U) \cap L(G_V)$. An algorithm that determines whether $\overline{L} = \emptyset$ can also be used to determine whether $L(G_U)$ and $L(G_V)$ are disjoint. ■

Theorem 11.9.3

There is no algorithm that determines whether an arbitrary context-free grammar is ambiguous.

Proof A grammar is ambiguous if it contains a string that can be generated by two distinct leftmost derivations. As before, we begin with an arbitrary Post correspondence

system C and construct G_U and G_V. These grammars are combined to obtain the grammar

$$\begin{aligned} G: V &= \{S, S_U, S_V\} \\ \Sigma &= \Sigma_C \\ P &= P_U \cup P_V \cup \{S \to S_U, S \to S_V\} \end{aligned}$$

with start symbol S.

Clearly all derivations of G are leftmost; every sentential form contains at most one variable. A derivation of G consists of the application of an S rule followed by a derivation of G_U or G_V. The grammars G_U and G_V are unambiguous; distinct derivations generate distinct suffixes of integers. This implies that G is ambiguous if, and only if, $L(G_U) \cap L(G_V) \neq \emptyset$. But this condition is equivalent to the existence of a solution to the original Post correspondence system C. Since the Post correspondence problem is reducible to the problem of determining whether a context-free grammar is ambiguous, the latter problem is also undecidable. ■

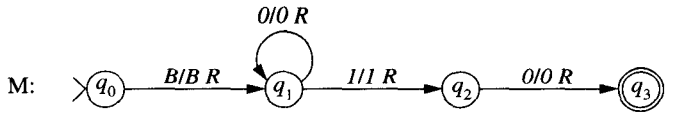
Exercises

In Exercises 1 through 5, describe a Turing machine that solves the specified decision problem. Use Example 11.1.2 as a model for defining the actions of a computation of the machine. You need not explicitly construct the transition function.

1. Design a two-tape Turing machine that determines whether two strings u and v over $\{0, 1\}$ are identical. The computation begins with $BuBvB$ on the tape and should require no more than $2(length(u) + 1)$ transitions.
2. Design a Turing machine whose computations decide whether a natural number is prime. Represent the natural number n by a sequence of $n + 1$ 1's.
3. Let $G = (V, \Sigma, P, S)$ be a regular grammar.
   a) Construct a representation for the grammar G over $\{0, 1\}$.
   b) Design a Turing machine that decides whether a string w is in L(G). The use of nondeterminism facilitates the construction of the desired machine.
4. A tour in a directed graph is a path $p_0, p_1, \ldots, p_n$ in which
   i) $p_0 = p_n$.
   ii) For $0 < i, j \leq n$, $i \neq j$ implies $p_i \neq p_j$.
   iii) Every node in the graph occurs in the path.

   Design a Turing machine that decides whether a directed graph contains a tour. Use the representation of a directed graph given in Example 11.1.2.
5. Design a Turing machine that solves the "2^n" problem. The representation of a natural number i is 1^{i+1}. The input is the representation of a natural number i and the output is yes if $i = 2^n$ for some n, no otherwise.

6. Let M be the Turing machine



   a) What is L(M)?

   b) Give the representation of M using the encoding from Section 11.3.

7. Construct a Turing machine that decides whether a string over $\{0, 1\}^*$ is the encoding of a nondeterministic Turing machine. What would be required to change this to a machine that decides whether the input is the representation of a deterministic Turing machine?

8. Design a Turing machine with input alphabet $\{0, 1\}$ that accepts an input string u if

   i) $u = R(\mathrm{M})w$ for some Turing machine M and input string w, and

   ii) when M is run with input w, there is a transition in the computation that prints a 1.

9. Given an arbitrary Turing machine M and input string w, will the computation of M with input w halt in fewer than 100 transitions? Describe a Turing machine that solves this decision problem.

10. The halting problem and the universal machine introduced in Sections 11.3 and 11.4 were concerned only with the halting of Turing machines. Consequently, the representation scheme $R(\mathrm{M})$ did not encode accepting states.

    a) Extend the representation $R(\mathrm{M})$ of a Turing machine M to explicitly encode the accepting states of M.

    b) Design a universal machine U_f that accepts input of the form $R(\mathrm{M})w$ if the machine M accepts input w by final state.

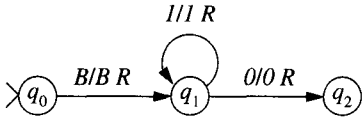
11. Let M be a deterministic Turing machine that accepts a nonrecursive language. Prove that the halting problem for M is undecidable. That is, there is no machine that takes input w and determines whether the computation of M halts with input w.

For Exercises 12 through 16, use reduction to establish the undecidability of the each of the decision problems.

12. Prove that there is no algorithm that determines whether an arbitrary Turing machine halts when run with the input string 101.

13. Prove that there is no algorithm that determines whether an arbitrary Turing machine halts for any input. That is, $R(\mathrm{M})$ is accepted if M halts for some string w. Otherwise, $R(\mathrm{M})$ is rejected.

14. Prove that there is no algorithm with input consisting of a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, a state $q_i \in Q$, and a string $w \in \Sigma^*$ that determines whether the computation of M with input w enters state q_i.
15. The computation of a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ with input w reenters the start state if the machine is in state q_0 at any time other than the initiation of the computation. Prove that there is no algorithm that determines whether a Turing machine reenters its start state.
16. Prove that there is no algorithm with input consisting of a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, a tape symbol x, and a string $w \in \Sigma^*$ that determines whether the computation of M with input w prints the symbol x.
17. Why can't we successfully argue that the blank tape problem is undecidable as follows: The blank tape problem is a subproblem of the halting problem which is undecidable and therefore must be undecidable itself.
18. Use Rice's theorem to show that the following properties of recursively enumerable languages are undecidable. To establish the undecidability, all you need do is show that the property is nontrivial.
    a) L contains a particular string w.
    b) L is finite.
    c) L is regular.
    d) L is $\{0, 1\}^*$.
19. Let $L = \{ R(M) \mid M \text{ halts when run with } R(M)\}$.
    a) Show that L is not recursive.
    b) Show that L is recursively enumerable.
20. Show that the language L_λ is recursively enumerable.
21. Show that the language $L_{\neq\emptyset} = \{R(M) \mid L(M) \text{ is nonempty}\}$ is recursively enumerable.
22. Give an example of a property of languages that is not satisfied by any recursively enumerable language.
23. Show that the property $\mathfrak{P}$, "L is context-free," is a nontrivial property of recursively enumerable languages.
24. Let M be the Turing machine



    a) Give the rules of the semi-Thue system S_M that simulate the computations of M.

b) Trace the computation of M with input *01* and give the corresponding derivation in S_M.

25. Find a solution for each of the following Post correspondence systems.
    a) $[a, aa]$, $[bb, b]$, $[a, bb]$
    b) $[a, aaa]$, $[aab, b]$, $[abaa, ab]$
    c) $[aa, aab]$, $[bb, ba]$, $[abb, b]$
    d) $[a, ab]$, $[ba, aba]$, $[b, aba]$, $[bba, b]$
26. Show that the following Post correspondence systems have no solutions.
    a) $[b, ba]$, $[aa, b]$, $[bab, aa]$, $[ab, ba]$
    b) $[ab, a]$, $[ba, bab]$, $[b, aa]$, $[ba, ab]$
    c) $[ab, aba]$, $[baa, aa]$, $[aba, baa]$
    d) $[ab, bb]$, $[aa, ba]$, $[ab, abb]$, $[bb, bab]$
    e) $[abb, ab]$, $[aba, ba]$, $[aab, abab]$
27. Prove that the Post correspondence problem for systems with a one-symbol alphabet is decidable.
28. Build the context-free grammars G_U and G_V that are constructed from the Post correspondence system $[b, bb]$, $[aa, baa]$, $[ab, a]$. Is $L(G_U) \cap L(G_V) = \emptyset$?
29. Let C be a Post correspondence system. Construct a context-free grammar that generates $\overline{L(G_U)}$.
30. Prove that there is no algorithm that determines whether the intersection of the languages of two context-free grammars contains infinitely many elements.
31. Prove that there is no algorithm that determines whether the complement of the language of a context-free grammar contains infinitely many elements.

Bibliographic Notes

Turing [1936] envisioned the theoretical computing machine he designed to be capable of performing all effective computations. This viewpoint, now known as the Church-Turing thesis, was formalized by Church [1936]. Turing's original paper also included the undecidability of the halting problem and the design of a universal machine. The proof of the undecidability of the halting problem presented in Section 11.3 is from Minsky [1967].

A proof that an arbitrary Turing machine can be simulated by a machine with tape alphabet $\{0, 1, B\}$ can be found in Hopcroft and Ullman [1979]. Techniques for establishing undecidability using properties of languages were presented in Rice [1953] and [1956]. The string transformation systems of Thue were introduced in Thue [1914]. The undecidability of the word problem for semi-Thue systems was established by Post [1947].

The undecidability of the Post correspondence problem was presented in Post [1946]. The proof of Theorem 11.8.1, based on the technique of Floyd [1964], is from Davis and Weyuker [1983]. Undecidability results for context-free languages, including Theorem 11.8.1, can be found in Bar-Hillel, Perles, and Shamir [1961]. The undecidability of ambiguity of context-free languages was established by Cantor [1962], Floyd [1962], and Chomsky and Schutzenberger [1963]. The question of inherent ambiguity was shown to be unsolvable by Ginsburg and Ullian [1966a].

CHAPTER 12

Numeric Computation

Turing machines have been used as a computational framework for constructing solutions to decision problems and 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 configuration permits an infinite number of possible outcomes. This technique will be used to construct Turing machines that compute number-theoretic functions.

12.1 Computation of Functions

A function $f : X \rightarrow Y$ can be thought of as a mapping that assigns at most one value of the range 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. Recall that the term function refers to both partial and total functions.

Definition 12.1.1

A deterministic one-tape Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$ computes the unary function $f : \Sigma^* \to \Sigma^*$ 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 $\delta(q_i, x) = [q_0, x, d]$ for any $q_i \in Q$, $x \in \Gamma$, and $d \in \{L, R\}$

iii) there are no transitions of the form $\delta(q_f, B)$

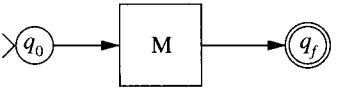
iv) the computation with input u halts in the configuration $q_f BvB$ whenever $f(u) = v$

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 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. Upon termination, the value of the function is written on the tape beginning at position one. The remainder of the tape is blank.

An arbitrary function need not have the same domain and range. Turing machines can be designed to compute functions from Σ^* to a specific set R by designating an input alphabet Σ 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

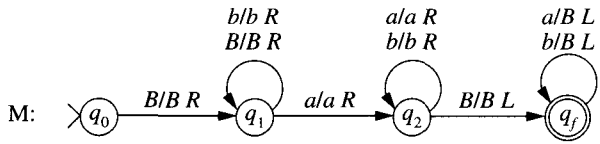


Intuitively, the computation remains inside the box labeled M until termination. This diagram is somewhat simplistic since Definition 12.1.1 permits multiple transitions to state q_f and transitions from q_f. However, condition (iii) ensures that there are no transitions from q_f when the machine is scanning a blank. When this occurs, the computation terminates with the result written on the tape.

Example 12.1.1

The Turing machine M computes the function f from $\{a, b\}^*$ to $\{a, b\}^*$ defined by

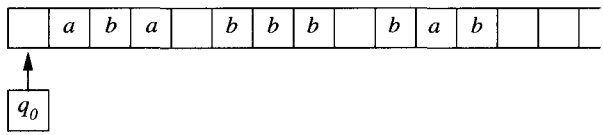
$$f(u) = \begin{cases} \lambda & \text{if } u \text{ contains an } a \\ \uparrow & \text{otherwise.} \end{cases}$$



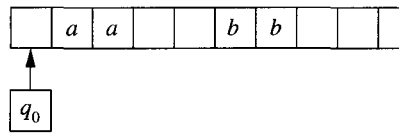
The function f is undefined if the input does not contain an a. In this case, the machine moves indefinitely to the right in state q_1. When an a is encountered, the machine enters state q_2 and reads the remainder of the input. The computation is completed by erasing the input while returning to the initial position. A computation that terminates produces the configuration $q_f BB$ designating the null string as the result. □

The machine M in Example 12.1.1 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 12.1.1 when the input does not have the anticipated form. A computation of M initiated with input $BbBbBaB$ terminates in the configuration $BbBbq_fB$. 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



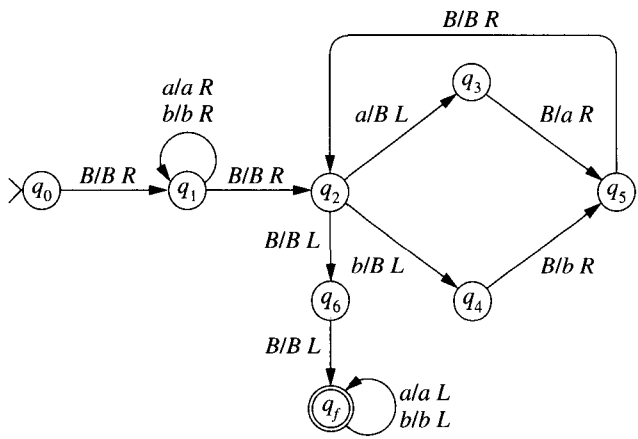
If $f(aba, bbb, bab)$ is defined, the computation terminates with the configuration $q_f B$ $f(aba, bbb, bab)B$. The initial configuration for the computation of $f(aa, \lambda, bb)$ is



The consecutive blanks in tape positions three and four indicate that the second argument is the null string.

Example 12.1.2

The Turing machine given below 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_fBuvB. □

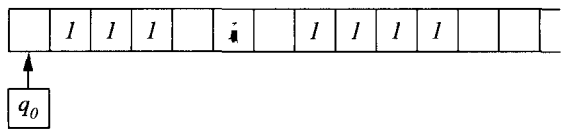
12.2 Numeric Computation

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 $f : \mathbf{N} \times \mathbf{N} \times \cdots \times \mathbf{N} \rightarrow \mathbf{N}$. 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 define binary number-theoretic functions.

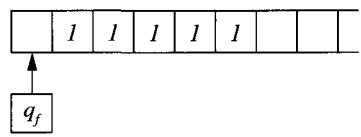
The transition from symbolic to numeric computation requires only a change of perspective since numbers are represented by strings of symbols. The input alphabet of the Turing machine is determined by the representation of the natural numbers used in the computation. We will represent the natural number n by the string 1^{n+1}. The number zero is represented by the string 1, the number one by 11, 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 $\overline{n}$. When numbers are encoded using the unary representation, the input alphabet for a machine that computes a number-theoretic function is the singleton set $\{1\}$.

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 $r : \mathbf{N} \times \mathbf{N} \times \cdots \times \mathbf{N} \to \{0, 1\}$ defines a k-ary relation R on the domain of the function. The relation is defined by

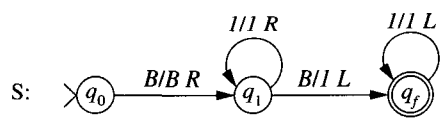
$$[n_1, n_2, \ldots, n_k] \in \mathrm{R} \text{ if } r(n_1, n_2, \ldots, n_k) = 1$$

$$[n_1, n_2, \ldots, n_k] \notin \mathrm{R} \text{ if } r(n_1, n_2, \ldots, n_k) = 0.$$

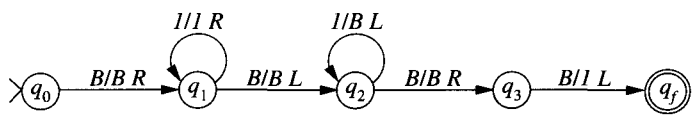
The function r is called the **characteristic function** of the relation R. A relation is Turing computable if its characteristic function is Turing computable.

Turing machines that compute several simple, but important, number-theoretic functions are given below. The functions are denoted by lowercase letters and the corresponding machines by capital letters.

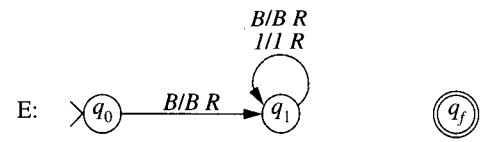
The successor function: $s(n) = n + 1$



The zero function: $z(n) = 0$

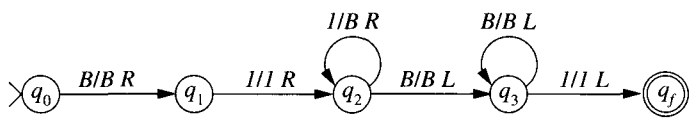


The empty function: $e(n) \uparrow$



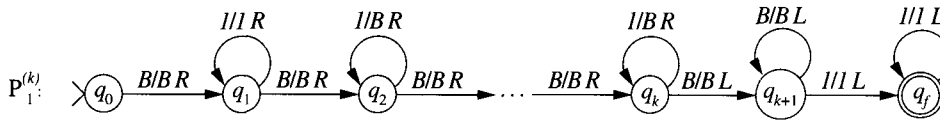
The machine that computes the successor simply adds a *1* to the right end of the input string. The zero function is computed by erasing the input and writing *1* in tape position one. The empty function is undefined for all arguments; the machine moves indefinitely to the right in state q_1.

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 maps 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 12.5 we will see that there are number-theoretic functions that cannot be computed by any Turing machine.

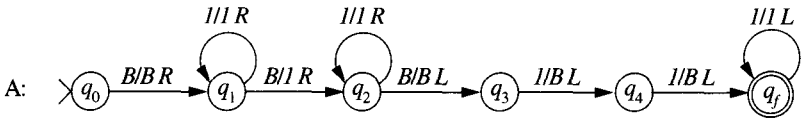
The value of the k-variable projection function $p_i^{(k)}$ is defined as the ith argument of the input, $p_i^{(k)}(n_1, n_2, \ldots, n_i, \ldots, n_k) = n_i$. The superscript k specifies the number of arguments and the subscript designates the argument that defines the result of the projection. The superscript is placed in parentheses so it is not mistaken for an exponent. The machine that computes $p_i^{(k)}$ leaves the first argument unchanged and erases the remaining arguments.



The function $p_1^{(1)}$ maps a single input to itself. This function is also called the *identity function* and is denoted id. Machines $\mathrm{P}_i^{(k)}$ that compute $p_i^{(k)}$ will be designed in Example 12.3.1.

Example 12.2.1

The Turing machine A computes the binary function defined by the addition of natural numbers.



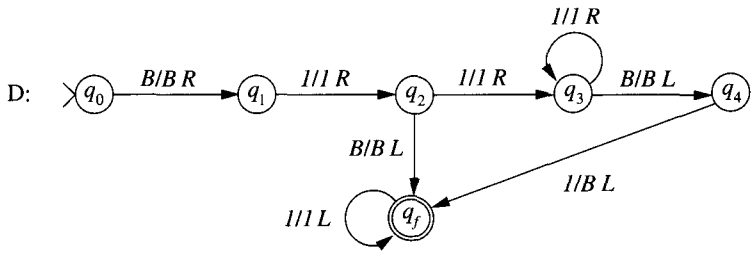
The unary representations of natural numbers n and m are 1^{n+1} and 1^{m+1}. The sum of these numbers is represented by 1^{n+m+1}. This string is generated by replacing the blank between the arguments with a *1* and erasing two *1*'s from the right end of the second argument. □

Example 12.2.2

The predecessor function

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

is computed by the machine D (decrement):



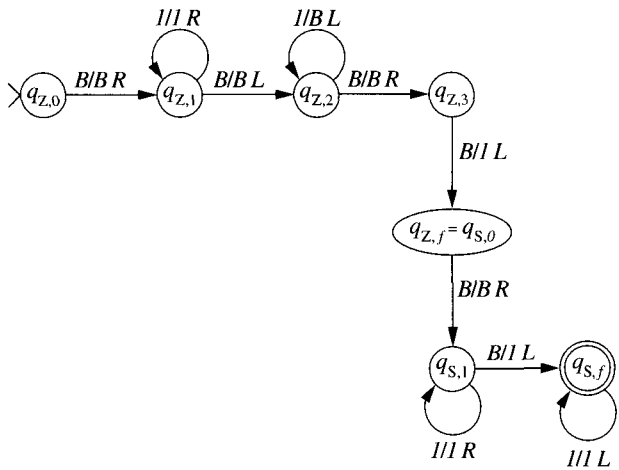
For input greater than zero, the computation erases the rightmost *1* on the tape. □

12.3 Sequential Operation of Turing Machines

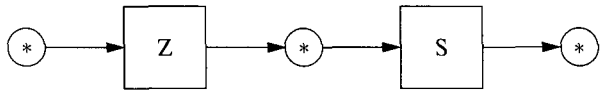
Turing machines designed to accomplish a single task can be combined to construct machines that perform complex computations. Intuitively, the combination is obtained by running the machines sequentially. The result of one computation becomes the input for the succeeding machine. A machine that computes the constant function $c(n) = 1$ can be constructed by combining the machines that compute the successor and the zero 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 12.1.1 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



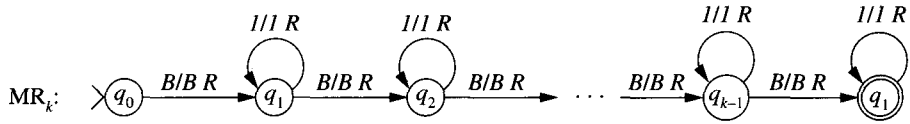
The state names are omitted from the initial and final nodes in the diagram since they may be inferred from the constituent machines.

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 12.1.1. 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 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 members (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 $B\overline{n}_1 q_0 B\overline{n}_2 B\overline{n}_3 B\overline{n}_4 B$ terminates in the configuration $B\overline{n}_1 B\overline{n}_2 B\overline{n}_3 q_f B\overline{n}_4 B$.

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 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):

$$\begin{array}{ll} B\overline{n}_1 B\overline{n}_2 B \ldots B\underline{\overline{n}_k B} & k \geq 0 \\ \updownarrow \qquad\qquad\qquad \updownarrow & \\ \underline{B\overline{n}_1} B\overline{n}_2 B \ldots B\overline{n}_k B & \end{array}$$

FR (find right):

$$\begin{array}{ll} \underline{B} B^i \overline{n} B & i \geq 0 \\ \updownarrow \quad \updownarrow & \\ B^i \underline{B\overline{n}} B & \end{array}$$

FL (find left):

$$\begin{array}{ll} B\overline{n}B^i\underline{B} & i \geq 0 \\ \updownarrow \quad \updownarrow & \\ \underline{B}\overline{n}B^iB & \end{array}$$

E_k (erase):

$$\begin{array}{lll} \underline{B}\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB & & k \geq 1 \\ \updownarrow \qquad\qquad\qquad \updownarrow & & \\ \underline{B}B \qquad \ldots \qquad BB & & \end{array}$$

CPY_k (copy):

$$\begin{array}{llll} \underline{B}\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kBBB & \ldots & BB & k \geq 1 \\ \updownarrow \qquad\qquad\qquad \updownarrow & & \updownarrow & \\ \underline{B}\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB & & & \end{array}$$

$CPY_{k,i}$ (copy through i numbers):

$$\begin{array}{llll} \underline{B}\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB\overline{n}_{k+1}\ldots B\overline{n}_{k+i}BB & \ldots & BB & k \geq 1 \\ \updownarrow \qquad\qquad \updownarrow \qquad\qquad\qquad \updownarrow & & \updownarrow & \\ \underline{B}\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB\overline{n}_{k+1}\ldots B\overline{n}_{k+i}B\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB & & & \end{array}$$

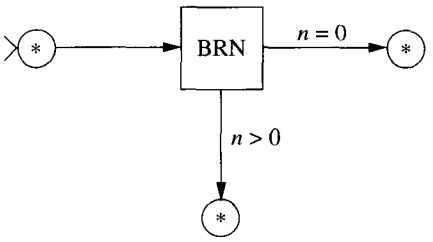
T (translate):

$$\begin{array}{ll} \underline{B}B^i\overline{n}B & i \geq 0 \\ \updownarrow \quad \updownarrow & \\ \underline{B}\overline{n}B^iB & \end{array}$$

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.

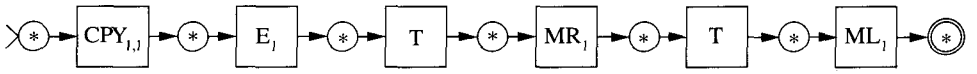
The copy machines produce a copy of the designated number of integers. The segment of the tape on which the copy is produced is assumed to be blank. $CPY_{k,i}$ expects a sequence of $k + i$ numbers followed by a blank segment large enough to hold a copy of the first k numbers. The translate macro changes 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 input to the macro BRN (branch on zero) is a single number. The value of the input is used to determine the halting state of the computation. 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 $\underline{B\overline{n}B}$. The branch macro is often used in the construction of loops in composite machines.

Additional macros can be created using those defined above. The machine



interchanges the order of two numbers. The tape configurations for this macro are

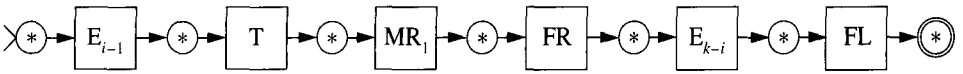
INT (interchange):

$$\begin{array}{c} \underline{B\overline{n}B\overline{m}BB^{n+1}B} \\ \updownarrow \qquad\qquad \updownarrow \\ \underline{B\overline{m}B\overline{n}BB^{n+1}B} \end{array}$$

In Exercise 3, you are asked to construct a Turing machine for the macro INT that does not leave the tape segment $B\overline{n}B\overline{m}B$.

Example 12.3.1

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 ith 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_i, MR_1, and T.

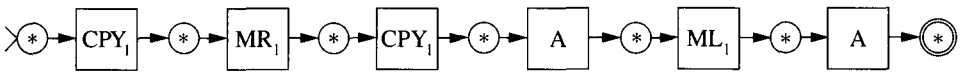


□

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 12.3.2

The macros and previously constructed machines can be used to design a Turing machine that computes the function $f(n) = 3n$.



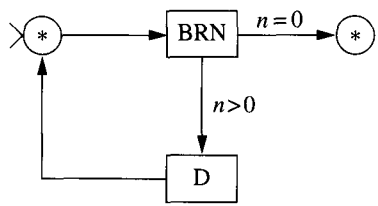
The initial state of the complete machine is that of the macro CPY_1. The machine A, constructed in Example 12.2.1, adds two natural numbers. A computation with input $\overline{n}$ generates the following sequence of tape configurations.

Machine	Configuration
	$\underline{B}\overline{n}B$
CPY_1	$\underline{B}\overline{n}B\overline{n}B$
MR_1	$B\overline{n}\underline{B}\overline{n}B$
CPY_1	$B\overline{n}\underline{B}\overline{n}B\overline{n}B$
A	$B\overline{n}\underline{B}\overline{n+n}B$
ML_1	$\underline{B}nB\overline{n+n}B$
A	$\underline{B}\overline{n+n+n}B$

Note that the addition machine A is run only when its arguments are the two rightmost encoded numbers on the tape. □

Example 12.3.3

The one-variable constant function zero defined by $z(n) = 0$, for all $n \in \mathbf{N}$, can be built from the BRN macro and the machine D that computes the predecessor function.

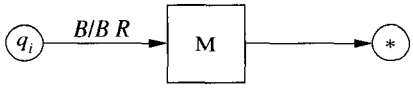


□

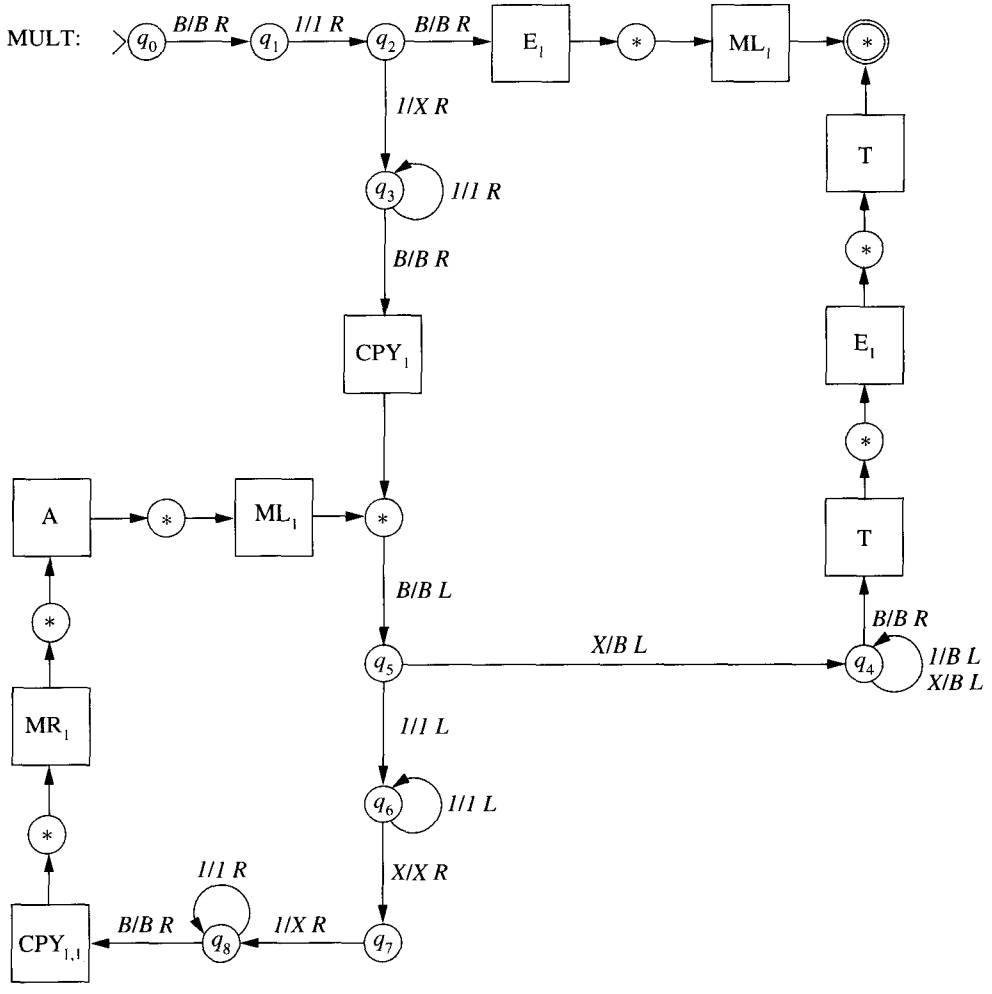
Example 12.3.4

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.

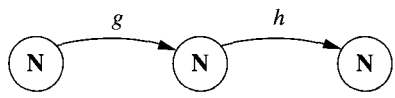


If the first argument is zero, the computation erases the second argument, returns to the initial position, and halts. Otherwise, a computation of MULT adds m to itself n times. The addition is performed by copying $\overline{m}$ and then adding the copy to the previous total. The number of iterations is recorded by replacing a 1 in the first argument with an X when a copy is made. □

12.4 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 12.4.1. Definition 12.4.2 extends the notion to n-variable functions.

Definition 12.4.1

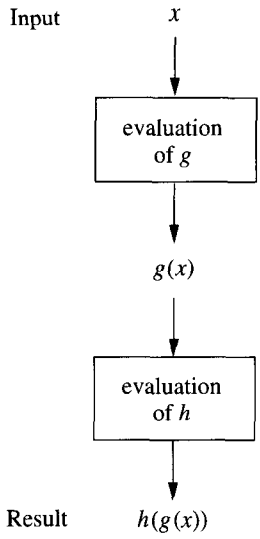
Let g and h be unary number-theoretic functions. The **composition** of h with g is the unary function $f : \mathbf{N} \to \mathbf{N}$ defined by

$$f(x) = \begin{cases} \uparrow & \text{if } g(x) \uparrow \\ \uparrow & \text{if } g(x) = y \text{ and } h(y) \uparrow \\ h(y) & \text{if } g(x) = y \text{ and } h(y) \downarrow. \end{cases}$$

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 is defined only when the preceding sequence of computations can be successfully completed.

Definition 12.4.2

Let $g_1, g_2, \ldots, g_n$ be k-variable number theoretic functions and let h be an n-variable number-theoretic function. The k-variable function f defined by

$$f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$$

is called the **composition** of h with $g_1, g_2, \ldots, g_n$ and written $f = h \circ (g_1, \ldots, g_n)$. $f(x_1, \ldots, x_k)$ is undefined if either

i) $g_i(x_1, \ldots, x_k) \uparrow$ for some $1 \leq i \leq n$

ii) $g_i(x_1, \ldots, x_k) = y_i$ for $1 \leq i \leq n$ and $h(y_1, \ldots, y_n) \uparrow$.

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 12.4.1

Consider the mapping defined by the composite function

$$add \circ (c_2^{(3)}, add \circ (p_1^{(3)}, p_3^{(3)})),$$

where $add(n, m) = n + m$ 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

$$\begin{aligned}
&add \circ (c_2^{(3)}, add \circ (p_1^{(3)}, p_3^{(3)}))(1,0,3) \\
&= add \circ (c_2^{(3)}(1,0,3), add \circ (p_1^{(3)}, p_3^{(3)})(1,0,3)) \\
&= add(2,\ add(p_1^{(3)}(1,0,3), p_3^{(3)}(1,0,3))) \\
&= add(2,\ add(1,3)) \\
&= add(2,4) \\
&= 6.
\end{aligned}$$

□

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**
	$\underline{B}\overline{n}_1 B\overline{n}_2 B\overline{n}_3 B$
CPY_3	$\underline{B}\overline{n}_1 B\overline{n}_2 B\overline{n}_3 B\overline{n}_1 B\overline{n}_2 B\overline{n}_3 B$
MR_3	$B\overline{n}_1 B\overline{n}_2 B\overline{n}_3 \underline{B}\overline{n}_1 B\overline{n}_2 B\overline{n}_3 B$
G_1	$B\overline{n}_1 B\overline{n}_2 B\overline{n}_3 \underline{B}\overline{g_1(n_1,n_2,n_3)}B$
ML_3	$\underline{B}\overline{n}_1 B\overline{n}_2 B\overline{n}_3 B\overline{g_1(n_1,n_2,n_3)}B$
$CPY_{3,1}$	$\underline{B}\overline{n}_1 B\overline{n}_2 B\overline{n}_3 B\overline{g_1(n_1,n_2,n_3)}B\overline{n}_1 B\overline{n}_2 B\overline{n}_3 B$
MR_4	$B\overline{n}_1 B\overline{n}_2 B\overline{n}_3 B\overline{g_1(n_1,n_2,n_3)}\underline{B}\overline{n}_1 B\overline{n}_2 B\overline{n}_3 B$
G_2	$B\overline{n}_1 B\overline{n}_2 B\overline{n}_3 B\overline{g_1(n_1,n_2,n_3)}\underline{B}\overline{g_2(n_1,n_2,n_3)}B$
ML_1	$B\overline{n}_1 B\overline{n}_2 B\overline{n}_3 \underline{B}\overline{g_1(n_1,n_2,n_3)}B\overline{g_2(n_1,n_2,n_3)}B$
H	$B\overline{n}_1 B\overline{n}_2 B\overline{n}_3 \underline{B}\overline{h(g_1(n_1,n_2,n_3), g_2(n_1,n_2,n_3))}B$
ML_3	$\underline{B}\overline{n}_1 B\overline{n}_2 B\overline{n}_3 B\overline{h(g_1(n_1,n_2,n_3), g_2(n_1,n_2,n_3))}B$
E_3	$\underline{B}B \quad \ldots \quad B\overline{h(g_1(n_1,n_2,n_3), g_2(n_1,n_2,n_3))}B$
T	$\underline{B}\overline{h(g_1(n_1,n_2,n_3), g_2(n_1,n_2,n_3))}B$

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.

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 12.4.3.

Theorem 12.4.3

The Turing computable functions are closed under the operation of composition.

Theorem 12.4.3 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 12.4.3, f is also Turing computable.

Example 12.4.2

The k-variable constant functions $c_i^{(k)}$ whose values are given by $c_i^{(k)}(n_1, \ldots, n_k) = i$ are Turing computable. The function $c_i^{(k)}$ can be defined by

$$c_i^{(k)} = \underbrace{s \circ s \circ \cdots \circ s}_{i \text{ times}} \circ z \circ p_1^{(k)}.$$

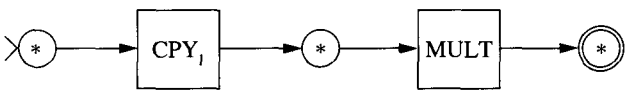
The projection function accepts the k-variable input and passes the first value to the zero function. The composition of i successor functions produces the desired value. Since each of the functions in the composition is Turing computable, the function $c_i^{(k)}$ is Turing computable by Theorem 12.4.3. □

Example 12.4.3

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 12.2.1 and sq by

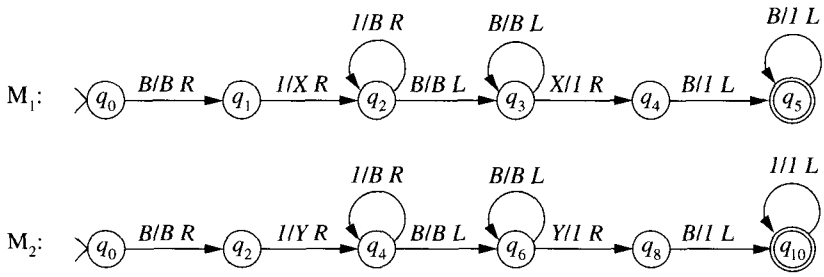


□

12.5 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 1 are immaterial, we adopt the following conventions concerning the naming of the components of a Turing machine:

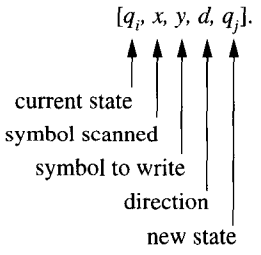
i) The set of states is a finite subset of $Q_0 = \{q_i \mid i \geq 0\}$.

ii) The input alphabet is $\{1\}$.

iii) The tape alphabet is a finite subset of the set $\Gamma_0 = \{B, 1, X_i \mid i \geq 0\}$.

iv) 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 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 12.4.2.

Theorem 12.5.1
The set of 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 12.5.1, we obtain Corollary 12.5.2.

Corollary 12.5.2
There is a total unary number-theoretic function that is not computable.

Corollary 12.5.2 vastly understates the relationship between computable and uncomputable functions. The former constitute a countable set and the latter an uncountable set.

12.6 Toward a Programming Language

High-level programming languages constitute a well-known family of computational systems. 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

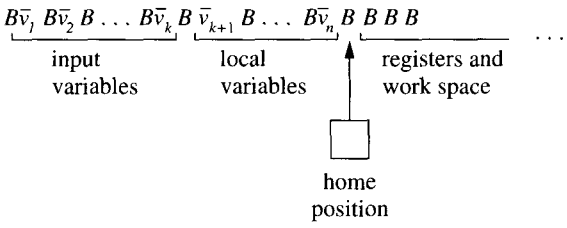


FIGURE 12.1 Turing machine architecture for high level computation.

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.

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 which, in turn, provides additional support for the Church-Turing thesis. In developing an 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. The assembly language TM is designed to bridge the gap between the Turing machine architecture and the 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 12.1. The input values are assigned to variables $v_1, \ldots, v_k$ and $v_{k+1}, \ldots, v_n$ are the local variables used in the program. The values of the variables are stored sequentially 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

TABLE 12.6.1 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

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, etc. The registers must be assigned sequentially; that is, register i may be written to or read from only if registers $1, 2, \ldots, i-1$ are assigned values. The instructions of the language TM are given in Table 12.6.1.

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 initiation, 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 3). The initialization phase of a program with one input and two local variables would produce the following sequence of Turing machine configurations:

Instruction	Configuration
	$\underline{B}\bar{i}B$
INIT 2	$\underline{B}\bar{i}B\bar{0}B$
INIT 3	$\underline{B}\bar{i}B\bar{0}B\bar{0}B$
HOME	$B\bar{i}B\bar{0}B\bar{0}\underline{B}$

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 alloted 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 $\underline{B}\bar{x}B\bar{y}B$ (Exercise 3).

The STOR command is defined by

Instruction	Definition
STOR $v_i, 1$	$\begin{pmatrix}\mathrm{ML}_1\\ \mathrm{INT}\end{pmatrix}^{n-i+1}$
	$\begin{pmatrix}\mathrm{MR}_1\\ \mathrm{INT}\end{pmatrix}^{n-i}$
	MR_1
	ER_1

Instruction	Definition
STOR v_i, t	MR_{t-2}
	INT
	$\begin{pmatrix}\mathrm{ML}_1\\ \mathrm{INT}\end{pmatrix}^{t+n-i-1}$
	$\begin{pmatrix}\mathrm{MR}_1\\ \mathrm{INT}\end{pmatrix}^{t+n-i-1}$
	MR_1
	ER_1
	ML_{t-1}

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
	$B\overline{v_1}B\overline{v_2}B\overline{v_3}\underline{B}\overline{r}B$
ML_1	$B\overline{v_1}B\overline{v_2}\underline{B}\overline{v_3}B\overline{r}B$
INT	$B\overline{v_1}B\overline{v_2}\underline{B}\overline{r}B\overline{v_3}B$
ML_1	$B\overline{v_1}\underline{B}\overline{v_2}B\overline{r}B\overline{v_3}B$
INT	$B\overline{v_1}\underline{B}\overline{r}B\overline{v_2}B\overline{v_3}B$
MR_1	$B\overline{v_1}B\overline{r}\underline{B}\overline{v_2}B\overline{v_3}B$
INT	$B\overline{v_1}B\overline{r}\underline{B}\overline{v_3}B\overline{v_2}B$
MR_1	$B\overline{v_1}B\overline{r}B\overline{v_3}\underline{B}\overline{v_2}B$
E_1	$B\overline{v_1}B\overline{r}B\overline{v_3}\underline{B}B$

The Turing machine implementation of the LOAD instructions 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 mentioned above, to load a value into register t requires registers $1, 2, \ldots, t-1$ to be filled. Thus the Turing machine must be in configuration

$$B\overline{v}_1\ B\overline{v}_2\ B\ldots B\overline{v}_k\ B\overline{v}_{k+1}\ B\ldots B\overline{v}_n\ B\overline{r}_1\ B\overline{r}_2\ B\ldots\ B\overline{r}_{t-1}\ B$$

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

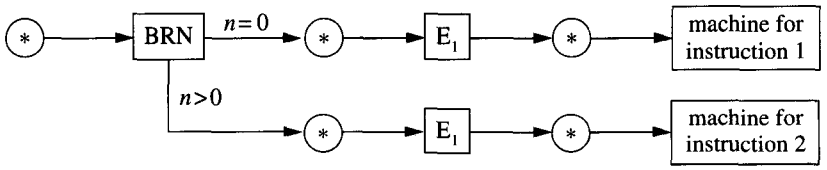
position and no registers allocated, the tape is rewritten placing the value of v_i in the Turing machine output position. CLEAR simply erases the value in the register.

Instruction	**Definition**
RETURN v_i	ML_n
	E_{i-1}
	T
	MR_1
	FR
	E_{n-i+1}
	FL
CLEAR	MR_{t-1}
	E_1
	ML_{t-1}

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 12.2.1. 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

```
        BRN L,1
        "instruction 1"
        ⋮
L       "instruction 2"
```



The value is tested, the register erased, and the machines that define the appropriate instruction are then executed.

Example 12.6.1

The TM program with one input variable and two local variables defined below computes the function $f(x_1) = 2x_1 + 1$. 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. □

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) $f(u) = \begin{cases} a & \text{if } length(u) \text{ is even} \\ b & \text{otherwise} \end{cases}$
   c) $f(u) = u^R$
   d) $f(u, v) = \begin{cases} u & \text{if } length(u) > length(v) \\ v & \text{otherwise} \end{cases}$
2. 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) $half(n) = \lfloor n/2 \rfloor$ 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) $even(n) = \begin{cases} 1 & \text{if } n \text{ is even} \\ 0 & \text{otherwise} \end{cases}$
   e) $eq(n, m) = \begin{cases} 1 & \text{if } n = m \\ 0 & \text{otherwise} \end{cases}$
   f) $lt(n, m) = \begin{cases} 1 & \text{if } n < m \\ 0 & \text{otherwise} \end{cases}$
   g) $n \dot{-} m = \begin{cases} n - m & \text{if } n \geq m \\ 0 & \text{otherwise} \end{cases}$
3. Construct Turing machines that perform the actions specified by the macros listed below. The computation should not leave the segment of the tape specified in the input configuration.
   a) ZR; input $\underline{B}BB$, output $\underline{B}\bar{0}B$
   b) FL; input $B\bar{n}B^i\underline{B}$, output $\underline{B}\bar{n}B^iB$
   c) E_2; input $\underline{B}\bar{n}B\bar{m}B$, output $\underline{B}B^{n+m+3}B$
   d) T; input $\underline{B}B^i\bar{n}B$, output $\underline{B}\bar{n}B^iB$
   e) BRN; input $\underline{B}\bar{n}B$, output $\underline{B}\bar{n}B$
   f) INT; input $\underline{B}\bar{n}B\bar{m}B$, output $\underline{B}\bar{m}B\bar{n}B$
4. Use the macros and machines constructed in Sections 12.2 through 12.4 to design machines that compute the following functions:

a) $f(n) = 2n + 3$

b) $f(n) = n^2 + 2n + 2$

c) $f(n_1, n_2, n_3) = n_1 + n_2 + n_3$

d) $f(n, m) = m^3$

e) $f(n_1, n_2, n_3) = n_2 + 2n_3$

5. Design machines that compute the following relations. You may use the macros and machines constructed in Sections 12.2 through 12.4 and the machines constructed in Exercise 2.

a) $gt(n, m) = \begin{cases} 1 & \text{if } n > m \\ 0 & \text{otherwise} \end{cases}$

b) $persq(n) = \begin{cases} 1 & \text{if } n \text{ is a perfect square} \\ 0 & \text{otherwise} \end{cases}$

c) $divides(n, m) = \begin{cases} 1 & \text{if } n > 0,\ m > 0, \text{ and } m \text{ divides } n \\ 0 & \text{otherwise} \end{cases}$

6. Trace the actions of the machine MULT for computations with input

a) $n = 0, m = 4$

b) $n = 1, m = 0$

c) $n = 2, m = 2$.

7. Describe the mapping defined by each of the following composite functions:

a) $add \circ (mult \circ (id, id), add \circ (id, id))$

b) $p_1^{(2)} \circ (s \circ p_1^{(2)}, e \circ p_2^{(2)})$

c) $mult \circ (c_2^{(3)}, add \circ (p_1^{(3)}, s \circ p_2^{(3)}))$

d) $mult \circ (mult \circ (p_1^{(1)}, p_1^{(1)}), p_1^{(1)})$

8. Give examples of total unary number-theoretic functions that satisfy the following conditions.

a) g is not id and h is not id but $g \circ h = id$

b) g is not a constant function and h is not a constant function but $g \circ h$ is a constant function

9. Give examples of unary number-theoretic functions that satisfy the following conditions:

a) g is not one-to-one, h is not total, $h \circ g$ is total

b) $g \neq e$, $h \neq e$, $h \circ g = e$, where e is the empty function

c) $g \neq id$, $h \neq id$, $h \circ g = id$, where id is the identity function

d) g is total, h is not one-to-one, $h \circ g = id$

10. Let F be a Turing machine that computes a total unary number-theoretic function f. Design a machine that returns the first natural number n such that $f(n) = 0$. A

computation should continue indefinitely if no such n exists. What will happen if the function computed by F is not total?

11. Let F be a Turing machine that computes a total unary number-theoretic function f. Design a machine that computes the function

$$g(n) = \sum_{i=0}^{n} f(i).$$

12. Let F and G be Turing machines that compute total unary number-theoretic functions f and g, respectively. Design a Turing machine that computes the function

$$h(n) = \sum_{i=0}^{n} eq(f(i), g(i)).$$

That is, $h(n)$ is the number of values in the range 0 to n for which the functions f and g assume the same value.

13. A unary relation R over **N** is Turing computable if its characteristic function is computable. Prove that every computable relation defines a recursive language. *Hint:* Construct a machine that accepts R from the machine that computes its characteristic function.
14. Let $R \subseteq \{1\}^+$ be a recursive language. Prove that R defines a computable relation on **N**.
15. Prove that there are unary relations over **N** that are not Turing computable.
16. Let F be the set consisting of all total unary number-theoretic functions that satisfy $f(i) = i$ for every even natural number i. Prove that there are functions in F that are not Turing computable.
17. Let L be a language over Σ and ch_L

$$ch_L(w) = \begin{cases} 1 & \text{if } w \in L \\ 0 & \text{otherwise} \end{cases}$$

be the characteristic function of L.

a) Let M be a Turing machine that computes ch_L. Prove that L is recursive.

b) If L is recursive, prove that there is a Turing machine that computes ch_L.

18. Let v_1, v_2, v_3, v_4 be a listing of the variables used in a TM program and assume register 1 contains a value. Trace the action of the instruction STOR v_2. To trace the actions, use the technique in Example 12.3.2.
19. Give a TM program that computes the function $f(v_1, v_2) = v_1 \dot{-} v_2$.

Bibliographic Notes

The Turing machine assembly language provides an architecture that resembles another family of abstract computing devices known as *random access machines*. Random access machines consist of an infinite number of memory locations and a finite number of registers, each of which is capable of storing a single integer. The instructions of a random access machine manipulate the registers and memory and perform arithmetic operations. These machines provide an abstraction of the standard von Neumann computer architecture. An introduction to random access machines and their equivalence to Turing machines can be found in Aho, Hopcroft, and Ullman [1974].

CHAPTER 13

Mu-Recursive Functions

Computable functions have been introduced from a mechanical perspective. Turing machines provide a microscopic approach to computability; the elementary operations of the Turing machine define an effective procedure for computing the values of a function. Computability is now presented from a macroscopic viewpoint. Rather than focusing on elementary operations, we make the functions themselves the fundamental objects of study. We introduce two families of functions, the primitive recursive functions and the μ-recursive functions. The primitive recursive functions are built from a set of intuitively computable functions using composition and primitive recursion. The μ-recursive functions are obtained by adding unbounded minimalization, a functional representation of sequential search, to the function building operations. Because of the emphasis on computability, the arguments of a function will be referred to as input and the evaluation of a function as a computation.

The computability of the primitive and μ-recursive functions is demonstrated by outlining an effective method for producing the values of the functions. The analysis of effective computation is completed by showing the equivalence of the notions of Turing computability and μ-recursivity.

13.1 Primitive Recursive Functions

A family of number-theoretic functions may be obtained from a set of basic functions and operators. Operators are used to combine the basic functions to construct additional functions. The set of basic primitive recursive functions consists of

i) the successor function s: $s(x) = x + 1$

ii) the zero function z: $z(x) = 0$

iii) the projection functions $p_i^{(n)}$: $p_i^{(n)}(x_1, \ldots, x_n) = x_i, 1 \leq i \leq n$.

The simplicity of the basic functions supports their intuitive computability. The successor function requires only the ability to add one to a natural number. Computing the zero function is even less complex; the value of the function is zero for every argument. The value of the projection function $p_i^{(n)}$ is obtained directly from its ith argument.

The primitive recursive functions are constructed from the basic functions by applications of two operations that preserve computability. The first operation is functional composition. Let f be defined by the composition of the n-variable function h with the k-variable functions $g_1, g_2, \ldots, g_n$. If each of the components of the composition is computable, then the value of $f(x_1, \ldots, x_k)$ can be obtained from h and $g_1(x_1, \ldots, x_k)$, $g_2(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k)$. The computability of f follows from the computability of its constituent functions.

The second method of generating functions is primitive recursion. Taken together, composition and primitive recursion provide a powerful tool for the construction of functions.

Definition 13.1.1

Let g and h be total number-theoretic functions with n and $n + 2$ variables, respectively. The $n + 1$-variable function f defined by

i) $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$

ii) $f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$

is said to be obtained from g and h by **primitive recursion**.

The x_i's are called the **parameters** of a definition by primitive recursion. The variable y is the **recursive variable**.

The operation of primitive recursion provides its own algorithm for computing the value of $f(x_1, \ldots, x_n, y)$ whenever g and h are computable. For a fixed set of parameters $x_1, \ldots, x_n$, $f(x_1, \ldots, x_n, 0)$ is obtained directly from the function g:

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$

The value $f(x_1, \ldots, x_n, y + 1)$ is obtained from the computable function h using

i) the parameters $x_1, \ldots, x_n$

ii) y, the previous value of the recursive variable

iii) $f(x_1, \ldots, x_n, y)$, the previous value of the function.

For example, $f(x_1, \ldots, x_n, y+1)$ is obtained by the sequence of computations

$$\begin{aligned} f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, 1) &= h(x_1, \ldots, x_n, 0, f(x_1, \ldots, x_n, 0)) \\ f(x_1, \ldots, x_n, 2) &= h(x_1, \ldots, x_n, 1, f(x_1, \ldots, x_n, 1)) \\ &\vdots \\ f(x_1, \ldots, x_n, y+1) &= h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)). \end{aligned}$$

Since h is computable, this iterative process can be used to determine $f(x_1, \ldots, x_n, y+1)$ for any value of the recursive variable y.

Definition 13.1.2

A function is **primitive recursive** if it can be obtained from the successor, zero, and projection functions by a finite number of applications of composition and primitive recursion.

A function defined by composition or primitive recursion from total functions is itself total. This is an immediate consequence of the definitions of the operations and is left as an exercise. Since the basic primitive recursive functions are total and the operations preserve totality, it follows that all primitive recursive functions are total.

Example 13.1.1

The constant functions $c_i^{(n)}(x_1, \ldots, x_n) = i$ are primitive recursive. Example 12.4.2 defines the constant functions as the composition of the successor, zero, and projection functions. □

Example 13.1.2

Let add be the function defined by primitive recursion from the functions $g(x) = x$ and $h(x, y, z) = z + 1$. Then

$$\begin{aligned} add(x, 0) &= g(x) = x \\ add(x, y+1) &= h(x, y, add(x, y)) = add(x, y) + 1 \end{aligned}$$

The function add computes the sum of two natural numbers. The definition of $add(x, 0)$ indicates that the sum of any number with zero is the number itself. The latter condition defines the sum of x and $y + 1$ as the sum of x and y (the result of add for the previous value of the recursive variable) incremented by one.

The preceding definition establishes that addition is primitive recursive. Both g and h, the components of the definition by primitive recursion, are primitive recursive since $g = p_1^{(1)}$ and $h = s \circ p_3^{(3)}$.

The value of add can be obtained from the primitive recursive definition by repeatedly applying the condition $add(x, y + 1) = add(x, y) + 1$ to reduce the value of the recursive variable. For example,

$$\begin{aligned} add(2,4) &= add(2,3) + 1 \\ &= (add(2,2) + 1) + 1 \\ &= ((add(2,1) + 1) + 1) + 1 \\ &= (((add(2,0) + 1) + 1) + 1) + 1 \\ &= (((2 + 1) + 1) + 1) + 1 \\ &= 6. \end{aligned}$$

When the recursive variable is zero, the function g is used to initiate the evaluation of the expression. □

Example 13.1.3

Let g and h be the primitive functions $g = z$ and $h = add \circ (p_3^{(3)}, p_3^{(3)})$. Multiplication can be defined by primitive recursion from g and h as follows:

$$mult(x, 0) = g(x) = 0$$
$$mult(x, y + 1) = h(x, y, mult(x, y)) = mult(x, y) + x$$

The infix expression corresponding to the primitive recursive definition is the identity $x \cdot (y + 1) = x \cdot y + x$, which follows from the distributive property of addition and multiplication. □

Adopting the convention that a zero-variable function is a constant, we can use Definition 13.1.1 to define one-variable functions using primitive recursion and a two-variable function h. The definition of such a function f has the form

i) $f(0) = n_0$ where $n_0 \in \mathbf{N}$

ii) $f(y + 1) = h(y, f(y))$.

Example 13.1.4

The one-variable factorial function defined by

$$fact(y) = \begin{cases} 1 & \text{if } y = 0 \\ \prod_{i=1}^{y} i & \text{otherwise} \end{cases}$$

is primitive recursive. Let $h(x, y) = mult \circ (p_2^{(2)}, s \circ p_1^{(2)}) = y \cdot (x + 1)$. The factorial function is defined using primitive recursion from h by

$$fact(0) = 1$$
$$fact(y + 1) = h(y, fact(y)) = fact(y) \cdot (y + 1).$$

Note that the definition uses $y + 1$, the value of the recursive variable. This is obtained by applying the successor function to y, the value provided to the function h.

The evaluation of the function *fact* for the first five input values illustrates how the primitive recursive definition generates the factorial function.

$$\begin{aligned} fact(0) &= 1 \\ fact(1) &= fact(0) \cdot (0 + 1) = 1 \\ fact(2) &= fact(1) \cdot (1 + 1) = 2 \\ fact(3) &= fact(2) \cdot (2 + 1) = 6 \\ fact(4) &= fact(3) \cdot (3 + 1) = 24 \end{aligned}$$ □

The primitive recursive functions were defined as a family of intuitively computable functions. As one might expect, these functions are also computable using our mechanical approach to effective computation.

Theorem 13.1.3

Every primitive recursive function is Turing computable.

Proof Turing machines that compute the basic functions were constructed in Section 12.2. To complete the proof it suffices to prove that the Turing computable functions are closed under composition and primitive recursion. The former was established in Section 12.4. All that remains is to show that the Turing computable functions are closed under primitive recursion; that is, if f is defined by primitive recursion from Turing computable functions g and h, then f is Turing computable.

Let g and h be Turing computable functions and let f be the function

$$\begin{aligned} f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, y + 1) &= h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) \end{aligned}$$

defined from g and h by primitive recursion. Since g and h are Turing computable, there are standard Turing machines G and H that compute them. A composite machine F is constructed to compute f. The computation of $f(x_1, x_2, \ldots, x_n, y)$ begins with tape configuration $B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{y}B$.

1. A counter, initially set to 0, is written to the immediate right of the input. The counter is used to record the value of the recursive variable for the current computation. The parameters are then written to the right of the counter, producing the tape configuration

$$B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{y}B\overline{0}B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B.$$

2. The machine G is run on the final n values of the tape, producing

$$B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{y}B\overline{0}B\overline{g(x_1, x_2, \ldots, x_n)}B.$$

The computation of G generates $g(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_n, 0)$.

3. The tape now has the form

$$B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{y} B\overline{i} B\overline{f(x_1, x_2, \ldots, x_n, i)} B.$$

If the counter i is equal to y, the computation of $f(x_1, x_2, \ldots, x_n, y)$ is completed by erasing the initial $n + 2$ numbers on the tape and translating the result to tape position one.

4. If $i < y$, the tape is configured to compute the next value of f.

$$B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{y} B\overline{i+1} B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{i} B\overline{f(x_1, x_2, \ldots, x_n, i)} B$$

The machine H is run on the final $n + 2$ values on the tape, producing

$$B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{y} B\overline{i+1} B\overline{h(x_1, x_2, \ldots, x_n, i, f(x_1, x_2, \ldots, x_n, i))} B,$$

where rightmost value on the tape is $f(x_1, x_2, \ldots, x_n, i + 1)$. The computation continues with the comparison in step 3. ■

13.2 Some Primitive Recursive Functions

A function is primitive recursive if it can be constructed from the zero, successor, and projection functions by a finite number of applications of composition and primitive recursion. Composition permits g and h, the functions used in a primitive recursive definition, to utilize any function that has previously been shown to be primitive recursive.

Primitive recursive definitions are constructed for several common arithmetic functions. Rather than explicitly detailing the functions g and h, a definition by primitive recursion is given in terms of the parameters, the recursive variable, the previous value of the function, and other primitive recursive functions. Note that the definitions of addition and multiplication are identical to the formal definitions given in Examples 13.1.2 and 13.1.3, with the intermediate step omitted.

Because of the compatibility with the operations of composition and primitive recursion, the definitions in Tables 13.2.1 and 13.2.2 are given using the functional notation. The standard infix representations of the binary arithmetic functions, given below the function names, are used in the arithmetic expressions throughout the chapter. The notation "+ 1" denotes the successor operator.

A primitive recursive predicate is a primitive recursive function whose range is the set $\{0, 1\}$. Zero and one are interpreted as false and true, respectively. The first two predicates in Table 13.2.2, the sign predicates, specify the sign of the argument. The function sg indicates whether the argument is positive. The complement of sg, denoted $cosg$, is true when the input is zero. Binary predicates that compare the input can be constructed from the arithmetic functions and the sign predicates using composition.

TABLE 13.2.1 Primitive Recursive Arithmetic Functions

Description	Function	Definition
addition	$add(x, y)$ $x + y$	$add(x, 0) = x$ $add(x, y + 1) = add(x, y) + 1$
multiplication	$mult(x, y)$ $x \cdot y$	$mult(x, 0) = 0$ $mult(x, y + 1) = mult(x, y) + x$
predecessor	$pred(y)$	$pred(0) = 0$ $pred(y + 1) = y$
proper subtraction	$sub(x, y)$ $x \dot{-} y$	$sub(x, 0) = x$ $sub(x, y + 1) = pred(sub(x, y))$
exponentation	$exp(x, y)$ x^y	$exp(x, 0) = 1$ $exp(x, y + 1) = exp(x, y) \cdot x$

Predicates are functions that exhibit the truth or falsity of a proposition. The logical operations negation, conjunction, and disjunction are constructed using the arithmetic functions and the sign predicates. Let p_1 and p_2 be two primitive recursive predicates. Logical operations on p_1 and p_2 can be defined as follows:

Predicate	Interpretation
$cosg(p_1)$	not p_1
$p_1 \cdot p_2$	p_1 and p_2
$sg(p_1 + p_2)$	p_1 or p_2

Applying *cosg* to the result of a predicate interchanges the values, yielding the negation of the predicate. This technique was used to define the predicate *ne* from the predicate *eq*. Determining the value of a disjunction begins by adding the truth values of the component predicates. Since the sum is 2 when both of the predicates are true, the disjunction is obtained by composing the addition with *sg*. The resulting predicates are primitive recursive since the components of the composition are primitive recursive.

Example 13.2.1

The equality predicates can be used to explicitly specify the value of a function for a finite set of arguments. For example, f is the identity function for all input values other than 0, 1, and 2:

TABLE 13.2.2 Primitive Recursive Predicates

Description	Predicate	Definition
sign	$sg(x)$	$sg(0) = 0$ $sg(y + 1) = 1$
sign complement	$cosg(x)$	$cosg(0) = 1$ $cosg(y + 1) = 0$
less than	$lt(x, y)$	$sg(y \dot{-} x)$
greater than	$gt(x, y)$	$sg(x \dot{-} y)$
equal to	$eq(x, y)$	$cosg(lt(x, y) + gt(x, y))$
not equal to	$ne(x, y)$	$cosg(eq(x, y))$

$$f(x) = \begin{cases} 2 & \text{if } x = 0 \\ 5 & \text{if } x = 1 \\ 4 & \text{if } x = 2 \\ x & \text{otherwise} \end{cases} \qquad \begin{aligned} f(x) = {} & eq(x,0) \cdot 2 \\ & + eq(x,1) \cdot 5 \\ & + eq(x,2) \cdot 4 \\ & + gt(x,2) \cdot x \end{aligned}$$

The function f is primitive recursive since it can be written as the composition of primitive recursive functions $eq, gt, +$, and $\cdot$. The four predicates in f are exhaustive and mutually exclusive; that is, one and only one of them is true for any natural number. The value of f is determined by the single predicate that holds for the input. □

The technique presented in the previous example, constructing a function from exhaustive and mutually exclusive primitive recursive predicates, is used to establish the following theorem.

Theorem 13.2.1

Let g be a primitive recursive function and f a total function that is identical to g for all but a finite number of input values. Then f is primitive recursive.

Proof Let g be primitive recursive and let f be defined by

$$f(x) = \begin{cases} y_1 & \text{if } x = n_1 \\ y_2 & \text{if } x = n_2 \\ \vdots & \\ y_k & \text{if } x = n_k \\ g(x) & \text{otherwise.} \end{cases}$$

The equality predicate is used to specify the values of f for input $n_1, \ldots, n_k$. For all other input values, $f(x) = g(x)$. The predicate obtained by the product

$$ne(x, n_1) \cdot ne(x, n_2) \cdot \cdots \cdot ne(x, n_k)$$

is true whenever the value of f is determined by g. Using these predicates, f can be written

$$f(x) = eq(x, n_1) \cdot y_1 + eq(x, n_2) \cdot y_2 + \cdots + eq(x, n_k) \cdot y_k + ne(x, n_1) \cdot ne(x, n_2) \cdot \cdots \cdot ne(x, n_k) \cdot g(x).$$

Thus f is also primitive recursive. ■

The order of the variables is an essential feature of a definition by primitive recursion. The initial variables are the parameters and the final variable is the recursive variable. Combining composition and the projection functions permits a great deal of flexibility in specifying the number and order of variables in a primitive recursive function. This flexibility is demonstrated by considering alterations to the variables in a two-variable function.

Theorem 13.2.2

Let $g(x, y)$ be a primitive recursive function. Then the functions obtained by

i) (adding dummy variables) $f(x, y, z_1, z_2, \ldots, z_n) = g(x, y)$
ii) (permuting variables) $f(x, y) = g(y, x)$
iii) (identifying variables) $f(x) = g(x, x)$

are primitive recursive.

Proof Each of the functions is primitive recursive since it can be obtained from g and the projections by composition as follows:

i) $f = g \circ (p_1^{(n+2)}, p_2^{(n+2)})$
ii) $f = g \circ (p_2^{(2)}, p_1^{(2)})$
iii) $f = g \circ (p_1^{(1)}, p_1^{(1)})$ ■

Dummy variables are used to make functions with different numbers of variables compatible for composition. The definition of the composition $h \circ (g_1, g_2)$ requires that g_1 and g_2 have the same number of variables. Consider the two-variable function f defined by

$f(x, y) = (x \cdot y) + x!$. The constituents of the addition are obtained from a multiplication and a factorial operation. The former function has two variables and the latter has one. Adding a dummy variable to the function *fact* produces a two-variable function *fact'* satisfying $fact'(x, y) = fact(x) = x!$. Finally, we note that $f = add \circ (mult, fact')$ so that f is also primitive recursive.

13.3 Bounded Operators

The sum of a sequence of natural numbers can be obtained by repeated applications of the binary operation of addition. Addition and projection can be combined to construct a function that adds a fixed number of arguments. For example, the primitive recursive function

$$add \circ (p_1^{(4)}, add \circ (p_2^{(4)}, add \circ (p_3^{(4)}, p_4^{(4)})))$$

returns the sum of its four arguments. This approach cannot be used when the number of summands is variable. Consider the function

$$f(y) = \sum_{i=0}^{y} g(i) = g(0) + g(1) + \cdot s + g(y).$$

The number of additions is determined by the input variable y. The function f is called the *bounded sum* of g. The variable i is the index of the summation. Computing a bounded sum consists of three actions: the generation of the summands, binary addition, and the comparison of the index with the input y.

We will prove that the bounded sum of a primitive recursive function is primitive recursive. The technique presented can be used to show that repeated applications of any binary primitive recursive operation is also primitive recursive.

Theorem 13.3.1

Let $g(x_1, \ldots, x_n, y)$ be a primitive recursive function. Then the functions

i) (bounded sum) $f(x_1, \ldots, x_n, y) = \sum_{i=0}^{y} g(x_1, \ldots, x_n, i)$

ii) (bounded product) $f(x_1, \ldots, x_n, y) = \prod_{i=0}^{y} g(x_1, \ldots, x_n, i)$

are primitive recursive.

Proof The sum

$$\sum_{i=0}^{y} g(x_1, \ldots, x_n, i)$$

is obtained by adding $g(x_1, \ldots, x_n, y)$ to

$$\sum_{i=0}^{y-1} g(x_1, \ldots, x_n, i).$$

Translating this into the language of primitive recursion, we get

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n, 0)$$
$$f(x_1, \ldots, x_n, y+1) = f(x_1, \ldots, x_n, y) + g(x_1, \ldots, x_n, y+1). \quad \blacksquare$$

The bounded operations just introduced begin with index zero and terminate when the index reaches the value specified by the argument y. Bounded operations can be generalized by having the range of the index variable determined by two computable functions. The functions l and u are used to determine the lower and upper bounds of the index.

Theorem 13.3.2

Let g be an $n+1$-variable primitive recursive function and let l and u be n-variable primitive recursive functions. Then the functions

i) $f(x_1, \ldots, x_n) = \sum_{i=l(x_1,\ldots,x_n)}^{u(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i)$

ii) $f(x_1, \ldots, x_n) = \prod_{i=l(x_1,\ldots,x_n)}^{u(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i)$

are primitive recursive.

Proof Since the lower and upper bounds of the summation are determined by the functions l and u, it is possible that the lower bound may be greater than the upper bound. When this occurs, the result of the summation is assigned the default value zero. The predicate

$$gt(l(x_1, \ldots, x_n), u(x_1, \ldots, x_n))$$

is true in precisely these instances.

If the lower bound is less than or equal to the upper bound, the summation begins with index $l(x_1, \ldots, x_n)$ and terminates when the index reaches $u(x_1, \ldots, x_n)$. Let g' be the primitive recursive function defined by

$$g'(x_1, \ldots, x_n, y) = g(x_1, \ldots, x_n, y + l(x_1, \ldots, x_n)).$$

The values of g' are obtained from those of g and $l(x_1, \ldots, x_n)$:

$$g'(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n, l(x_1, \ldots, x_n))$$
$$g'(x_1, \ldots, x_n, 1) = g(x_1, \ldots, x_n, 1 + l(x_1, \ldots, x_n))$$
$$\vdots$$
$$g'(x_1, \ldots, x_n, y) = g(x_1, \ldots, x_n, y + l(x_1, \ldots, x_n))$$

By Theorem 13.3.1, the function

$$f'(x_1, \ldots, x_n, y) = \sum_{i=0}^{y} g'(x_1, \ldots, x_n, i)$$
$$= \sum_{i=l(x_1,\ldots,x_n)}^{y+l(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i)$$

is primitive recursive. The generalized bounded sum can be obtained by composing f' with the functions u and l:

$$f'(x_1, \ldots, x_n, (u(x_1, \ldots, x_n) \dot{-} l(x_1, \ldots, x_n))) = \sum_{i=l(x_1,\ldots,x_n)}^{u(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i)$$

Multiplying this function by the predicate that compares the upper and lower bounds ensures that the bounded sum returns the default value whenever the lower bound exceeds the upper bound. Thus

$$f(x_1, \ldots, x_n) = cosg(gt(l(x_1, \ldots, x_n), u(x_1, \ldots, x_n)))$$
$$\cdot f'(x_1, \ldots, x_n, (u(x_1, \ldots, x_n) \dot{-} l(x_1, \ldots, x_n))).$$

Since each of the constituent functions is primitive recursive, it follows that f is also primitive recursive.

A similar argument can be used to show that the generalized bounded product is primitive recursive. When the lower bound is greater than the upper, the bounded product defaults to one. ■

The value returned by a predicate p designates whether the input satisfies the property represented by p. For fixed values $x_1, \ldots, x_n$,

$$\mu z[p(x_1, \ldots, x_n, z)]$$

is defined to be the smallest natural number z such that $p(x_1, \ldots, x_n, z) = 1$. The notation $\mu z[p(x_1, \ldots, x_n, z)]$ is read "the least z satisfying $p(x_1, \ldots, x_n, z)$." This construction is called the *minimalization* of p, and μz is called the μ-operator. The minimalization of an $n + 1$-variable predicate defines an n-variable function

$$f(x_1, \ldots, x_n) = \mu z[p(x_1, \ldots, x_n, z)].$$

A function defined by minimalization can be thought of as a search procedure. Initially, the variable z is set to zero. The search sequentially examines the natural numbers until a value of z for which $p(x_1, \ldots, x_n, z) = 1$ is encountered.

Unfortunately, the function obtained by the minimalization of a primitive recursive predicate need not be primitive recursive. In fact, such a function may not even be total. Consider the function

$$f(x) = \mu z[eq(x, z \cdot z)].$$

If x is a perfect square, then $f(x)$ returns the square root of x. Otherwise, f is undefined.

By restricting the range over which the minimalization occurs, we obtain a bounded minimalization operator. An $n + 1$-variable predicate defines an $n + 1$-variable function

$$\begin{aligned} f(x_1, \ldots, x_n, y) &= \overset{y}{\mu} z[p(x_1, \ldots, x_n, z)] \\ &= \begin{cases} z & \text{if } p(x_1, \ldots, x_n, i) = 0 \text{ for } 0 \leq i < z \leq y \\ & \quad \text{and } p(x_1, \ldots, x_n, z) = 1 \\ y + 1 & \text{otherwise.} \end{cases} \end{aligned}$$

The bounded μ-operator returns the first natural number z less than or equal to y for which $p(x_1, \ldots, x_n, z) = 1$. If no such value exists, the default value of $y + 1$ is assigned. Limiting the search to the range of natural members between zero and y ensures the totality of the function

$$f(x_1, \ldots, x_n, y) = \overset{y}{\mu} z[p(x_1, \ldots, x_n, z)].$$

In fact, the bounded minimalization operator defines a primitive recursive function whenever the predicate is primitive recursive.

Theorem 13.3.3

Let $p(x_1, \ldots, x_n, y)$ be a primitive recursive predicate. Then the function

$$f(x_1, \ldots, x_n, y) = \overset{y}{\mu} z[p(x_1, \ldots, x_n, z)]$$

is primitive recursive.

Proof The proof is given for a two-variable predicate $p(x, y)$. The technique presented easily generalizes to n-variable predicates. We begin by defining an auxiliary predicate

$$\begin{aligned} g(x, y) &= \begin{cases} 1 & \text{if } p(x, i) = 0 \text{ for } 0 \leq i \leq y \\ 0 & \text{otherwise} \end{cases} \\ &= \prod_{i=0}^{y} cosg(p(x, i)). \end{aligned}$$

This predicate is primitive recursive since it is a bounded product of the primitive recursive predicate $cosg \circ p$.

The bounded sum of the predicate g produces the bounded μ-operator. To illustrate the use of g in constructing the minimalization operator, consider a two-variable predicate p with argument n whose values are given below:

$$\begin{array}{lll}
p(n,0)=0 & g(n,0)=1 & \sum_{i=0}^{0} g(n,i)=1 \\
p(n,1)=0 & g(n,1)=1 & \sum_{i=0}^{1} g(n,i)=2 \\
p(n,2)=0 & g(n,2)=1 & \sum_{i=0}^{2} g(n,i)=3 \\
p(n,3)=1 & g(n,3)=0 & \sum_{i=0}^{3} g(n,i)=3 \\
p(n,4)=0 & g(n,4)=0 & \sum_{i=0}^{4} g(n,i)=3 \\
p(n,5)=1 & g(n,5)=0 & \sum_{i=0}^{5} g(n,i)=3 \\
\vdots & \vdots & \vdots
\end{array}$$

The value of g is one until the first number z with $p(n,z)=1$ is encountered. All subsequent values of g are zero. The bounded sum adds the results generated by g. Thus

$$\sum_{i=0}^{y} g(n,i) = \begin{cases} y+1 & \text{if } z > y \\ z & \text{otherwise.} \end{cases}$$

The first condition also includes the possibility that there is no z satisfying $p(n,z)=1$. In this case, the default value is returned regardless of the specified range.

By the preceding argument, we see that the bounded minimalization of a primitive recursive predicate p is given by the function

$$f(x,y) = \overset{y}{\mu} z[p(x,z)] = \sum_{i=0}^{y} g(x,i),$$

and consequently is primitive recursive. ■

Bounded minimalization can be generalized by generating the upper bound with a function u. When u is primitive recursive, so is the resulting function. The proof is similar to that of Theorem 13.3.2 and is left as an exercise.

Theorem 13.3.4
Let p be an $n+1$-variable primitive recursive predicate and let u be an n-variable primitive recursive function. Then the function

$$f(x_1, \ldots, x_n) = {}^{u(x_1,\ldots,x_n)}\mu z\,[p(x_1, \ldots, x_n, z)]$$

is primitive recursive.

13.4 Division Functions

The fundamental operation of integer division, *div*, is not total. The function $div(x, y)$ returns the quotient, the integer part of the division of x by y, when the second argument is nonzero. The function is undefined when y is zero. Since all primitive recursive functions are total, it follows that *div* is not primitive recursive. A primitive recursive division function *quo* is defined by assigning a default value when the denominator is zero:

$$quo(x, y) = \begin{cases} 0 & \text{if } y = 0 \\ div(x, y) & \text{otherwise.} \end{cases}$$

The division function *quo* is constructed using the primitive recursive operation of multiplication. For values of y other than zero, $quo(x, y) = z$ implies that z satisfies $z \cdot y \leq x < (z+1) \cdot y$. That is, $quo(x, y)$ is the smallest natural number z such that $(z+1) \cdot y$ is greater than x. The search for the value of z that satisfies the inequality succeeds before z reaches x since $(x+1) \cdot y$ is greater than x. The function

$$\overset{x}{\mu} z[gt((z+1) \cdot y, x)]$$

determines the quotient of x and y whenever the division is defined. The default value is obtained by multiplying the minimalization by $sg(y)$. Thus

$$quo(x, y) = sg(y) \cdot \overset{x}{\mu} z[gt((z+1) \cdot y, x)],$$

where the bound is determined by the primitive recursive function $p_1^{(2)}$. The previous definition demonstrates that *quo* is primitive recursive since it has the form prescribed by Theorem 13.3.4.

The quotient function can be used to define a number of division related functions and predicates (Table 13.4.1). The function *rem* returns the remainder of the division of x by y whenever the division is defined. Otherwise, $rem(x, 0) = x$. The predicate *divides* is true whenever y divides x. By convention, zero is not considered to be divisible by

TABLE 13.4.1 Primitive Recursive Division Functions

$$quo(x, y) = sg(y) \cdot \overset{x}{\mu z}[gt((z+1) \cdot y, x)]$$

$$rem(x, y) = x \dot{-} (y \cdot quo(x, y))$$

$$divides(x, y) = \begin{cases} 1 & \text{if } x > 0, y > 0, \text{ and } y \text{ is a divisor of } x \\ 0 & \text{otherwise} \end{cases}$$
$$= eq(rem(x, y), 0) \cdot sg(x)$$

$$ndivisors(x) = \sum_{i=0}^{x} divides(x, i)$$

$$prime(x) = eq(ndivisors(x), 2)$$

any number. The factor $sg(x)$ enforces this condition. The default value of the remainder function guarantees that $divides\ (x, 0) = 0$.

The generalized bounded sum can be used to count the number of divisors of a number. The upper bound of the sum is computed from the input by the primitive recursive function $p_1^{(1)}$. This bound is satisfactory since no number greater than x is a divisor of x. A prime number is a number whose only divisors are 1 and itself. The predicate *prime* simply consults the function *ndivisors*.

The predicate prime and bounded minimalization can be used to construct a primitive recursive function pn that enumerates the primes. The value of $pn(i)$ is the ith prime. Thus, $pn(0) = 2$, $pn(1) = 3$, $pn(2) = 5$, $pn(3) = 7$, ... The $x + 1$st prime is the first prime number greater than $pn(x)$. Bounded minimalization is ideally suited for performing this type of search. To employ the bounded μ-operator, we must determine an upper bound for the minimalization. By Theorem 13.3.4, the bound may be calculated using the input value x.

Lemma 13.4.1

Let $pn(x)$ denote the xth prime. Then $pn(x + 1) \leq pn(x)! + 1$.

Proof Each of the primes $pn(i)$, $i = 0, 1, \ldots, x$, divides $pn(x)!$. Since a prime cannot divide two consecutive numbers, either $pn(x)! + 1$ is prime or its prime decomposition contains a prime other than $pn(0)$, $pn(1)$, . . . , $pn(x)$. In either case, $pn(x + 1) \leq pn(x)! + 1$. ■

The bound provided by the preceding lemma is computed by the primitive recursive function $fact(x) + 1$. The xth prime function is obtained by primitive recursion as follows:

$$pn(0) = 2$$

$$pn(x + 1) = \overset{fact(pn(x))+1}{\mu z}[prime(z) \cdot gt(z, pn(x))]$$

Let us take a moment to reflect on the consequences of the relationship between the family of primitive recursive functions and Turing computability. By Theorem 13.1.3,

every primitive recursive function is Turing computable. The bounded operators and bounded minimalization are powerful tools for constructing complex primitive recursive functions. Designing Turing machines that explicitly compute functions such as *pn* or *ndivisors* would require a large number of states and a complicated transition function. Using the macroscopic approach to computation, these functions are easily shown to be computable. Without the tedium inherent in constructing complicated Turing machines, we have shown that many useful functions and predicates are Turing computable.

13.5 Gödel Numbering and Course-of-Values Recursion

Many common computations involving natural numbers are not number-theoretic functions. Sorting a sequence of numbers returns a sequence, not a single number. However, there are many sorting algorithms that we consider effective procedures. We now introduce primitive recursive constructions that allow us to perform this type of operation. The essential feature is the ability to encode a sequence of numbers in a single value. The coding scheme utilizes the unique decomposition of a natural number into a product of primes. Such codes are called *Gödel numberings* after German logician Kurt Gödel, who developed the technique.

A sequence $x_0, x_1, \ldots, x_n$ of length $n + 1$ is encoded by

$$pn(0)^{x_0+1} \cdot pn(1)^{x_1+1} \cdot \cdots \cdot pn(n)^{x_n+1} = 2^{x_0+1} \cdot 3^{x_1+1} \cdot \cdots \cdot pn(n)^{x_n+1}.$$

Since our numbering begins with zero, the elements of a sequence of length n are numbered $0, 1, \ldots, n - 1$.

Sequence	Encoding
1, 2	$2^2 3^3 = 108$
0, 1, 3	$2^1 3^2 5^4 = 11,250$
0, 1, 0, 1	$2^1 3^2 5^1 7^2 = 4,410$

An encoded sequence of length n is a product of powers of the first n primes. The choice of the exponent $x_i + 1$ guarantees that $pn(i)$ occurs in the encoding even when x_i is zero.

The definition of a function that encodes a fixed number of inputs can be obtained directly from the definition of the Gödel numbering. We let

$$gn_n(x_0, \ldots, x_n) = pn(0)^{x_0+1} \cdot \cdots \cdot pn(n)^{x_n+1} = \prod_{i=0}^{n} pn(i)^{x_i+1}$$

be the $n + 1$-variable function that encodes a sequence $x_0, x_1, \ldots, x_n$. The function gn_{n-1} can be used to encode the components of an ordered n-tuple. The Gödel number associated with the ordered pair $[x_0, x_1]$ is $gn_1(x_0, x_1)$.

A decoding function is constructed to retrieve the components of an encoded sequence. The function

$$dec(i,x) = \overset{x}{\mu z}[cosg(divides(x, pn(i)^{z+1}))] \dot{-} 1$$

returns the ith element of the sequence encoded in the Gödel number x. The bounded μ-operator is used to find the power of $pn(i)$ in the prime decomposition of x. The minimalization returns the first value of z for which $pn(i)^{z+1}$ does not divide x. The ith element in an encoded sequence is one less than the power of $pn(i)$ in the encoding. The decoding function returns zero for every prime that does not occur the prime decomposition of x.

When a computation requires n previously computed values, the Gödel encoding function gn_{n-1} can be used to encode the values. The encoded values can be retrieved when they are needed by the computation.

Example 13.5.1

The Fibonacci numbers are defined as the sequence 0, 1, 1, 2, 3, 5, 8, 13, . . . , where an element in the sequence is the sum of its two predecessors. The function

$$f(0) = 0$$
$$f(1) = 1$$
$$f(y+1) = f(y) + f(y-1) \text{ for } y > 1$$

generates the Fibonacci numbers. This is not a definition by primitive recursion since the computation of $f(y+1)$ utilizes both $f(y)$ and $f(y-1)$. To show that the Fibonacci numbers are generated by a primitive recursive function, the Gödel numbering function gn_1 is used to store the two values as a single number. An auxiliary function h encodes the ordered pair with first component $f(y-1)$ and second component $f(y)$:

$$h(0) = gn_1(0,1) = 2^1 3^2 = 18$$
$$h(y+1) = gn_1(dec(1,h(y)),\ dec(0,h(y)) + dec(1,h(y)))$$

The initial value of h is the encoded pair $[f(0), f(1)]$. The calculation of $h(y+1)$ begins by producing another ordered pair

$$[dec(1,h(y)),\ dec(0,h(y)) + dec(1,h(y))] = [f(y),\ f(y-1) + f(y)].$$

Encoding the pair with gn_1 completes the evaluation of $h(y+1)$. This process constructs the sequence of Gödel numbers of the pairs $[f(0), f(1)]$, $[f(1), f(2)]$, $[f(2), f(3)]$, ... The primitive recursive function $f(y) = dec(0, h(y))$ extracts the Fibonacci numbers from the first components of the ordered pairs. □

The Gödel numbering functions gn_i encode a fixed number of arguments. A Gödel numbering function can be constructed in which the number of elements to be encoded is variable. The approach is similar to that taken in constructing the bounded sum and product operations. The values of a one-variable primitive recursive function f with input

$0, 1, \ldots, n$ define a sequence $f(0), f(1), \ldots, f(n)$ of length $n + 1$. The bounded product

$$\prod_{i=0}^{y} pn(i)^{f(i)+1}$$

encodes the first $y + 1$ values of f. A Gödel numbering function $gn_f(x_1, \ldots, x_n, y)$ can be defined from a total $n + 1$-variable function f. The relationship between a function f and its encoding function gn_f is established in Theorem 13.5.1.

Theorem 13.5.1

Let f be an $n + 1$-variable function and gn_f the encoding function defined from f. Then f is primitive recursive if, and only if, gn_f is primitive recursive.

Proof If $f(x_1, \ldots, x_n, y)$ is primitive recursive, then the bounded product

$$gn_f(x_1, \ldots, x_n, y) = \prod_{i=0}^{y} pn(i)^{f(x_1,\ldots,x_n,i)+1}$$

computes the Gödel encoding function. On the other hand, the decoding function can be used to recover the values of f from the Gödel number generated by gn_f.

$$f(x_1, \ldots, x_n, y) = dec(y, gn_f(x_1, \ldots, x_n, y))$$

Thus f is primitive recursive whenever gn_f is. ■

The primitive recursive functions have been introduced because of their intuitive computability. In a definition by primitive recursion, the computation is permitted to use the result of the function with the previous value of the recursive variable. Consider the function defined by

$$\begin{aligned}
f(0) &= 1 \\
f(1) &= f(0) \cdot 1 = 1 \\
f(2) &= f(0) \cdot 2 + f(1) \cdot 1 = 3 \\
f(3) &= f(0) \cdot 3 + f(1) \cdot 2 + f(2) \cdot 1 = 8 \\
f(4) &= f(0) \cdot 4 + f(1) \cdot 3 + f(2) \cdot 2 + f(3) \cdot 1 = 21 \\
&\vdots
\end{aligned}$$

The function f can be written as

$$\begin{aligned}
f(0) &= 1 \\
f(y+1) &= \sum_{i=0}^{y} f(i) \cdot (y + 1 - i).
\end{aligned}$$

The definition, as formulated, is not primitive recursive since the computation of $f(y)$ utilizes all of the previously computed values. The function, however, is intuitively computable; the definition itself outlines an algorithm by which any value can be calculated.

When the result of a function with recursive variable $y + 1$ is defined in terms of $f(0), f(1), \ldots, f(y)$, the function f is said to be defined by **course-of-values recursion**. Determining the result of a function defined by course-of-values recursion appears to utilize a different number of inputs for each value of the recursive variable. In the preceding example, $f(2)$ requires only $f(0)$ and $f(1)$, while $f(4)$ requires $f(0), f(1), f(2)$, and $f(3)$. No single function can be used to compute both $f(2)$ and $f(4)$ directly from the preceding values since a function has a fixed number of arguments.

Regardless of the value of the recursive variable $y + 1$, the preceding results can be encoded in the Gödel number $gn_f(y)$. This observation provides the framework for a formal definition of course-of-values recursion.

Definition 13.5.2

Let g and h be $n + 2$-variable total number-theoretic functions, respectively. The $n + 1$-variable function f defined by

i) $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$

ii) $f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, gn_f(x_1, \ldots, x_n, y))$

is said to be obtained from g and h by course-of-values recursion.

Theorem 13.5.3

Let f be an $n + 1$-variable function defined by course-of-values recursion from primitive recursive functions g and h. Then f is primitive recursive.

Proof We begin by defining gn_f by primitive recursion directly from the primitive recursive functions g and h.

$$\begin{aligned} gn_f(x_1, \ldots, x_n, 0) &= 2^{f(x_1,\ldots,x_n,0)+1} \\ &= 2^{g(x_1,\ldots,x_n)+1} \\ gn_f(x_1, \ldots, x_n, y + 1) &= gn_f(x_1, \ldots, x_n, y) \cdot pn(y + 1)^{f(x_1,\ldots,x_n,y+1)+1} \\ &= gn_f(x_1, \ldots, x_n, y) \cdot pn(y + 1)^{h(x_1,\ldots,x_n,y,gn_f(x_1,\ldots,x_n,y))+1} \end{aligned}$$

The evaluation of $gn_f(x_1, \ldots, x_n, y + 1)$ uses only

i) the parameters $x_0, \ldots, x_n$

ii) y, the previous value of the recursive variable

iii) $gn_f(z_1, \ldots, x_n, y)$, the previous value of gn_f

iv) the primitive recursive functions h, pn, $\cdot$, $+$, and exponentiation.

Thus, the function gn_f is primitive recursive. By Theorem 13.5.1, it follows that f is also primitive recursive. ■

In mechanical terms, the Gödel numbering gives computation the equivalent of unlimited memory. A single Gödel number is capable of storing any number of preliminary results. The Gödel numbering encodes the values $f(x_0, \ldots, x_n, 0)$, $f(x_0, \ldots, x_n, 1)$, $\ldots$, $f(x_0, \ldots, x_n, y)$ that are required for the computation of $f(x_0, \ldots, x_n, y + 1)$. The decoding function provides the connection between the memory and the computation. Whenever a stored value is needed by the computation, the decoding function makes it available.

Example 13.5.2

Let h be the primitive recursive function

$$h(x, y) = \sum_{i=0}^{x} dec(i, y) \cdot (x + 1 - i).$$

The function f, which was defined earlier to introduce course-of-values computation, can be defined by course-of-values recursion from h.

$$f(0) = 1$$

$$f(y + 1) = h(y, gn_f(y)) = \sum_{i=0}^{y} dec(i, gn_f(y)) \cdot (y + 1 - i)$$

$$= \sum_{i=0}^{y} f(i) \cdot (y + 1 - i) \qquad \square$$

13.6 Computable Partial Functions

The primitive recursive functions were defined as a family of intuitively computable functions. We have established that all primitive recursive functions are total. Conversely, are all computable total functions primitive recursive? Moreover, should we restrict our analysis of computability to total functions? In this section we will present arguments for a negative response to both of these questions.

Theorem 13.6.1

The set of primitive recursive functions is a proper subset of the set of effectively computable total number-theoretic functions.

Proof The primitive recursive functions can be represented as strings over the alphabet $\Sigma = \{s, p, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), \circ, :, \langle, \rangle\}$. The basic functions s, z, and $p_i^{(j)}$ are represented by $\langle s \rangle, \langle z \rangle$, and $\langle pi(j) \rangle$. The composition $h \circ (g_1, \ldots, g_n)$ is encoded $\langle \langle h \rangle \circ \langle \langle g_1 \rangle, \ldots, \langle g_n \rangle \rangle \rangle$, where $\langle h \rangle$ and $\langle g_i \rangle$ are the representations of the constituent functions. A function defined by primitive recursion from functions g and h is represented by $\langle \langle g \rangle : \langle h \rangle \rangle$.

The strings in Σ^* can be generated by length: first the null string, followed by strings of length one, length two, and so on. A straightforward mechanical process can be designed to determine whether a string represents a correctly formed primitive recursive function. The enumeration of the primitive recursive functions is accomplished by repeatedly generating a string and determining if it is a syntactically correct representation of a function. The first correctly formed string is denoted f_0, the next f_1, and so on. In the same manner, we can enumerate the one-variable primitive recursive functions. This is accomplished by deleting all n-variable functions, $n > 1$, from the list generated above. This sequence is denoted $f_0^{(1)}, f_1^{(1)}, f_2^{(1)}, \ldots$.

The total one-variable function

$$g(i) = f_i^{(1)}(i) + 1$$

is effectively computable. The effective enumeration of the one-variable primitive recursive functions establishes the computability of g. The value $g(i)$ is obtained by

i) determining the ith one-variable primitive recursive function $f_i^{(1)}$

ii) computing $f_i^{(1)}(i)$

iii) adding one to $f_i^{(1)}(i)$.

Since each of these steps is effective, we conclude that g is computable. By the familiar diagonalization argument,

$$g(i) \neq f_i^{(1)}(i)$$

for any i. Consequently, g is total and computable but not primitive recursive. ■

Theorem 13.6.1 uses a counting argument to demonstrate the existence of computable functions that are not primitive recursive. This can also be accomplished directly by constructing a computable function that is not primitive recursive. The two-variable number-theoretic function, known as *Ackermann's function*, defined by

i) $A(0, y) = y + 1$

ii) $A(x + 1, 0) = A(x, 1)$

iii) $A(x + 1, y + 1) = A(x, A(x + 1, y))$,

is one such function. The values of A are defined recursively with the basis given in condition (i). A proof by induction on x establishes that A is uniquely defined for every pair of input values (Exercise 20). The computations in Example 13.6.1 illustrate the computability of Ackermann's function.

Example 13.6.1

The values $A(1, 1)$ and $A(3, 0)$ are constructed from the definition of Ackermann's function. The column on the right gives the justification for the substitution.

a) $A(1,1) = A(0, A(1,0))$ (iii)
$= A(0, A(0,1))$ (ii)
$= A(0,2)$ (i)
$= 3$

b) $A(2,1) = A(1, A(2,0))$ (iii)
$= A(1, A(1,1))$ (ii)
$= A(1,3)$ (a)
$= A(0, A(1,2))$ (iii)
$= A(0, A(0, A(1,1)))$ (iii)
$= A(0, A(0,3))$ (a)
$= A(0,4)$ (i)
$= 5$ (i) □

The values of Ackermann's function exhibit a remarkable rate of growth. By fixing the first variable, Ackermann's function generates the one-variable functions

$$A(1,y) = y + 2$$
$$A(2,y) = 2y + 3$$
$$A(3,y) = 2^{y+3} - 3$$
$$A(4,y) = 2^{2^{\cdot^{\cdot^{\cdot^{2^{16}}}}}} - 3.$$

The number of 2's in the exponential chain in $A(4,y)$ is y. For example, $A(4,0) = 16 - 3$, $A(4,1) = 2^{16} - 3$, and $A(4,2) = 2^{2^{16}} - 3$. The first variable of Ackermann's function determines the rate of growth of the function values. We state, without proof, the following theorem that compares the rate of growth of Ackermann's function with that of the primitive recursive functions.

Theorem 13.6.2
For every one-variable primitive recursive function f, there is some $i \in \mathbf{N}$ such that $f(i) < A(i,i)$.

Clearly, the one-variable function $A(i,i)$ obtained by indentifying the variables of A is not primitive recursive. It follows that Ackermann's function is not primitive recursive. If it were, then $A(i,i)$, which can be obtained by the composition $A \circ (p_1^{(1)}, p_1^{(1)})$, would also be primitive recursive.

Regardless of the set of total functions that we consider computable, the diagonal argument presented in the proof of Theorem 13.6.1 can be used to show that there is no effective enumeration of these functions. Therefore, we must conclude that the computable

functions cannot be effectively generated or that there are computable nontotal functions. If we accept the latter proposition, the contradiction from the diagonalization argument disappears. The reason we can claim that g is not one of the f_i's is that $g(i) \neq f_i^{(1)}(i)$. If $f_i^{(1)}(i) \uparrow$, then $g(i) = f_i^{(i)}(i) + 1$ is also undefined. If we wish to be able to effectively enumerate the computable functions, it is necessary to include partial functions in the enumeration.

We now consider the computability of partial functions. Since composition and primitive recursion preserve totality, an additional operation is needed to construct partial functions from the basic functions. Minimalization has been informally described as a search procedure. Placing a bound on the range of the natural numbers to be examined ensured that the bounded minimalization operation produces total functions. Unbounded minimalization is obtained by performing the search without an upper limit on the set of natural numbers to be considered. The function

$$f(x) = \mu z[eq(x, z \cdot z)]$$

defined by unbounded minimalization returns the square root of x whenever x is a perfect square. Otherwise, the search for the first natural number satisfying the predicate continues ad infinitum. Although eq is a total function, the resulting function f is not. For example, $f(3) \uparrow$. A function defined by unbounded minimalization is undefined for input x whenever the search fails to return a value.

The introduction of partial functions forces us to reexamine the operations of composition and primitive recursion. The possibility of undefined values was considered in the definition of composition. The function $h \circ (g_1, \ldots, g_n)$ is undefined for input $x_1, \ldots, x_k$ if either

i) $g_i(x_1, \ldots, x_k) \uparrow$ for some $1 \leq i \leq n$

ii) $g_i(x_1, \ldots, x_k) \downarrow$ for all $1 \leq i \leq n$ and $h(g_1(x_1, \ldots, x_k), \ldots g_n(x_1, \ldots, x_k)) \uparrow$.

An undefined value propagates from any of the g_i's to the composite function.

The operation of primitive recursion required both of the defining functions g and h to be total. This restriction is relaxed to permit definitions by primitive recursion using partial functions. Let f be defined by primitive recursion from partial functions g and h.

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$
$$f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$$

Determining the value of a function defined by primitive recursion is an iterative process. The function f is defined for recursive variable y only if the following conditions are satisfied:

i) $f(x_1, \ldots, x_n, 0) \downarrow$ if $g(x_1, \ldots, x_n) \downarrow$

ii) $f(x_1, \ldots, x_n, y + 1) \downarrow$ if $f(x_1, \ldots, x_n, i) \downarrow$ for $0 \leq i \leq y$
and $h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) \downarrow$.

An undefined value for the primitive recursive variable causes f to be undefined for all the subsequent values of the primitive recursive variable.

With the conventions established for definitions with partial functions, a family of computable partial functions can be defined using the operations composition, primitive recursion, and unbounded minimalization.

Definition 13.6.3

The family of **μ-recursive functions** is defined as follows:

i) The successor, zero, and projection functions are μ-recursive.

ii) If h is an n-variable μ-recursive function and $g_1, \ldots, g_n$ are k-variable μ-recursive functions, then $f = h \circ (g_1, \ldots, g_n)$ is μ-recursive.

iii) If g and h are n and $n+2$-variable μ-recursive functions, then the function f defined from g and h by primitive recursion is μ-recursive.

iv) If $p(x_1, \ldots, x_n, y)$ is a total μ-recursive predicate, then $f = \mu z[p(x_1, \ldots, x_n, z)]$ is μ-recursive.

v) A function is μ-recursive only if it can be obtained from (i) by a finite number of applications of the rules in (ii), (iii), and (iv).

Conditions (i), (ii), and (iii) imply that all primitive recursive functions are μ-recursive. Notice that unbounded minimalization is not defined for all predicates. The unbounded μ-operator can be applied only to total μ-recursive predicates.

The notion of Turing computability encompasses partial functions in a natural way. A Turing machine computes a partial number-theoretic function f if

i) the computation terminates with result $f(x_1, \ldots, x_n)$ whenever $f(x_1, \ldots, x_n) \downarrow$

ii) the computation does not terminate whenever $f(x_1, \ldots, x_n) \uparrow$.

The Turing machine computes the value of the function whenever possible. Otherwise, the computation continues indefinitely.

We will now establish the relationship between the μ-recursive and Turing computable functions. The first step is to show that every μ-recursive function is Turing computable. This is not a surprising result; it simply extends Theorem 13.1.3 to partial functions.

Theorem 13.6.4

Every μ-recursive function is Turing computable.

Proof Since the basic functions are known to be Turing computable, the proof consists of showing that the Turing computable partial functions are closed under operations of composition, primitive recursion, and unbounded minimalization. The techniques developed in Theorems 12.4.3 and 13.1.3 demonstrate the closure of Turing computable total functions under composition and primitive recursion, respectively. These machines also

establish the closure for partial functions. An undefined value in one of the constituent computations causes the entire computation to continue indefinitely.

The proof is completed by showing that the unbounded minimalization of a Turing computable total predicate is Turing computable. Let $f(x_1, \ldots, x_n) = \mu z[p(x_1, \ldots, x_n, y)]$ where $p(x_1, \ldots, x_n, y)$ is a total Turing computable predicate. A Turing machine to compute f can be constructed from P, the machine that computes the predicate p. The initial configuration of the tape is $B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B$.

1. The representation of the number zero is added to the right of the input. The search specified by the minimalization operator begins with the tape configuration

$$B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{0}B.$$

   The number to the right of the input, call it j, is the index for the minimalization operator.

2. A working copy of the parameters and j is made, producing the tape configuration

$$B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{j} B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{j}B.$$

3. The machine P is run with the input consisting of the copy of the parameters and j, producing

$$B\overline{x}_1 B\overline{x}_2 B \ldots B\overline{x}_n B\overline{j} B\overline{p(x_1, x_2, \ldots, x_n, j)}B.$$

4. If $p(x_1, x_2, \ldots, x_n, j) = 1$, the value of the minimalization of p is j. Otherwise, the $p(x_1, x_2, \ldots, x_n, j)$ is erased, j is incremented, and steps 2 through 4 are repeated.

A computation terminates at step 4 when the first j for which $p(x_1, \ldots, x_n, j) = 1$ is encountered. If no such value exists, the computation loops indefinitely, indicating that the function f is undefined. ■

13.7 Turing Computability and Mu-Recursive Functions

It has already been established that every μ-recursive function can be computed by a Turing machine. We now turn our attention to the opposite inclusion, that every Turing computable function is μ-recursive. A number-theoretic function is designed to simulate the computations of a Turing machine. The construction of the simulating function requires moving from the domain of machines to the domain of natural numbers. The process of translating machine computations to functions is known as the *arithmetization* of Turing machines.

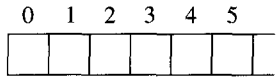
The arithmetization begins by assigning a number to a Turing machine configuration. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_n)$ be a standard Turing machine that computes a one-variable

number-theoretic function f. The states and tape alphabet of M are denoted

$$Q = \{q_0, q_1, \ldots, q_n\}$$
$$\Gamma = \{B = a_0, 1 = a_1, \ldots, a_k\}.$$

A μ-recursive function is constructed to numerically simulate the computations of M. The construction easily generalizes to functions of more than one variable.

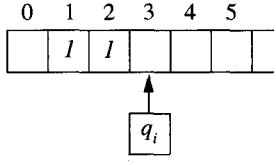
A configuration of the Turing machine M consists of the state, the position of the tape head, and the segment of the tape from the left boundary to the rightmost nonblank symbol. Each of these components must be represented by a natural number. The subscripts provide a numbering for the states and the tape alphabet. The tape symbols B and 1 are assigned zero and one, respectively. The location of the tape head can be encoded using the numbering of the tape positions.



The symbols on the tape to the rightmost nonblank square form a string over Σ^*. Encoding the tape uses the numeric representation of the elements of the tape alphabet. The string $a_{i_0}a_{i_1}\ldots a_{i_n}$ is encoded by the Gödel number associated with the sequence $i_0, i_1, \ldots, i_n$. The number representing the nonblank tape segment is called the **tape number**.

Representing the blank by the number zero permits the correct decoding of any tape position regardless of the segment of the tape encoded in the tape number. If $dec(i, z) = 0$ and $pn(i)$ divides z, then the blank is specifically encoded in the tape number z. On the other hand, if $dec(i, z) = 0$ and $pn(i)$ does not divide z, then position i is to the right of the encoded segment of the tape. Since the tape number encodes the entire nonblank segment of the tape, it follows that position i must be blank.

The tape number of the nonblank segment of the machine configuration



is $2^1 3^2 5^2 = 450$. Explicitly encoding the blank in position three produces $2^1 3^2 5^2 7^1 = 3150$, another tape number representing the tape. Any number of blanks to the right of the rightmost nonblank square may be included in the tape number.

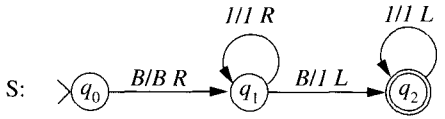
A Turing machine configuration is defined by the state number, tape head position, and tape number. The configuration number incorporates these values into the single number

$$gn_2(\text{state number, tape head position, tape number}),$$

where gn_2 is the Gödel numbering function that encodes ordered triples.

Example 13.7.1

The Turing machine S computes the successor function.



The configuration numbers are given for each configuration produced by the computation of the successor of 1. Recall that the tape symbols B and 1 are assigned the numbers zero and one, respectively.

	State	Position	Tape Number	Configuration Number
q_0B11B	0	0	$2^1 3^2 5^2 = 450$	$gn_2(0,0,450)$
$\vdash Bq_111B$	1	1	$2^1 3^2 5^2 = 450$	$gn_2(1,1,450)$
$\vdash B1q_11B$	1	2	$2^1 3^2 5^2 = 450$	$gn_2(1,2,450)$
$\vdash B11q_1B$	1	3	$2^1 3^2 5^2 7^1 = 3150$	$gn_2(1,3,3150)$
$\vdash B1q_211B$	2	2	$2^1 3^2 5^2 7^2 11^1 = 242550$	$gn_2(2,2,242550)$
$\vdash Bq_2111B$	2	1	$2^1 3^2 5^2 7^2 11^1 = 242550$	$gn_2(2,1,242550)$
$\vdash q_2B111B$	2	0	$2^1 3^2 5^2 7^2 11^1 = 242550$	$gn_2(2,0,242550)$

□

A Turing machine transition need not alter the tape or the state, but it must move the tape head. The change in the tape head position and the uniqueness of the Gödel numbering ensure that no two consecutive configuration numbers of a computation are identical.

A function tr_M is constructed to trace the computations of a Turing machine M. Tracing a computation means generating the sequence of configuration numbers that correspond to the machine configurations produced by the computation. The value of $tr_M(x,i)$ is the number of the configuration after i transitions when M is run with input x. Since the initial configuration of M is $q_0B\bar{x}B$,

$$tr_M(x,0) = gn_2(0,0,2^1 \cdot \prod_{i=1}^{x+1} pn(i)^2).$$

The value of $tr_M(x, y+1)$ is obtained by manipulating the configuration number $tr_M(x,y)$ to construct the encoding of the subsequent machine configuration.

The state and symbol in the position scanned by the tape head determine the transition to be applied by the machine M. The primitive recursive functions

$$cs(z) = dec(0, z)$$
$$ctp(z) = dec(1, z)$$
$$cts(z) = dec(ctp(z), dec(2, z))$$

return the state number, tape head position, and the number of the symbol scanned by the tape head from a configuration number z. The position of the tape head is obtained by a direct decoding of the configuration number. The numeric representation of the scanned symbol is encoded as the $ctp(z)$th element of the tape number. The c's in cs, ctp, and cts stand for the components of the current configuration: current state, current tape position, and current tape symbol.

A transition specifies the alterations to the machine configuration and, hence, the configuration number. A transition of M is written

$$\delta(q_i, b) = [q_j, c, d],$$

where $q_i, q_j \in Q$; $b, c \in \Gamma$; and $d \in \{R, L\}$. Functions are defined to simulate the effects of a transition of M. We begin by listing the transitions of M:

$$\delta(q_{i_0}, b_0) = [q_{j_0}, c_0, d_0]$$
$$\delta(q_{i_1}, b_1) = [q_{j_1}, c_1, d_1]$$
$$\vdots$$
$$\delta(q_{i_m}, b_m) = [q_{j_m}, c_m, d_m].$$

The determinism of the machine ensures that the arguments of the transitions are distinct. The number assigned to the tape symbol a is denoted $n(a)$. A function that returns the number of the state entered by a transition from a configuration with configuration number z is

$$ns(z) = \begin{cases} j_0 & \text{if } cs(z) = i_0 \text{ and } cts(z) = n(b_0) \\ j_1 & \text{if } cs(z) = i_1 \text{ and } cts(z) = n(b_1) \\ \vdots & \vdots \\ j_m & \text{if } cs(z) = i_m \text{ and } cts(z) = n(b_m) \\ cs(z) & \text{otherwise.} \end{cases}$$

The first condition can be interpreted "if the number of the current state is i_0 (state q_{i_0}) and the current tape symbol is b_0, then the new state number has number j_0 (state q_{j_0})." This is a direct translation of the initial transition into the numeric representation. The conditions define a set of exhaustive and mutually exclusive primitive recursive predicates. Thus, $ns(z)$ is primitive recursive. A function nts that computes the number of the new tape symbol can be defined in a completely analogous manner.

A function that computes the new tape head position alters the number of the current position as specified by the direction in the transition. The transitions designate the directions as L (left) or R (right). A movement to the left subtracts one from the current position

number and a movement to the right adds one. To numerically represent the direction we use the notation

$$n(d) = \begin{cases} 0 & \text{if } d = L \\ 2 & \text{if } d = R. \end{cases}$$

The new tape position is computed by

$$ntp(z) = \begin{cases} ctp(z) + n(d_0) \dot{-} 1 & \text{if } cs(z) = i_0 \text{ and } cts(z) = n(b_0) \\ ctp(z) + n(d_1) \dot{-} 1 & \text{if } cs(z) = i_1 \text{ and } cts(z) = n(b_1) \\ \quad \vdots & \quad \vdots \\ ctp(z) + n(d_m) \dot{-} 1 & \text{if } cs(z) = i_m \text{ and } cts(z) = n(b_m) \\ ctp(z) & \text{otherwise.} \end{cases}$$

The addition of $n(d_i) \dot{-} 1$ to the current position number increments the value by one when the transition moves the tape head to the right. Similarly, one is subtracted on a move to the left.

We have almost completed the construction of the components of the trace function. Given a machine configuration, the functions ns and ntp compute the state number and tape head position of the new configuration. All that remains is to compute the new tape number.

A transition replaces the tape symbol occupying the position scanned by the tape head. In our functional approach, the location of the tape head is obtained from the configuration number z by the function ctp. The tape symbol to be written at position $ctp(z)$ is represented numerically by $nts(z)$. The new tape number is obtained by changing the power of $pn(ctp(z))$ in the current tape number. Before the transition, the decomposition of z contains $pn(ctp(z))^{cts(z)+1}$, encoding the value of the current tape symbol at position $ctp(z)$. After the transition, position $ctp(z)$ contains the symbol represented by $nts(z)$. The primitive recursive function

$$ntn(z) = quo(ctn(z), pn(ctp(z))^{cts(z)+1}) \cdot pn(ctp(z))^{nts(z)+1}$$

makes the desired substitution. The division removes the factor that encodes the current symbol at position $ctp(z)$ from the tape number $ctn(z)$. The result is then multiplied by $pn(ctp(z))^{nts(z)+1}$, encoding the new tape symbol.

The trace function is defined by primitive recursion from the functions that simulate the effects of a transition of M on the components of the configuration.

$$tr_{\mathrm{M}}(x, 0) = gn_2(0, 0, 2^1 \cdot \prod_{i=1}^{x+1} pn(i)^2)$$

$$tr_{\mathrm{M}}(x, y + 1) = gn_2(ns(tr_{\mathrm{M}}(x, y)),\ ntp(tr_{\mathrm{M}}(x, y)),\ ntn(tr_{\mathrm{M}}(x, y)))$$

Since each of the component functions has been shown to be primitive recursive, we conclude that the tr_{M} is not only μ-recursive but also primitive recursive. The trace function

is not the culmination of our functional simulation of a Turing machine; it does not return the result of a computation but rather a configuration number.

The result of the computation of the Turing machine M that computes the number-theoretic function f with input x may be obtained from the function tr_{M}. We first note that the computation of M may never terminate; $f(x)$ may be undefined. The question of termination can be determined from the values of tr_{M}. If M specifies a transition for configuration $tr_{\mathrm{M}}(x,i)$, then $tr_{\mathrm{M}}(x,i) \neq tr_{\mathrm{M}}(x,i+1)$ since the movement of the head changes the Gödel number. On the other hand, if M halts after transition i, then $tr_{\mathrm{M}}(x,i) = tr_{\mathrm{M}}(x,i+1)$. The functions *nts*, *ntp*, and *ntn* return the preceding value when the configuration number represents a halting configuration. Consequently, the machine halts after the zth transition, where z is the first number that satisfies $tr_{\mathrm{M}}(x,z) = tr_{\mathrm{M}}(x,z+1)$.

Since no bound can be placed on the number of transitions that occur before an arbitrary Turing machine computation terminates, unbounded minimalization is required to determine this value. The μ-recursive function

$$term(x) = \mu z[eq(tr_{\mathrm{M}}(x,z),\ tr_{\mathrm{M}}(x,z+1))]$$

computes the number of the transition after which the computation of M with input x terminates. When a computation terminates, the halting configuration of the machine is encoded in the value $tr_{\mathrm{M}}(x,term(x))$. Upon termination, the tape has the form $B\overline{f(x)}B$. The terminal tape number, ttn, is obtained from the terminal configuration number by

$$ttn(x) = dec(2,\ tr_{\mathrm{M}}(x,term(x))).$$

The result of the computation is obtained by counting the number of *1*'s on the tape or, equivalently, determining the number of primes that are raised to the power of 2 in the terminal tape number. The latter computation is performed by the bounded sum

$$sim_{\mathrm{M}}(x) = \left(\sum_{i=0}^{y} eq(1, dec(i, ttn(x))\right) \dot{-}\ 1,$$

where y is the length of the tape segment encoded in the terminal tape number. The bound y is computed by the primitive recursive function $gdln(ttn(x))$ (Exercise 15). One is subtracted from the bounded sum since the tape contains the unary representation of $f(x)$.

Whenever f is defined for input x, the computation of M and the simulation of M both compute the $f(x)$. If $f(x)$ is undefined, the unbounded minimalization fails to return a value and $sim_{\mathrm{M}}(x)$ is undefined. The construction of sim_{M} completes the proof of the following theorem.

Theorem 13.7.1

Every Turing computable function is μ-recursive.

Theorems 13.6.4 and 13.7.1 establish the equivalence of the microscopic and macroscopic approaches to computation.

Corollary 13.7.2
A function is Turing computable if, and only if, it is μ-recursive.

13.8 The Church-Turing Thesis Revisited

The Church-Turing thesis, as presented in Chapter 11, asserted that every effectively solvable decision problem admits a Turing machine solution. We have subsequently designed Turing machines to compute number theoretic functions. In its functional form, the Church-Turing thesis associates the effective computation of functions with Turing computability.

The Church-Turing thesis A partial function is computable if, and only if, it is μ-recursive.

Turing machines that accept input by final state provided the formal framework for constructing solutions to decision problems. A functional approach to solving decision problems uses the computed values one and zero to designate affirmative and negative responses. The method of specifying the answer does not affect the set of decision problems that have Turing machine solutions (Exercise 12.13). The formulation of the Church-Turing thesis in terms of computable functions includes the assertion concerning decision problems presented earlier.

As before, no proof can be put forward for the Church-Turing thesis. It is accepted by the community of mathematicians and computer scientists because of the accumulation of evidence supporting the claim. Many attempts, employing a wide variety of techniques and formalisms, have been made to classify the computable functions. The approaches to computability fall into three general categories: mechanical, functional, and transformational. Turing computability and μ-recursive functions are examples of the first two categories. An example of the latter is provided by the computational scheme introduced by Markov that defines algorithmic computation as a sequence of string transformations. Each of these, and all other computational systems that have been constructed for this purpose, generates precisely the functions that can be computed by Turing machines, the μ-recursive functions.

Accepting the Church-Turing thesis is tantamount to bestowing the title "most general computing device" on the Turing machine. The thesis implies that any number-theoretic function that can be effectively computed by any machine or technique can also be computed by a Turing machine. This contention extends to nonnumeric computation as well.

We begin by observing that the computation of any digital computer can be interpreted as a numeric computation. Character strings are often used to communicate with the computer, but this is only a convenience to facilitate the input of the data and the interpretation of the output. The input is immediately translated to a string over $\{0, 1\}$ using either the ASCII or EBCDIC encoding schemes. After the translation, the input string can

be considered the binary representation of a natural number. The computation progresses, generating another sequence of *0*'s and *1*'s, again a binary natural number. The output is then translated back to character data because of our inability to interpret and appreciate the output in its internal representation.

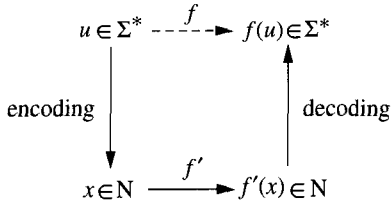
Following this example, we can design effective procedures that transform a string computation to a number-theoretic computation. The Gödel encoding can be used to translate strings to numbers. Let $\Sigma = \{a_0, a_1, \ldots, a_n\}$ and f be a function from Σ^* to Σ^*. The generation of a Gödel number from a string begins by assigning a unique number to each element in the alphabet. The numbering for Σ is defined by its subscript. The encoding of a string $a_{i_0}a_{i_1}\ldots a_{i_n}$ is generated by the bounded product

$$pn(0)^{i_0+1} \cdot pn(1)^{i_1+1} \cdot \ldots \cdot pn(n)^{i_n+1} = \prod_{j=0}^{y} pn(j)^{i_j+1},$$

where y is the length of the string to be encoded.

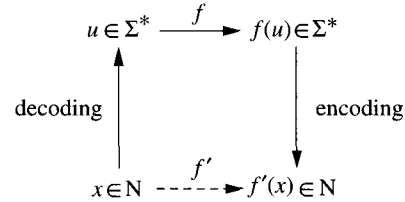
The decoding function retrieves the exponent of each prime in the prime decomposition of the Gödel number. The string can be reconstructed from the decoding function and the numbering of the alphabet. If x is the encoding of a string $a_{i_0}a_{i_1}\ldots a_{i_n}$ over Σ, then $dec(j,x) = i_j$. The original string then is obtained by concatenating the results of the decoding. Once the elements of the alphabet have been identified with natural numbers, the encoding and decoding are primitive recursive and therefore Turing computable.

The transformation of a string function f to a numeric function is obtained using character to number encoding and number to character decoding:



The numeric evaluation of f consists of three distinct computations: the transformation of the input string into a natural number, the computation of the number-theoretic function f', and the translation of the result back to a string.

The Church-Turing thesis asserts that the function f' obtained from an effectively computable nonnumeric function f is Turing-computable. This can be established by observing that the encoding and decoding functions are reversible.



An effective procedure to compute f' consists of generating a string u from the input x, computing $f(u)$, and then encoding $f(u)$ to obtain $f'(x)$. The Turing computability of f' follows from the Church-Turing thesis. Since f' is computable whenever f is, any nonnumeric function can be computed by a numeric function accompanied by encoding and decoding functions.

Example 13.8.1

Let Σ be the alphabet $\{a, b\}$. Consider the function $f : \Sigma^* \to \Sigma^*$ that interchanges the a's and the b's in the input string. A number-theoretic function f' is constructed which, when combined with the functions that encode and decode strings over Σ, computes f. The elements of the alphabet are numbered by the function n: $n(a) = 0$ and $n(b) = 1$. A string $u = u_0 u_1 \ldots u_n$ is encoded as the number

$$pn(0)^{n(u_0)+1} \cdot pn(1)^{n(u_1)+1} \cdot \ldots \cdot pn(n)^{n(u_n)+1}.$$

The power of $pn(i)$ in the encoding is one or two depending upon whether the ith element of the string is a or b, respectively.

Let x be the encoding of a string u over Σ. Recall that $gdln(x)$ returns the length of the sequence encoded by x. The bounded product

$$f'(x) = \prod_{i=0}^{gdln(x)} (eq(dec(i,x),0) \cdot pn(i) \cdot pn(i) + eq(dec(i,x),1) \cdot pn(i))$$

generates the encoding of a string of the same length as the string u. When $eq(dec(i,x),0) = 1$, the ith symbol in u is a. This is represented by $pn(i)^1$ in the encoding of u. The product

$$eq(dec(i,x),0) \cdot pn(i) \cdot pn(i)$$

contributes the factor $pn(i)^2$ to $f'(x)$. Similarly, the power of $pn(i)$ in $f'(x)$ is one whenever the ith element of u is b. Thus f' constructs a number whose prime decomposition can be obtained from that of x by interchanging the exponents 1 and 2. The translation of $f'(x)$ to a string generates $f(u)$. □

Exercises

1. Using only the basic functions, composition, and primitive recursion, show that the following functions are primitive recursive. When using primitive recursion, give the functions g and h.
   a) $c_2^{(3)}$
   b) *pred*
   c) $f(x) = 2x + 2$
2. The functions below were defined by primitive recursion in Table 13.2.1. Explicitly, give the functions g and h that constitute the definition by primitive recursion.
   a) *sg*
   b) *sub*
   c) *exp*
3. a) Prove that a function f defined by the composition of total functions h and $g_1, \ldots, g_n$ is total.
   b) Prove that a function f defined by primitive recursion from total functions g and h is total.
   c) Conclude that all primitive recursive functions are total.
4. Let $g = id$, $h = p_1^{(3)} + p_3^{(3)}$, and let f be defined from g and h by primitive recursion.
   a) Compute the values $f(3,0)$, $f(3,1)$, and $f(3,2)$.
   b) Give a closed form (nonrecursive) definition of the function f.
5. Let $g(x, y, z)$ be a primitive recursive function. Show that each of the following functions is primitive recursive.
   a) $f(x, y) = g(x, y, x)$
   b) $f(x, y, z, w) = g(x, y, x)$
   c) $f(x) = g(1, 2, x)$
6. Show that the following functions are primitive recursive. You may use the functions and predicates from Tables 13.2.1 and 13.2.2. Do not use the bounded operations.
   a) $max(x, y) = \begin{cases} x & \text{if } x \geq y \\ y & \text{otherwise} \end{cases}$
   b) $min(x, y) = \begin{cases} x & \text{if } x \leq y \\ y & \text{otherwise} \end{cases}$
   c) $min(x, y, z) = \begin{cases} x & \text{if } x \leq y \text{ and } x \leq z \\ y & \text{if } y \leq x \text{ and } y \leq z \\ z & \text{if } z \leq z \text{ and } z \leq y \end{cases}$

d) $even(x) = \begin{cases} 1 & \text{if } x \text{ is even} \\ 0 & \text{otherwise} \end{cases}$

e) $half(x) = div(x, 2)$

f) $sqrt(x) = \lfloor \sqrt{x} \rfloor$

7. Show that the following predicates are primitive recursive. You may use the functions and predicates from Tables 13.2.1 and 13.2.2 and Exercise 6. Do not use the bounded operators.

a) $le(x, y) = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{otherwise} \end{cases}$

b) $ge(x, y) = \begin{cases} 1 & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$

c) $btw(x, y, z) = \begin{cases} 1 & \text{if } y < x < z \\ 0 & \text{otherwise} \end{cases}$

d) $prsq(x) = \begin{cases} 1 & \text{if } x \text{ is a perfect square} \\ 0 & \text{otherwise} \end{cases}$

8. Let t be a two-variable primitive recursive function and define f as follows:

$$f(x, 0) = t(x, 0)$$
$$f(x, y + 1) = f(x, y) + t(x, y + 1)$$

Explicitly give the functions g and h that define f by primitive recursion.

9. Let g and h be primitive recursive functions. Use bounded operators to show that the following functions are primitive recursive. You may use any functions and predicates that have been shown to be primitive recursive.

a) $f(x, y) = \begin{cases} 1 & \text{if } g(i) < g(x) \text{ for all } 0 \leq i \leq y \\ 0 & \text{otherwise} \end{cases}$

b) $f(x, y) = \begin{cases} 1 & \text{if } g(i) = x \text{ for some } 0 \leq i \leq y \\ 0 & \text{otherwise} \end{cases}$

c) $f(y) = \begin{cases} 1 & \text{if } g(i) = h(j) \text{ for some } 0 \leq i, j \leq y \\ 0 & \text{otherwise} \end{cases}$

d) $f(y) = \begin{cases} 1 & \text{if } g(i) < g(i + 1) \text{ for all } 0 \leq i \leq y \\ 0 & \text{otherwise} \end{cases}$

e) $nt(x, y) =$ the number of times $g(i) = x$ in the range $0 \leq i \leq y$

f) $thrd(x, y) = \begin{cases} 0 & \text{if } g(i) \text{ does not assume the value } x \text{ at least} \\ & \text{three times in the range } 0 \leq i \leq y \\ j & \text{if } j \text{ is the third integer in the range } 0 \leq i \leq y \\ & \text{for which } g(i) = x \end{cases}$

g) $lrg(x, y) =$ the largest value in the range $0 \leq i \leq y$ for which $g(i) = x$

10. Show that the following functions are primitive recursive.

a) $gcd(x, y) =$ the greatest common divisor of x and y

b) $lcm(x, y) =$ the least common multiple of x and y

c) $pw2(x) = \begin{cases} 1 & \text{if } x = 2^n \text{ for some } n \\ 0 & \text{otherwise} \end{cases}$

11. Let g be a one-variable primitive recursive function. Prove that the function

$$f(x) = \min_{i=0}^{x}(g(i))$$
$$= min\{g(0), \ldots, g(x)\}$$

is primitive recursive.

12. Prove that the function

$$f(x_1, \ldots, x_n) = {}^{u(x_1,\ldots,x_n)}\mu z\,[p(x_1, \ldots, x_n, z)]$$

is primitive recursive whenever p and u are primitive recursive.

13. Compute the Gödel number for the following sequence:
    a) 3, 0
    b) 0, 0, 1
    c) 1, 0, 1, 2
    d) 0, 1, 1, 2, 0

14. Determine the sequences encoded by the following Gödel numbers:
    a) 18,000
    b) 131,072
    c) 2,286,900
    d) 510,510

15. Prove that the following functions are primitive recursive:
    a) $gdn(x) = \begin{cases} 1 & \text{if } x \text{ is the Gödel number of some sequence} \\ 0 & \text{otherwise} \end{cases}$
    b) $gdln(x) = \begin{cases} n & \text{if } x \text{ is the Gödel number of a sequence of length } n \\ 0 & \text{otherwise} \end{cases}$
    c) $g(x, y) = \begin{cases} 1 & \text{if } x \text{ is a Gödel number and } y \text{ occurs in the sequence encoded in } x \\ 0 & \text{otherwise} \end{cases}$

16. Construct a primitive recursive function whose input is an encoded ordered pair and whose output is the encoding of an ordered pair in which the positions of the elements have been swapped. For example, if the input is the encoding of $[x, y]$ then the output is the encoding of $[y, x]$.

17. Let f be the function defined by

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ 2 & \text{if } x = 1 \\ 3 & \text{if } x = 2 \\ f(x-3) + f(x-1) & \text{otherwise.} \end{cases}$$

Prove that f is primitive recursive.

18. Let g_1 and g_2 be one-variable primitive recursive functions. Also let h_1 and h_2 be four-variable primitive recursive functions. The two functions f_1 and f_2 defined by

$$f_1(x, 0) = g_1(x)$$
$$f_2(x, 0) = g_2(x)$$
$$f_1(x, y+1) = h_1(x, y, f_1(x, y), f_2(x, y))$$
$$f_2(x, y+1) = h_2(x, y, f_1(x, y), f_2(x, y))$$

are said to be constructed by simultaneous recursion from g_1, g_2, h_1, and h_2. The values $f_1(x, y+1)$ and $f_2(x, y+1)$ are defined in terms of the previous values of both of the functions. Prove that f_1 and f_2 are primitive recursive.

19. Let f be the function defined by

$$f(0) = 1$$
$$f(y+1) = \sum_{i=0}^{y} f(i)^y.$$

a) Compute $f(1)$, $f(2)$, and $f(3)$.

b) Use course of values recursion to show that f is primitive recursive.

20. Let A be Ackermann's function. (See Section 13.6.)

a) Compute $A(2, 2)$.

b) Prove that $A(x, y)$ has a unique value for every $x, y \in \mathbf{N}$.

c) Prove that $A(1, y) = y + 2$.

d) Prove that $A(2, y) = 2y + 3$.

21. Prove that the following functions are μ-recursive. The functions g and h are assumed to be primitive recursive.

a) $cube(x) = \begin{cases} 1 & \text{if } x \text{ is a perfect cube} \\ \uparrow & \text{otherwise} \end{cases}$

b) $root(c_0, c_1, c_2) =$ the smallest natural number root of the quadratic polynomial $c_2 \cdot x_2 + c_1 \cdot x + c_0$

c) $r(x) = \begin{cases} 1 & \text{if } g(i) = g(i+x) \text{ for some } i \geq 0 \\ \uparrow & \text{otherwise} \end{cases}$

d) $l(x) = \begin{cases} \uparrow & \text{if } g(i) - h(i) < x \text{ for all } i \geq 0 \\ 0 & \text{otherwise} \end{cases}$

e) $f(x) = \begin{cases} 1 & \text{if } g(i) + h(j) = x \text{ for some } i, j \in \mathbf{N} \\ \uparrow & \text{otherwise} \end{cases}$

f) $f(x) = \begin{cases} 1 & \text{if } g(y) = h(z) \text{ for some } y > x,\ z > x \\ \uparrow & \text{otherwise.} \end{cases}$

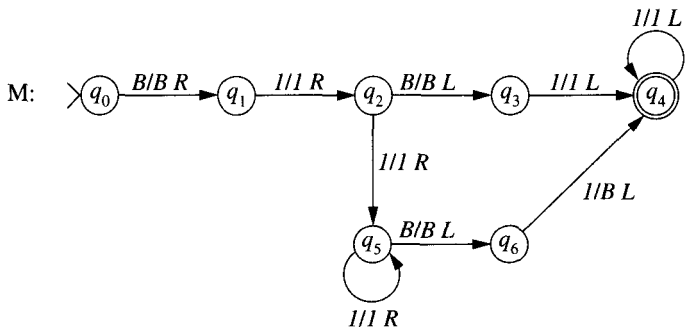
22. The unbounded μ-operator can be defined for partial predicates as follows:

$$\mu z[p(x_1, \ldots, x_n, z)] = \begin{cases} j & \text{if } p(x_1, \ldots, x_n, i) = 0 \text{ for } 0 \leq i < j \\ & \quad \text{and } p(x_1, \ldots, x_n, j) = 1 \\ \uparrow & \text{otherwise.} \end{cases}$$

That is, the value is undefined if $p(x_1, \ldots, x_n, i) \uparrow$ for some i occurring before the first value j for which $p(x_1, \ldots, x_n, j) = 1$. Prove that the family of functions obtained by replacing the unbounded minimalization operator in Definition 13.6.3 with the preceding μ-operator is the family of Turing computable functions.

23. Construct the functions *ns*, *ntp*, and *nts* for the Turing machine S given in Example 13.7.1.

24. Let M be the machine



a) What unary number-theoretic function does M compute?

b) Give the tape numbers for each configuration that occurs in the computation of M with input $\bar{0}$.

c) Give the tape numbers for each configuration that occurs in the computation of M with input $\bar{2}$.

25. Let f be the function defined by

$$f(x) = \begin{cases} x + 1 & \text{if } x \text{ even} \\ x - 1 & \text{otherwise.} \end{cases}$$

a) Give the state diagram of a Turing machine M that computes f.

b) Trace the computation of your machine for input 1 ($B11B$). Give the tape number for each configuration in the computation. Give the value of $tr_{\mathrm{M}}(1, i)$ for each step in the computation.

c) Show that f is primitive recursive. You may use the functions from the text that have been shown to be primitive recursive in Sections 13.1, 13.2 and 13.4.

26. Let M be a Turing machine and tr_{M} the trace function of M.

a) Show that the function

$$prt(x, y) = \begin{cases} 1 & \text{if the } y\text{th transition of M with input } x \text{ prints} \\ & \text{a blank} \\ 0 & \text{otherwise} \end{cases}$$

is primitive recursive.

b) Show that the function

$$fprt(x) = \begin{cases} y & \text{if } y \text{ is the number of the first transition of M} \\ & \text{with input } x \text{ that prints a blank} \\ \uparrow & \text{otherwise} \end{cases}$$

is μ-recursive.

c) In light of undecidability of the printing problem (Exercise 11.16), explain why *fprt* cannot be primitive recursive.

27. Give an example of a function that is not μ-recursive. *Hint*: Consider a language that is not recursively enumerable.

28. Let f be the function from $\{a, b\}^*$ to $\{a, b\}^*$ defined by $f(u) = u^R$. Construct the primitive recursive function f' that, along with the encoding and decoding functions, computes f.

29. A number-theoretic function is said to be **macro-computable** if it can be computed by a Turing machine defined using only the machines S and D that compute the successor and predecessor functions and the macros from Section 12.3. Prove that every μ-recursive function is macro-computable. To do this you must show that

i) The successor, zero, and projection functions are macro-computable.

ii) The macro-computable functions are closed under composition, primitive recursion, and unbounded minimimalization.

30. Prove that the programming language TM defined in Section 12.6 computes the entire set of μ-recursive functions.

Bibliographic Notes

The functional and mechanical development of computability flourished in the 1930s. Gödel [1931] defined a method of computation now referred to as Herbrand-Gödel computability. The properties of Herbrand-Gödel computability and μ-recursive functions were developed extensively by Kleene. The equivalence of μ-recursive functions and Turing computability was established in Kleene [1936]. Post machines, Post [1936], provide an alternative mechanical approach to numeric computation. The transformational work of Markov can be found in Markov [1961]. Needless to say, all these systems compute precisely the μ-recursive functions.

Ackermann's function was introduced in Ackermann [1928]. An excellent exposition of the features of Ackermann's function can be found in Hennie [1977].

The classic book by Kleene [1952] presents computability, the Church-Turing thesis, and recursive functions. A further examination of recursive function theory can be found in Hermes [1965], Péter [1967], and Rogers [1967]. Hennie [1977] develops computability from the notion of an abstract family of algorithms.

PART V

Computational Complexity

A decision problem is solvable or a function computable if there is an effective procedure that determines the result. The objective of the preceding chapters was to characterize computability. We now shift our attention from exhibiting the existence of algorithms to analyzing their efficiency. The performance of an algorithm is measured by the resources required by a computation. Thus we begin a formal analysis of the question *how much* first posed in the introduction.

Since it is the inherent properties of an algorithm that are of interest to us, the analysis should be independent of any particular implementation. To isolate the features of an algorithm from those of the implementation, a single machine must be chosen for analyzing computational complexity. The choice should not place any unnecessary restrictions, such as limiting the time or memory available, upon the computation since these limitations are properties of the implementation and not of the algorithm itself. The standard Turing machine, which fulfills all of these requirements, provides the underlying computational framework for the analysis of algorithms. Moreover, the Church-Turing thesis assures us that any effective procedure can be implemented on such a machine.

Complexity analysis allows us to classify decision problems upon the feasibility of their solutions. A solvable problem may have no practical solutions; every algorithmic solution may require an unacceptable amount of resources. The class $\mathcal{P}$ of decision problems that are considered tractable is defined as the set of problems solvable in polynomial time by a deterministic Turing machine. Another class of problems, $\mathcal{NP}$, consists of all decision problems that can be solved by nondeterministic polynomial time algorithms. Clearly, $\mathcal{P}$ is a subset of $\mathcal{NP}$. It is currently unknown if these two classes of problems are identical. Answering the $\mathcal{P} = \mathcal{NP}$? question is equivalent to deciding whether constructing a solution to a problem and checking whether a single possibility is a solution are of equal difficulty, since a nondeterministic algorithm using a "guess-and-check" strategy need only determine whether the nondeterministic guess is a solution.

CHAPTER 14

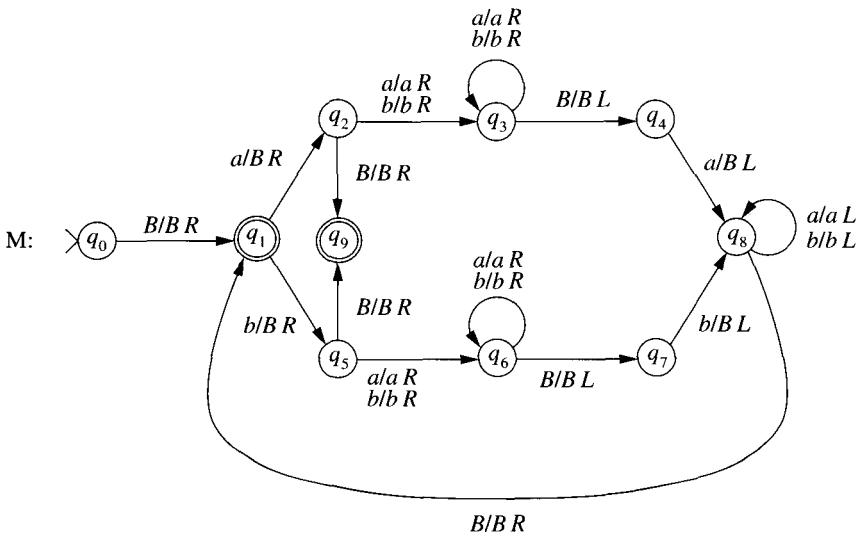
Computational Complexity

The time complexity of a Turing machine computation is measured by the number of transitions in the computation. Because of the variation in the number of transitions in computations initiated with strings of the same length, rates of growth are commonly used to describe the complexity of a Turing machine. We will show that the time complexity of multitrack and multitape Turing machines differs only polynomially from an equivalent standard Turing machine.

The complexity of a language is determined by the complexities of the machines that accept the language. Several important properties of the complexities of languages are established. A machine that accepts a language can be "sped up" to produce another machine whose complexity is reduced by any desired linear factor. We will also see that there is no upper bound on the complexity of languages; for any computable function there is a language whose complexity grows faster than the values of the function.

14.1 Time Complexity of a Turing Machine

The time complexity of a computation measures the amount of work expended by the computation. The time of a computation of a Turing machine is quantified by the number of transitions processed. The issues involved in determining the time complexity of a Turing machine are presented by analyzing the computations of the machine M that accepts palindromes over the alphabet $\{a, b\}$.



A computation of M consists of a loop that compares the first nonblank symbol on the tape with the last. The first symbol is recorded and replaced with a blank by the transition from state q_1. Depending upon the path taken from q_1, the final nonblank symbol is checked for a match in state q_4 or q_7. The machine then moves to the left through the nonblank segment of the tape and the comparison cycle is repeated. When a blank is read in states q_2 or q_5, the string is an odd-length palindrome and is accepted in state q_9. Even-length palindromes are accepted in state q_1.

The computations of M are symmetric with respect to the symbols a and b. The upper path from q_1 to q_8 is traversed when processing an a and the lower path when processing a b. The computations in Table 14.1.1 contain all significant combinations of symbols in strings of length zero, one, two, and three.

As expected, the computations show that the number of transitions in a computation depends upon the particular input string. Indeed, the amount of work may differ radically for strings of the same length. Rather than attempting to determine the exact number of transitions for each input string, the time complexity of a Turing machine is measured by the work required for strings of a fixed length.

Definition 14.1.1

Let M be a standard Turing machine. The **time complexity** of M is the function tc_M : $\mathbf{N} \rightarrow \mathbf{N}$ such that $tc_M(n)$ is the maximum number of transitions processed by a computation of M when initiated with an input string of length n.

TABLE 14.1.1 Computations of M

Length 0	Length 1	Length 2		Length 3	
q_0BB	q_0BaB	q_0BaaB	q_0BabB	q_0BabaB	q_0BaabB
$\vdash Bq_1B$	$\vdash Bq_1aB$	$\vdash Bq_1aaB$	$\vdash Bq_1abB$	$\vdash Bq_1abaB$	$\vdash Bq_1aabB$
	$\vdash BBq_2B$	$\vdash BBq_2aB$	$\vdash BBq_2bB$	$\vdash BBq_2baB$	$\vdash BBq_2abB$
	$\vdash BBBq_9B$	$\vdash BBaq_3B$	$\vdash BBbq_3B$	$\vdash BBbq_3aB$	$\vdash BBaq_3bB$
		$\vdash BBq_4aB$	$\vdash BBq_4bB$	$\vdash BBbaq_3B$	$\vdash BBabq_3B$
		$\vdash Bq_8BBB$		$\vdash BBbq_4aB$	$\vdash BBaq_4bB$
		$\vdash BBq_1BB$		$\vdash BBq_8bBB$	
				$\vdash Bq_8BbBB$	
				$\vdash BBq_1bBB$	
				$\vdash BBBq_5BB$	
				$\vdash BBBBq_9B$	

When evaluating the time complexity of a Turing machine, we assume that the computations terminate for every input string. It makes no sense to attempt to discuss the efficiency, or more accurately the complete inefficiency, of a computation that continues indefinitely.

Definition 14.1.1 serves equally well for machines that accept languages and compute functions. The time complexity of deterministic multitrack and multitape machines is defined in a similar manner. The complexity of nondeterministic machines will be discussed in Section 14.6.

The time complexity gives the worst case performance of the Turing machine. In analyzing an algorithm, we choose the worst case performance for two reasons. The first is that we are considering the limitations of algorithmic computation. The value $tc_M(n)$ specifies the minimum resources required to guarantee that the computation of M terminates when initiated with an input string of length n. The other reason is strictly pragmatic; the worst case performance is often easier to evaluate than the average performance.

The machine M accepting the palindromes over $\{a, b\}$ is used to demonstrate the process of determining the time complexity of a Turing machine. A computation of M terminates when the entire input string has been replaced with blanks or the first nonmatching pair of symbols is discovered. Since the time complexity measures the worst case performance, we need only concern ourselves with the strings whose computations cause the machine to do largest possible number of match-and-erase cycles. This condition is satisfied when the input is accepted by M.

Using these observations, we can obtain the initial values of the function tc_M from the computations in Table 14.1.1.

$$tc_M(0) = 1$$
$$tc_M(1) = 3$$
$$tc_M(2) = 6$$
$$tc_M(3) = 10$$

Determining the remainder of the values of tc_M requires a detailed analysis of the computations of M. Consider the actions of M when processing an even-length input string. The computation alternates between sequences of right and left movements of the machine. Initially, the tape head is positioned to the immediate left of the nonblank segment of the tape.

- *Rightward movement:* The machine moves to the right, erasing the leftmost nonblank symbol. The remainder of the string is read and the machine enters state q_4 or q_7. This requires $k + 1$ transitions, where k is the length of the nonblank portion of the tape.
- *Leftward movement:* M moves left, erasing the matching symbol, and continues through the nonblank portion of the tape. This requires k transitions.

The preceding actions reduce the length of the nonblank portion of the tape by two. The cycle of comparisons and erasures is repeated until the tape is completely blank. As previously noted, the worst case performance for an even-length string occurs when M accepts the input. A computation accepting a string of length n requires $n/2$ iterations of the preceding loop.

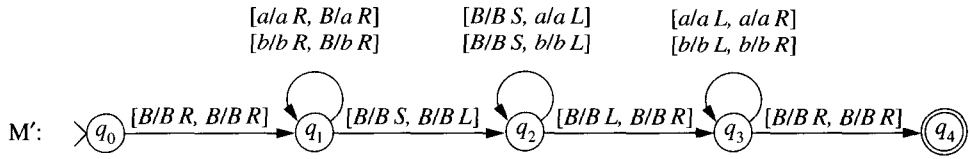
Iteration	Direction	Transitions
1	right	$n + 1$
	left	n
2	right	$n - 1$
	left	$n - 2$
3	right	$n - 3$
	left	$n - 4$
⋮		⋮
$n/2$	right	1

The total number of transitions of a computation can be obtained by adding those of each iteration. As indicated by the preceding table, the maximum number of transitions in a computation of a string of even length n is the sum of the first $n + 1$ natural numbers. An analysis of odd-length strings produces the same result. Consequently, the time complexity of M is given by the function

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

Example 14.1.1

The two-tape machine M′



also accepts the set of palindromes over $\{a, b\}$. A computation of M′ traverses the input, making a copy on tape 2. The head on tape 2 is then moved back to tape position 0. At this point, the heads move across the input, tape 1 right to left and tape 2 left to right, comparing the symbols on tape 1 and tape 2. If the tape heads ever encounter different symbols, the input is not a palindrome and the computation halts in a nonaccepting state. When the input is a palindrome, the computation halts and accepts when blanks are simultaneously read on tapes 1 and 2.

For an input of length n, the maximum number of transitions occurs when the string is a palindrome. An accepting computation requires three complete passes: the copy, the rewind, and the comparison. Counting the number of transitions in each pass, we see that the time complexity M′ is $tc_{M'}(n) = 3(n + 1) + 1$. □

The definition of time complexity tc_M is predicated on the computations of the machine M and not on the underlying language accepted by the machine. We know that many different machines can be constructed to accept the same language, each with different time complexities. We say that a language L is accepted in deterministic time $f(n)$ if there is a deterministic Turing machine M of any variety with $tc_M(n) \leq f(n)$ for all $n \in \mathbf{N}$. The machine M shows that set of palindromes over $\{a, b\}$ is accepted in time $(n^2 + 3n + 2)/2$ while the two-tape machine M′ exhibits acceptance in time $3n + 4$.

A transition of the two-tape machine utilizes more information and performs a more complicated operation than that of a one-tape machine. The additional structure of the two-tape transition allowed the reduction in the number of transitions required by M′ to process a string over that required by the one-tape machine M. Thus we see a trade-off between the complexity of the transitions and the number that must be processed. The precise relationship between the time complexity of one-tape and multitape Turing machines is established in Section 14.4.

14.2 Linear Speedup

The time complexity function $tc_M(n)$ of a Turing machine M gives the maximum number of transitions required for a computation with an input string of length n. In this section

we show that a machine that accepts a language L can be "sped up" to produce another machine that accepts L in time that is faster by an arbitrary multiplicative constant.

Let M = (Q, Σ, Γ, δ, q_0, F) be a k-tape Turing machine, $k > 1$, that accepts a language L. The underlying strategy involved in the speedup is to construct a machine N that accepts L in which a block of six transitions of N simulates m transitions of M, where the value of m is determined by the desired degree of speedup. Since the machines M and N accept the same language, the input alphabet of N is also Σ. The tape alphabet of N includes that of M as well as the symbol # and all ordered m-tuples of symbols of Γ.

During the simulation of M, a tape symbol of N is an m-tuple of symbols of M and the states of N are used to record the portion of the tapes of M that may be affected by the next m transitions. In this phase of the computation of N, a state of N consists of

i) the state of M

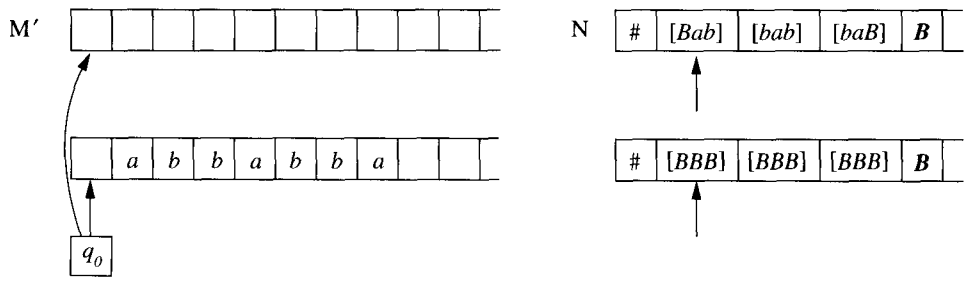
ii) for $i = 1$ to k, the m-tuple currently scanned on tape i of N and the m-tuples to the immediate right and left

iii) an ordered k-tuple $[i_1, \ldots, i_k]$, where i_j is the position of the symbol on tape j being scanned by M in the m-tuple being scanned by N.

A sequence of six transitions of N uses the information in the state to simulate m transitions of M.

The process will be demonstrated using the two-tape machine M′ from Example 14.1.1, with $m = 3$ and input *abbabba*. The input configuration of N is exactly that of M′, with the input string on tape 1 and tape 2 entirely blank. The first step of N is to encode the input string into m-tuples. The process begins by writing # on position 1 of both tapes. For every three consecutive symbols on tape 1, an ordered triple is written on tape 2. The final ordered triple written on tape 2 is padded with blanks since the length of *Babbabba* is not evenly divisible by three. The tape heads of N are repositioned at tape position one and the original input string is erased from tape 1. For the remainder of the computation, tape 2 of N will simulate tape 1 of M′ and tape 1 of N will simulate tape 2 of M′.

The diagram below shows the initial configuration of M′ with input *abbabba* and the configuration of N after the encoding.

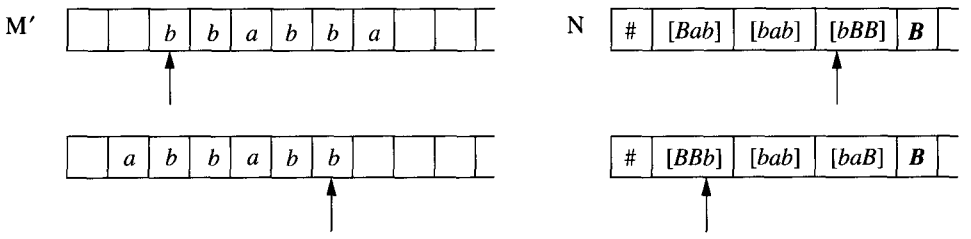


Each blank on the tape of N will be considered to represent an the encoded triple $[BBB]$ of blanks of M′. To illustrate the difference in the diagrams, the blanks of N will be written $\boldsymbol{B}$. After the encoding of the input, N will enter the state

$$(q_0;\ ?, BBB, ?;\ ?, Bab, ?;\ [1, 1]).$$

The strings *BBB* and *Bab* are those currently scanned by N on tapes 1 and 2, respectively. The ordered pair [1, 1] indicates that the computation of M′ is scanning the symbol that occurs in the first position in each of the triples *BBB* and *Bab* in the state of N. Subsequent transitions will cause N to enter states in which each ? has been replaced with information concerning the triples to the left and right of the position currently being scanned.

The simulation of m moves of M′ is demonstrated by considering the configurations of M′ and N



that would be obtained during the processing of *abbabba*. Upon entering this configuration, the state of N is

$$(q_3;\ ?, BBb, ?;\ ?, bBB, ?;\ [3, 1]).$$

The ordered pair [3, 1] in the state indicates that the computation of M′ is reading the b in the triple $[BBb]$ on tape 1 and the b in the triple $[bBB]$ on tape 2.

N then makes a move to the left on each tape, scans the squares, and enters state

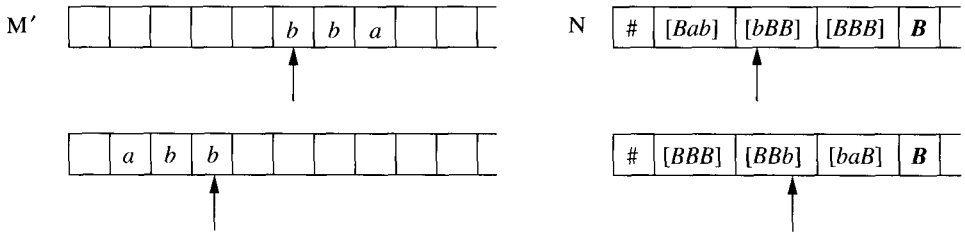
$$(q_3;\ \#, BBb, ?;\ bab, bBB, ?;\ [3, 1]),$$

which records the triples to the left of the originally scanned squares in the state. The role of the marker # is to ensure that N will not cross a left-hand tape boundary in this phase of the simulation. Two moves to the right leaves N in state

$$(q_3;\ \#, BBb, bab;\ bab, bBB, BBB;\ [3, 1]),$$

recording the triple to the right of the originally scanned positions. N then moves its tape heads left to return to the original position. At this point, the state of N contains all the information that can be altered by three transitions of M′.

N then rewrites its tapes to match the configuration that M′ will enter after three transitions



and enters state

$$(q_3;\ ?, BBb, ?;\ ?, bBB, ?;\ [3, 1])$$

to begin the simulation of the next three transitions of M′. Rewriting the tape of N requires at most two transitions. Since each tape square of N has three symbols of M′, the portion of the tape of M′ that can be altered by three transitions is contained in the tape square currently being scanned by N and either the square to the immediate right or immediate left of the square being scanned but not both. Consequently, at most two transitions of N are required to update its tape and prepare for the continuation of the simulation of M′. The simulation of transitions of M′ continues until M′ halts, in which case N will halt and return the same indication of membership as M′.

Theorem 14.2.1

Let M be a k-tape Turing machine, $k > 1$, that accepts L with $tc_{\mathrm{M}}(n) = f(n)$. Then there is a k-tape machine N that accepts L with $tc_{\mathrm{N}}(n) \leq \lceil cf(n) \rceil + 2n + 3$ for any constant $c > 0$.

Proof The construction of the machine N is outlined above. Encoding an input string of length n as m-tuples and repositioning the tape heads require $2n + 3$ transitions.

The remainder of the computation of N consists of the simulation of the computation of M. To obtain and record the information needed to simulate m transitions of M, N takes one move to the left, two to the right, and one to reposition the head at the original position. At most two transitions are then required to reconfigure the tapes of N. Thus six transitions of N are sufficient to produce the same result as m of the machine M. Choosing $m > 6/c$ produces

$$\begin{aligned} tc_{\mathrm{N}}(n) &= \lceil (6/m) f(n) \rceil + 2n + 3 \\ &\leq \lceil cf(n) \rceil + 2n + 3, \end{aligned}$$

as desired. ■

Corollary 14.2.2

Let M be a one-tape Turing machine that accepts L with $tc_{\mathrm{M}}(n) = f(n)$. Then there is a two-tape machine N that accepts L with $tc_{\mathrm{N}}(n) \leq \lceil cf(n) \rceil + 2n + 3$ for any constant $c > 0$.

TABLE 14.3.1 Growth of Functions

n	0	5	10	25	50	100	1,000
$20n + 500$	500	600	700	1,000	1,500	2,500	20,500
n^2	0	25	100	625	2,500	10,000	1,000,000
$n^2 + 2n + 5$	5	40	125	680	2,605	10,205	1,002,005
$n^2/(n^2 + 2n + 5)$	0	0.625	0.800	0.919	0.960	0.980	0.998

Proof In the standard manner, the one-tape machine M can be considered to be a two-tape machine in which the second tape is not referenced in the computation. Theorem 14.2.1 can then be used to speed up the two-tape machine. ■

14.3 Rates of Growth

The speedup theorem showed that determining the exact complexity of a language is futile, since a machine can always be "improved" to accept the language in time that is faster by any multiplicative constant. This is one reason why the time complexity is often represented by the rate of growth of a function rather than by the function itself. Before continuing with our evaluation of the time complexity of Turing machines, we detour for a brief introduction to the mathematical analysis of the rate of growth of functions.

The rate of growth of a function measures the increase of the function values as the input gets arbitrarily large. Intuitively, the rate of growth is determined by the most significant contributor to the growth of the function. The contribution of the individual terms to the values of a function can be seen by examining the growth of the functions n^2 and $n^2 + 2n + 5$ in Table 14.3.1. The contribution of n^2 to $n^2 + 2n + 5$ is measured by the ratio of the function values in the bottom row. The linear and constant terms of the function $n^2 + 2n + 5$ are called the *lower-order terms*. Lower-order terms may exert undue influence on the initial values of the functions. As n gets large, it is clear that the lower-order terms do not significantly contribute to the growth of the function values. Although n^2 grows faster than the linear function $20n + 500$, this is not exhibited for input values less than 25.

Definition 14.3.1

Let f and g be one-variable number-theoretic functions. The function f is said to be of **order** g, written $f = O(g)$, if there is a positive constant c and a natural number n_0 such that

$$f(n) \leq c \cdot g(n)$$

for all $n \geq n_0$.

The order of a function provides an upper bound to the values of the function as the input increases. The function f is of order g if the growth of f is bounded by a constant multiple of the values of g. Because of the influence of the lower-order terms, the inequality $f(n) \leq c \cdot g(n)$ is required to hold only for input values greater than some specified number. The notation $f = O(g)$ is read "f is big oh of g."

Two functions f and g are said to have the same rate of growth if $f = O(g)$ and $g = O(f)$. When f and g have the same rate of growth, Definition 14.3.1 provides the two inequalities

$$f(n) \leq c_1 \cdot g(n) \quad \text{for } n \geq n_1$$
$$g(n) \leq c_2 \cdot f(n) \quad \text{for } n \geq n_2,$$

where c_1 and c_2 are positive constants. Combining these inequalities, we see that each of these functions is bounded above and below by constant multiples of the other.

$$f(n)/c_1 \leq g(n) \leq c_2 \cdot f(n)$$
$$g(n)/c_2 \leq f(n) \leq c_1 \cdot g(n)$$

These relationships hold for all n greater than the maximum of n_1 and n_2. Because of these bounds, it is clear that neither f nor g can grow faster than the other.

Example 14.3.1

Let $f(n) = n^2 + 2n + 5$ and $g(n) = n^2$. Then $f = O(g)$ and $g = O(f)$. Since

$$n^2 \leq n^2 + 2n + 5$$

for all natural numbers, setting c to 1 and n_0 to 0 satisfies the conditions of Definition 14.3.1. Consequently, $g = O(f)$.

To establish the opposite relationship, we begin by noting that $2n \leq 2n^2$ and $5 \leq 5n^2$ for all $n \geq 1$. Then,

$$\begin{aligned} f(n) &= n^2 + 2n + 5 \\ &\leq n^2 + 2n^2 + 5n^2 \\ &= 8n^2 \\ &= 8 \cdot g(n) \end{aligned}$$

whenever $n \geq 1$. In the big oh terminology, the preceding inequality shows that $n^2 + 2n + 5 = O(n^2)$. □

Example 14.3.2

Let $f(n) = n^2$ and $g(n) = n^3$. Then $f = O(g)$ and $g \neq O(f)$. Clearly, $n^2 = O(n^3)$ since $n^2 \leq n^3$ for all natural numbers.

Let us suppose that $n^3 = O(n^2)$. Then there are constants c and n_0 such that

$$n^3 \leq c \cdot n^2 \quad \text{for all } n \geq n_0.$$

Choose n_1 to be the maximum of $n_0 + 1$ and $c + 1$. Then, $n_1^3 = n_1 \cdot n_1^2 > c \cdot n_1^2$ and $n_1 > n_0$, contradicting the inequality. Thus our assumption is false and $n^3 \neq O(n^2)$. □

A **polynomial with integral coefficients** is a function of the form

$$f(n) = c_r \cdot n^r + c_{r-1} \cdot n^{r-1} + \cdots + c_1 \cdot n + c_0,$$

where the $c_0, c_1, \ldots, c_{r-1}$ are arbitrary integers, c_r is a nonzero integer, and r is a positive integer. The constants c_i are the coefficients of f and r is the degree of the polynomial. A polynomial with integral coefficients defines a function from the natural numbers into the integers. The presence of negative coefficients may produce negative values. For example, if $f(n) = n^2 - 3n - 4$, then $f(0) = -4$, $f(1) = -6$, $f(2) = -6$, and $f(3) = -4$. The values of the polynomial $g(n) = -n^2 - 1$ are negative for all natural numbers n.

The rate of growth has been defined only for number-theoretic functions. The absolute value function can be used to transform an arbitrary polynomial into a number-theoretic function. The absolute value of an integer i is the nonnegative integer defined by

$$|i| = \begin{cases} i & \text{if } i \geq 0 \\ -i & \text{otherwise.} \end{cases}$$

Composing a polynomial f with the absolute value produces a number-theoretic function $|f|$. The rate of growth of a polynomial f is defined to be that of $|f|$.

The techniques presented in Examples 14.3.1 and 14.3.2 can be used to establish a general relationship between the degree of a polynomial and its rate of growth.

Theorem 14.3.2

Let f be a polynomial of degree r. Then

i) $f = O(n^r)$

ii) $n^r = O(f)$

iii) $f = O(n^k)$ for all $k > r$

iv) $f \neq O(n^k)$ for all $k < r$.

One of the consequences of Theorem 14.3.2 is that the rate of growth of any polynomial can be characterized by a function of the form n^r. The first two conditions show that a polynomial of degree r has the same rate of growth as n^r. Moreover, by conditions (iii) and (iv), its growth is not equivalent to that of n^k for any k other than r.

Other important functions used in measuring the performance of algorithms are the logarithmic, exponential, and factorial functions. A number-theoretic logarithmic function with base a is defined by

$$f(n) = \lfloor \log_a(n) \rfloor.$$

Changing the base of a logarithmic function alters the value by a constant multiple. More precisely,

$$\log_a(n) = \log_b(n)\log_a(b).$$

This relationship indicates that the rate of growth of the logarithmic functions is independent of the base.

Theorem 14.3.3 compares the growth of these functions with each other and the polynomials. The proofs are left as exercises.

Theorem 14.3.3

Let r be a natural number and let a and b be real numbers greater than 1. Then

i) $\log_a(n) = O(n)$
ii) $n \neq O(\log_a(n))$
iii) $n^r = O(b^n)$
iv) $b^n \neq O(n^r)$
v) $b^n = O(n!)$
vi) $n! \neq O(b^n)$.

A function f is said to **polynomially bounded** if $f = O(n^r)$ for some natural number r. Although not a polynomial, it follows from condition (i) that $n \log_2(n)$ is bounded by the polynomial n^2. The polynomially bounded functions, which of course include the polynomials, constitute an important family of functions that will be associated with the time complexity of efficiently solvable algorithms. Conditions (iv) and (vi) show that the exponential and factorial functions are not polynomially bounded. A big oh hierarchy can be constructed from the relationships outlined in Theorems 14.3.2 and 14.3.3. The elements in Table 14.3.2 are listed in the order of their rates of growth. It is standard practice to refer to a function f for which $2^n = O(f)$ as having *exponential growth*. With this convention, n^n and $n!$ are both said to exhibit exponential growth.

The efficiency of an algorithm is commonly characterized by its rate of growth. A polynomial time algorithm, or simply a polynomial algorithm, is one whose time complexity is polynomially bounded. That is, $tc_M = O(n^r)$ for some $r \in \mathbf{N}$, where M is a standard Turing machine that computes the algorithm. The distinction between polynomial and nonpolynomial algorithms is apparent when considering the number of transitions of a computation as the length of the input increases. Table 14.3.3 illustrates the enormous resources required by an algorithm whose time complexity is not polynomial.

TABLE 14.3.2 A Big O Hierarchy

Big Oh	Name
$O(1)$	constant
$O(\log_a(n))$	logarithmic
$O(n)$	linear
$O(n \log_a(n))$	$n \log n$
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^r)$	polynomial $r \geq 0$
$O(b^n)$	exponential $b > 1$
$O(n!)$	factorial

TABLE 14.3.3 Number of Transitions of Machine with Time Complexity tc_M with Input of Length n

n	tc_M					
	$\log_2(n)$	n	n^2	n^3	2^n	$n!$
5	2	5	25	125	32	120
10	3	10	100	1,000	1,024	3,628,800
20	4	20	400	8,000	1,048,576	$2.4 \cdot 10^{18}$
30	4	30	900	27,000	$1.0 \cdot 10^9$	$2.6 \cdot 10^{32}$
40	5	40	1,600	64,000	$1.1 \cdot 10^{12}$	$8.1 \cdot 10^{47}$
50	5	50	2,500	125,000	$1.1 \cdot 10^{15}$	$3.0 \cdot 10^{64}$
100	6	100	10,000	1,000,000	$1.2 \cdot 10^{30}$	$> 10^{157}$
200	7	200	40,000	8,000,000	$1.6 \cdot 10^{60}$	$> 10^{374}$

14.4 Complexity and Turing Machine Variations

Several variations on the Turing machine model were presented in Chapter 9 to facilitate the design of machines to perform complex computations. The machines designed in Section 14.1 to accept the palindromes over $\{a, b\}$ exhibit the potential differences in computational resources required by one-tape and two-tape machines. In this section we examine the relationship between the complexity of computations in various Turing machines models.

Theorem 14.4.1

Let L be accepted by a k-track deterministic Turing machine M with time complexity $tc_M(n) = f(n)$. Then L is accepted by a standard Turing machine M′ with time complexity $tc_{M'}(n) = f(n)$.

Proof This follows directly from the construction of a one-track Turing machine M′ from a k-track machine in Section 9.4. The alphabet of the one-track machine consists of k-tuples of symbols from the tape alphabet of M. A transition of M has the form $\delta(q_i, x_1, \ldots, x_k)$, where $x_1, \ldots, x_k$ are the symbols on track 1, track 2, . . . , track k. The associated transition of M′ has the form $\delta(q_i, [x_1, \ldots, x_k])$, where the k-tuple is the single alphabet symbol of M′. Thus the number of transitions processed by M and M′ are identical for every input string and $tc_M = tc_{M'}$. ■

Theorem 14.4.2

Let L be accepted by a k-tape deterministic Turing machine M with time complexity $tc_M(n) = f(n)$. Then L is accepted by a standard Turing machine N with time complexity $tc_N(n) = O(f(n)^2)$.

Proof The construction of an equivalent one-tape machine from a k-tape machine uses a $2k + 1$-track machine M′ as an intermediary. By Theorem 14.4.1, all that is required is to show that $tc_{M'} = O(f(n)^2)$.

The argument follows the construction of the multitrack machine M′ that simulates the actions of a multitape machine described in Section 9.6. We begin by analyzing the number of transitions of M′ that are required to simulate a single transition of M.

Assume that we are simulating the tth transition of M. The furthest right that a tape head of M may be at this time is tape position t. The first step in the simulation consists of recording the symbols on the odd-numbered tapes marked by the X's on the even-numbered tapes. This consists of the following sequence of transitions of M′:

Action	Maximum Number of Transitions of M′
find X on second track	
and return to tape position 0	$2t$
find X on fourth track	
and return to tape position 0	$2t$
⋮	⋮
find X on $2k$th track	
and return to tape position 0	$2t$

After finding the symbol under each X, M′ uses one transition to record the action taken by M. The simulation of the transition of M is completed by

Action	Maximum Number of Transitions of M′
write symbol on track 1, reposition X on track 2, and return to tape position 0	$2(t+1)$
write symbol on track 3, reposition X on track 4, and return to tape position 0	$2(t+1)$
⋮	⋮
write symbol on track $2k-1$, reposition X on track $2k$, and return to tape position 0	$2(t+1)$

Consequently, the simulation of the tth transition of M requires at most $4kt + 2k + 1$ transitions of M′. The computation of M′ begins with a single transition that places the markers on the even-numbered tracks and the # on track $2k + 1$. The remainder of the computation consists of the simulation of the transitions of M. An upper bound on number of transitions of M′ needed to simulate the computation of M with input of length n is

$$tc_{M'}(n) \leq 1 + \sum_{t=1}^{f(n)} (4kt + 2k + 1) = O(f(n)^2). \quad \blacksquare$$

14.5 Properties of Time Complexity

This section establishes two interesting results on the bounds of the time complexity of languages. First we show that, for any computable total function $f(n)$, there is a language whose time complexity is not bounded by $f(n)$. That is, there is a sufficiently complex recursive language such that, for some n, more than $f(n)$ transitions are required to determine membership for strings of length n.

We then show that there are languages for which no "best" accepting Turing machine exists. Theorem 14.2.1 has already demonstrated that a machine accepting a language can be sped up linearly. That process, however, does not change the rate of growth of the accepting machine. We will now show that there are languages which, when accepted by any machine, are also accepted by a machine whose time complexity grows at a strictly smaller rate than the original machine.

Both of these results utilize the ability to encode and enumerate all multitape Turing machines. An encoding of one-tape machines as strings over $\{0, 1\}$ was outlined in Section 11.3. This approach can be extended to an encoding of all multitape machines with input alphabet $\{0, 1\}$. The tape alphabet is assumed to consist of elements $\{0, 1, B, x_1, \ldots, x_n\}$. The tape symbols are encoded as follows:

Symbol	Encoding
0	*1*
1	*11*
B	*111*
x_1	*1111*
⋮	⋮
x_n	1^{n+3}

As before, a number is encoded by its unary representation and a transition by its encoded components separated by *0*'s; encoded transitions are separated by *00*. With these conventions, a k-tape machine may be encoded

$$\overline{k}000en(\text{accepting states})000en(\text{transitions})000,$$

where $\overline{k}$ is the unary representation of k and *en* denotes the encoding of the items in parantheses.

With this representation, every string over $u \in \{0, 1\}^*$ can be considered to be the encoding of some multitape Turing machine. If u does not satisfy the syntactic conditions for the encoding of a multitape Turing machine, the string is interpreted as the representation of the one-tape, one-state Turing machine with no transitions.

In Exercise 9.28, a Turing machine E that enumerated all strings over $\{0, 1\}$ was constructed. Since every such string also represents a multitape Turing machine, the machine E can equally well be thought of as enumerating all Turing machines with input alphabet $\{0, 1\}$. The strings enumerated by E will be written $u_0, u_1, u_2, \ldots$ and the corresponding machines by $M_0, M_1, M_2, \ldots$.

Theorem 14.5.1

Let f be a total computable function. Then there is a language L such that tc_M is not bounded by f for any deterministic Turing machine M that accepts L.

Proof Let F be a Turing machine that computes the function f. Consider the language $L = \{u_i \mid M_i \text{ does not accept } u_i \text{ in } f(n) \text{ or fewer moves, where } n = length(u_i)\}$. First we show that L is recursive and then that the number of transitions of any machine that accepts L is not bounded by $f(n)$.

A machine M that accepts L is described below. The input to M is a string u_i in $\{0, 1\}^*$. Recall that the string u_i represents the encoding of the Turing machine M_i in the enumeration of all multitape Turing machines. A computation of M

1. Determines the length of u_i, say $length(u_i) = n$.
2. Simulates the computation of F to determine $f(n)$.
3. Simulates M_i on u_i until either M_i halts or completes $f(n)$ transitions, whichever comes first.

4. M accepts u_i if either M_i halted without accepting u_i or M_i did not halt in the first $f(n)$ transitions. Otherwise, u_i is rejected by M.

Clearly the language L(M) is recursive, since step 3 ensures that each computation will terminate.

Now let M be any Turing machine that accepts L. Then M occurs somewhere in the enumeration of Turing machines, say $M = M_j$. A diagonalization argument is used to show that the computations of M_j are not bounded by f. That is, we will show that it is not the case that $tc_{M_j}(n) \leq f(n)$ for all $n \in \mathbf{N}$. The diagonalization is obtained by considering the membership of u_j in L. Since $L(M_j) = L$, M_j accepts u_j if, and only if, M_j halts without accepting u_j in $f(n)$ or fewer transitions or M_j does not halt in the first $f(n)$ transitions.

The proof M_j is not bounded by f is by contradiction. Assume that the time complexity of M_j is bounded by f and let $n = length(u_j)$. There are two cases to consider: either $u_j \in L$ or $u_j \notin L$.

If $u_j \in L$, then M_j accepts u_j in $f(n)$ or fewer transitions (since the computations of M_j are assumed to be bounded by f). But, as previously noted, M_j accepts u_j if, and only if, M_j halts without accepting u_j or M_j does not halt in the first $f(n)$ transitions.

If $u_j \notin L$, then the computation of M_j halts within the bound of $f(n)$ steps or fewer and does not accept u_j. In this case, $u_j \in L$ by the definition of L.

In either case, the assumption that the number of transitions of M_j is bounded by f leads to a contradiction. Consequently, we conclude that time complexity of any machine that accepts L is not bounded by f. ■

Next we show that there is a language that has no fastest accepting machine. To illustrate how this might occur, consider a sequence of machines N_0, N_1, . . . that all accept the same language over 0^*. The argument uses the function t that is defined recursively by

i) $t(1) = 2$

ii) $t(n) = 2^{t(n-1)}$.

Thus $t(2) = 2^2$, $t(3) = 2^{2^2}$, and $t(n)$ is a series of n 2's as a sequence of exponents. The number of transitions of machine N_i when run with input 0^j is given in the $[i, j]$th position of the Table 14.5.1. A * in the $[i, j]$th position indicates that number of transitions of this computation is irrelevant.

If such a sequence of machines exists, then

$$tc_{N_i}(n) = \log(tc_{N_{i-1}}(n))$$

for all $n \geq i + 1$. Consequently, we have a sequence of machines that accept the same language in which each machine has a strictly smaller rate of growth than its predecessor. In Theorem 14.5.2 a language is constructed that exhibits the "this can always be accepted more efficiently" property.

The speedup in both the motivating discussion and in the construction in Theorem 14.5.2 uses the property that rates of growth measure the performance of the function as the input gets arbitrarily large. From the pattern in the Table 14.5.1, we see that the

TABLE 14.5.1 Machines N_i and Their Computations

	λ	0	00	0^3	0^4	0^5	0^6	...
N_0	*	2	4	$t(3)$	$t(4)$	$t(5)$	$t(6)$	
N_1	*	*	2	4	$t(3)$	$t(4)$	$t(5)$	
N_2	*	*	*	2	4	$t(3)$	$t(4)$	
N_3	*	*	*	*	2	4	$t(3)$	
N_4	*	*	*	*	*	2	4	
⋮								

computations of machines N_i and N_{i+1} are compared only on input strings of length $i + 2$ or greater.

Theorem 14.5.2

There is a language L such that for any machine M that accepts L there is another machine M′ that accepts L with $tc_{M'}(n) = O(\log(tc_M(n)))$.

Let t be the function defined recursively by $t(1) = 2$ and $t(n) = 2^{t(n-1)}$ for $n > 1$ as above. A recursive language $L \subseteq \{0\}^*$ is constructed so that

1. If M_i accepts L, then $tc_{M_i}(n) \geq t(n - i)$ for all n greater than some n_i.
2. For each k, there is a Turing machine M_j with $L(M_j) = L$ and $tc_{M_j}(n) \leq t(n - k)$ for all n greater than some n_k.

Assume that L has been constructed to satisfy the preceding conditions; then for every machine M_i that accepts L there is an M_j that also accepts L with

$$tc_{M_j}(n) = O(\log(tc_{M_i}(n))),$$

as desired. To see this, set $k = i + 1$. By condition 2, there is a machine M_j that accepts L and $tc_{M_j}(n) \leq t(n - i - 1)$ for all $n \geq n_k$. However, by condition 1,

$$tc_{M_i}(n) \geq t(n - i) \text{ for all } n > n_i.$$

Combining the two inequalities with definition of t yields

$$tc_{M_i}(n) \geq t(n - i) = 2^{t(n-i-1)} \geq 2^{tc_{M_j}(n)} \text{ for all } n > \max\{n_i, n_k\}.$$

That is, $tc_{M_j}(n) \leq \log(tc_{M_i}(n))$ for all $n > \max\{n_i, n_k\}$.

We now define the construction of the language L. Sequentially, we determine whether strings 0^n, $n = 0, 1, 2, \ldots$, are in L. During this construction, Turing machines in the enumeration $M_0, M_1, M_2, \ldots$ are marked as canceled. In determining whether $0^n \in L$, we examine a machine $M_{g(n)}$ where $g(n)$ is the least value j in the range $0, \ldots, n$ such that

i) M_j has not been previously canceled

ii) $tc_{M_j}(n) < t(n - j)$.

It is possible that no such value j may exist, in which case $g(n)$ is undefined. $0^n \in L$ if, and only if, $g(n)$ is defined and $M_{g(n)}$ does not accept 0^n. If $g(n)$ is defined, then $M_{g(n)}$ is marked as canceled. The definition of L ensures that a canceled machine cannot accept L. If $M_{g(n)}$ is canceled, then $0^n \in L$ if, and only if, 0^n is not accepted by $M_{g(n)}$. Consequently, $L(M_{g(n)}) \neq L$.

The proof of Theorem 14.5.2 consists of establishing the following three lemmas. The first shows that the language L is recursive. The final two demonstrate that conditions 1 and 2 stated above are satisfied by L.

Lemma 14.5.3

The language L is recursive.

Proof The definition of L provides a method for deciding whether $0^n \in L$. The decision process for 0^n begins by determining the index $g(n)$, if it exists, of the first machine in the sequence $M_0, \ldots, M_n$ that satisfies conditions (i) and (ii). To accomplish this, it is necessary to determine the machines in the sequence $M_0, M_1, \ldots, M_{n-1}$ that have been canceled in the analysis of input $\lambda, 0, \ldots, 0^{n-1}$. This requires comparing the value of the complexity functions in Table 14.5.2 with the appropriate value of t. Since the input alphabet consists of the single character 0, $tc_{M_i}(m)$ can be determined by simuluating the computation of M_i with input 0^m.

After the machines that have been canceled are recorded, the computations with input 0^n are used to determine $g(n)$. Beginning with $j = 0$, if M_j has not previously been canceled, then $t(n - j)$ is computed and the computation of M_j on 0^n is simulated. If $tc_{M_j}(n) < t(n - j)$, then $g(n) = j$. If not, j is incremented and the comparison is repeated until $g(n)$ is found or until all the machines $M_0, \ldots, M_n$ have been tested.

If $g(n)$ exists, $M_{g(n)}$ is run with input 0^n. The result of this computation determines the membership of 0^n in L: $0^n \in L$ if, and only if, $M_{g(n)}$ does not accept it. The preceding process describes a decision procedure that determines the membership of any string 0^n in L; hence, L is recursive. ■

Lemma 14.5.4

L satisfies condition 1.

Proof Assume M_i accepts L. First note that there is some integer p_i such that if a machine $M_0, M_1, \ldots, M_i$ is ever canceled, it is canceled prior to examination of the string 0^{p_i}. Since the number of machines in the sequence $M_0, M_1, \ldots, M_i$ that are canceled is finite, at some point in the generation of L all of those that are canceled will be so marked. We may not know for what value of p_i this occurs, but it must occur sometime and that is all that we require.

For any 0^n with n greater than the maximum of p_i and i, no M_k with $k < i$ can be canceled. Suppose $tc_{M_i}(n) < t(n - i)$. Then M_i would be canceled in the examination of

TABLE 14.5.2 Computations to Determine Canceled Machines

Input	m	Comparison $tc_{M_i}(m) \leq t(m-i)$
λ	0	$tc_{M_0}(0) \leq t(0-0) = t(0)$
0	1	$tc_{M_0}(1) \leq t(1-0) = t(1)$
		$tc_{M_1}(1) \leq t(1-1) = t(0)$
00	2	$tc_{M_0}(2) \leq t(2-0) = t(2)$
		$tc_{M_1}(2) \leq t(2-1) = t(1)$
		$tc_{M_2}(2) \leq t(2-2) = t(0)$
$\vdots$	$\vdots$	$\vdots$
0^{n-1}	$n-1$	$tc_{M_0}(n-1) \leq t(n-1-0) = t(n-1)$
		$tc_{M_1}(n-1) \leq t(n-1-1) = t(n-2)$
		$tc_{M_2}(n-1) \leq t(n-1-2) = t(n-3)$
		$\vdots$
		$tc_{M_{n-1}}(n-1) \leq t(n-1-(n-1)) = t(0)$

0^n. However, a Turing machine that is canceled cannot accept L. It follows that $tc_{M_i}(n) \geq t(n-i)$ for all $n > \max\{p_i, i\}$. ■

Lemma 14.5.5

L satisfies condition 2.

Proof We must prove, for any integer k, that there is a machine M that accepts L and $tc_M(n) \leq t(n-k)$ for all n greater than some value n_k. We begin with the machine M that accepts L described in Lemma 14.5.3. To decide if 0^n is in L, M determines the value $g(n)$ and simulates $M_{g(n)}$ on 0^n. To establish $g(n)$, M must determine which of the M_i's, $i < n$, have been canceled during the analysis of strings λ, 0, 00, . . . , 0^{n-1}. Unfortunately, a straightforward evaluation of these cases as illustrated in Table 14.5.2 may require more than $t(n-k)$ transitions.

As noted in Lemma 14.5.4, any Turing machine M_i, $i \leq k$, that is ever canceled is canceled when considering some initial sequence λ, 0, 00, . . . , 0^{p_k} of input strings. This value p_k can be used to reduce the complexity of the computation described above. For each $m \leq p_k$, the information on whether 0^m is accepted is stored in states of the machine M that accepts L.

The computation of machine M with input 0^n then can be split into two cases.

Case 1: $n \leq p_k$ The membership of 0^n in L is determined solely using the information recorded in the states of M.

Case 2: $n > p_k$ The first step is to determine $g(n)$. This is accomplished by simulating the computation of Turing machines M_i, $i = k+1, \ldots, n$ on inputs 0^m, $m = k+1, \ldots,$

n to see if M_i is canceled on or before 0^n. We only need to check machines in the range $M_{k+1}, \ldots, M_n$ since no machine $M_0, \ldots, M_k$ will be canceled by an input of length greater than p_k.

The ability to skip the simulations of machines $M_0, \ldots, M_k$ reduces the number of transitions needed to evaluate an input string 0^n with $n > p_k$. The number of simulations indicated in Table 14.5.2 is reduced to

Input	m	Comparison $tc_{M_i}(m) \leq t(m-i)$
0^{k+1}	$k+1$	$tc_{M_{k+1}}(k+1) \leq t(k+1-(k+1)) = t(0)$
0^{k+2}	$k+2$	$tc_{M_{k+1}}(k+2) \leq t(k+2-(k+1)) = t(1)$ $tc_{M_{k+2}}(k+2) \leq t(k+2-(k+2)) = t(0)$
$\vdots$	$\vdots$	$\vdots$
0^n	n	$tc_{M_{k+1}}(n) \leq t(n-(k+1))$ $tc_{M_{k+2}}(n) \leq t(n-(k+2))$ $\vdots$ $tc_{M_n}(n) \leq t(n-n) = t(0)$

Checking whether machine M_i is canceled with input 0^m requires at most $t(m-i)$ transitions. The maximum number of transitions required for any computation in the sequence above is $t(n-k-1)$, which occurs for $i = k+1$ and $m = n$.

The machine M must perform each the comparisons indicated above. At most $t(n-k-1)$ transitions are required to simulate the computation of M_i on 0^m. Erasing the tape after the simulation and preparing the subsequent simulation can be accomplished in an additional $2t(n-k-1)$ transitions. The simulation and comparison cycle must be repeated for each machine M_i, $i = k+1, \ldots, n$, and input 0^m, $m = k+1, \ldots, n$. Thus the process of simulation is repeated at most $(n-k)(n-k+1)/2$ times. Consequently, the number of transitions required by M is less than $3(n-k)(n-k+1)t(n-k-1)/2$. That is,

$$tc_M(n) \leq 3(n-k)(n-k+1)t(n-k-1)/2.$$

However, the rate of growth of $3(n-k)(n-k+1)t(n-k-1)/2$ is less than that of $t(n-k) = 2^{t(n-k-1)}$. Consequently, $tc_M(n) \leq t(n-k)$ for all n greater than some n_k. ■

The preceding proof demonstrated that for any machine M accepting L, there is a machine M′ that accepts L more efficiently than M. Now, M′ accepts L so, again by Theorem 14.5.2, there is a more efficient machine M″ that accepts L. This process can

continue indefinitely, producing a sequence of machines each of which accepts L with strictly smaller rate of growth than its predecessor.

Theorem 14.5.2 reveals a rather unintuitive property of algorithmic computation; there are decision problems that have no best solution. Given any algorithmic solution to such a problem, there is another solution that is significantly more efficient.

14.6 Nondeterministic Complexity

Nondeterministic computations are fundamentally different from their deterministic counterparts. A deterministic machine often solves a decision problem by generating a solution. A nondeterministic machine need only determine if one of the possibilities is a solution. Consider the problem of deciding whether a natural number n is a composite (not a prime). A constructive, deterministic solution can be obtained by sequentially examining every number in the interval from 2 to $\lfloor \sqrt{n} \rfloor$. If a factor is discovered, then n is not prime. A nondeterministic computation begins by arbitrarily choosing a value in the designated range. A single division determines whether the guess is a factor. If n is a composite, one of the nondeterministic choices will produce a factor and that computation returns the affirmative response.

A string is accepted by a nondeterministic machine if at least one computation terminates in an accepting state. The acceptance of the string is unaffected by the existence of other computations that halt in nonaccepting states or do not halt at all. The worst case performance of the algorithm, however, measures the efficiency over all computations.

Definition 14.6.1

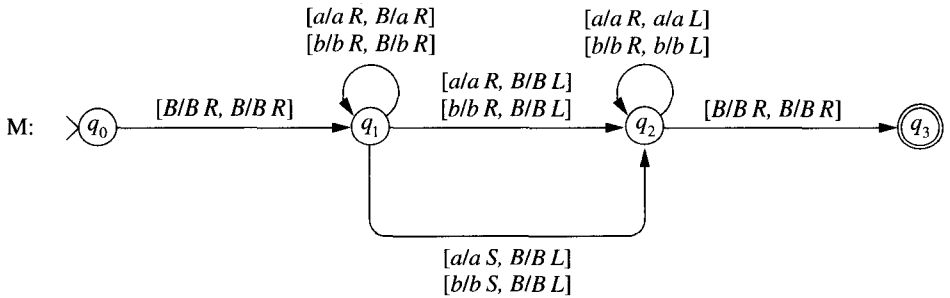
Let M be a nondeterministic Turing machine. The time complexity of M is the function $tc_M : \mathbf{N} \rightarrow \mathbf{N}$ such that $tc_M(n)$ is the maximum number of transitions processed by a computation, employing any choice of transitions, of an input string of length n.

This definition is identical to that of the time complexity of a deterministic machine. It is included to emphasize that the nondeterministic analysis must consider all possible computations for an input string. As before, our definition of time complexity assumes that every computation of M terminates.

Nondeterministic computations utilizing a guess-and-check strategy are generally simpler than their deterministic counterparts. The simplicity reduces the number of transitions required for a single computation. Employing this strategy, we can construct a nondeterministic machine to accept the palindromes over $\{a, b\}$.

Example 14.6.1

The two-tape nondeterministic machine M



accepts that palindromes over $\{a, b\}$ in time $tc_M(n) = n + 1$. Both tape heads move to the right with the input being copied on tape 2. The transition from state q_1 "guesses" the center of the string. A transition from q_1 that moves the tape head on tape 1 to the right and tape 2 to the left is checking for an odd-length palindrome, while a transition that leaves the head on tape 2 in the same location is checking for an even-length palindrome. The maximum number of transitions for a computation with input string of length n is $n + 1$, the number of transitions required to read the entire input string on tape 1. □

The machines designed to accept the palindromes illustrate the reduction of the time complexity that can be achieved by using nondeterminism. In the next chapter we will explore the relationship between the class of problems that can be solved deterministically in polynomial time and the class that can be solved nondeterministically in polynomial time.

14.7 Space Complexity

The focus of this chapter has been the time complexity of a Turing machine. We could equally as well have chosen to analyze the space required by a computation. In high-level algorithmic problem solving, it is often the case that amount of time and memory required by a program are related. We will show that the time complexity of a Turing machine provides an upper bound on the space required and vice versa. The Turing machines in this section are assumed to be deterministic. Nondeterministic space complexity can be defined following the techniques in Section 14.6.

The Turing machine architecture depicted in Figure 14.1 is used for measuring the space required by a computation. Tape 1, which contains the input, is read-only. With an input string of length n, the head on the input tape must remain within tape positions 0 through $n + 1$. The Turing machine reads the input tape but performs its work on the remaining tapes and the space occupied by the input is not included in the complexity analysis. Providing a read-only input tape separates the amount of space required for the

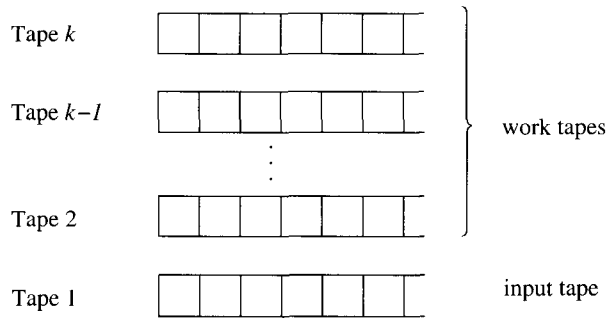


FIGURE 14.1 Turing machine architecture for space complexity.

input from the work space needed by the computation. The space complexity measures the amount of space on the work tapes used by a computation.

Definition 14.7.1

Let M be a k-tape Turing machine designed for space complexity analysis. The **space complexity** of M is the function $sc_M : \mathbf{N} \rightarrow \mathbf{N}$ such that $sc_M(n)$ is the maximum number of tape squares read on tapes $2, \ldots, k$ by a computation of M when initiated with an input string of length n.

Since the space complexity measures only the work tapes, it is possible that $sc_M(n) < n$. That is, the space needed for computation may be less than the length of the input. As with time complexity, we assume that the computations of the Turing machine terminate for every input string.

Example 14.7.1

Example 14.1.1 presented a two-tape Turing machine that accepts the palindromes over $\{a, b\}$. This machine conforms to the specifications of a machine designed for space complexity analysis: The input tape is read-only and the tape head reads only the input string and the blanks on either side of the input. The space complexity of M′ is $n + 2$; a computation reproduces the input on tape 2 and compares the strings on tapes 1 and 2 by reading the strings in opposite directions.

We now design a three-tape machine M that accepts the palindromes with $sc_M(n) = O(\log(n))$. The work tapes, which hold the binary representation of integers, are used as counters. A computation of M with input u of length n consists of the following steps:

1. A single *1* is written on tape 3.
2. Tape 3 is copied to tape 2.
3. The input tape head is positioned at the leftmost square.

   Let i be the integer whose binary representation is on tapes 2 and 3.

4. While the number on tape 2 is not 0,
   a) Move the tape head on the input tape one square to the right.
   b) Decrement the value on tape 2.
5. If the symbol read on the input tape is a blank, halt and accept.
6. The ith symbol of the input is recorded using machine states.
7. The input tape head is moved to the immediate right of the input (tape position $n + 1$).
8. Tape 3 is copied to tape 2.
9. While the number on tape 2 is not 0,
   a) Move the tape head of the input tape one square to the left.
   b) Decrement the value on tape 2.
10. If the $(n - i + 1)$th symbol matches the ith symbol, then increment the value on tape 3 and go to step 2. Else halt and reject.

The operations on tapes 2 and 3 increment and decrement the binary representation of integers. Since $n + 1$ is the largest number written on either of these tapes, each tape uses at most $\log (n + 1) + 2$ tape squares. □

Theorem 14.7.2
Let M be a k-tape Turing machine with time complexity $tc_M(n) = f(n)$. Then $sc_M(n) \leq (k - 1) \cdot f(n)$.

Proof The maximum amount of tape is read when each transition of M moves the heads on tapes $2, \ldots, k$ to the right. In this case, each head on a work tape reads at most $f(n)$ different squares. ■

Obtaining the restriction on time complexity imposed by a known space bound is more complicated since a machine may read a particular segment of the tape multiple times. A two-tape machine is used to demonstrate the bound on the time of a computation that can be obtained from the space complexity. The generalization to k-tape machines is straightforward.

Theorem 14.7.3
Let M be a two-tape Turing machine with space complexity $sc_M(n) = f(n)$. Then $tc_M(n) \leq c^{f(n)}$, where c is dependent upon n, $f(n)$, the number of tape symbols of M, and the number of states of M.

Proof Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a two-tape Turing machine with $m = card(Q)$ and $t = card(\Gamma)$. For an input of length n, the space complexity restricts the computation of M to the first $f(n)$ positions of tape 2. Limiting the computation to a finite length segment of the tape allows us to count the number of distinct machine configurations that M may enter.

The work tape may have any of the t symbols in any position, yielding $t^{f(n)}$ possible configurations. The head on tape 1 may read any of the first $n + 2$ positions while the head on tape 2 may read positions 0 through $f(n)$. Thus there are $f(n) \cdot (n + 2) \cdot t^{f(n)}$ possible combinations of tape configurations and head positions. For any of these, the machine may be in one of m states, yielding a total of $m \cdot f(n) \cdot (n + 2) \cdot t^{f(n)}$ distinct configurations.

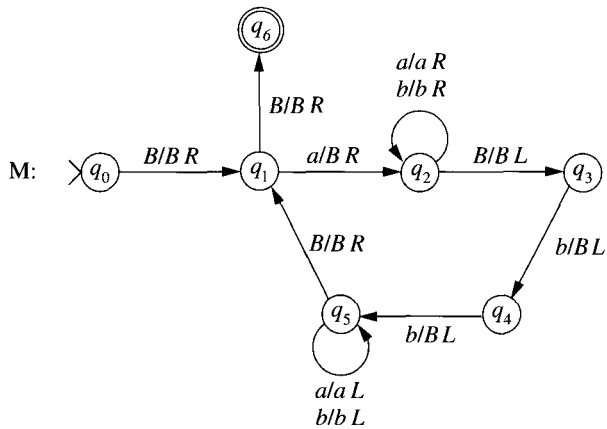
If a computation of M ever encounters identical configurations, the computation will enter a nonterminating loop. Consequently,

$$tc_{\mathrm{M}}(n) \leq m \cdot f(n) \cdot (n + 2) \cdot t^{f(n)} \leq c^{f(n)},$$

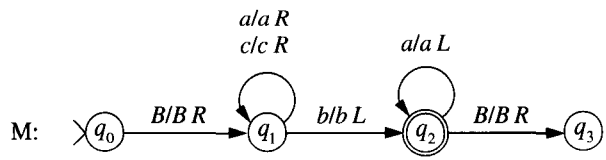
where the constant c depends only on n, $f(n)$, m, and t. ■

Exercises

1. Determine the time complexity of the following Turing machines.
   a) Example 9.2.1
   b) Example 9.6.2
   c) Example 12.1.2
   d) Example 12.2.1
2. Let M be the Turing machine



   a) Trace the computation of M with input λ, a, and abb.
   b) Describe the string of length n for which the computation of M requires the maximum number of transitions.
   c) Give the function tc_{M}.
3. Let M be the Turing machine



a) Trace the computation of M with input *abc*, *aab*, and *cab*.

b) Describe the string of length n for which the computation of M requires the maximum number of transitions.

c) Give a regular expression for L(M).

d) Give the function tc_M.

4. For each of the functions below, choose the "best" big O from Table 14.3.2 that describes the rate of growth of the function.

a) $6n^2 + 500$

b) $2n^2 + n^2 \log_2(n)$

c) $\lfloor (n^3 + 2n)(n + 5)/n^2 \rfloor$

d) $n^2 \cdot 2^n + n!$

e) $25 \cdot n \cdot sqrt(n) + 5n^2 + 23$

5. Let f be a polynomial of degree r. Prove that f and n^r have the same rate of growth.

6. Use Definition 14.3.1 to establish the following relationships.

a) $n \log_2(n) = O(n^2)$

b) $n^r = O(2^n)$

c) $2^n \neq O(n^r)$

d) $2^n = O(n!)$

e) $n! \neq O(2^n)$

7. Is $3^n = O(2^n)$? Prove your answer.

8. Let f and g be two unary functions such that $f = O(n^r)$ and $g = O(n^t)$. Give the best polynomial big O that has the same rate of growth as the following functions.

a) $f + g$

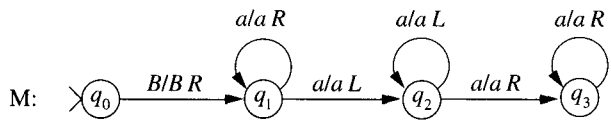
b) fg

c) f^2

d) $f \circ g$

9. Determine the time complexity of the nondeterministic Turing machine in Example 9.7.1.

10. Let M be the Turing machine



a) Trace all computations of M with input λ, a, and aa.

b) Describe the computation of M with input a^n that requires the maximum number of transitions.

c) Give the function tc_M.

Bibliographic Notes

An introduction to computational complexity can be found in the books by Machtey and Young [1978], Hopcroft and Ullman [1979], Rawlins [1992], and Papadimitriou [1994]. Karp [1986], in the 1985 Association of Computing Machinery Turing Award Lecture, gives an interesting personal history of the development and directions of complexity theory.

We have presented computational complexity in terms of the time and space of a computation. An axiomatic approach to abstract complexity measures was introduced in Blum [1967] and developed further by Hartmanis and Hopcroft [1971]. Chapters 12 and 13 of Hopcroft and Ullman [1979] give an introduction to abstract complexity as well as to time and space complexity.

CHAPTER 15

Tractability and NP-Complete Problems

Computational complexity was introduced to provide a measure of the resources required by an algorithm. The complexity of a decision problem is determined by that of the algorithms that solve the problem. Complexity analysis provides criteria for classifying problems based upon the practicality of their solutions. A problem that is theoretically solvable may not have a practical solution; there may be no algorithm that solves the problem without requiring an extraordinary amount of time or memory. In this chapter, the class of decision problems is partitioned into two subclasses: tractable and intractable problems. Problems are considered **tractable** if there is a deterministic Turing machine that solves the problem whose computations are polynomially bounded.

There are a number of famous problems for which it is not known whether there are polynomial-time deterministic solutions. Among these is the Hamiltonian circuit problem which, as we will see, has a nondeterministic polynomial-time solution. This chapter is concerned with exploring the relationship between solvability using deterministic and nondeterministic polynomial-time algorithms. Whether every problem that can be solved in polynomial time by a nondeterministic algorithm can also be solved deterministically in polynomial time is currently the outstanding open question of theoretical computer science.

15.1 Tractable and Intractable Decision Problems

A language L over Σ is *decidable in polynomial time,* or simply polynomial, if there is an algorithm that determines membership in L for which the time required by a computation grows polynomially with the length of the input string. Since there may be many algorithms that decide membership in a language L, the time complexity of L is characterized by the rate of growth of the most efficient solution. The notion of polynomial time decidability is formally defined using transitions of the standard Turing machine as the computational framework for measuring the time of a computation. The language of palindromes over $\{a, b\}$ is polynomial since the machine M from Section 14.1 accepts the language in $O(n^2)$ time.

Because of the equivalence between recursive languages and solvable decision problems, we will also refer to the class of decision problems that are *solvable in polynomial time.* There is one additional consideration when determining the time complexity of a decision problem, the representation of the problem instances as input strings to the Turing machine that solves the problem. The effect of this component of the solution on the time complexity will be examined after the formal definition of polynomial-time decidability is presented.

Definition 15.1.1

A language L is **decidable in polynomial time** if there is a standard Turing machine M that accepts L with $tc_M = O(n^r)$, where r is a natural number independent of n. The family of languages decidable in polynomial time is denoted $\mathcal{P}$.

Computability theory is concerned with establishing whether decision problems are theoretically decidable. Complexity theory attempts to distinguish problems that are solvable in practice from those that are solvable in principle only. A solution to a decision problem may be impractical because of the resources required by the computations. Problems for which there is no efficient algorithm are said to be **intractable.** Because of the rate of growth of the time complexity, nonpolynomial algorithms are considered unfeasible for all but the simplest cases of the problem. The division of the class of solvable decision problems into polynomial and nonpolynomial problems is generally considered to distinguish the efficiently solvable problems from the intractable problems.

The class $\mathcal{P}$ was defined in terms of the time complexity of an implementation of an algorithm on a standard Turing machine. We could just as easily have chosen a multitrack, multitape, or two-way deterministic machine as the computational model on which algorithms are evaluated. The class $\mathcal{P}$ of polynomial decision problems is invariant under the choice of the deterministic Turing machine chosen for the analysis. In Section 14.3 it was shown that a language accepted by a multitrack machine in time $O(n^r)$ is also accepted by a standard Turing machine in time $O(n^r)$. The transition from multitape to standard machine also preserves polynomial solutions. A language accepted in time $O(n^r)$ by a multitape machine is accepted in $O(n^{2r})$ time by a standard machine.

We will now consider the additional complication in determining the time complexity of a decision problem. As mentioned above, the first step in constructing an algorithm to solve a decision problem is to select a representation of the problem instances suitable for input to a Turing machine. Not surprisingly, the representation has important consequences for the amount of work required by a computation and the determination of the problem complexity since the time complexity of a Turing machine relates the length of the input to the number of transitions in the computations.

For example, consider an algorithm whose input is a natural number. If a computation with input n requires n transitions, regardless of the representation, the time complexity differs according to the representation selected. With a unary representation, the number of transitions is linear with respect to the size of the input. However, a binary representation of the number n has length $\lceil \lg n \rceil$. Consequently, the function that relates the size of the input to the number of transitions is exponential. Is this problem polynomial or exponential? Assessing problem complexity requires a concise representation of the input. In particular, numbers will be represented in binary so that the representation of an integer n requires only $\lceil \lg n \rceil$ tape positions. A problem like the one discussed above, which has a polynomial solution using the unary representation of natural numbers for the input but no polynomial solution using the binary representation, is sometimes called **pseudo-polynomial**.

In addition to the computional issues, another reason for distinguishing polynomial and nonpolynomial algorithms is that nonpolynomial algorithms often do not provide insight into the nature of the problem being solved, but rather have the flavor of an exhaustive search. This type of behavior is exhibited by the machine constructed in Example 15.1.1 that solves the Hamiltonian circuit problem.

Let G be a directed graph with n vertices numbered 1 to n. An arc from vertex i to vertex j is represented by the ordered pair $[i, j]$. A Hamiltonian circuit is a path $i_0, i_1, \ldots, i_n$ in G that satisfies

i) $i_0 = i_n$

ii) $i_i \neq i_j$ whenever $i \neq j$ and $0 \leq i, j < n$.

That is, a Hamiltonian circuit is a path that visits every vertex exactly once and terminates at its starting point. A Hamiltonian circuit is frequently called a *tour*. Since each vertex is contained in a tour, we may assume that every tour begins and ends at vertex 1. The Hamiltonian circuit problem is to determine whether a directed graph has a tour.

Example 15.1.1

Let G be a directed graph with n vertices. The composition of a deterministic four-tape machine that solves the Hamiltonian circuit problem is outlined. A vertex of the graph is denoted by its binary representation. The alphabet of the representation is $\{0, 1, \#\}$. A graph with n vertices and m arcs is represented by the input string

$$\overline{x}_1\#\overline{y}_1\#\# \ldots \#\#\overline{x}_m\#\overline{y}_m\#\#\#\overline{n}$$

where $[x_i, y_i]$ are the arcs of the graph and $\overline{x}$ denotes the binary representation of the number x.

Throughout the computation, tape 1 maintains the representation of the arcs. The computation generates and examines sequences of $n + 1$ vertices $1, i_1, \ldots, i_{n-1}, 1$ to determine if they form a cycle. The sequences are generated in numeric order on tape 2. The representation of the sequence $1, n, \ldots, n, 1$ is written on tape 4 and used to trigger the halting condition. The techniques employed by the machine in Figure 9.1 can be used to generate the sequences on tape 2.

A computation is a loop that

1. generates a sequence $B\overline{1}B\overline{i}_1B\overline{i}_2B \ldots B\overline{i}_{n-1}B\overline{1}B$ on tape 2
2. halts if tapes 2 and 4 are identical
3. examines the sequence $1, i_1, \ldots, i_{n-1}, 1$ and halts if it is a tour of the graph.

If the computation halts in step 2, all sequences have been examined and the graph does not contain a Hamiltonian circuit.

The analysis in step 3 begins with the machine configuration

$$\begin{aligned}
&\text{Tape 4} \quad B\overline{1}(B\overline{n})^{n-1}B\overline{1}B\\
&\text{Tape 3} \quad B\overline{1}B\\
&\text{Tape 2} \quad B\overline{1}B\overline{i}_1B \ldots B\overline{i}_{n-1}B\overline{1}B\\
&\text{Tape 1} \quad B\overline{x}_1\#\overline{y}_1\#\# \ldots \#\#\overline{x}_m\#\overline{y}_mB\#\#\#\overline{n}B.
\end{aligned}$$

Sequentially the vertices $i_1, \ldots, i_{n-1}$ are examined. Vertex i_j is added to the sequence on tape 3 if

i) $i_j \neq 1$

ii) $i_j \neq i_k$ for $1 \leq k \leq j - 1$

iii) there is an arc $[i_{j-1}, i_j]$ represented on tape 1;

that is, if $1, i_1, \ldots, i_j$ is an acyclic path in the graph. If every vertex i_j, $j = i, \ldots, n - 1$, in the sequence on tape 2 is added to tape 3 and there is an arc from i_{n-1} to 1, the path on tape 2 is a tour and the computation accepts the input. □

The algorithm outlined in Example 15.1.1 is exponential. A computation examines and rejects each sequence $1, i_1, i_2, \ldots, i_{n-1}, 1$ when the input graph does not contain a tour. For a graph with n vertices, there are n^{n-1} such sequences. Disregarding the computations involved in checking a sequence, the number of sequences grows exponentially with the number of vertices of the graph. Since the binary representation is used to encode the vertices, increasing the number of vertices to $2n$ (but adding no arcs to the graph) increases the length of the input string by a single character. Consequently, incrementing the length of the input causes an exponential increase in the number of possible sequences that must be examined.

We have shown that the Hamiltonian circuit problem is solvable in exponential time. It does not follow that the problem cannot be solved in polynomial time. So far, no polynomial algorithm has been discovered. This may be because no such solution exists or maybe we have just not been clever enough to find one! The likelihood and ramifications of the discovery of a polynomial time solution are discussed in Section 15.3.

15.2 The Class $\mathcal{NP}$

The computation of a nondeterministic machine that solves a decision problem examines one of the possible solutions to the problem. The ability to nondeterministically select a single potential solution, rather than systematically examining all possible solutions, reduces the complexity of the computation of the nondeterministic machine.

Definition 15.2.1

A language L is said to be accepted in **nondeterministic polynomial time** if there is a nondeterministic Turing machine M that accepts L with $tc_M = O(n^r)$, where r is a natural number independent of n. The family of languages accepted in nondeterministic polynomial time is denoted $\mathcal{NP}$.

The family $\mathcal{NP}$ is a subset of the recursive languages; the polynomial bound on the number of transitions ensures that all computations of M eventually terminate. Since every deterministic machine is also nondeterministic, $\mathcal{P} \subseteq \mathcal{NP}$. The status of the reverse inclusion is the topic of the remainder of this chapter.

Nondeterministic computations utilizing a guess-and-check strategy are generally simpler than their deterministic counterparts. The simplicity reduces the number of transitions required for a single computation. A nondeterministic machine employing this strategy is constructed that solves the Hamiltonian circuit problem in polynomial time.

Example 15.2.1

A three-tape nondeterministic machine that solves the Hamiltonian circuit problem in polynomial time is obtained by altering the deterministic machine constructed in Example 15.1.1. The fourth tape, which is used to terminate the computation when the graph does not contain a tour, is not required in the nondeterministic machine. The computation

1. halts and rejects the input whenever there are fewer than $n + 1$ arcs in the graph
2. nondeterministically generates a sequence $1, i_1, \ldots, i_{n-1}, 1$ on tape 2
3. uses tapes 1 and 3 to determine whether the sequence on tape 2 defines a tour.

To show that the nondeterministic machine is polynomial, we construct an upper bound to the number of transitions in a computation. The maximum number of transitions occurs when the string defines a tour. Otherwise, the computation terminates without

examining each of the nodes represented on tape 2. Since the nodes are represented in binary, the maximum amount of tape needed to encode any node is $\lceil \lg n \rceil + 1$.

The length of the input depends upon the number of arcs in the graph. Let k be the number of arcs. We will show that the rate of growth of the number of transitions is polynomial in k. Since the length of the input cannot grow more slowly than k (each arc requires at least three tape positions), it follows that the time complexity is polynomial.

Rejecting the input in step 1 requires the computation to compare the number of arcs in the input with the number of nodes. This can be accomplished in time that grows polynomially with the number of arcs.

For the remainder of the computation, we know that the number of arcs is greater than the number of nodes. Generating the sequence on tape 2 and repositioning the tape head processes $O(n \lg n)$ transitions. Now assume that tape 3 contains the initial subsequence $B\bar{1}\#\bar{i}_1\# \ldots \#\bar{i}_{j-1}$ of the sequence on tape 2. The remainder of the computation consists of a loop that

i) moves tape heads 2 and 3 to the position of the first blank on tape 3 ($O(n \lg n)$ transitions)

ii) checks if the encoded vertex on tape 2 is already on tape 3 ($O(n \lg n)$ transitions)

iii) checks if there is an arc from i_{j-1} to i_j ($O(k \lg n)$ transitions examining all the arcs and repositioning the tape head)

iv) writes $\bar{i}_j$ on tape 3 and repositions the tape heads ($O(n \lg n)$ transitions).

A computation consists of the generation of the sequence on tape 2 followed by examination of the sequence. The loop that checks the sequence is repeated for each vertex $i_1, \ldots, i_{n-1}$ on tape 2. The repetition of step (iii) causes the number of transitions of the entire computation to grow at the rate $O(k^2 \lg k)$. The rate of growth of the time complexity of the nondeterministic machine is determined by the portion of the computation that searches for the presence of a particular arc in the arc list. This differs from the deterministic machine in which the exhaustive search of the entire set of sequences of n vertices defines the rate of growth. □

15.3 $\mathcal{P} = \mathcal{NP}$?

A language accepted in polynomial time by a deterministic multitrack or multitape machine is in $\mathcal{P}$. The construction of an equivalent standard Turing machine from one of these alternatives preserves polynomial time complexity. A technique for constructing an equivalent deterministic machine from the transitions of a nondeterministic machine was presented in Section 9.7. Unfortunately, this construction does not preserve polynomial time complexity.

Let L be a language accepted by a nondeterministic machine M and let k be the maximum number of alternative transitions for a state, symbol pair of M. The equiva-

lent deterministic machine M′ constructed using the technique developed in Section 9.7 sequentially examines all possible computations of M. For an input string of length n, the number of computations of M that are simulated by M′ is k^m, where $m = tc_M(n)$. When the nondeterministic machine M has alternative transitions, k is greater than one and the number of transitions of M′ grows exponentially with the length of the computation of M. Another strategy must be devised to show that a language accepted in nondeterministic polynomial time is also accepted in polynomial time.

The two solutions to the Hamiltonian circuit problem dramatically illustrate the difference between deterministic and nondeterministic computations. To obtain an answer, the deterministic solution generates sequences of vertices in an attempt to discover a tour. In the worst case, this process requires examining all possible sequences of vertices that may constitute a tour of the graph. The nondeterministic machine avoided this by "guessing" a single sequence of vertices and determining if this sequence is a tour. The philosophic interpretation of the $\mathcal{P} = \mathcal{NP}$ question is whether constructing a solution to a problem is inherently more difficult than checking to see if a single possibility satisfies the conditions of the problem. Because of the additional complexity of currently known deterministic solutions over nondeterministic solutions across a wide range of important problems, it is generally believed that $\mathcal{P} \neq \mathcal{NP}$. The $\mathcal{P} = \mathcal{NP}$ question is, however, a precisely formulated mathematical problem and will be resolved only when the equality of the two classes or the proper inclusion of $\mathcal{P}$ in $\mathcal{NP}$ is proven.

One approach for determining whether $\mathcal{P} = \mathcal{NP}$ is to examine the properties of each language or decision problem on an individual basis. For example, considerable effort has been expended attempting to develop a deterministic polynomial algorithm to solve the Hamiltonian circuit problem. On the face of it, finding such a solution would resolve the question for only one language. What is needed is a universal approach that resolves the issue of deterministic polynomial solvability for all languages in $\mathcal{NP}$ at once. To accomplish this we introduce the notion of polynomial-time reducibility of languages.

Definition 15.3.1
Let Q and L be languages over alphabets Σ_1 and Σ_2, respectively. We say that Q is reducible to L in polynomial-time if there is a polynomial-time computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that $u \in Q$ if, and only if, $f(u) \in L$.

A reduction of Q to L transforms the problem of accepting Q to that of accepting L. Let F be a Turing machine that computes the function f. If L is accepted by a machine M, then Q is accepted by a machine that

i) runs F on an input string $u \in \Sigma_1^*$
ii) runs M on $f(u)$.

The resulting string $f(u)$ is accepted by M if, and only if, $u \in Q$. The time complexity of the composite machine can be obtained from those of F and M.

Theorem 15.3.2
Let Q be reducible to L in polynomial time and let $L \in \mathcal{P}$. Then $Q \in \mathcal{P}$.

Proof As before, we let F denote the machine that computes the reduction and M the machine that recognizes L. Q is accepted by a machine that sequentially runs F and M. The time complexities tc_F and tc_M combine to produce an upper bound on the number of transitions of a computation of the composite machine. The computation of F with input string u generates the string $f(u) \in L$, which provides the input to M. The function tc_F can be used to establish an upper bound to the length of $f(u)$. If the input string $u \in L$ has length n, then the length of $f(u)$ cannot exceed the maximum of n and $tc_F(n)$.

The computation of M processes at most $tc_M(k)$ transitions, where k is the bound on the length of the input string. The number of transitions of the composite machine is bounded by the sum of the estimates of the two separate computations. If $tc_F = O(n^r)$ and $tc_M = O(n^t)$, then

$$tc_F(n) + tc_M(tc_F(n)) = O(n^{rt}).$$ ■

Definition 15.3.3
A language L is called **NP-hard** if for every $Q \in \mathcal{NP}$, Q is reducible to L in polynomial time. An NP-hard language that is also in $\mathcal{NP}$ is called **NP-complete**.

One can consider an NP-complete language as a universal language in the class $\mathcal{NP}$. The discovery of a polynomial-time machine that accepts an NP-complete language can be used to construct machines to accept every language in $\mathcal{NP}$ in deterministic polynomial time. This, in turn, yields an affirmative answer to the $\mathcal{P} = \mathcal{NP}$ question.

Theorem 15.3.4
If there is an NP-complete language that is also in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

Proof Assume that L is an NP-complete language that is accepted in polynomial time by a deterministic Turing machine. Let Q be any language in $\mathcal{NP}$. Since L is NP-hard, there is a polynomial time reduction of Q to L. Now, by Theorem 15.3.2, Q is also in $\mathcal{P}$. ■

The definition of NP-completeness utilized the terminology of recursive languages and Turing computable functions because of the preciseness afforded by the concepts and notation of Turing computability. The duality between recursive languages and solvable decision problems permits us to speak of NP-hard and NP-complete decision problems. It is worthwhile to reexamine these definitions in the context of decision problems.

Reducibility of languages using Turing computable functions is a formalization of the notion reduction of decision problems that was developed in Chapter 11. A decision problem is NP-hard or NP-complete whenever the language accepted by a machine that solves the problem is. In an intuitive sense, an NP-complete problem **P** is a universal problem in the class $\mathcal{NP}$. Utilizing the reducibility to an NP-hard problem, we can obtain a solution to any $\mathcal{NP}$ problem by combining the reduction with the machine that solves **P**.

Regardless of whether we approach NP-completeness from the perspective of languages or decision problems, it is clear that this is an important class of problems. Unfortunately, we have not yet shown that such a universal problem exists. Although it requires a substantial amount of work, this omission is remedied in the next section.

15.4 The Satisfiability Problem

Historically, the satisfiability problem was the first decision problem shown to be NP-complete. The satisfiability problem is concerned with the truth values of formulas in propositional logic. The truth value of a formula is obtained from those of the elementary propositions occurring in the formula. The objective of the satisfiability problem is to determine whether there is an assignment of truth values to propositions that makes the formula true. Before demonstrating that the satisfiability problem is NP-complete, we will briefly review the fundamentals of propositional logic.

A **Boolean variable** is a variable that takes on values 0 and 1. Boolean variables are considered to be propositions, the elementary objects of propositional logic. The value of the variable specifies the truth or falsity of the proposition. The proposition x is true when the Boolean variable is assigned the value 1. The value 0 designates a false proposition. A **truth assignment** is a function that assigns a value 0 or 1 to every Boolean variable.

The logical connectives $\wedge$ (and), $\vee$ (or), and $\sim$ (not) are used to construct propositions known as well-formed formulas from a set of Boolean variables. The symbols x, y, and z are used to denote Boolean variables while u, v, and w represent well-formed formulas.

Definition 15.4.1

Let V be a set of Boolean variables.

i) If $x \in$ V, then x is a well-formed formula.

ii) If u, v are well-formed formulas, then (u), $(\sim u)$, $(u \wedge v)$, and $(u \vee v)$ are well-formed formulas.

iii) An expression is a well-formed formula over V only if it can be obtained from the Boolean variables in the set V by a finite number of applications of the operations in (ii).

The expressions $((\sim(x \vee y)) \wedge z)$, $(((x \wedge y) \vee z) \vee \sim(x))$, and $(((\sim x) \vee y) \wedge (x \vee z))$ are well-formed formulas over the Boolean variables x, y, and z. The number of parentheses in a well-formed formula can be reduced by defining a precedence relation on the logical operators. Negation is considered the most binding operation, followed by conjunction and then disjunction. Additionally, the associativity of conjunction and disjunction permits the parentheses in sequences of these operations to be omitted. Utilizing these conventions, we can write the preceding formulas $\sim(x \vee y) \wedge z$, $x \wedge y \vee z \vee \sim x$, and $(\sim x \vee y) \wedge (x \vee z)$.

The truth values of the variables are obtained directly from the truth assignment. The standard interpretation of the logical operations can be used to extend the assignment of truth values to the well-formed formulas. The truth values of formulas $\sim u$, $u \wedge v$, and $u \vee v$ are obtained from the values of u and v according to the rules given in the following tables.

u	$\sim u$
0	1
1	0

u	v	$u \wedge v$
0	0	0
0	1	0
1	0	0
1	1	1

u	v	$u \vee v$
0	0	0
0	1	1
1	0	1
1	1	1

A formula u is satisfied by a truth assignment if the values of the variables cause u to assume the value 1. Two well-formed formulas are equivalent if they are satisfied by the same truth assignments.

A **clause** is a well-formed formula that consists of a disjunction of variables or the negation of variables. An unnegated variable is called a *positive literal* and a negated variable a *negative literal*. Using this terminology, a clause is a *disjunction of literals*. The formulas $x \vee \sim y$, $\sim x \vee z \vee \sim y$, and $x \vee z \vee \sim x$ are clauses over the set of Boolean variables $\{x, y, z\}$. A formula is in **conjunctive normal form** if it has the form

$$u_1 \wedge u_2 \wedge \cdots \wedge u_n,$$

where each u_i is a clause. A classical theorem of propositional logic asserts that every well-formed formula can be transformed into an equivalent formula in conjunctive normal form.

Stated precisely, the **satisfiability problem** is the problem of deciding if a formula in conjunctive normal form is satisfied by some truth assignment. Let V be the set of variables $\{x, y, z\}$. The formulas u and v are satisfied by the truth assignment t.

$$u = (x \vee y) \wedge (\sim y \vee \sim z)$$

$$v = (x \vee \sim y \vee \sim z) \wedge (x \vee z) \wedge (\sim x \vee \sim y)$$

$$w = \sim x \wedge (x \vee y) \wedge (\sim y \vee x)$$

t	
x	1
y	0
z	1

The first clause in u is satisfied by x and the second by $\sim y$. The formula w is not satisfied by t. Moreover, it is not difficult to see that w is not satisfied by any truth assignment.

A deterministic solution to the satisfiability problem can be obtained by checking every truth assignment. The number of possible truth assignments is 2^n, where n is the number of variables. An implementation of this strategy is essentially a mechanical method of constructing the complete truth table for the formula. Clearly, the complexity of this

exhaustive approach is exponential. The work expended in checking a particular truth assignment, however, grows polynomially with the number of variables and the length of the formula. This observation provides the insight needed for designing a polynomial-time nondeterministic machine that solves the satisfiability problem.

Theorem 15.4.2

The satisfiability problem is $\mathcal{NP}$.

Proof We begin by developing a representation for the well-formed formulas over a set of Boolean variables $\{x_1, \ldots, x_n\}$. A variable is encoded by the binary representation of its subscript. The encoding of a literal consists of the encoded variable followed by *#1* if the literal is positive and *#0* if it is negative.

Literal	Encoding
x_i	$\bar{i}$*#1*
$\sim x_i$	$\bar{i}$*#0*

The number following the encoding of the variable specifies the Boolean value that satisfies the literal.

A well-formed formula is encoded by concatenating the literals with the symbols representing disjunction and conjunction. The conjunctive normal-form formula

$$(x_1 \vee \sim x_2) \wedge (\sim x_1 \vee x_3)$$

is encoded as

$$1\#1 \vee 10\#0 \wedge 1\#0 \vee 11\#1.$$

Finally, the input to the machine consists of the encoding of the variables in the formula followed by ## and then the encoding of the formula itself. The input string representing the preceding formula is

$$\underbrace{1\#10\#11\#\#1\#1}_{\text{variables}} \vee \underbrace{10\#0 \wedge 1\#0 \vee 11\#1}_{\text{formula}}$$

The representation of an instance of the satisfiability problem is a string over the alphabet $\Sigma = \{0, 1, \wedge, \vee, \#\}$. The language L_{SAT} consists of all strings over Σ that represent satisfiable conjunctive normal form formulas.

A two-tape nondeterministic machine M that solves the satisfiability problem is described below. M employs the guess-and-check strategy; the guess nondeterministically generates a truth assignment. Configurations corresponding to the computation initiated

with the input string representing the formula $(x_1 \vee \sim x_2) \wedge (\sim x_1 \vee x_3)$ are given to illustrate the actions of the machine. The initial configuration of the tape contains the representation of the formula on tape 1 with tape 2 blank:

1. If the input does not have the anticipated form, the computation halts and rejects the string.

   Tape $2\,B\,B$

   Tape $1\,B\,1\#10\#11\#\#1\#1 \vee 10\#0 \wedge 1\#0 \vee 11\#1\,B$

2. The encoding of the first variable on tape 1 is copied onto tape 2. This is followed by printing # and nondeterministically writing 0 or 1. If this is not the last variable, ## is written and the procedure is repeated for the next variable. Nondeterministically choosing a value for each variable defines a truth assignment t. The value assigned to variable x_i is denoted $t(x_i)$.

   Tape $2\,B\,1\#t(x_1)\#\#10\#t(x_2)\#\#11\#t(x_3)\,B$

   Tape $1\,B\,1\#10\#11\#\#1\#1 \vee 10\#0 \wedge 1\#0 \vee 11\#1\,B$

   The tape head on tape 2 is repositioned at the leftmost position. The head on tape 1 is moved past ## into a position to read the first variable of the formula.

The generation of the truth assignment is the only instance of nondeterminism of M. The remainder of the computation determines whether the formula is satisfied by the nondeterministically selected truth assignment.

3. Assume that the encoding of the variable x_i is scanned on tape 1. The encoding of x_i is found on tape 2. The subsequent actions of the machine are determined by the result of comparing the value $t(x_i)$ on tape 2 with the Boolean value following x_i on tape 1.
4. If the values do not match, the current literal is not satisfied by the truth assignment. If the symbol following the literal is a B or $\wedge$, every literal in the current clause has been examined and failed. When this occurs, the truth assignment does not satisfy the formula and the computation halts, rejecting the string. If $\vee$ is read, the tape heads are positioned to examine the next literal in the clause (step 3).
5. If the values do match, the literal and current clause are satisfied by the truth assignment. The head on tape 1 moves to the right to the next $\wedge$ or B. If a B is encountered, the computation halts, accepting the input. Otherwise, the next clause is processed by returning to step 3.

The matching procedure in step 3 determines the rate of growth of the length of the computations. In the worst case, the matching requires comparing the variable on tape 1 with each of the variables on tape 2 to discover the match. This can be accomplished in $O(k \cdot n^2)$ time, where n is the number of variables and k the number of literals in the input. ■

We now must show that L_{SAT} is NP-hard, that is, that every language in $\mathcal{NP}$ is polynomial time reducible to L_{SAT}. At the outset, this may seem like an impossible task. There are infinitely many languages in $\mathcal{NP}$, and they appear to have little in common. They are not even restricted to having the same alphabet. The lone universal feature of the languages in $\mathcal{NP}$ is that they are all accepted by a polynomial time-bounded nondeterministic Turing machine. Fortunately, this is enough. Rather than concentrating on the languages, the proof will exploit the properties of the machines that accept the languages. In this manner, a general procedure is developed that can be used to reduce any $\mathcal{NP}$ language to L_{SAT}.

We begin with the following technical lemma. The notation

$$\bigwedge_{i=1}^{k} v_i \qquad \bigvee_{i=i}^{k} v_i$$

represents the conjunction and disjunction of the literals $v_1, v_2, \ldots, v_k$.

Lemma 15.4.3

Let $u = w_1 \vee w_2 \vee \cdots \vee w_n$ be the disjunction of conjunctive normal form formulas $w_1, w_2, \ldots, w_n$ over the set of Boolean variables V. Also let $V' = V \cup \{y_1, y_2, \ldots, y_{n-1}\}$ where the variables y_i are not in V. The formula u can be transformed into a formula u' over V′ such that

i) u' is in conjunctive normal form

ii) u' is satisfiable over V′ if, and only if, u is satisfiable over V

iii) The transformation can be accomplished in $O(m \cdot n^2)$, where m is the number of clauses in the w's.

Proof The transformation of the disjunction of two conjunctive normal form formulas is presented. This technique may be repeated $n - 1$ times to transform the disjunction of n formulas. Let $u = w_1 \vee w_2$ be the disjunction of two conjunctive normal form formulas. Then w_1 and w_2 can be written

$$w_1 = \bigwedge_{j=1}^{r_1} \left(\bigvee_{k=1}^{s_j} v_{j,k} \right)$$

$$w_2 = \bigwedge_{j=1}^{r_2} \left(\bigvee_{k=1}^{t_j} p_{j,k} \right),$$

where r_i is the number of clauses in w_i, s_j is the number of literals in the jth clause of w_1, and t_j is the number of literals in the jth clause of w_2. Define

$$u' = \bigwedge_{j=1}^{r_1} \left(y \vee \bigvee_{k=1}^{s_j} v_{j,k} \right) \wedge \bigwedge_{j=1}^{r_2} \left(\sim y \vee \bigvee_{k=1}^{t_j} p_{j,k} \right).$$

The formula u' is obtained by disjoining y to each clause in w_1 and $\sim y$ to each clause in w_2.

We now show that u' is satisfiable whenever u is. Assume that w_1 is satisfied by a truth assignment t over V. Then the truth assignment t'

$$t'(x) = \begin{cases} t(x) & \text{if } x \in \mathrm{V} \\ 0 & \text{if } x = y \end{cases}$$

satisfies u'. When w_2 is satisfied by t, the truth assignment t' may be obtained by extending t by setting $t'(y) = 1$.

Conversely, assume that u' is satisfied by the truth assignment t'. Then the restriction of t' to V satisfies u. If $t'(y) = 0$, then w_1 must be true. On the other hand, if $t'(y) = 1$, then w_2 is true.

The transformation of

$$u = w_1 \vee w_2 \vee \ldots \vee w_n$$

requires $n - 1$ iterations of the preceding process. The repetition adds $n - 1$ literals to each clause in w_1 and w_2, $n - 2$ literals to each clause in w_3, $n - 3$ literals to each clause in w_4, and so on. The transformation requires fewer than $m \cdot n^2$ steps, where m is the total number of clauses in the formulas $w_1, w_2, \ldots, w_n$. ■

Theorem 15.4.4

The satisfiability problem is NP-hard.

Proof Let M be a nondeterministic machine whose computations are bounded by the polynomial p. Without loss of generality, we assume that all computations of M halt in one of two states. All accepting computations terminate in state q_A and rejecting computations in q_R. Moreover, we assume that there are no transitions leaving these states. An arbitrary machine can be transformed into an equivalent one satisfying these restrictions by adding transitions from every accepting configuration to q_A and from every rejecting configuration to q_R. This alteration adds a single transition to every computation of the original machine. The transformation from computation to well-formed formula assumes that all computations with input of length n contain $p(n)$ configurations. The terminating configuration is repeated, if necessary, to ensure that the correct number of configurations are present.

The states and elements of the alphabets of M are denoted

$$\mathrm{Q} = \{q_0, q_1, \ldots, q_m\}$$
$$\Gamma = \{B = a_0, a_1, \ldots, a_s, a_{s+1}, \ldots, a_t\}$$
$$\Sigma = \{a_{s+1}, a_{s+2}, \ldots, a_t\}$$
$$\mathrm{F} = \{q_m\}.$$

The blank is assumed to be the tape symbol numbered 0. The input alphabet consists of the elements of the tape alphabet numbered $s + 1$ to t. The lone accepting state is q_m and the rejecting state is q_{m-1}.

Let $u \in \Sigma^*$ be a string of length n. Our goal is to define a formula $f(u)$ that encodes the computations of M with input u. The length of $f(u)$ depends on $p(n)$, the maximum length of a computation of M with input of length n. The encoding is designed so that there is a truth assignment satisfying $f(u)$ if, and only if, $u \in L(M)$. The formulas are built from three classes of variables; each class is introduced to represent a property of a machine configuration.

Variable		Interpretation (When Satisfied)
$Q_{i,k}$	$0 \le i \le m$ $0 \le k \le p(n)$	M is in state q_i at time k
$P_{j,k}$	$0 \le j \le p(n)$ $0 \le k \le p(n)$	M is scanning position j at time k
$S_{j,r,k}$	$0 \le j \le p(n)$ $0 \le r \le t$ $0 \le k \le p(n)$	Tape position j contains symbol a_r at time k

The set of variables V is the union of the three sets defined above. A computation of M defines a truth assignment on V. For example, if tape position 3 initially contains symbol a_i, then $S_{3,i,0}$ is true. Necessarily, $S_{3,j,0}$ must be false for all $i \neq j$. A truth assignment obtained in this manner specifies the state, position of the tape head, and the symbols on the tape for each time k in the range $0 \le k \le p(n)$. This is precisely the information contained in the sequence of configurations produced by the computation.

An arbitrary assignment of truth values to the variables in V need not correspond to a computation of M. Assigning 1 to both $P_{0,0}$ and $P_{1,0}$ specifies that the tape head is at two distinct positions at time 0. Similarly, a truth assignment might specify that the machine is in several states at a given time or designate the presence of multiple symbols in a single position.

The formula $f(u)$ should impose restrictions on the variables to ensure that the interpretations of the variables are identical with those generated by the truth assignment obtained from a computation. Eight sets of formulas are defined from the input string u and the transitions of M. Seven of the eight families of formulas are given directly in clause form. The clauses are accompanied by a brief description of their interpretation in terms of Turing machine configurations and computations.

A truth assignment that satisfies the set of clauses defined in (i) in the table below indicates that the machine is in a unique state at each time. Satisfying the first disjunction guarantees that at least one of the variables $Q_{i,k}$ holds. The pairwise negations specify

that no two states are satisfied at the same time. The interpretation of this disjunction is most easily obtained from the equivalent formula $Q_{i,k} \Rightarrow \sim Q_{i',k}$, written using the logical connective implication.

	Clause	**Conditions**	**Interpretation**
i)	State		
	$\bigvee_{i=0}^{m} Q_{i,k}$	$0 \leq k \leq p(n)$	For each time k, M is in at least one state.
	$\sim Q_{i,k} \vee \sim Q_{i',k}$	$0 \leq i < i' \leq m$ $0 \leq k \leq p(n)$	M is in at most one state (not two different states at the same time).
ii)	Tape head position		
	$\bigvee_{j=0}^{p(n)} P_{j,k}$	$0 \leq k \leq p(n)$	For each time k, the tape head is in at least one position.
	$\sim P_{j,k} \vee \sim P_{j',k}$	$0 \leq j < j' \leq p(n)$ $0 \leq k \leq p(n)$	At most one position.
iii)	Symbols on tape		
	$\bigvee_{r=0}^{t} S_{j,r,k}$	$0 \leq j \leq p(n)$ $0 \leq k \leq p(n)$	For each time k and position j, position j contains at least one symbol.
	$\sim S_{j,r,k} \vee \sim S_{j,r',k}$	$0 \leq j \leq p(n)$ $0 \leq r < r' \leq t$ $0 \leq k \leq p(n)$	At most one symbol.
iv)	Initial conditions for input string $u = a_{r_1} a_{r_2} \ldots a_{r_n}$		
	$Q_{0,0}$ $P_{0,0}$ $S_{0,0,0}$		The computation begins reading the leftmost blank.
	$S_{1,r_1,0}$		The string u is in the input

	$S_{2,r_2,0}$	position at time 0.
	$\vdots$	
	$S_{n,r_n,0}$	
	$S_{n+1,0,0}$	The remainder of the tape is
	$\vdots$	blank at time 0.
	$S_{p(n),0,0}$	
v)	Accepting condition	The halting state of the
	$Q_{m,p(n)}$	computations is q_m.

Since the computation of M with input of length n cannot access the tape beyond position $p(n)$, a machine configuration is completely defined by the contents of the initial $p(n)$ positions of the tape. A truth assignment that satisfies the clauses in (i), (ii), and (iii) defines a machine configuration for each time between 0 and $p(n)$. The conjunction of the clauses (i) and (ii) indicates that the machine is in a unique state scanning a single tape position at each time. The clauses in (iii) ensure that the tape is well-defined; that is, the tape contains precisely one symbol in each position that may be referenced during the computation.

A computation, however, does not consist of a sequence of unrelated configurations. Each configuration must be obtained from its successor by the application of a transition. Assume that the machine is in state q_i, scanning symbol a_r in position j at time k. The final three sets of formulas are introduced to generate the permissible configurations at time $k + 1$ based on the variables that define the configuration at time k.

The effect of a transition on the tape is to rewrite the position scanned by the tape head. With the possible exception of position $P_{j,k}$, every tape position at time $k + 1$ contains the same symbol as at time k. Clauses must be added to the formula to ensure that the remainder of the tape is unaffected by a transition.

Clause	Conditions	Interpretation
vi) Tape consistency		
$\sim S_{j,r,k} \vee P_{j,k} \vee S_{j,r,k+1}$	$0 \leq j \leq p(n)$	Symbols not at the position of
	$0 \leq r \leq t$	the tape head are unchanged.
	$0 \leq k \leq p(n)$	

This clause is not satisfied if a change occurs to a tape position other than the one scanned by the tape head. This can be seen by noting that

$$\sim S_{j,r,k} \vee P_{j,k} \vee S_{j,r,k+1}$$

is equivalent to

$$\sim P_{j,k} \Rightarrow (S_{j,r,k} \Rightarrow S_{j,r,k+1}),$$

which clearly indicates that if the tape head is not a position j at time k then symbol at position j is the same at time $k+1$ as it was at time k.

Now assume that for a given time k, the machine is in state q_i scanning symbol a_r in position j. These features of a configuration are designated by the assignment of 1 to the Boolean variables $Q_{i,k}$, $P_{j,k}$, and $S_{j,r,k}$. The clause

a) $\sim Q_{i,k} \vee \sim P_{j,k} \vee \sim S_{j,r,k} \vee Q_{i',k+1}$

is satisfied only when $Q_{i',k+1}$ is true. In terms of the computation, this signifies that M has entered state $q_{i'}$ at time $k+1$. Similarly, the symbol in position j at time $k+1$ and the tape head position are specified by the clauses

b) $\sim Q_{i,k} \vee \sim P_{j,k} \vee \sim S_{j,r,k} \vee S_{j,r',k+1}$

c) $\sim Q_{i,k} \vee \sim P_{j,k} \vee \sim S_{j,r,k} \vee P_{j+n(d),k+1}$,

where $n(L) = -1$ and $n(R) = 1$. The conjunction of clauses of (a), (b), and (c) is satisfied only if the configuration at time $k+1$ is obtained from the configuration at time k by the application of the transition $[q_{i'}, a_{r'}, d] \in \delta(q_i, a_r)$.

The clausal representation of transitions is used to construct a formula whose satisfaction guarantees that the time $k+1$ variables define a configuration obtained from the configuration defined by the time k variables by the application of a transition of M. Except for states q_m and q_{m-1}, the restrictions on M ensure that at least one transition is defined for every state, symbol pair.

The conjunctive normal form formula

$$\begin{aligned}
&(\sim Q_{i,k} \vee \sim P_{j,k} \vee \sim S_{j,r,k} \vee Q_{i',k+1}) && \text{(new state)}\\
\wedge&(\sim Q_{i,k} \vee \sim P_{j,k} \vee \sim S_{j,r,k} \vee P_{j+n(d),k+1}) && \text{(new tape head position)}\\
\wedge&(\sim Q_{i,k} \vee \sim P_{j,k} \vee \sim S_{j,r,k} \vee S_{j,r',k+1}) && \text{(new symbol at position } r)
\end{aligned}$$

is constructed for every

$$\begin{aligned}
&0 \le k \le p(n) && \text{(time)}\\
&0 \le i < m-1 && \text{(nonhalting state)}\\
&0 \le j \le p(n) && \text{(tape head position)}\\
&0 \le r \le t && \text{(tape symbol)}
\end{aligned}$$

where $[q_{i'}, a_{r'}, d] \in \delta(q_i, a_r)$ except when the position is 0 and the direction L is specified by the transition. The exception occurs when the application of a transition would cause the tape head to cross the left-hand boundary of the tape. In clausal form, this is represented by having the succeeding configuration contain the rejecting state q_{m-1}. This

special case is encoded by the formulas

$$(\sim Q_{i,k} \vee \sim P_{0,k} \vee \sim S_{0,r,k} \vee Q_{m-1,k+1}) \quad \text{(entering the rejecting state)}$$
$$\wedge(\sim Q_{i,k} \vee \sim P_{0,k} \vee \sim S_{0,r,k} \vee P_{0,k+1}) \quad \text{(same tape head position)}$$
$$\wedge(\sim Q_{i,k} \vee \sim P_{0,k} \vee \sim S_{0,r,k} \vee S_{0,r,k+1}) \quad \text{(same symbol at position } r\text{)}$$

for all transitions $[q_{i'}, a_{r'}, L] \in \delta(q_i, a_r)$.

Since M is nondeterministic, there may be several transitions that can be applied to a given configuration. The result of the application of any of these alternatives is a permissible succeeding configuration in a computation. Let $trans(i, j, r, k)$ denote disjunction of the conjunctive normal form formulas that represent the alternative transitions for a configuration at time k in state q_i, tape head position j, and tape symbol r. The formula $trans(i, j, r, k)$ is satisfied only if the values of the variables encoding the configuration at time $k + 1$ represent a legitimate successor to the configuration encoded in the variables with time k.

	Clause
vii)	Generation of successor configuration $trans(i, j, r, k)$

The formulas $trans(i, j, r, k)$ do not specify the actions to be taken when the machine is in state q_m or q_{m-1}, the halting states of the machine. In this case, the subsequent configuration is identical to its predecessor.

	Clause	**Interpretation**
viii)	Halted computation	
	$\sim Q_{i,k} \vee \sim P_{j,k} \vee \sim S_{j,r,k} \vee Q_{i,k+1}$	(same state)
	$\sim Q_{i,k} \vee \sim P_{j,k} \vee \sim S_{j,r,k} \vee P_{j,k+1}$	(same tape head position)
	$\sim Q_{i,k} \vee \sim P_{j,k} \vee \sim S_{j,r,k} \vee S_{j,r,k+1}$	(same symbol at position r)

These clauses are built for all j, r, k in the appropriate ranges and $i = q_{m-1}, q_m$.

Let $f'(u)$ be the conjunction of the formulas constructed in (i) through (viii). When $f'(u)$ is satisfied by a truth assignment on V, the variables define the configurations of a computation of M that accepts the input string u. The clauses in condition (iv) specify that the configuration at time 0 is the initial configuration of a computation of M with input u. Each subsequent configuration is obtained from its successor by the result of the application of a transition. The string u is accepted by M since condition (v) indicates that the final configuration contains the accepting state q_m.

A conjunctive normal form formula $f(u)$ can be obtained from $f'(u)$. This is accomplished by converting each formula $trans(i, j, r, k)$ into conjunctive normal form using

the technique presented in Lemma 15.4.3. All that remains is to show that the transformation of a string $u \in \Sigma^*$ to $f(u)$ can be done in polynomial time.

The transformation of u to $f(u)$ consists of the construction of the clauses and the conversion of *trans* to conjunctive normal form. The number of clauses is a function of

i) the number of states m and the number of tape symbols t

ii) the length n of the input string u

iii) the bound $p(n)$ on the length of the computation of M.

The values m and t obtained from the Turing machine M are independent of the input string. From the range of the subscripts, we see that the number of clauses is polynomial in $p(n)$. The development of $f(u)$ is completed with the transformation into conjunctive normal form, which is polynomial in the number of clauses in the formulas $trans(i, j, r, k)$.

We have shown that the conjunctive normal form formula can be constructed in a number of steps that grows polynomially with the length of the input string. What is really needed is the representation of the formula that serves as input to a Turing machine that solves the satisfiability problem. Any reasonable encoding, including the one developed in Theorem 15.4.2, requires only polynomial time to convert the high-level representation to the machine representation. ■

15.5 Additional NP-Complete Problems

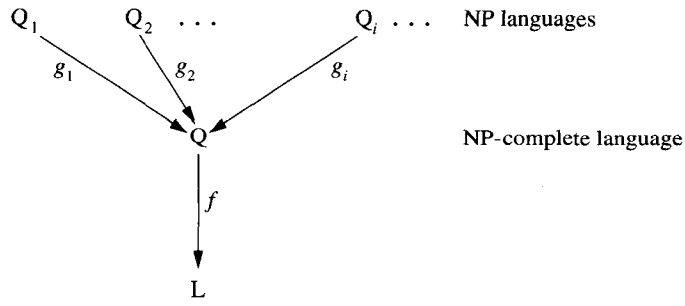
The proof that L_{SAT} is NP-complete was accomplished by associating Turing machine computations with conjunctive normal form formulas. If every proof of NP-completeness required the ingenuity of Theorem 15.4.4, the number of languages known to be NP-complete would not be very large. Fortunately, once one language has been shown to be NP-complete the reduction of languages provides a simpler technique to demonstrate that other languages are also NP-hard.

Theorem 15.5.1

Let Q be an NP-complete language. If Q is reducible to L in polynomial time, then L is NP-hard.

Proof Let f be the computable function that reduces Q to L in polynomial time and let Q_i be any language in $\mathcal{NP}$. Since Q is NP-complete, there is a computable function g_i that reduces Q_i to Q. The composite function $f \circ g_i$ is a reduction of Q_i to L. A polynomial bound to the reduction can be obtained from the bounds on f and g_i. ■

The functional composition used in establishing NP-completeness by reduction can represented pictorally as a two-step process.



The first level shows the polynomial reducibility of any language Q_i in $\mathcal{NP}$ to Q via a function g_i. Following the arrows from Q_i to L illustrates the reducibility of any $\mathcal{NP}$ language to L.

Reduction is used to show that several additional problems are NP-complete. We begin with a special case of the satisfiability problem known as the 3-satisfiability problem. A formula is said to be in **3-conjunctive normal form** if it is in conjunctive normal form and each clause contains precisely three literals. The objective of the 3-satisfiability problem is to determine whether a 3-conjunctive normal form formula is satisfiable.

Theorem 15.5.2

The 3-satisfiability problem is NP-complete.

Proof Clearly, the 3-satisfiability problem is in $\mathcal{NP}$. The machine that solves the satisfiability problem for arbitrary conjunctive normal form formulas also solves it for the subclass of 3-conjunctive normal form formulas.

Now we must show that every conjunctive normal form formula can be transformed to a 3-conjunctive normal form formula. Each clause u whose length is not three is independently transformed into a 3-conjunctive normal form formula. The resulting formula u' is satisfiable if, and only if, there is a truth assignment that satisfies the original clause. The variables added in the transformation are assumed not to occur in the clause u.

Length 1: $u = v_1$

Transformation $u' = (v_1 \vee y \vee z) \wedge (v_1 \vee \sim y \vee z) \wedge (v_1 \vee y \vee \sim z) \wedge (v_1 \vee \sim y \vee \sim x)$

Length 2: $u = v_1 \vee v_2$

Transformation $u' = (v_1 \vee v_2 \vee y) \wedge (v_1 \vee v_2 \vee \sim y)$

Length $n > 3$: $u = v_1 \vee v_2 \vee \cdots \vee v_n$

Transformation $u' = (v_1 \vee v_2 \vee y_1) \wedge (v_3 \vee \sim y_1 \vee y_2) \wedge \cdots \wedge (v_j \vee \sim y_{j-2} \vee y_{j-1}) \wedge \cdots$

$\wedge (v_{n-2} \vee \sim y_{n-4} \vee y_{n-3}) \wedge (v_{n-1} \vee v_n \vee \sim y_{n-3})$

Establishing the relationship between satisfiability of clauses of length one and two and their transformations is left as an exercise. Let V be the variables in the clause $u = v_1 \vee v_2 \vee \cdots \vee v_n$ and let t be a truth assignment that satisfies u. Since u is satisfied by t, there is at least one literal satisfied by t. Let v_j be the first such literal. Then the truth assignment

$$t'(x) = \begin{cases} t(x) & \text{if } x \in \mathrm{V} \\ 1 & \text{if } x = y_1, \ldots, y_{j-2} \\ 0 & \text{if } x = y_{j-1}, \ldots, y_{n-3} \end{cases}$$

satisfies u'. The first $j-2$ clauses are satisfied by literals $y_1, \ldots, y_{j-2}$. The final $n-j+1$ clauses are satisfied by $\sim y_{j-1}, \ldots, \sim y_{n-3}$. The remaining clause, $v_j \vee \sim y_{j-2} \vee y_{j-1}$, is satisfied by v_j.

Conversely, let t' be a truth assignment that satisfies u'. The truth assignment t obtained by restricting t' to V satisfies u. The proof is by contradiction. Assume that t does not satisfy u. Then no literal v_j, $1 \leq j \leq n$, is satisfied by t. Since the first clause of u' has the value 1, it follows that $t'(y_1) = 1$. Now, $t'(y_2) = 1$ since the second clause also has the value 1. Employing the same reasoning, we conclude that $t'(y_k) = 1$ for all $1 \leq k \leq n-3$. This implies that the final clause of u' has value 0, a contradiction since t' was assumed to satisfy u'.

The transformation of each clause into a 3-conjunctive normal form formula is clearly polynomial in the number of literals in the clause. The work required for the construction of the 3-conjunctive normal form formula is the sum of the work of the transformation of the individual clauses. Thus, the construction is polynomial in the number of clauses in the original form. ■

Let G = (N, A) be an undirected graph. A subset VC of N is said to be a **vertex cover** of G if for every arc $[u, v]$ in A at least one of u or v is in VC. The vertex cover problem can be stated as follows: For an undirected graph G and an integer k, is there a vertex cover of G containing k vertices?

Theorem 15.5.3

The vertex cover problem is NP-complete.

Proof The vertex cover problem can easily be seen to be in $\mathcal{NP}$. The nondeterministic solution strategy consists of choosing a set of k vertices and determining whether they cover the arcs of the graph. We show that the vertex cover problem is NP-hard by reducing the 3-satisfiability problem to it. That is, for any 3-conjunctive normal form formula u we must construct a graph G so that G has a vertex cover of some predetermined size k if, and only if, u is satisfiable.

Let

$$u = (u_{1,1} \vee u_{1,2} \vee u_{1,3}) \wedge \cdots \wedge (u_{m,1} \vee u_{m,2} \vee u_{m,3})$$

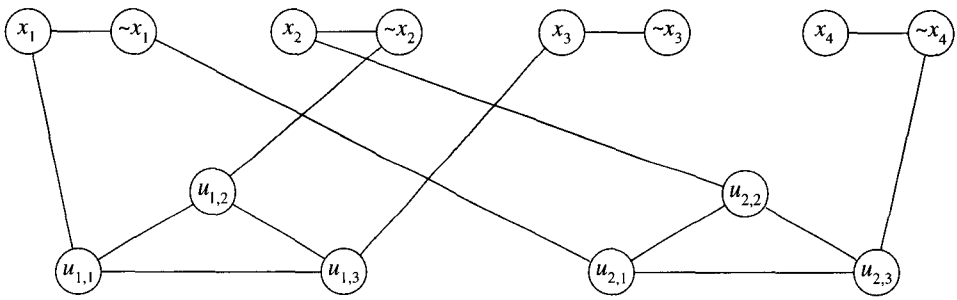


FIGURE 15.1 Graph representing reduction of $(x_1 \vee \sim x_2 \vee x_3) \wedge (\sim x_1 \vee x_2 \vee \sim x_4)$.

be a 3-conjunctive normal form formula where each $u_{i,j}, 1 \leq i \leq m$ and $1 \leq j \leq 3$, is a literal over the set $V = \{x_1, \ldots, x_n\}$ of Boolean variables. The symbol $u_{i,j}$ is used to indicate the position of a literal in a 3-conjuctive normal form formula; the first subscript indicates the clause and the second subscript the position of the literal in the clause. The reduction consists of constructing a graph G from the 3-conjunctive normal form formula. The satisfiability of u will be equivalent to the existence of a cover of G containing $n + 2m$ vertices.

The vertices of G consist of the sets

i) $\{x_i, \sim x_i \mid 1 \leq i \leq n\}$

ii) $\{u_{ij} \mid 1 \leq i \leq m, 1 \leq j \leq 3\}$.

The arcs of G are obtained from the union of the sets

$$T = \{[x_i, \sim x_i] \mid 1 \leq i \leq n\}$$

$$C_k = \{[u_{k,1}, u_{k,2}], [u_{k,2}, u_{k,3}], [u_{k,3}, u_{k,1}]\} \quad \text{for } 1 \leq k \leq m$$

$$L_k = \{[u_{k,1}, v_{k,1}], [u_{k,2}, v_{k,2}], [u_{k,3}, v_{k,3}]\} \quad \text{for } 1 \leq k \leq m,$$

where $v_{k,j}$ is the literal from $\{x_i, \sim x_i \mid 1 \leq i \leq n\}$ that occurs in position $u_{k,j}$ of the formula.

An arc in T connects a positive literal x_i to its corresponding negative literal $\sim x_i$. A vertex cover must include one of these two vertices. At least n vertices are needed to cover the arcs in T. Each clause $u_{j,1} \vee u_{j,2} \vee u_{j,3}$ generates a subgraph C_j. C_j can be pictured as a triangle connecting the literals $u_{j,1}$, $u_{j,2}$, and $u_{j,3}$. A set of vertices that covers C_j must contain at least two vertices. Thus a cover of G must contain $n + 2m$ vertices.

The arcs in L_j link the symbols $u_{i,j}$ that indicate the positions of the literals to the corresponding literal x_k or $\sim x_k$ in the formula. Figure 15.1 gives the graph corresponding to the formula $(x_1 \vee \sim x_2 \vee x_3) \wedge (\sim x_1 \vee x_2 \vee \sim x_4)$. It is easy to show that the construction of the graph is polynomially dependent upon the number of variables and clauses in the formula. All that remains is to show that the formula u is satisfiable if, and only if, the associated graph has a cover of size $n + 2m$.

First we show that a cover VC of size $n + 2m$ defines a truth assignment on V that satisfies the formula u. By the previous remarks, we know that every cover must contain at least $n + 2m$ vertices. Consequently, exactly one vertex from each arc $[x_i, \sim x_i]$ and two vertices from each subgraph C_j are in VC. A truth assignment is obtained from VC by

$$t(x_i) = \begin{cases} 1 & \text{if } x_i \in \text{VC} \\ 0 & \text{otherwise.} \end{cases}$$

That is, the literal from the pair x_i or $\sim x_i$ in the vertex cover is assigned truth value 1 by t.

To see that t satisfies each clause, consider the covering of the subgraph C_j. Only two of the vertices $u_{j,1}, u_{j,2}$, and $u_{j,3}$ can be in VC. Assume $u_{j,k}$ is not in VC. Then the arc $[u_{j,k}, v_{j,k}]$ must be covered by $v_{j,k}$ in VC. This implies that $t(u_{j,k}) = 1$ and the clause is satisfied.

Now assume that $t : \text{V} \to \{0, 1\}$ is a truth assignment that satisfies u. A vertex cover VC of the associated graph can be constructed from the truth assignment. VC contains the vertex x_i if $t(x_i) = 1$ and $\sim x_i$ if $t(x_i) = 0$. Let $u_{j,k}$ be a literal in clause j that is satisfied by t. The arc $[u_{j,k}, v_{j,k}]$ is covered by $v_{j,k}$. Adding the two other vertices of C_j completes the cover. Clearly, $card(\text{VC}) = n + 2m$, as desired. ■

We now return to our old friend, the Hamiltonian circuit problem. This problem has already been shown to be solvable in exponential time by a deterministic machine (Example 15.1.1) and in polynomial time by a nondeterministic machine (Example 15.2.1). A reduction of 3-satisfiability to the Hamiltonian circuit problem establishes that the latter is NP-complete.

Theorem 15.5.4

The Hamiltonian circuit problem is NP-complete.

Proof Following the strategy employed in Theorem 15.5.3, a directed graph $G(u)$ is constructed from a 3-conjunctive normal form formula u. The construction is designed so that the presence of a Hamiltonian circuit in $G(u)$ is equivalent to the satisfiability of u.

Let $u = w_1 \wedge w_2 \wedge \cdots \wedge w_m$ be a 3-conjunctive normal form formula, where $\text{V} = \{x_1, x_2, \ldots, x_n\}$ is the set of variables of u. The jth clause of u is denoted $u_{j,1} \vee u_{j,2} \vee u_{j,3}$, where each $u_{j,k}$ is a literal over V. For each variable x_i, let r_i be the larger of the number of occurrences of x_i in u or the number of occurrences of $\sim x_i$ in u. A graph V_i is constructed for each variable x_i as illustrated in Figure 15.2(a). Node e_i is considered the entrance to V_i and o_i the exit. There are precisely two paths through V_i that begin with e_i, end with o_i, and visit each vertex once. These are depicted in Figure 15.2(b) and (c). The arc from e_i to $t_{i,0}$ or $f_{i,0}$ determines the remainder of the path through V_i.

The subgraphs V_i are joined to construct the graph G′ depicted in Figure 15.2(d). The two paths through each V_i combine to generate 2^n Hamiltonian circuits through the graph G′. A Hamiltonian circuit in G′ represents a truth assignment on V. The value of x_i is specified by the arc from e_i. An arc from e_i to $t_{i,0}$ designates a truth assignment of 1 for x_i.

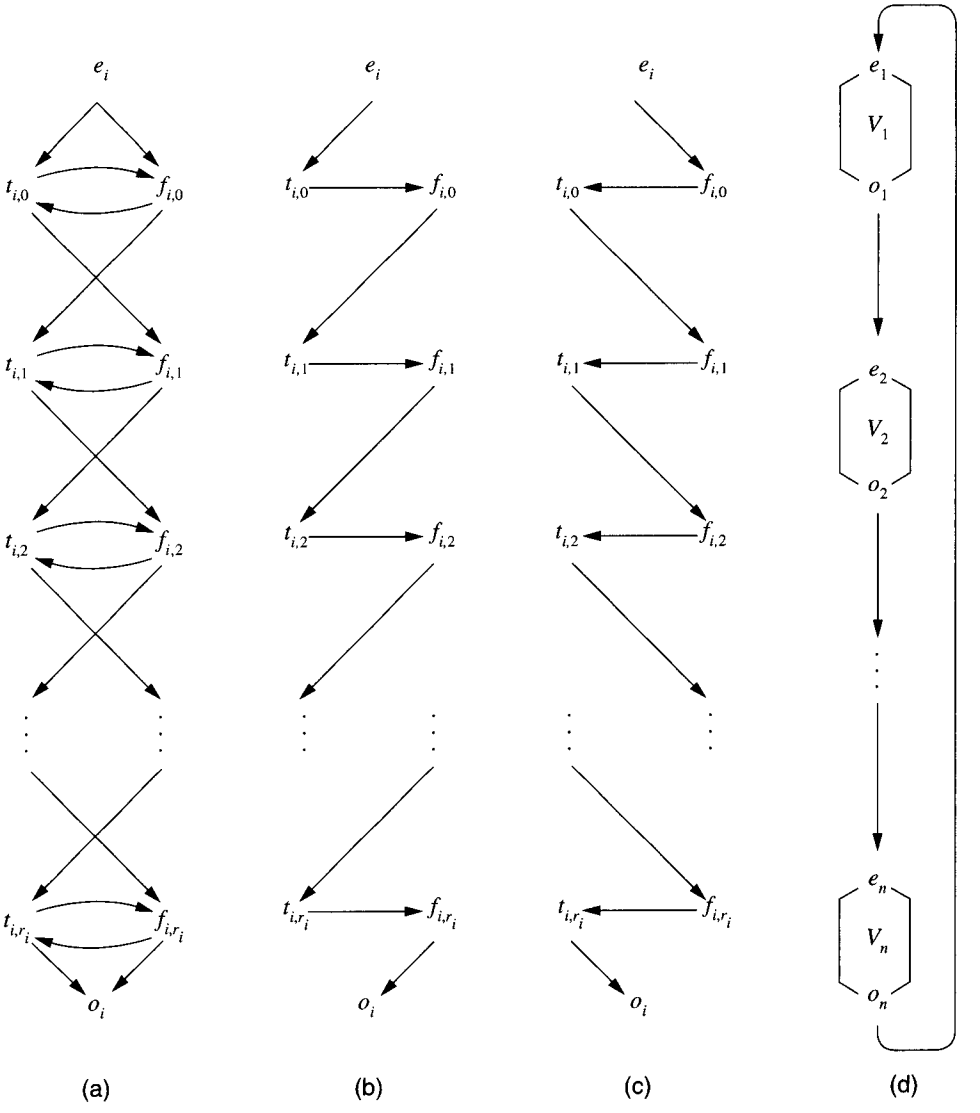


FIGURE 15.2 Subgraph for each variable x_i.

Otherwise, x_i is assigned 0. The graph in Figure 15.3 is constructed from the formula

$$(x_1 \vee x_2 \vee \sim x_3) \wedge (\sim x_1 \vee x_2 \vee \sim x_4) \wedge (x_1 \vee \sim x_2 \vee x_4) \wedge (\sim x_1 \vee x_3 \vee x_4).$$

The tour highlighted by bold arcs in the graph defines the truth assignment

$$t(x_1) = 1$$
$$t(x_2) = 0$$
$$t(x_3) = 0$$
$$t(x_4) = 1.$$

The Hamiltonian circuits of G′ encode the possible truth assignments of V. We now augment G′ with subgraphs that encode the clauses of the 3-conjunctive form formula.

For each clause w_j, we construct a subgraph C_j that has the form outlined in Figure 15.4. The graph G(u) is constructed by connecting these subgraphs to G′ as follows:

i) If x_i is a literal in w_j, then pick some $f_{i,k}$ that has not previously been connected to a graph C. Add an arc from $f_{i,k}$ to a vertex $in_{j,m}$ of C_j that has not already been connected to G′. Then add an arc from $out_{j,m}$ to $t_{i,k+1}$.

ii) If $\sim x_i$ is a literal in w_j, then pick some $t_{i,k}$ that has not previously been connected to a graph C. Add an arc from $t_{i,k}$ to a vertex $in_{j,m}$ of C_j that has not already been connected to G′. Then add an arc from $out_{j,m}$ to $f_{i,k+1}$.

The graph in Figure 15.5 is obtained by connecting the subgraph representing the clause $(x_1 \vee x_2 \vee \sim x_3)$ to the graph G′ from Figure 15.3.

A truth assignment is represented by a Hamiltonian circuit in the graph G′. If x_i is a positive literal in the clause w_j, then there is an arc from some vertex $f_{i,k}$ to one of the in vertices of C_j. Similarly, if $\sim x_i$ is in w_j, then there is an arc from some vertex $t_{i,k}$ to one of the in vertices of C_j. These arcs are used to extend the Hamiltonian circuit in G′ to a tour of G(u) when the associated truth assignment satisfies u.

Let t be a truth assignment on V that satisfies u. We will construct a Hamiltonian circuit through G(u). Begin with the tour through the V_i's that represents t. We now detour the path through the subgraphs that encode the clauses. An arc $[t_{i,k}, f_{i,k}]$ in the path V_i indicates that the value of the truth assignment $t(x_i) = 1$. If the path reaches $f_{i,k}$ by an arc $[t_{i,k}, f_{i,k}]$ and $f_{i,k}$ contains an arc to a subgraph C_j that is not already in the path, then connect C_j to the tour in G′ as follows:

i) Detour to C_j via the arc from $f_{i,k}$ to $in_{j,m}$ in C_j.

ii) Visit each vertex of C_j once.

iii) Return to V_i via the arc from $out_{j,m}$ to $t_{i,k+1}$.

The presence of a detour to C_j indicates that the truth assignment encoded in G′ satisfies the clause w_j.

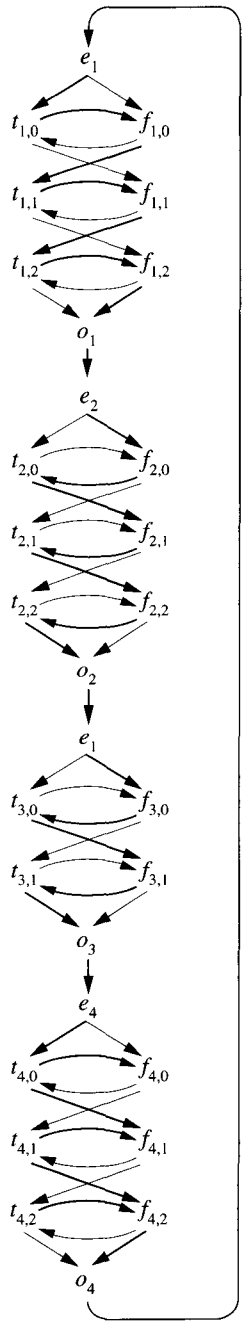


FIGURE 15.3 Truth assignment by Hamiltonian circuit.

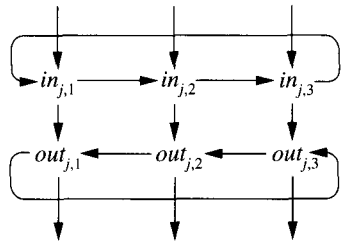


FIGURE 15.4 Subgraph representing clause w_j.

On the other hand, a clause can also be satisfied by the presence of a negative literal $\sim x_i$ for which $t(x_i) = 0$. A similar detour can be constructed from a vertex $t_{i,k}$. Since $t(x_i) = 0$, the vertices $t_{i,k}$ are entered by an arc $[f_{i,k}, t_{i,k}]$. Choose a $t_{i,k}$ that has not already been connected to one of the subgraphs C_j. Construct the detour as follows:

i) Detour to C_j via the arc from $t_{i,k}$ to $in_{j,m}$ in C_j.

ii) Visit each vertex in C_j once.

iii) Return to V_i via the arc from $out_{j,m}$ to $f_{i,k+1}$.

Since each clause is satisfied by the truth assignment, a detour from G′ can be constructed that visits each subgraph C_j. In this manner, the Hamiltonian cycle of G′ defined by a satisfying truth assignment can be extended to a tour of G(u).

Now assume that a graph G(u) contains a Hamiltonian circuit. We must show that u is satisfiable. The Hamiltonian circuit defines a truth assignment as follows:

$$t(x_1) = \begin{cases} 1 & \text{if the arc } [e_i, t_{i,0}] \text{ is in the tour} \\ 0 & \text{if the arc } [e_i, f_{i,0}] \text{ is in the tour.} \end{cases}$$

If $t(x_i) = 1$, then the arcs $[t_{i,k}, f_{i,k}]$ are in the tour. On the other hand, the tour contains the arcs $[f_{i,k}, t_{i,k}]$ whenever $t(x_i) = 0$.

Before proving that t satisfies u, we examine several properties of a tour that enters the subgraph C_j. Upon entering at the vertex $in_{j,m}$, the path may visit two, four, or all six vertices in C_j. A path that exits C_j at any position other than $out_{j,m}$ cannot be a subpath of a tour. Assume that C_j is entered at $in_{j,1}$; the following paths in C_j are not subpaths of a tour because the vertices listed cannot be reached without visiting some vertex twice.

Path	Unreachable Vertices
$in_{j,1}, in_{j,2}, in_{j,3}, out_{j,3}$	$out_{j,2}, out_{j,1}$
$in_{j,1}, in_{j,2}, in_{j,3}, out_{j,3}, out_{j,2}$	$out_{j,1}$
$in_{j,1}, in_{j,2}, out_{j,2}$	$out_{j,1}$
$in_{j,1}, in_{j,2}, out_{j,2}, out_{j,1}, out_{j,3}$	$in_{j,3}$

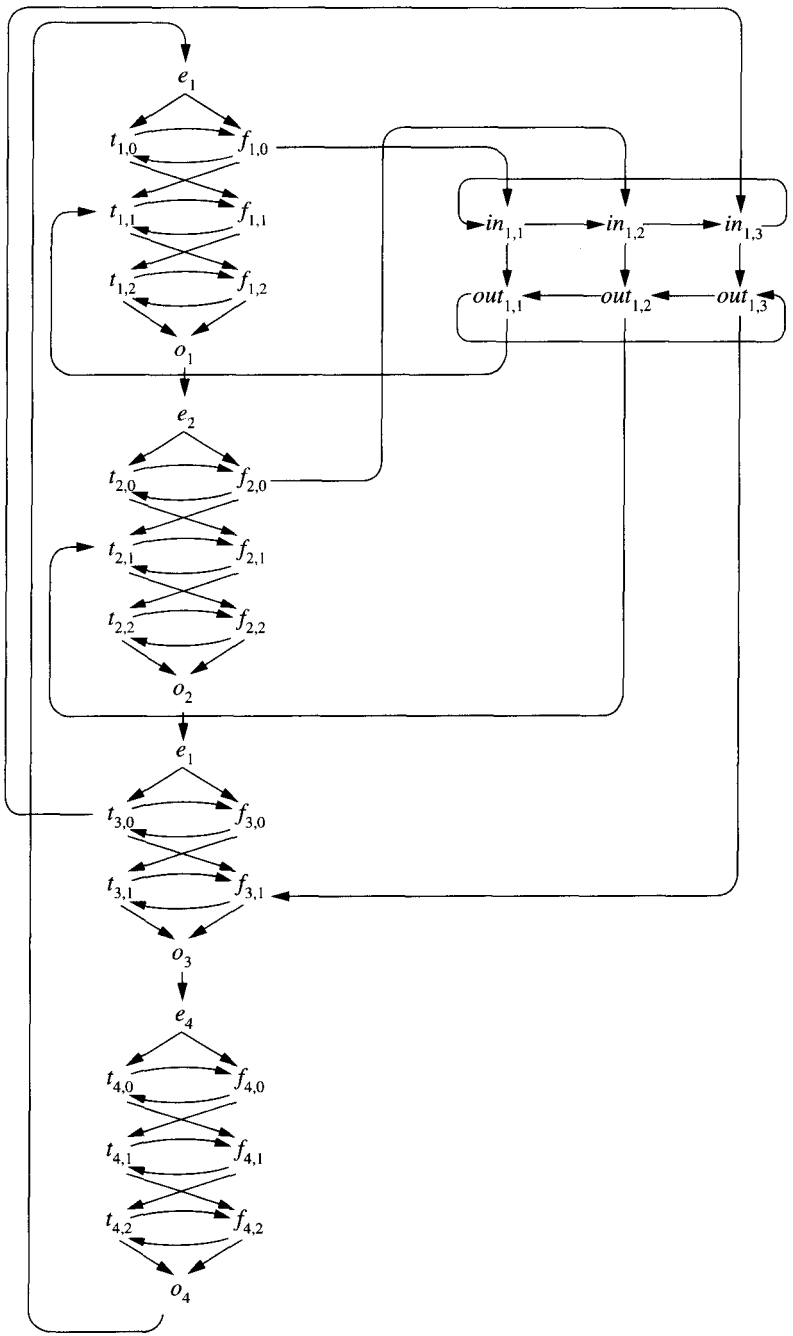


FIGURE 15.5 Connection of C_1 to G′.

Thus the only paths entering C_j at $in_{j,1}$ that are subpaths of tours must exit at $out_{j,1}$. The same property holds for $in_{j,2}$ and $in_{j,3}$.

Each of the C_j's must be entered by the tour. If C_j is entered at vertex $in_{j,m}$ by an arc from a vertex $f_{i,k}$, then the tour exits C_j via the arc from $out_{j,m}$ to $t_{i,k+1}$. The presence of the arc $[f_{i,k}, in_{j,m}]$ in $G(u)$ indicates that w_j, the clause encoded by C_j, contains the literal x_i. Moreover, when C_j is entered by an arc $[f_{i,k}, in_{j,m}]$, the vertex $f_{i,k}$ must be entered by the arc $[t_{i,k}, f_{i,k}]$. Otherwise, the vertex $t_{i,k}$ is not in the tour. Since $[t_{i,k}, f_{i,k}]$ is in the tour, we conclude that $t(x_i) = 1$. Thus, w_j is satisfied by t. Similarly, if C_j is entered by an arc $[t_{i,k}, in_{j,m}]$, then $\sim x_i$ is in w_j and $t(x_i) = 0$.

Combining the previous observations, we see that the truth assignment generated by a Hamiltonian circuit through $G(u)$ satisfies each of the clauses of u and hence u itself. All that remains is to show that the construction of $G(u)$ is polynomial in the number of literals in the formula u. The number of vertices and arcs in a subgraph V_i increases linearly with the number of occurrences of the variable x_i in u. For each clause, the construction of C_j adds six vertices and fifteen arcs to $G(u)$. ■

15.6 Derivative Complexity Classes

Our study of tractability introduced two complexity classes: the class $\mathcal{P}$ of languages recognizable deterministically in polynomial time and the class $\mathcal{NP}$ of languages recognizable in polynomial time by nondeterministic computations. The question of whether these classes are identical is currently unknown. In this section we consider several additional classes of languages that provide insight to the $\mathcal{P} = \mathcal{NP}$ question. Interestingly enough, properties of these classes are often dependent upon the relationship between $\mathcal{P}$ and $\mathcal{NP}$. The majority of the following discussion will proceed under the assumption that $\mathcal{P} \neq \mathcal{NP}$. However, this condition will be explicitly stated in any theorem whose proof utilizes the assumption.

Let $\mathcal{NPC}$ represent the class of all NP-complete languages. Our investigations in the preceding sections have established that $\mathcal{NPC}$ is not empty. Both $\mathcal{P}$ and $\mathcal{NPC}$ are subsets of $\mathcal{NP}$, but what is relationship between these two classes? By Theorem 15.3.4, if $\mathcal{P} \cap \mathcal{NPC}$ is nonempty then $\mathcal{P} = \mathcal{NP}$. Consequently, under the assumption $\mathcal{P} \neq \mathcal{NP}$, $\mathcal{P}$ and $\mathcal{NPC}$ must be disjoint.

The Venn diagram in Figure 15.6 shows the inclusions of $\mathcal{P}$ and $\mathcal{NPC}$ in $\mathcal{NP}$ if $\mathcal{P} \neq \mathcal{NP}$. Another question immediately arises when looking at this diagram: Are there languages in $\mathcal{NP}$ that are not in either $\mathcal{P}$ or $\mathcal{NPC}$? We define the family of languages $\mathcal{NPI}$, where the letter $\mathcal{I}$ represents intermediate, to consist of all languages in $\mathcal{NP} - (\mathcal{NPC} \cup \mathcal{P})$. The use of the word *intermediate* in this context is best explained in terms of solving decision problems. The universal reducibility of problems in $\mathcal{NP}$ to an NP-complete problem **P** guarantees that no problem in $\mathcal{NP}$ is more difficult to solve than **P**. The solution to **P**, when combined with the appropriate reduction, produces a solution any problem in $\mathcal{NP}$. Since problems in $\mathcal{NPI}$ are not NP-hard, there are some problems in $\mathcal{NP}$ that cannot be

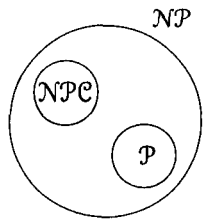


FIGURE 15.6 Classes $\mathcal{P}$, $\mathcal{NP}$, and $\mathcal{NPC}$ if $\mathcal{P} \neq \mathcal{NP}$.

reduced to an $\mathcal{NPI}$ problem. Solving an intermediate problem does not provide solutions to all problems in $\mathcal{NP}$. Thus the problems in $\mathcal{NPI}$ are more difficult to solve than those in $\mathcal{P}$ but not as comprehensive as the problems in $\mathcal{NPC}$. If $\mathcal{P} = \mathcal{NP}$, the class $\mathcal{NPI}$ is vacuously empty. Theorem 15.6.1, stated without proof, guarantees the existence of intermediate problems if $\mathcal{P} \neq \mathcal{NP}$.

Theorem 15.6.1
If $\mathcal{P} \neq \mathcal{NP}$, then $\mathcal{NPI}$ is not empty.

The complement of a language L over an alphabet Σ, denoted $\overline{\mathrm{L}}$, consists of all strings not in L, that is, $\overline{\mathrm{L}} = \Sigma^* - \mathrm{L}$. A family of languages $\mathcal{F}$ is said to be closed under complementation if $\overline{\mathrm{L}} \in \mathcal{F}$ whenever $\mathrm{L} \in \mathcal{F}$. The family $\mathcal{P}$ is closed under complementation; a deterministic Turing machine that accepts a language in polynomial time can be transformed to accept the complement with the same polynomial bound. The transformation simply consists of interchanging the accepting and rejecting states of the Turing machine.

The asymmetry of nondeterminism, however, has a dramatic impact on the complexity of machines that accept a language and those that accept its complement. To obtain an affirmative answer, a single nondeterministic "guess" that solves the problem is all that is required. A negative answer is obtained only if all possible guesses fail. The satisfiability problem is used to demonstrate the asymmetry of complexity of nondeterministic acceptance of a language and its complement.

The input to the satisfiability problem is a conjunctive normal form formula u over a set of Boolean variables V, and the output is yes if u is satisfiable and no otherwise. Theorem 15.4.2 described the computation of a nondeterministic machine that solves that satisfiability problem in polynomial time. This was accomplished by guessing a truth assignment on V. Checking whether a formula u is satisfied by a single truth assignment is a straightforward process that can be accomplished in time polynomial to the length of u.

The complement of the satisfiability problem is to determine whether a conjunctive normal form formula is unsatisfiable, that is, it is not satisfied by any truth assignment. Thus, an affirmative answer is obtained for a formula u if the truth value of u is 0 for every possible truth assignment. Guessing a single truth assignment and discovering that it

does not satisfy u does not provide sufficient information to conclude that u is unsatisfiable. Intuitively, the values of u under all truth assignments are required to determine the unsatisfiability. If $card(V) = n$, there are 2^n truth assignments to be examined. It seems reasonable to conclude that this problem is not in $\mathcal{NP}$. Note the use of the terms *intuitively* and *it seems reasonable* in the previous sentences. These hedges have been included because it is not known whether the unsatisfiability problem is in $\mathcal{NP}$.

Rather than only considering the complement of the satisfiability problem, we will examine the family of languages consisting of the complements of all languages in $\mathcal{NP}$. The family co-$\mathcal{NP} = \{\overline{L} \mid L \in \mathcal{NP}\}$.

Theorem 15.6.2

If $\mathcal{NP} \neq$ co-$\mathcal{NP}$, then $\mathcal{P} \neq \mathcal{NP}$.

Proof As noted previously, $\mathcal{P}$ is closed under complementation. Thus if $\mathcal{NP}$ is not closed under complementation, the two classes of languages cannot be identical. ■

Theorem 15.6.2 provides another method for answering the $\mathcal{P} = \mathcal{NP}$ question. It is sufficient to find a language $L \in \mathcal{NP}$ with $\overline{L} \notin \mathcal{NP}$. Proving that $\mathcal{NP} =$ co-$\mathcal{NP}$ does not answer the question of the identity of $\mathcal{P}$ and $\mathcal{NP}$. At this time, it is unknown whether $\mathcal{NP} =$ co-$\mathcal{NP}$. Just as it is generally believed that $\mathcal{P} \neq \mathcal{NP}$, it is also the consensus of theoretical computer scientists that $\mathcal{NP} \neq$ co-$\mathcal{NP}$. However, the majority does not rule in deciding mathematical properties and the search for a proof of these inequalities continues.

Theorem 15.6.3

If there is an NP-complete language L with $\overline{L} \in \mathcal{NP}$, then $\mathcal{NP} =$ co-$\mathcal{NP}$.

Proof Let L be a language over alphabet Σ that satisfies the above conditions. Since L is NP-complete, there is a reduction of any NP language Q to L. This reduction also serves as a reduction of $\overline{Q}$ to $\overline{L}$.

Choose any language Q in $\mathcal{NP}$. As noted above, $\overline{Q}$ can be reduced to $\overline{L}$ in polynomial time. Since $\overline{L} \in \mathcal{NP}$, $\overline{L}$ is accepted in polynomial time by a nondeterministic Turing machine. Combining the machine that performs the reduction of $\overline{Q}$ to $\overline{L}$ with the machine that accepts $\overline{L}$ produces a nondeterministic machine that accepts $\overline{Q}$ in polynomial time. Thus, co-$\mathcal{NP} \subseteq \mathcal{NP}$.

To complete the proof, it is necessary to establish the opposite inclusion. Let Q be any language in $\mathcal{NP}$. By the preceding argument, $\overline{Q}$ is also in $\mathcal{NP}$. The complement of $\overline{Q}$, which is Q itself, is then in co-$\mathcal{NP}$. ■

The satisfiability problem and its complement were used to initiate the examination of the family co-$\mathcal{NP}$. At that point we said that it seems reasonable to believe that the complement of the satisfiability problem is not in $\mathcal{NP}$. By Theorem 15.6.2, $\overline{L_{SAT}}$ is in $\mathcal{NP}$ if, and only if, $\mathcal{NP} =$ co-$\mathcal{NP}$.

We end this discussion with the consideration of two complementary languages over the alphabet $\Sigma = \{0, 1\}$. A string over Σ may be thought of as a binary representation

of an integer. Let $int(w)$ be the integer represented by the string $w \in \Sigma^*$. Consider the languages

$$\text{PRIME} = \{w \in \Sigma^* \mid int(w) \text{ is a prime number}\}.$$
$$\text{COMP} = \{w \in \Sigma^* \mid int(w) = 0, 1, \text{ or } k \text{ where } k = m \cdot n,\ m > 1,\ n > 1\}.$$

PRIME consists of the binary representation of prime numbers. COMP, containing the representation of composite numbers, is the complement of PRIME.

COMP is easily shown to be in $\mathcal{NP}$. For input w whose value is not 0 or 1, all that is required is to guess two integers in the appropriate range and determine if their product equals w. Establishing the membership of PRIME in $\mathcal{NP}$ is nowhere near as straightforward. Modular arthimetic and Fermat's theorem have been used to show that the primality of an integer k can be nondeterministically determined in time polynomial in $\log(k)$.

If either COMP or PRIME is NP-complete, it follows from Theorem 15.6.2 that $\mathcal{NP} = \text{co-}\mathcal{NP}$. Needless to say, neither of these languages has been proven to be NP-complete. These languages are, excuse the pun, prime candidates for membership in $\mathcal{NPI}$.

Exercises

1. Let M be a nondeterministic machine and p a polynomial. Assume that every string of length n in L(M) is accepted by at least one computation of $p(n)$ or fewer transitions. Note this makes no claim about the length of nonaccepting computations or other accepting computations. Prove that L(M) is in $\mathcal{NP}$.
2. Construct a deterministic Turing machine that reduces the language L_1 to L_2 in polynomial time. Using the big oh notation, give the time complexity of the machine that computes the reduction.
   a) $L_1 = \{a^i b^j c^i \mid i \geq 0, j \geq 0\}$ $\quad L_2 = \{a^i c^i \mid i \geq 0, j \geq 0\}$
   b) $L_1 = \{a^i b^i a^j \mid i \geq 0, j \geq 0\}$ $\quad L_2 = \{c^i d^i \mid i \geq 0, j \geq 0\}$
   c) $L_1 = \{a^i (bb)^i \mid i \geq 0\}$ $\quad L_2 = \{a^i b^i \mid i \geq 0\}$
   d) $L_1 = \{a^i b^i a^i \mid i \geq 0\}$ $\quad L_2 = \{c^i d^i \mid i \geq 0\}$
3. For each of the formulas below, give a truth assignment that satisfies the formula.
   a) $(x \vee y \vee \sim z) \wedge (\sim x \vee y) \wedge (\sim x \vee \sim y \vee \sim z)$
   b) $(\sim x \vee y \vee \sim z) \wedge (x \vee \sim y) \wedge (y \vee \sim z) \wedge (\sim x \vee \sim y \vee z)$
   c) $(x \vee y) \wedge (\sim x \vee \sim y \vee z) \wedge (x \vee \sim z) \wedge (\sim y \vee \sim z)$
4. Show that the formula $(x \vee \sim y) \wedge (\sim x \vee z) \wedge (y \vee \sim z) \wedge (\sim x \vee \sim y) \wedge (y \vee z)$ is not satisfiable.
5. Construct four clauses over $\{x, y, z\}$ such that the conjunction of any three is satisfiable but the conjunction of all four is unsatisfiable.
6. Prove that the formula u' is satisfiable if, and only if, u is satisfiable.

a) $u = v$, $u' = (v \vee y \vee z) \wedge (v \vee \sim y \vee z) \wedge (v \vee y \vee \sim z) \wedge (v \vee \sim y \vee \sim z)$

b) $u = v \vee w$, $u' = (v \vee w \vee y) \wedge (v \vee w \vee \sim y)$

7. A formula is in 4-conjunctive normal form if it is the conjunction of clauses consisting of the disjunction of four literals. Prove that the satisfiability problem for 4-conjunctive normal-form formulas is NP-complete.

8. A clique in an undirected graph G is a subgraph of G in which every two vertices are connected by an arc. The clique problem is to determine, for an arbitrary graph G and integer k, whether G has a clique of size k. Prove that the clique problem is NP-complete. *Hint*: To show that the clique problem is NP-hard, establish a relationship between cliques in a graph G and vertex covers in the complement graph $\overline{G}$. There is an arc between vertices x and y in $\overline{G}$ if, and only if, there is no arc connecting these vertices in G.

9. Let G be a weighted, directed graph. The cost of a path is the sum of the weights of the arcs in the path. The traveling salesman problem can be formulated as a decision problem as follows: Is there a tour of G with cost k or less? Prove that the traveling salesman problem is NP-complete. *Hint*: Reduce the Hamiltonian circuit problem to the traveling salesman problem.

10. The integer linear programming problem is: Given an n by m matrix $\mathbf{A}$ and a column vector $\mathbf{b}$ of length n, does there exist a column vector $\mathbf{x}$ such that $\mathbf{Ax} \geq \mathbf{b}$? Use a reduction of 3-SAT to prove that the integer linear programming problem is NP-hard. (The integer linear programming problem is also in $\mathcal{NP}$; the proof requires knowledge of some of the elementary properties of linear algebra.)

11. Assume that $\mathcal{P} = \mathcal{NP}$.

    a) Let L be a language in $\mathcal{NP}$ with $L \neq \emptyset$ and $\overline{L} \neq \emptyset$. Prove that L is NP-complete.

    b) Why is $\mathcal{NPC}$ a proper subset of $\mathcal{NP}$?

Bibliographic Notes

The family $\mathcal{P}$ was introduced in Cobham [1964]. $\mathcal{NP}$ was first studied by Edmonds [1965]. The foundations of the theory of NP-completeness were presented in Cook [1971]. This work includes the proof that the satisfiability problem is NP-complete. Karp [1972] demonstrated the importance of NP-completeness by proving that several well-known problems fall within this class. These include the 3-SAT, the vertex cover problem, and the Hamiltonian circuit problem. The first seven chapters of the book by Garey and Johnson [1979] give an entertaining introduction to the theory of NP-completeness. The remainder of the book is a virtual encyclopedia of NP-complete problems.

Theorem 15.6.1 follows from a result of Ladner [1975]. The proof that the language PRIME can be recognized in nondeterministic polynomial time can be found in Pratt [1975].

PART VI

Deterministic Parsing

The objective of a parser is to determine whether a string satisfies the syntactic requirements of a language. The parsers introduced in Chapter 4, when implemented with a Greibach normal form grammar, provide an algorithm for determining membership in a context-free language. While these algorithms demonstrate the feasibility of algorithmic syntax analysis, their inefficiency makes them inadequate for incorporation in frequently used applications such as commercial compilers or interpreters.

Building a derivation in a context-free grammar is inherently a nondeterministic process. In transforming a variable, any rule with the variable on the left-hand side may be applied. The breadth-first parsers built a search tree by applying all permissible rules, while the depth-first parsers explored derivations initiated with one rule and backed up to try another whenever the search was determined to be a dead-end. In either case, these algorithms have the potential for examining many extraneous derivations.

We will now introduce two families of context-free grammars that can be parsed deterministically. To ensure the selection of the appropriate action, the parser "looks ahead" in the string being analyzed. LL(k) grammars permit deterministic top-down parsing with a k symbol lookahead. LR(k) parsers use a finite automaton and k symbol lookahead to select a reduction or shift in a bottom-up parse.

CHAPTER 16

LL(k) Grammars

The LL(k) grammars constitute the largest subclass of context-free grammars that permits deterministic top-down parsing using a k-symbol lookahead. The notation LL describes the parsing strategy for which these grammars are designed; the input string is scanned in a left-to-right manner and the parser generates a leftmost derivation.

Throughout this chapter, all derivations and rule applications are leftmost. We also assume that the grammars do not contain useless symbols. Techniques for detecting and removing useless symbols were presented in Section 5.3.

16.1 Lookahead in Context-Free Grammars

A top-down parser attempts to construct a leftmost derivation of an input string p. The parser extends derivations of the form $S \stackrel{*}{\Rightarrow} uAv$, where u is a prefix of p, by applying an A rule. "Looking ahead" in the input string can reduce the number of A rules that must be examined. If $p = uaw$, the terminal a is obtained by looking one symbol beyond the prefix of the input string that has been generated by the parser. Using the lookahead symbol permits an A rule whose right-hand side begins with a terminal other than a to be eliminated from consideration. The application of any such rule generates a terminal string that is not a prefix of p.

Consider a derivation of the string $acbb$ in the regular grammar

$$\begin{aligned} \text{G: } & S \to aS \mid cA \\ & A \to bA \mid cB \mid \lambda \\ & B \to cB \mid a \mid \lambda. \end{aligned}$$

The derivation begins with the start symbol S and lookahead symbol a. The grammar contains two S rules, $S \to aS$ and $S \to cA$. Clearly, applying $S \to cA$ cannot lead to a derivation of $acbb$ since c does not match the lookahead symbol. It follows that the derivation must begin with an application of the rule $S \to aS$.

After the application of the S rule, the lookahead symbol is advanced to c. Again, there is only one S rule that generates c. Comparing the lookahead symbol with the terminal in each of the appropriate rules permits the deterministic construction of derivations in G.

Prefix Generated	Lookahead Symbol	Rule	Derivation
λ	a	$S \to aS$	$S \Rightarrow aS$
a	c	$S \to cA$	$\Rightarrow acA$
ac	b	$A \to bA$	$\Rightarrow acbA$
acb	b	$A \to bA$	$\Rightarrow acbbA$
$acbb$	λ	$A \to \lambda$	$\Rightarrow acbb$

Looking ahead one symbol is sufficient to construct derivations deterministically in the grammar G. A more general approach allows the lookahead to consist of the portion of the input string that has not been generated. An intermediate step in a derivation of a terminal string p has the form $S \stackrel{*}{\Rightarrow} uAv$, where $p = ux$. The string x is called a **lookahead string** for the variable A. The lookahead set of A consists of all lookahead strings for that variable.

Definition 16.1.1

Let G = (V, Σ, P, S) be a context-free grammar and $A \in$ V.

i) The **lookahead set** of the variable A, LA(A), is defined by

$$\text{LA}(A) = \{x \mid S \stackrel{*}{\Rightarrow} uAv \stackrel{*}{\Rightarrow} ux \in \Sigma^*\}.$$

ii) For each rule $A \to w$ in P, the lookahead set of the rule $A \to w$ is defined by

$$\text{LA}(A \to w) = \{x \mid wv \stackrel{*}{\Rightarrow} x \text{ where } x \in \Sigma^* \text{ and } S \stackrel{*}{\Rightarrow} uAv\}.$$

LA(A) consists of all terminal strings derivable from strings Av, where uAv is a left sentential form of the grammar. LA($A \to w$) is the subset of LA(A) in which the subderivations $Av \stackrel{*}{\Rightarrow} x$ are initiated with the rule $A \to w$.

Let $A \to w_1, \ldots, A \to w_n$ be the A rules of a grammar G. The lookahead string can be used to select the appropriate A rule whenever the sets $\mathrm{LA}(A \to w_i)$ partition $\mathrm{LA}(A)$, that is, when the sets $\mathrm{LA}(A \to w_i)$ satisfy

i) $\mathrm{LA}(A) = \bigcup_{i=1}^{n} \mathrm{LA}(A \to w_i)$

ii) $\mathrm{LA}(A \to w_i) \cap \mathrm{LA}(A \to w_j) = \emptyset$ for all $1 \leq i < j \leq n$.

The first condition is satisfied for every context-free grammar; it follows directly from the definition of the lookahead sets. If the lookahead sets satisfy (ii) and $S \overset{*}{\Rightarrow} uAv$ is a partial derivation of a string $p = ux \in \mathrm{L(G)}$, then x is an element of exactly one set $\mathrm{LA}(A \to w_k)$. Consequently, $A \to w_k$ is the only A rule whose application can lead to a successful completion of the derivation.

Example 16.1.1

The lookahead sets are constructed for the variables and the rules of the grammar

$$\begin{aligned} \mathrm{G_1}\colon\ & S \to Aabd \mid cAbcd \\ & A \to a \mid b \mid \lambda. \end{aligned}$$

$\mathrm{LA}(S)$ consists of all terminal strings derivable from S. Every terminal string derivable from the rule $S \to Aabd$ begins with a or b. On the other hand, derivations initiated by the rule $S \to cAbcd$ generate strings beginning with c.

$$\begin{aligned} \mathrm{LA}(S) &= \{aabd, babd, abd, cabcd, cbbcd, cbcd\} \\ \mathrm{LA}(S \to Aabd) &= \{aabd, babd, abd\} \\ \mathrm{LA}(S \to cAbcd) &= \{cabcd, cbbcd, cbcd\} \end{aligned}$$

Knowledge of the first symbol of the lookahead string is sufficient to select the appropriate S rule.

To construct the lookahead set for the variable A we must consider derivations from all the left sentential forms of $\mathrm{G_1}$ that contain A. There are only two such sentential forms, $Aabd$ and $cAbcd$. The lookahead sets consist of terminal strings derivable from $Aabd$ and $Abcd$.

$$\begin{aligned} \mathrm{LA}(A \to a) &= \{aabd, abcd\} \\ \mathrm{LA}(A \to b) &= \{babd, bbcd\} \\ \mathrm{LA}(A \to \lambda) &= \{abd, bcd\} \end{aligned}$$

The substring ab can be obtained by applying $A \to a$ to $Abcd$ and by applying $A \to \lambda$ to $Aabd$. Looking ahead three symbols in the input string provides sufficient information to discriminate between these rules. A top-down parser with a three-symbol lookahead can deterministically construct derivations in the grammar $\mathrm{G_1}$. □

A lookahead string of the variable A is the concatenation of the results of two derivations, one from the variable A and one from the portion of the sentential form following A. Example 16.1.2 emphasizes the dependence of the lookahead set on the sentential form.

Example 16.1.2

A lookahead string of G_2 receives at most one terminal from each of the variables A, B, and C.

$$\begin{aligned} G_2\colon\ S &\to ABCabcd \\ A &\to a \mid \lambda \\ B &\to b \mid \lambda \\ C &\to c \mid \lambda \end{aligned}$$

The only left sentential form of G_2 that contains A is $ABCabcd$. The variable B appears in $aBCabcd$ and $BCabcd$, both of which can be obtained by the application of an A rule to $ABCabcd$. In either case, $BCabcd$ is used to construct the lookahead set. Similarly, the lookahead set LA(C) consists of strings derivable from $Cabcd$.

$$\begin{aligned} \text{LA}(A \to a) &= \{abcabcd, acabcd, ababcd, aabcd\} \\ \text{LA}(A \to \lambda) &= \{bcabcd, cabcd, babcd, abcd\} \\ \text{LA}(B \to b) &= \{bcabcd, babcd\} \\ \text{LA}(B \to \lambda) &= \{cabcd, abcd\} \\ \text{LA}(C \to c) &= \{cabcd\} \\ \text{LA}(C \to \lambda) &= \{abcd\} \end{aligned}$$

A string with prefix abc can be derived from the sentential form $ABCabcd$ using the rule $A \to a$ or $A \to \lambda$. One-symbol lookahead is sufficient for selecting the B and C rules. Four-symbol lookahead is required to parse the strings of G_2 deterministically. □

The lookahead sets LA(A) and LA($A \to w$) may contain strings of arbitrary length. The selection of rules in the previous examples needed only fixed-length prefixes of strings in the lookahead sets. The k-symbol lookahead sets are obtained by truncating the strings of the sets LA(A) and LA($A \to w$). A function $trunc_k$ is introduced to simplify the definition of the fixed-length lookahead sets.

Definition 16.1.2

Let G = (V, Σ, P, S) be a context-free grammar and let k be a natural number greater than zero.

i) $trunc_k$ is a function from $\mathcal{P}(\Sigma^*)$ to $\mathcal{P}(\Sigma^*)$ defined by

$$trunc_k(X) = \{u \mid u \in X \text{ with } length(u) \leq k \text{ or } uv \in X \text{ with } length(u) = k\}$$

for all $X \in \mathcal{P}(\Sigma^*)$.

ii) The length-k lookahead set of the variable A is the set

$$\mathrm{LA}_k(A) = trunc_k(\mathrm{LA}(A)).$$

iii) The length-k lookahead set of the rule $A \to w$ is the set

$$\mathrm{LA}_k(A \to w) = trunc_k(\mathrm{LA}(A \to w)).$$

Example 16.1.3

The length-three lookahead sets for the rules of the grammar G_1 from Example 16.1.1 are

$$\mathrm{LA}_3(S \to Aabd) = \{aab, bab, abd\}$$
$$\mathrm{LA}_3(S \to cAbcd) = \{cab, cbb, cbc\}$$
$$\mathrm{LA}_3(A \to a) = \{aab, abc\}$$
$$\mathrm{LA}_3(A \to b) = \{bab, bbc\}$$
$$\mathrm{LA}_3(A \to \lambda) = \{abd, bcd\}.$$

Since there is no string in common in the length three lookahead sets of the S rules or the A rules, a three symbol lookahead is sufficient to determine the appropriate rule of G_1. □

Example 16.1.4

The language $\{a^i abc^i \mid i > 0\}$ is generated by each of the grammars G_1, G_2, and G_3. The minimal length lookahead sets necessary for discriminating between alternative productions are given for these grammars.

Rule		Lookahead Set
G_1:	$S \to aSc$	$\{aaa\}$
	$S \to aabc$	$\{aab\}$
G_2:	$S \to aA$	
	$A \to Sc$	$\{aa\}$
	$A \to abc$	$\{ab\}$
G_3:	$S \to aaAc$	
	$A \to aAc$	$\{a\}$
	$A \to b$	$\{b\}$

A one-symbol lookahead is insufficient for determining the S rule in G_1 since both of the alternatives begin with the symbol a. In fact, three symbol lookahead is required to determine the appropriate rule. G_2 is constructed from G_1 by using the S rule to generate the leading a. The variable A is added to generate the remainder of the right-hand side of the S rules of G_1. This technique is known as **left factoring** since the leading a is factored out of the rules $S \to aSc$ and $S \to aabc$. Left factoring the S rule reduces the length of the lookahead needed to select the rules.

A lookahead of length one is sufficient to parse strings with the rules of G_3. The recursive A rule generates an a while the nonrecursive rule terminates the derivation by generating a b. □

16.2 FIRST, FOLLOW, and Lookahead Sets

The lookahead set $LA_k(A)$ contains prefixes of length k of strings that can be derived from the variable A. If A derives strings of length less than k, the remainder of the lookahead strings comes from derivations that follow A in the sentential forms of the grammar. For each variable A, sets $FIRST_k(A)$ and $FOLLOW_k(A)$ are introduced to provide the information required for constructing the lookahead sets. $FIRST_k(A)$ contains prefixes of terminal strings derivable from A. $FOLLOW_k(A)$ contains prefixes of terminal strings that can follow the strings derivable from A. For convenience, a set $FIRST_k$ is defined for every string in $(V \cup \Sigma)^*$.

Definition 16.2.1

Let G be a context-free grammar. For every string $u \in (V \cup \Sigma)^*$ and $k > 0$, the set $FIRST_k(u)$ is defined by

$$FIRST_k(u) = trunc_k(\{x \mid u \stackrel{*}{\Rightarrow} x,\ x \in \Sigma^*\}).$$

Example 16.2.1

FIRST sets are constructed for the strings S and ABC using the grammar G_2 from Example 16.1.2.

$$FIRST_1(ABC) = \{a, b, c, \lambda\}$$
$$FIRST_2(ABC) = \{ab, ac, bc, a, b, c, \lambda\}$$
$$FIRST_3(S) = \{abc, aba, aca, bca, bab, cab\}$$ □

Recall that the concatenation of two sets X and Y is denoted by juxtaposition, $XY = \{xy \mid x \in X \text{ and } y \in Y\}$. Using this notation, we can establish the following relationships for the $FIRST_k$ sets.

Lemma 16.2.2

For every $k > 0$,

1. $FIRST_k(\lambda) = \{\lambda\}$
2. $FIRST_k(a) = \{a\}$
3. $FIRST_k(au) = \{av \mid v \in FIRST_{k-1}(u)\}$
4. $FIRST_k(uv) = trunc_k(FIRST_k(u)FIRST_k(v))$
5. if $A \rightarrow w$ is a rule in G, then $FIRST_k(w) \subseteq FIRST_k(A)$.

Definition 16.2.3

Let G be a context-free grammar. For every $A \in V$ and $k > 0$, the set $\text{FOLLOW}_k(A)$ is defined by

$$\text{FOLLOW}_k(A) = \{x \mid S \overset{*}{\Rightarrow} uAv \text{ and } x \in \text{FIRST}_k(v)\}.$$

The set $\text{FOLLOW}_k(A)$ consists of prefixes of terminal strings that can follow the variable A in derivations in G. Since the null string follows every derivation from the sentential form consisting solely of the start symbol, $\lambda \in \text{FOLLOW}_k(S)$.

Example 16.2.2

The FOLLOW sets of length one and two are given for the variables of G_2.

$$\begin{aligned} \text{FOLLOW}_1(S) &= \{\lambda\} & \text{FOLLOW}_2(S) &= \{\lambda\} \\ \text{FOLLOW}_1(A) &= \{a, b, c\} & \text{FOLLOW}_2(A) &= \{bc, ba, ca\} \\ \text{FOLLOW}_1(B) &= \{a, c\} & \text{FOLLOW}_2(B) &= \{ca, ab\} \\ \text{FOLLOW}_1(C) &= \{a\} & \text{FOLLOW}_2(C) &= \{ab\} \end{aligned}$$

□

The FOLLOW sets of a variable B are obtained from the rules in which B occurs on the right-hand side. Consider the relationships generated by a rule of the form $A \to uBv$. The strings that follow B include those generated by v concatenated with all terminal strings that follow A. If the grammar contains a rule $A \to uB$, any string that follows A can also follow B. The preceding discussion is summarized in Lemma 16.2.4.

Lemma 16.2.4

For every $k > 0$,

1. $\text{FOLLOW}_k(S)$ contains λ, where S is the start symbol of G
2. If $A \to uB$ is a rule of G, then $\text{FOLLOW}_k(A) \subseteq \text{FOLLOW}_k(B)$
3. If $A \to uBv$ is a rule of G, then $\textit{trunc}_k(\text{FIRST}_k(v)\text{FOLLOW}_k(A)) \subseteq \text{FOLLOW}_k(B)$.

The FIRST_k and FOLLOW_k sets are used to construct the lookahead sets for the rules of a grammar. Theorem 16.2.5 follows immediately from the definitions of the length-k lookahead sets and the function $\textit{trunc}_k$.

Theorem 16.2.5

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For every $k > 0$, $A \in V$, and rule $A \to w = u_1 u_2 \ldots u_n$ in P,

i) $\text{LA}_k(A) = \textit{trunc}_k(\text{FIRST}_k(A)\text{FOLLOW}_k(A))$

ii) $$\begin{aligned} \text{LA}_k(A \to w) &= \textit{trunc}_k(\text{FIRST}_k(w)\text{FOLLOW}_k(A)) \\ &= \textit{trunc}_k(\text{FIRST}_k(u_1) \ldots \text{FIRST}_k(u_n)\text{FOLLOW}_k(A)). \end{aligned}$$

Example 16.2.3

The FIRST_3 and FOLLOW_3 sets for the symbols in the grammar

$$G_1: \; S \rightarrow Aabd \mid cAbcd$$
$$A \rightarrow a \mid b \mid \lambda$$

from Example 16.1.1 are given below.

$$\text{FIRST}_3(S) = \{aab, bab, abd, cab, cbb, cbc\}$$
$$\text{FIRST}_3(A) = \{a, b, \lambda\}$$
$$\text{FIRST}_3(a) = \{a\}$$
$$\text{FIRST}_3(b) = \{b\}$$
$$\text{FIRST}_3(c) = \{c\}$$
$$\text{FIRST}_3(d) = \{d\}$$
$$\text{FOLLOW}_3(S) = \{\lambda\}$$
$$\text{FOLLOW}_3(A) = \{abd, bcd\}$$

The set $\text{LA}_3(S \rightarrow Aabd)$ is explicitly constructed from the sets $\text{FIRST}_3(A)$, $\text{FIRST}_3(a)$, $\text{FIRST}_3(b)$, $\text{FIRST}_3(d)$, and $\text{FOLLOW}_3(S)$ using the strategy outlined in Theorem 16.2.5.

$$\begin{aligned}\text{LA}_3(S \rightarrow Aabd) &= trunc_3(\text{FIRST}_3(A)\text{FIRST}_3(a)\text{FIRST}_3(b)\text{FIRST}_3(d)\text{FOLLOW}_3(S)) \\ &= trunc_3(\{a, b, \lambda\}\{a\}\{b\}\{d\}\{\lambda\}) \\ &= trunc_3(\{aabd, babd, abd\}) \\ &= \{aab, bab, abd\}\end{aligned}$$

The remainder of the length-three lookahead sets for the rules of G_1 can be found in Example 16.1.3. □

16.3 Strong LL(k) Grammars

We have seen that the lookahead sets can be used to select the A rule in a top-down parse when $\text{LA}(A)$ is partitioned by the sets $\text{LA}(A \rightarrow w_i)$. This section introduces a subclass of context-free grammars known as the strong $\text{LL}(k)$ grammars. The strong $\text{LL}(k)$ condition guarantees that the lookahead sets $\text{LA}_k(A)$ are partitioned by the sets $\text{LA}_k(A \rightarrow w_i)$.

When employing a k-symbol lookahead, it is often helpful if there are k symbols to be examined. An endmarker $\#^k$ is concatenated to the end of each string in the language to guarantee that every lookahead string contains exactly k symbols. If the start symbol S of the grammar is nonrecursive, the endmarker can be concatenated to the right-hand side of each S rule. Otherwise, the grammar can be augmented with a new start symbol S' and rule $S' \rightarrow S\#^k$.

Definition 16.3.1
Let G = (V, Σ, P, S) be a context-free grammar with endmarker $\#^k$. G is **strong LL(k)** if whenever there are two leftmost derivations

$$S \overset{*}{\Rightarrow} u_1 A v_1 \Rightarrow u_1 x v_1 \overset{*}{\Rightarrow} u_1 z w_1$$

$$S \overset{*}{\Rightarrow} u_2 A v_2 \Rightarrow u_2 y v_2 \overset{*}{\Rightarrow} u_2 z w_2,$$

where $u_i, w_i, z \in \Sigma^*$ and $length(z) = k$, then $x = y$.

We now establish several properties of strong LL(k) grammars. First we show that the length-k lookahead sets can be used to parse strings deterministically in a strong LL(k) grammar.

Theorem 16.3.2
A grammar G is strong LL(k) if, and only if, the sets $\mathrm{LA}_k(A \to w_i)$ partition $\mathrm{LA}(A)$ for each variable $A \in \mathrm{V}$.

Proof Assume that the sets $\mathrm{LA}_k(A \to w_i)$ partition $\mathrm{LA}(A)$ for each variable $A \in \mathrm{V}$. Let z be a terminal string of length k that can be obtained by the derivations

$$S \overset{*}{\Rightarrow} u_1 A v_1 \Rightarrow u_1 x v_1 \overset{*}{\Rightarrow} u_1 z w_1$$

$$S \overset{*}{\Rightarrow} u_2 A v_2 \Rightarrow u_2 y v_2 \overset{*}{\Rightarrow} u_2 z w_2.$$

Then z is in both $\mathrm{LA}_k(A \to x)$ and $\mathrm{LA}_k(A \to y)$. Since the sets $\mathrm{LA}_k(A \to w_i)$ partition $\mathrm{LA}(A)$, $x = y$ and G is strong LL(k).

Conversely, let G be a strong LL(k) grammar and let z be an element of $\mathrm{LA}_k(A)$. The strong LL(k) condition ensures that there is only one A rule that can be used to derive terminal strings of the form uzw from the sentential forms uAv of G. Consequently, z is in the lookahead set of exactly one A rule. This implies that the sets $\mathrm{LA}_k(A \to w_i)$ partition $\mathrm{LA}_k(A)$. ■

Theorem 16.3.3
If G is strong LL(k) for some k, then G is unambiguous.

Intuitively, a grammar that can be deterministically parsed must be unambiguous; there is exactly one rule that can be applied at each step in the derivation of a terminal string. The formal proof of this proposition is left as an exercise.

Theorem 16.3.4
If G has a left recursive variable, then G is not strong LL(k), for any $k > 0$.

Proof Let A be a left recursive variable. Since G does not contain useless variables, there is a derivation of a terminal string containing a left recursive subderivation of the variable A. The proof is presented in two cases.

Case 1: A is directly left recursive: A derivation containing direct left recursion uses A rules of the form $A \to Ay$ and $A \to x$, where the first symbol of x is not A.

$$S \overset{*}{\Rightarrow} uAv \Rightarrow uAyv \Rightarrow uxyv \overset{*}{\Rightarrow} uw \in \Sigma^*$$

The prefix of w of length k is in both $\mathrm{LA}_k(A \to Ay)$ and $\mathrm{LA}_k(A \to x)$. By Theorem 16.3.2, G is not strong LL(k).

Case 2: A is indirectly left recursive: A derivation with indirect recursion has the form

$$S \overset{*}{\Rightarrow} uAv \Rightarrow uB_1yv \Rightarrow \cdots \Rightarrow uB_nv_n \Rightarrow uAv_{n+1} \Rightarrow uxv_{n+1} \overset{*}{\Rightarrow} uw \in \Sigma^*.$$

Again, G is not strong LL(k) since the sets $\mathrm{LA}_k(A \to B_1y)$ and $\mathrm{LA}_k(A \to x)$ are not disjoint. ■

16.4 Construction of FIRST$_k$ Sets

We now present algorithms to construct the length-k lookahead sets for a context-free grammar with endmarker $\#^k$. This is accomplished by generating the FIRST_k and FOLLOW_k sets for the variables of the grammar. The lookahead sets can then be constructed using the technique presented in Theorem 16.2.5.

The initial step in the construction of the lookahead sets begins with the generation of the FIRST_k sets. Consider a rule of the form $A \to u_1u_2 \ldots u_n$. The subset of $\mathrm{FIRST}_k(A)$ generated by this rule can be constructed from the sets $\mathrm{FIRST}_k(u_1)$, $\mathrm{FIRST}_k(u_2)$, ..., $\mathrm{FIRST}_k(u_n)$, and $\mathrm{FOLLOW}_k(A)$. The problem of constructing FIRST_k sets for a string reduces to that of finding the sets for the variables in the string.

Algorithm 16.4.1
Construction of FIRST$_k$ Sets

input: context-free grammar $\mathrm{G} = (\mathrm{V}, \Sigma, \mathrm{P}, S)$

1. **for** each $a \in \Sigma$ **do** $\mathrm{F}'(a) := \{a\}$
2. **for** each $A \in \mathrm{V}$ **do** $\mathrm{F}(A) := \begin{cases} \{\lambda\} & \text{if } A \to \lambda \text{ is a rule in P} \\ \emptyset & \text{otherwise} \end{cases}$
3. **repeat**
   - 3.1 **for** each $A \in \mathrm{V}$ **do** $\mathrm{F}'(A) := \mathrm{F}(A)$
   - 3.2 **for** each rule $A \to u_1u_2 \ldots u_n$ with $n > 0$ **do**
     $\mathrm{F}(A) := \mathrm{F}(A) \cup \mathit{trunc}_k(\mathrm{F}'(u_1)\mathrm{F}'(u_2) \ldots \mathrm{F}'(u_n))$

   until $\mathrm{F}(A) = \mathrm{F}'(A)$ for all $A \in \mathrm{V}$
4. $\mathrm{FIRST}_k(A) = \mathrm{F}(A)$

The elements of FIRST$_k(A)$ are generated in step 3.2. At the beginning of each iteration of the repeat-until loop, the auxiliary set F′(A) is assigned the current value of F(A). Strings obtained from the concatenation F′(u_1)F′$(u_2)\ldots$F′(u_n), where $A \to u_1u_2\ldots u_n$ is a rule of G, are then added to F(A). The algorithm halts when an iteration occurs in which none of the sets F(A) are altered.

Example 16.4.1

Algorithm 16.4.1 is used to construct the FIRST$_2$ sets for the variables of the grammar

$$\begin{aligned} \text{G: } & S \to A\#\# \\ & A \to aAd \mid BC \\ & B \to bBc \mid \lambda \\ & C \to acC \mid ad. \end{aligned}$$

The sets F′(a) are initialized to $\{a\}$ for each $a \in \Sigma$. The action of the repeat-until loop is prescribed by the right-hand side of the rules of the grammar. Step 3.2 generates the assignment statements

$$\begin{aligned} &\text{F}(S) := \text{F}(S) \cup trunc_2(F'(A)\{\#\}\{\#\}) \\ &\text{F}(A) := \text{F}(A) \cup trunc_2(\{a\}\text{F}'(A)\{d\}) \cup trunc_2(\text{F}'(B)\text{F}'(C)) \\ &\text{F}(B) := \text{F}(B) \cup trunc_2(\{b\}\text{F}'(B)\{c\}) \\ &\text{F}(C) := \text{F}(C) \cup trunc_2(\{a\}\{c\}\text{F}'(C)) \cup trunc_2(\{a\}\{d\}) \end{aligned}$$

from the rules of G. The generation of the FIRST$_2$ sets is traced by giving the status of the sets F(S), F(A), F(B), and F(C) after each iteration of the loop. Recall that the concatenation of the empty set with any set yields the empty set.

	F(S)	F(A)	F(B)	F(C)
0	$\emptyset$	$\emptyset$	$\{\lambda\}$	$\emptyset$
1	$\emptyset$	$\emptyset$	$\{\lambda, bc\}$	$\{ad\}$
2	$\emptyset$	$\{ab, bc\}$	$\{\lambda, bc, bb\}$	$\{ad, ac\}$
3	$\{ad, bc\}$	$\{ad, bc, aa, ab, bb, ac\}$	$\{\lambda, bc, bb\}$	$\{ad, ac\}$
4	$\{ad, bc, aa, ab, bb, ac\}$	$\{ad, bc, aa, ab, bb, ac\}$	$\{\lambda, bc, bb\}$	$\{ad, ac\}$
5	$\{ad, bc, aa, ab, bb, ac\}$	$\{ad, bc, aa, ab, bb, ac\}$	$\{\lambda, bc, bb\}$	$\{ad, ac\}$

□

Theorem 16.4.2

Let G $= (\text{V}, \Sigma, \text{P}, S)$ be a context-free grammar. Algorithm 16.4.1 generates the sets FIRST$_k(A)$, for every variable $A \in \text{V}$.

Proof The proof consists of showing that the repeat-until loop in step 3 terminates and, upon termination, F(A) = FIRST$_k(A)$.

i) Algorithm 16.4.1 terminates: The number of iterations of the repeat-until loop is bounded since there are only a finite number of lookahead strings of length k or less.
ii) $\mathrm{F}(A) = \mathrm{FIRST}_k(A)$: First we prove that $\mathrm{F}(A) \subseteq \mathrm{FIRST}_k(A)$ for all variables $A \in \mathrm{V}$. To accomplish this we show that $\mathrm{F}(A) \subseteq \mathrm{FIRST}_k(A)$ at the beginning of each iteration of the repeat-until loop. By inspection, this inclusion holds prior to the first iteration. Assume $\mathrm{F}(A) \subseteq \mathrm{FIRST}_k(A)$ for all variables A after m iterations of the loop.

During the $m + 1$st iteration, the only additions to $\mathrm{F}(A)$ come from assignment statements of the form

$$\mathrm{F}(A) := \mathrm{F}(A) \cup \mathit{trunc}_k(\mathrm{F}'(u_1)\mathrm{F}'(u_2)\ldots\mathrm{F}'(u_n)),$$

where $A \to u_1u_2\ldots u_n$ is a rule of G. By the inductive hypothesis, each of the sets $\mathrm{F}'(u_i)$ is the subset of $\mathrm{FIRST}_k(u_i)$. If u is added to $\mathrm{F}(A)$ on the iteration then

$$\begin{aligned} u &\in \mathit{trunc}_k(\mathrm{F}'(u_1)\mathrm{F}'(u_2)\ldots\mathrm{F}'(u_n)) \\ &\subseteq \mathit{trunc}_k(\mathrm{FIRST}_k(u_1)\mathrm{FIRST}_k(u_2)\ldots\mathrm{FIRST}_k(u_n)) \\ &= \mathrm{FIRST}_k(u_1u_2\ldots u_n) \\ &\subseteq \mathrm{FIRST}_k(A) \end{aligned}$$

and $u \in \mathrm{FIRST}_k(A)$. The final two steps follow from parts 4 and 5 of Lemma 16.2.2.

We now show that $\mathrm{FIRST}_k(A) \subseteq \mathrm{F}(A)$ upon completion of the loop. Let $\mathrm{F}_m(A)$ be the value of the set $\mathrm{F}(A)$ after m iterations. Assume the repeat-until loop halts after j iterations. We begin with the observation that if a string can be shown to be in $\mathrm{F}_m(A)$ for some $m > j$ then it is in $\mathrm{F}_j(A)$. This follows since the sets $\mathrm{F}(A)$ and $\mathrm{F}'(A)$ would be identical for all iterations of the loop past iteration j. We will show that $\mathrm{FIRST}_k(A) \subseteq \mathrm{F}_j(A)$.

Let x be a string in $\mathrm{FIRST}_k(A)$. Then there is a derivation $A \overset{m}{\Rightarrow} w$, where $w \in \sum^*$ and x is the prefix of w of length k. We show that $x \in \mathrm{F}_m(A)$. The proof is by induction on the length of the derivation. The basis consists of terminal strings that can be derived with one rule application. If $A \to w \in \mathrm{P}$, then x is added to $\mathrm{F}_1(A)$.

Assume that $\mathit{trunc}_k(\{w \mid A \overset{m}{\Rightarrow} w \in \Sigma^*\}) \subseteq \mathrm{F}_m(A)$ for all variables A in V. Let $x \in \mathit{trunc}_k(\{w \mid A \overset{m+1}{\Longrightarrow} w \in \Sigma^*\})$; that is, x is a prefix of terminal string derivable from A by $m + 1$ rule applications. We will show that $x \in \mathrm{F}_{m+1}(A)$. The derivation of w can be written

$$A \Rightarrow u_1u_2\ldots u_n \overset{m}{\Rightarrow} x_1x_2\ldots x_n = w,$$

where $u_i \in \mathrm{V} \cup \Sigma$ and $u_i \overset{*}{\Rightarrow} x_i$. Clearly, each subderivation $u_i \overset{*}{\Rightarrow} x_i$ has length less than $m + 1$. By the inductive hypothesis, the string obtained by truncating x_i at length k is in $\mathrm{F}_m(u_i)$.

On the $m + 1$st iteration, $\mathrm{F}_{m+1}(A)$ is augmented with the set

$$\mathit{trunc}_k(\mathrm{F}'_{m+1}(u_1)\ldots\mathrm{F}'_{m+1}(u_n)) = \mathit{trunc}_k(\mathrm{F}_m(u_1)\ldots\mathrm{F}_m(u_n)).$$

Thus,

$$\{x\} = trunc_k(x_1x_2 \ldots x_n) \subseteq trunc_k(\mathrm{F}_m(u_1) \ldots \mathrm{F}_m(u_n))$$

and x is an element of $\mathrm{F}_{m+1}(A)$. It follows that every string in $\mathrm{FIRST}_k(A)$ is in $\mathrm{F}_j(A)$, as desired. ■

16.5 Construction of FOLLOW$_k$ Sets

The inclusions in Lemma 16.2.4 form the basis of an algorithm to generate the FOLLOW_k sets. $\mathrm{FOLLOW}_k(A)$ is constructed from the FIRST_k sets and the rules in which A occurs on the right-hand side. Algorithm 16.5.1 generates $\mathrm{FOLLOW}_k(A)$ using the auxiliary set $\mathrm{FL}(A)$. The set $\mathrm{FL}'(A)$, which triggers the halting condition, maintains the value assigned to $\mathrm{FL}(A)$ on the preceding iteration.

Algorithm 16.5.1
Construction of FOLLOW$_k$ Sets

input: context-free grammar $\mathrm{G} = (\mathrm{V}, \Sigma, \mathrm{P}, S)$
$\mathrm{FIRST}_k(A)$ for every $A \in \mathrm{V}$

1. $\mathrm{FL}(S) := \{\lambda\}$
2. **for** each $A \in \mathrm{V} - \{S\}$ **do** $\mathrm{FL}(A) := \emptyset$
2. **repeat**
   3.1 **for** each $A \in \mathrm{V}$ **do** $\mathrm{FL}'(A) := \mathrm{FL}(A)$
   3.2 **for** each rule $A \rightarrow w = u_1u_2 \ldots u_n$ with $w \notin \Sigma^*$ **do**
   3.2.1. $\mathrm{L} := \mathrm{FL}'(A)$
   3.2.2. **if** $u_n \in \mathrm{V}$ **then** $\mathrm{FL}(u_n) := \mathrm{FL}(u_n) \cup \mathrm{L}$
   3.2.3. **for** $i := n - 1$ **to** 1 **do**
   3.2.3.1. $\mathrm{L} := trunc_k(\mathrm{FIRST}_k(u_{i+1})\mathrm{L})$
   3.2.3.2. **if** $u_i \in \mathrm{V}$ **then** $\mathrm{FL}(u_i) := \mathrm{FL}(u_i) \cup \mathrm{L}$
   end for
   end for
   until $\mathrm{FL}(A) = \mathrm{FL}'(A)$ for every $A \in \mathrm{V}$
4. $\mathrm{FOLLOW}_k(A) := \mathrm{FL}(A)$

The inclusion $\mathrm{FL}(A) \subseteq \mathrm{FOLLOW}_k(A)$ is established by showing that every element added to $\mathrm{FL}(A)$ in statements 3.2.2 or 3.2.3.2 is in $\mathrm{FOLLOW}_k(A)$. The opposite inclusion is obtained by demonstrating that every element of $\mathrm{FOLLOW}_k(A)$ is added to $\mathrm{FL}(A)$ prior to the termination of the repeat-until loop. The details are left as an exercise.

Example 16.5.1

Algorithm 16.5.1 is used to construct the set FOLLOW_2 for every variable of the grammar G from Example 16.4.1. The interior of the repeat-until loop processes each rule in a right-to-left fashion. The action of the loop is specified by the assignment statements obtained from the rules of the grammar.

Rule	Assignments
$S \to A\#\#$	$\text{FL}(A) := \text{FL}(A) \cup trunc_2(\{\#\#\}\text{FL}'(S))$
$A \to aAd$	$\text{FL}(A) := \text{FL}(A) \cup trunc_2(\{d\}\text{FL}'(A))$
$A \to BC$	$\text{FL}(C) := \text{FL}(C) \cup \text{FL}'(A)$
	$\text{FL}(B) := \text{FL}(B) \cup trunc_2(\text{FIRST}_2(C)\text{FL}'(A))$
	$= \text{FL}(B) \cup trunc_2(\{ad, ac\}\text{FL}'(A))$
$B \to bBc$	$\text{FL}(B) := \text{FL}(B) \cup trunc_2(\{c\}\text{FL}'(B))$

The rule $C \to acC$ has been omitted from the list since the assignment generated by this rule is $\text{FL}(C) := \text{FL}(C) \cup \text{FL}'(C)$. Tracing Algorithm 16.5.1 yields

	FL(S)	FL(A)	FL(B)	FL(C)
0	$\{\lambda\}$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{\lambda\}$	$\{\#\#\}$	$\emptyset$	$\emptyset$
2	$\{\lambda\}$	$\{\#\#, d\#\}$	$\{ad, ac\}$	$\{\#\#\}$
3	$\{\lambda\}$	$\{\#\#, d\#, dd\}$	$\{ad, ac, ca\}$	$\{\#\#, d\#\}$
4	$\{\lambda\}$	$\{\#\#, d\#, dd\}$	$\{ad, ac, ca, cc\}$	$\{\#\#, d\#, dd\}$
5	$\{\lambda\}$	$\{\#\#, d\#, dd\}$	$\{ad, ac, ca, cc\}$	$\{\#\#, d\#, dd\}$

□

Example 16.5.2

The length-two lookahead sets for the rules of the grammar G are constructed from the FIRST_2 and FOLLOW_2 sets generated in Examples 16.4.1 and 16.5.1.

$$\text{LA}_2(S \to A\#\#) = \{ad, bc, aa, ab, bb, ac\}$$

$$\text{LA}_2(A \to aAd) = \{aa, ab\}$$

$$\text{LA}_2(A \to BC) = \{bc, bb, ad, ac\}$$

$$\text{LA}_2(B \to bBc) = \{bb, bc\}$$

$$\text{LA}_2(B \to \lambda) = \{ad, ac, ca, cc\}$$

$$\text{LA}_2(C \to acC) = \{ac\}$$

$$\text{LA}_2(C \to ad) = \{ad\}$$

G is strong LL(2) since the lookahead sets are disjoint for each pair of alternative rules. □

The preceding algorithms provide a decision procedure to determine whether a grammar is strong LL(k). The process begins by generating the FIRST_k and FOLLOW_k sets using Algorithms 16.4.1 and 16.5.1. The techniques presented in Theorem 16.2.5 are then used to construct the length-k lookahead sets. By Theorem 16.3.2, the grammar is strong LL(k) if, and only if, the sets $\text{LA}_k(A \to x)$ and $\text{LA}_k(A \to y)$ are disjoint for each pair of distinct A rules.

16.6 A Strong LL(1) Grammar

The grammar AE was introduced in Section 4.3 to generate infix additive expressions containing a single variable b. AE is not strong LL(k) since it contains a directly left recursive A rule. In this section we modify AE to obtain a strong LL(1) grammar that generates the additive expressions. To guarantee that the resulting grammar is strong LL(1), the length-one lookahead sets are constructed for each rule.

The transformation begins by adding the endmarker # to the strings generated by AE. This ensures that a lookahead set does not contain the null string. The grammar

$$\begin{aligned} \text{AE: } & S \to A\# \\ & A \to T \\ & A \to A + T \\ & T \to b \\ & T \to (A) \end{aligned}$$

generates the strings in L(AE) concatenated with the endmarker #. The direct left recursion can be removed using the techniques presented in Section 5.5. The variable Z is used to convert the left recursion to right recursion, yielding the equivalent grammar AE_1.

$$\begin{aligned} \text{AE}_1\text{: } & S \to A\# \\ & A \to T \\ & A \to TZ \\ & Z \to +T \\ & Z \to +TZ \\ & T \to b \\ & T \to (A) \end{aligned}$$

AE_1 still cannot be strong LL(1) since the A rules both have T as the first symbol occurring on the right-hand side. This difficulty is removed by left factoring the A rules using the new variable B. Similarly, the right-hand side of the Z rules begin with identical

substrings. The variable Y is introduced by the factoring of the Z rules. AE_2 results from making these modifications to AE_1.

$$
\begin{aligned}
AE_2\colon\ & S \to A\# \\
& A \to TB \\
& B \to Z \\
& B \to \lambda \\
& Z \to +TY \\
& Y \to Z \\
& Y \to \lambda \\
& T \to b \\
& T \to (A)
\end{aligned}
$$

To show that AE_2 is strong LL(1), the length-one lookahead sets for the variables of the grammar must satisfy the partition condition of Theorem 16.3.2. We begin by tracing the sequence of sets generated by Algorithm 16.4.1 in the construction of the $FIRST_1$ sets.

	F(S)	**F**(A)	**F**(B)	**F**(Z)	**F**(Y)	**F**(T)
0	$\emptyset$	$\emptyset$	$\{\lambda\}$	$\emptyset$	$\{\lambda\}$	$\emptyset$
1	$\emptyset$	$\emptyset$	$\{\lambda\}$	$\{+\}$	$\{\lambda\}$	$\{b, (\}$
2	$\emptyset$	$\{b, (\}$	$\{\lambda, +\}$	$\{+\}$	$\{\lambda, +\}$	$\{b, (\}$
3	$\{b, (\}$	$\{b, (\}$	$\{\lambda, +\}$	$\{+\}$	$\{\lambda, +\}$	$\{b, (\}$
4	$\{b, (\}$	$\{b, (\}$	$\{\lambda, +\}$	$\{+\}$	$\{\lambda, +\}$	$\{b, (\}$

Similarly, the $FOLLOW_2$ sets are generated using Algorithm 16.5.1.

	FL(S)	FL(A)	FL(B)	FL(Z)	FL(Y)	FL(T)
0	{λ}	∅	∅	∅	∅	∅
1	{λ}	{#,)}	∅	∅	∅	∅
2	{λ}	{#,)}	{#,)}	∅	∅	∅
3	{λ}	{#,)}	{#,)}	{#,)}	∅	∅
4	{λ}	{#,)}	{#,)}	{#,)}	{#,)}	∅
5	{λ}	{#,)}	{#,)}	{#,)}	{#,)}	{#,)}
6	{λ}	{#,)}	{#,)}	{#,)}	{#,)}	{#,)}

The length-one lookahead sets are obtained from the FIRST_1 and FOLLOW_1 sets generated above.

$$\text{LA}_1(S \to A\#) = \{b, (\}$$
$$\text{LA}_1(A \to TB) = \{b, (\}$$
$$\text{LA}_1(B \to Z) = \{+\}$$
$$\text{LA}_1(B \to \lambda) = \{\#,)\}$$
$$\text{LA}_1(Z \to +TY) = \{+\}$$
$$\text{LA}_1(Z \to \lambda) = \{\#,)\}$$
$$\text{LA}_1(Y \to Z) = \{+\}$$
$$\text{LA}_1(Y \to \lambda) = \{\#,)\}$$
$$\text{LA}_1(T \to b) = \{b\}$$
$$\text{LA}_1(T \to (A)) = \{(\}$$

Since the lookahead sets for alternative rules are disjoint, the grammar AE_2 is strong LL(1).

16.7 A Strong LL(*k*) Parser

Parsing with a strong LL(k) grammar begins with the construction of the lookahead sets for each of the rules of the grammar. Once these sets have been built, they are available for the parsing of any number of strings. The strategy for parsing strong LL(k) grammars presented in Algorithm 16.7.1 consists of a loop that compares the lookahead string with the lookahead sets and applies the appropriate rule.

Algorithm 16.7.1
Deterministic Parser for a Strong $LL_{(k)}$ Grammar

input: strong LL(k) grammar G = (V, Σ, P, S)
string $p \in \Sigma^*$
lookahead sets $LA_k(A \rightarrow w)$ for each rule in P

1. $q := S$
2. **repeat**
   Let $q = uAv$ where A is the leftmost variable in q and
   let $p = uyz$ where $length(y) = k$.
   2.1. **if** $y \in LA_k(A \rightarrow w)$ for some A rule **then** $q := uwv$
   until $q = p$ **or** $y \notin LA_k(A \rightarrow w)$ for all A rules
3. **if** $q = p$ **then** accept **else** reject

The presence of the endmarker in the grammar ensures that the lookahead string y contains k symbols. The input string is rejected whenever the lookahead string is not an element of one of the lookahead sets. When the lookahead string is in $LA_k(A \rightarrow w)$, a new sentential form is constructed by applying $A \rightarrow w$ to the current string uAv. The input is accepted if this rule application generates the input string. Otherwise, the loop is repeated for the sentential form uwv.

Example 16.7.1

Algorithm 16.7.1 and the lookahead sets from Section 16.6 are used to parse the string $(b + b)\#$ using the strong LL(1) grammar AE_2. Each row in the table below represents one iteration of step 2 of Algorithm 16.7.1.

u	A	v	Lookahead	Rule	Derivation
λ	S	λ	(	$S \rightarrow A\#$	$S \Rightarrow A\#$
λ	A	#	(	$A \rightarrow TB$	$\Rightarrow TB\#$
λ	T	$B\#$	(	$T \rightarrow (A)$	$\Rightarrow (A)B\#$
(	A	$)B\#$	b	$A \rightarrow TB$	$\Rightarrow (TB)B\#$
(	T	$B)B\#$	b	$T \rightarrow b$	$\Rightarrow (bB)B\#$
$(b$	B	$)B\#$	+	$B \rightarrow Z$	$\Rightarrow (bZ)B\#$
$(b$	Z	$)B\#$	+	$Z \rightarrow +TY$	$\Rightarrow (b + TY)B\#$
$(b+$	T	$Y)B\#$	b	$T \rightarrow b$	$\Rightarrow (b + bY)B\#$
$(b+b$	Y	$)B\#$	)	$Y \rightarrow \lambda$	$\Rightarrow (b + b)B\#$
$(b+b)$	B	#	#	$B \rightarrow \lambda$	$\Rightarrow (b + b)\#$

□

16.8 LL(*k*) Grammars

The lookahead sets in a strong LL(k) grammar provide a global criterion for selecting a rule. When A is the leftmost variable in the sentential form being extended by the parser, the lookahead string generated by the parser and the lookahead sets provide sufficient information to select the appropriate A rule. This choice does not depend upon the sentential form containing A. The LL(k) grammars provide a local selection criterion; the choice of the rule depends upon both the lookahead and the sentential form.

Definition 16.8.1
Let G = (V, Σ, P, S) be a context-free grammar with endmarker $\#^k$. G is LL(k) if whenever there are two leftmost derivations

$$S \overset{*}{\Rightarrow} uAv \Rightarrow uxv \overset{*}{\Rightarrow} uzw_1$$

$$S \overset{*}{\Rightarrow} uAv \Rightarrow uyv \overset{*}{\Rightarrow} uzw_2,$$

where $u, w_i, z \in \Sigma^*$ and $length(z) = k$, then $x = y$.

Notice the difference between the derivations in Definitions 16.3.1 and 16.8.1. The strong LL(k) condition requires that there be a unique A rule that can derive the lookahead string z from any sentential form containing A. An LL(k) grammar only requires the rule to be unique for a fixed sentential form uAv. The lookahead sets for an LL(k) grammar must be defined for each sentential form.

Definition 16.8.2
Let G = (V, Σ, P, S) be a context-free grammar with endmarker $\#^k$ and uAv a sentential form of G.

i) The lookahead set of the sentential form uAv is defined by $\text{LA}_k(uAv) = \text{FIRST}_k(Av)$.

ii) The lookahead set for the sentential form uAv and rule $A \to w$ is defined by $\text{LA}_k(uAv, A \to w) = \text{FIRST}_k(wv)$.

A result similar to Theorem 16.3.2 can be established for LL(k) grammars. The unique selection of a rule for the sentential form uAv requires the set $\text{LA}_k(uAv)$ to be partitioned by the lookahead sets $\text{LA}_k(uAv, A \to w_i)$ generated by the A rules. If the grammar is strong LL(k), then the partition is guaranteed and the grammar is also LL(k).

Example 16.8.1

An LL(k) grammar need not be strong LL(k). Consider the grammar

$$\text{G}_1\text{:}\ S \to Aabd \mid cAbcd$$
$$A \to a \mid b \mid \lambda$$

whose lookahead sets were given in Example 16.1.1. G_1 is strong LL(3) but not strong LL(2) since the string ab is in both $LA_2(A \to a)$ and $LA_2(A \to \lambda)$. The length-two lookahead sets for the sentential forms containing the variables S and A are

$$LA_2(S, S \to Aabd) = \{aa, ba, ab\}$$
$$LA_2(S, S \to cAbcd) = \{ca, cb\}$$
$$LA_2(Aabd, A \to a) = \{aa\} \qquad LA_2(cAbcd, A \to a) = \{ab\}$$
$$LA_2(Aabd, A \to b) = \{ba\} \qquad LA_2(cAbcd, A \to b) = \{bb\}$$
$$LA_2(Aabd, A \to \lambda) = \{ab\} \qquad LA_2(cAbcd, A \to \lambda) = \{bc\}.$$

Since the alternatives for a given sentential form are disjoint, the grammar is LL(2). □

Example 16.8.2

A three-symbol lookahead is sufficient for a local selection of rules in the grammar

$$\begin{aligned} G:\ & S \to aBAd \mid bBbAd \\ & A \to abA \mid c \\ & B \to ab \mid a. \end{aligned}$$

The S and A rules can be selected with a one-symbol lookahead; so we turn our attention to selecting the B rule. The lookahead sets for the B rules are

$$LA_3(aBAd, B \to ab) = \{aba, abc\}$$
$$LA_3(aBAd, B \to a) = \{aab, acd\}$$
$$LA_3(bBbAd, B \to ab) = \{abb\}$$
$$LA_3(bBbAd, B \to a) = \{aba, abc\}.$$

The length-three lookahead sets for the two sentential forms that contain B are partitioned by the B rules. Consequently, G is LL(3). The strong LL(k) conditions can be checked by examining the lookahead sets for the B rules.

$$LA(B \to ab) = \boldsymbol{ab(ab)^*cd \cup abb(ab)^*cd}$$
$$LA(B \to a) = \boldsymbol{a(ab)^*cd \cup ab(ab)^*cd}$$

For any integer k, there is a string of length greater than k in both $LA(B \to ab)$ and $LA(B \to a)$. Consequently, G is not strong LL(k) for any k. □

Parsing deterministically with LL(k) grammars requires the construction of the local lookahead sets for the sentential forms generated during the parse. The lookahead set for a sentential form can be constructed directly from the $FIRST_k$ sets of the variables and terminals of the grammar. The lookahead set $LA_k(uAv, A \to w)$, where $w = w_1 \ldots w_n$

and $v = v_1 \ldots v_m$, is given by

$$trunc_k(\text{FIRST}_k(w_1) \ldots \text{FIRST}_k(w_n)\text{FIRST}_k(v_1) \ldots \text{FIRST}_k(v_w)).$$

A parsing algorithm for LL(k) grammars can be obtained from Algorithm 16.7.1 by adding the construction of the local lookahead sets.

Algorithm 16.8.3
Deterministic Parser for an LL(k) Grammar

input: LL(k) grammar G = (V, Σ, P, S)
string $p \in \Sigma^*$
$\text{FIRST}_k(A)$ for every $A \in$ V

1. $q := S$
2. **repeat**
   Let $q = uAv$ where A is the leftmost variable in q and
   let $p = uyz$ where $length(y) = k$.
   2.1. **for** each rule $A \rightarrow w$ construct the set $\text{LA}_k(uAv, A \rightarrow w)$
   2.2. **if** $y \in \text{LA}_k(uAv, A \rightarrow w)$ for some A rule **then** $q := uwv$
   until $q = p$ **or** $y \notin \text{LA}_k(uAv, A \rightarrow w)$ for all A rules
3. **if** $q = p$ **then** *accept* **else** *reject*

Exercises

1. Let G be a context-free grammar with start symbol S. Prove that LA(S)=L(G).
2. Give the lookahead sets for each variable and rule of the following grammars.
   a) $S \rightarrow ABab \mid bAcc$
      $A \rightarrow a \mid c$
      $B \rightarrow b \mid c \mid \lambda$
   b) $S \rightarrow aS \mid A$
      $A \rightarrow ab \mid b$
   c) $S \rightarrow AB \mid ab$
      $A \rightarrow aA \mid \lambda$
      $B \rightarrow bB \mid \lambda$
   d) $S \rightarrow aAbBc$
      $A \rightarrow aA \mid cA \mid \lambda$
      $B \rightarrow bBc \mid bc$
3. Give the FIRST_1 and FOLLOW_1 sets for each of the variables of the following grammars. Which of these grammars are strong LL(1)?

a) $S \to aAB\#$
$A \to a \mid \lambda$
$B \to b \mid \lambda$

b) $S \to AB\#$
$A \to aAb \mid B$
$B \to aBc \mid \lambda$

c) $S \to ABC\#$
$A \to aA \mid \lambda$
$B \to bBc \mid \lambda$
$C \to cA \mid dB \mid \lambda$

d) $S \to aAd\#$
$A \to BCD$
$B \to bB \mid \lambda$
$C \to cC \mid \lambda$
$D \to bD \mid \lambda$

4. Use Algorithms 16.4.1 and 16.5.1 to construct the FIRST_2 and FOLLOW_2 sets for variables of the following grammars. Construct the length-two lookahead sets for the rules of the grammars. Are these grammars strong LL(2)?

a) $S \to ABC\#\#$
$A \to aA \mid a$
$B \to bB \mid \lambda$
$C \to cC \mid a \mid b \mid c$

b) $S \to A\#\#$
$A \to bBA \mid BcAa \mid \lambda$
$B \to acB \mid b$

5. Prove parts 3, 4, and 5 of Lemma 16.2.2.

6. Prove Theorem 16.3.3.

7. Show that each of the grammars defined below is not strong LL(k) for any k. Construct a deterministic PDA that accepts the language generated by the grammar.

a) $S \to aSb \mid A$
$A \to aAc \mid \lambda$

b) $S \to A \mid B$
$A \to aAb \mid ab$
$B \to aBc \mid ac$

c) $S \to A$
$A \to aAb \mid B$
$B \to aB \mid a$

8. Prove that Algorithm 16.5.1 generates the sets $\text{FOLLOW}_k(A)$.

9. Modify the grammars given below to obtain an equivalent strong LL(1) grammar. Build the lookahead sets to ensure that the modified grammar is strong LL(1).

   a) $S \rightarrow A\#$
   $A \rightarrow aB \mid Ab \mid Ac$
   $B \rightarrow bBc \mid \lambda$

   b) $S \rightarrow aA\# \mid abB\# \mid abcC\#$
   $A \rightarrow aA \mid \lambda$
   $B \rightarrow bB \mid \lambda$
   $C \rightarrow cC \mid \lambda$

10. Parse the following strings with the LL(1) parser and the grammar AE_2. Trace the actions of the parser using the format of Example 16.7.1. The lookahead sets for AE_2 are given in Section 16.6.

    a) $b + (b)\#$

    b) $((b))\#$

    c) $b + b + b\#$

    d) $b + +b\#$

11. Construct the lookahead sets for the rules of the grammar. What is the minimal k such that the grammar is strong LL(k)? Construct the lookahead sets for the combination of each sentential form and rule. What is the minimal k such that the grammar is LL(k)?

    a) $S \rightarrow aAcaa \mid bAbcc$
    $A \rightarrow a \mid ab \mid \lambda$

    b) $S \rightarrow aAbc \mid bABbd$
    $A \rightarrow a \mid \lambda$
    $B \rightarrow a \mid b$

    c) $S \rightarrow aAbB \mid bAbA$
    $A \rightarrow ab \mid a$
    $B \rightarrow aB \mid b$

12. Prove that a grammar is strong LL(1) if, and only if, it is LL(1).

13. Prove that a context-free grammar G is LL(k) if, and only if, the lookahead set $LA_k(uAv)$ is partitioned by the sets $LA_k(uAv, A \rightarrow w_i)$ for each left sentential form uAv.

Bibliographic Notes

Parsing with LL(k) grammars was introduced by Lewis and Stearns [1968]. The theory of LL(k) grammars and deterministic parsing was further developed in Rosenkrantz and Stearns [1970]. Relationships between the class of LL(k) languages and other classes of

languages that can be parsed deterministically are examined in Aho and Ullman [1973]. The LL(k) hierarchy was presented in Kurki-Suonio [1969]. Foster [1968], Wood [1969], Stearns [1971], and Soisalon-Soininen and Ukkonen [1979] introduced techniques for modifying grammars to satisfy the LL(k) or strong LL(k) conditions.

The construction of compilers for languages defined by LL(1) grammars frequently employs the method of recursive descent. This approach allows the generation of machine code to accompany the syntax analysis. Several textbooks develop the relationships between LL(k) grammars, recursive descent syntax analysis, and compiler design. These include Lewis, Rosenkrantz, and Stearns [1976], Backhouse [1979], Barrett, Bates, Gustafson, and Couch [1986], and Pyster [1980]. A comprehensive introduction to syntax analysis and compiling can be found in Aho, Sethi, and Ullman [1986].

CHAPTER 17

LR(k) Grammars

A bottom-up parser generates a sequence of shifts and reductions to reduce the input string to the start symbol of the grammar. A deterministic parser must incorporate additional information into the process to select the correct alternative when more than one operation is possible. A grammar is LR(k) if a k-symbol lookahead provides sufficient information to make this selection. LR signifies that these strings are parsed in a left-to-right manner to construct a rightmost derivation.

All derivations in this chapter are rightmost. We also assume that grammars have a nonrecursive start symbol and that all the symbols in a grammar are useful.

17.1 LR(0) Contexts

A deterministic bottom-up parser attempts to reduce the input string to the start symbol of the grammar. Nondeterminism in bottom-up parsing is illustrated by examining reductions of the string *aabb* using the grammar

$$\begin{aligned} \text{G: } & S \rightarrow aAb \mid BaAa \\ & A \rightarrow ab \mid b \\ & B \rightarrow Bb \mid b. \end{aligned}$$

The parser scans the prefix *aab* before finding a reducible substring. The suffixes *b* and *ab* of *aab* both constitute the right-hand side of a rule of G. Three reductions of *aabb* can be obtained by replacing these substrings.

Rule	Reduction
$A \rightarrow b$	$aaAb$
$A \rightarrow ab$	aAb
$B \rightarrow b$	$aaBb$

The objective of a bottom-up parser is to repeatedly reduce the input string until the start symbol is obtained. Can a reduction of *aabb* initiated with the rule $A \rightarrow b$ eventually produce the start symbol? Equivalently, is *aaAb* a right sentential form of G? Rightmost derivations of the grammar G have the form

$$S \Rightarrow aAb \Rightarrow aabb$$

$$S \Rightarrow aAb \Rightarrow abb$$

$$S \Rightarrow BaAa \Rightarrow Baaba \overset{i}{\Rightarrow} Bb^iaaba \Rightarrow bb^iaaba \qquad i \geq 0$$

$$S \Rightarrow BaAa \Rightarrow Baba \overset{i}{\Rightarrow} Bb^iaba \Rightarrow bb^iaba \qquad i \geq 0.$$

Successful reductions of strings in L(G) can be obtained by "reversing the arrows" in the preceding derivations. Since the strings *aaAb* and *aaBb* do not occur in any of these derivations, a reduction of *aabb* initiated by the rule $A \rightarrow b$ or $B \rightarrow b$ cannot produce S. With this additional information, the parser need only reduce *aab* using the rule $A \rightarrow ab$.

Successful reductions were obtained by examining rightmost derivations of G. A parser that does not use lookahead must decide whether to perform a reduction with a rule $A \rightarrow w$ as soon as a string uw is scanned by the parser.

Definition 17.1.1

Let G = (V, Σ, P, S) be a context-free grammar. The string uw is an **LR(0) context** of a rule $A \rightarrow w$ if there is a derivation

$$S \underset{R}{\overset{*}{\Rightarrow}} uAv \underset{R}{\Rightarrow} uwv,$$

where $u \in (V \cup \Sigma)^*$ and $v \in \Sigma^*$. The set of LR(0) contexts of the rule $A \rightarrow w$ is denoted LR(0)-CONTEXT($A \rightarrow w$).

The LR(0) contexts of a rule $A \rightarrow w$ are obtained from the rightmost derivations that terminate with the application of the rule. In terms of reductions, uw is an LR(0) context of $A \rightarrow w$ if there is a reduction of a string uwv to S that begins by replacing w with A. If $uw \notin$ LR(0)-CONTEXT($A \rightarrow w$) then there is no sequence of reductions beginning with $A \rightarrow w$ that produces S from a string of the form uwv with $v \in \Sigma^*$. The LR(0) contexts, if known, can be used to eliminate reductions from consideration by the parser.

The parser need only reduce a string uw with the rule $A \to w$ when uw is an LR(0) context of $A \to w$.

The LR(0) contexts of the rules of G are constructed from the rightmost derivations of G. To determine the LR(0) contexts of $S \to aAb$ we consider all rightmost derivations that contain an application of the rule $S \to aAb$. The only two such derivations are

$$S \Rightarrow aAb \Rightarrow aabb$$
$$S \Rightarrow aAb \Rightarrow abb.$$

The only rightmost derivation terminating with the application of $S \to aAb$ is $S \Rightarrow aAb$. Thus LR(0)-CONTEXT$(S \to aAb) = \{aAb\}$.

The LR(0) contexts of $A \to ab$ are obtained from the rightmost derivations that terminate with an application of $A \to ab$. There are only two such derivations. The reduction is indicated by the arrow from ab to A. The context is the prefix of the sentential form up to and including the occurrence of ab that is reduced.

$$S \Rightarrow aAb \Rightarrow aabb$$

$$S \Rightarrow BaAa \Rightarrow Baaba$$

Consequently, the LR(0) contexts of $A \to ab$ are aab and $Baab$. In a similar manner we can obtain the LR(0) contexts for all the rules of G.

Rule	LR(0) Contexts
$S \to aAb$	$\{aAb\}$
$S \to BaAa$	$\{BaAa\}$
$A \to ab$	$\{aab, Baab\}$
$A \to b$	$\{ab, Bab\}$
$B \to Bb$	$\{Bb\}$
$B \to b$	$\{b\}$

Example 17.1.1

The LR(0) contexts are constructed for the rules of the grammar

$$S \to aA \mid bB$$
$$A \to abA \mid bB$$
$$B \to bBc \mid bc.$$

The rightmost derivations initiated by the rule $S \to aA$ have the form

$$S \Rightarrow aA \overset{i}{\Rightarrow} a(ab)^i A \Rightarrow a(ab)^i bB \overset{j}{\Rightarrow} a(ab)^i bb^j Bc^j \Rightarrow a(ab)^i bb^j bcc^j,$$

where $i, j \geq 0$. Derivations beginning with $S \rightarrow bB$ can be written

$$S \Rightarrow bB \overset{i}{\Rightarrow} bb^i Bc^i \Rightarrow bb^i bcc^i.$$

The LR(0) contexts can be obtained from the sentential forms generated in the preceding derivations.

Rule	LR(0) Contexts
$S \rightarrow aA$	$\{aA\}$
$S \rightarrow bB$	$\{bB\}$
$A \rightarrow abA$	$\{a(ab)^i A \mid i > 0\}$
$A \rightarrow bB$	$\{a(ab)^i bB \mid i \geq 0\}$
$B \rightarrow bBc$	$\{a(ab)^i bb^j Bc, bb^j Bc \mid i \geq 0, j > 0\}$
$B \rightarrow bc$	$\{a(ab)^i bb^j c, bb^j c \mid i \geq 0, j > 0\}$

□

The contexts can be used to eliminate reductions from consideration by the parser. When the LR(0) contexts provide sufficient information to eliminate all but one action, the grammar is called an LR(0) grammar.

Definition 17.1.2

A context-free grammar $G = (V, \Sigma, P, S)$ with nonrecursive start symbol S is **LR(0)** if, for every $u \in (V \cup \Sigma)^*$ and $v \in \Sigma^*$,

$$u \in \text{LR(0)-CONTEXT}(A \rightarrow w_1) \quad \text{and} \quad uv \in \text{LR(0)-CONTEXT}(B \rightarrow w_2)$$

implies $v = \lambda$, $A = B$, and $w_1 = w_2$.

The grammar from Example 17.1.1 is LR(0). Examining the table of LR(0) contexts we see that there is no LR(0) context of a rule that is a prefix of an LR(0) context of another rule.

The contexts of an LR(0) grammar provide the information needed to select the appropriate action. Upon scanning the string u, the parser takes one of three mutually exclusive actions:

1. If $u \in \text{LR(0)-CONTEXT}(A \rightarrow w)$, then u is reduced with the rule $A \rightarrow w$.
2. If u is not an LR(0) context but is a prefix of some LR(0) context, then the parser effects a shift.
3. If u is not the prefix of any LR(0) context, then the input string is rejected.

Since a string u is an LR(0) context for at most one rule $A \to w$, the first condition specifies a unique action. A string u is called a **viable prefix** if there is a string $v \in (V \cup \Sigma)^*$ such that uv is an LR(0) context. If u is a viable prefix and not an LR(0) context, a sequence of shift operations produces the LR(0) context uv.

Example 17.1.2

The grammar

$$\begin{aligned} \text{G: } S &\to aA \mid aB \\ A &\to aAb \mid b \\ B &\to bBa \mid b \end{aligned}$$

is not LR(0). The rightmost derivations of G have the form

$$S \Rightarrow aA \overset{i}{\Rightarrow} aa^iAb^i \Rightarrow aa^ibb^i$$

$$S \Rightarrow aB \overset{i}{\Rightarrow} ab^iBa^i \Rightarrow ab^iba^i$$

for $i \geq 0$. The LR(0) contexts for the rules of the grammar can be obtained from the right sentential forms in the preceding derivations.

Rule	LR(0) Contexts
$S \to aA$	$\{aA\}$
$S \to aB$	$\{aB\}$
$A \to aAb$	$\{aa^iAb \mid i > 0\}$
$A \to b$	$\{aa^ib \mid i \geq 0\}$
$B \to bBa$	$\{ab^iBa \mid i > 0\}$
$B \to b$	$\{ab^i \mid i > 0\}$

The grammar G is not LR(0) since ab is an LR(0) context of both $B \to b$ and $A \to b$. □

17.2 An LR(0) Parser

Incorporating the information provided by the LR(0) contexts of the rules of an LR(0) grammar into a bottom-up parser produces a deterministic algorithm. The input string p is scanned in a left-to-right manner. The action of the parser in Algorithm 17.2.1 is determined by comparing the LR(0) contexts with the string scanned. The string u is the prefix of the sentential form scanned by the parser, and v is the remainder of the input string. The operation $shift(u, v)$ removes the first symbol from v and concatenates it to the right end of u.

Algorithm 17.2.1
Parser for an LR(0) Grammar

input: LR(0) grammar G = (V, Σ, P, *S*)
string $p \in \Sigma^*$

1. $u := \lambda, v := p$
2. dead-end := *false*
3. **repeat**
   3.1. **if** $u \in$ LR(0)-CONTEXT($A \rightarrow w$) for the rule $A \rightarrow w$ in P
   where $u = xw$ **then** $u := xA$
   else if u is a viable prefix **and** $v \neq \lambda$ **then** *shift*(u, v)
   else dead-end := *true*
   until $u = S$ **or** dead-end
4. **if** $u = S$ **then** accept **else** reject

The decision to reduce with the rule $A \rightarrow w$ is made as soon as a substring $u = xw$ is encountered. The decision does not use any information contained in v, the unscanned portion of the string. The parser does not look beyond the string xw, hence the zero in LR(0) indicating no lookahead is required.

One detail has been overlooked in Algorithm 17.2.1. No technique has been provided for deciding whether a string is a viable prefix or an LR(0) context of a rule of the grammar. This omission is remedied in the next section.

Example 17.2.1

The string *aabbbbcc* is parsed using the rules and LR(0) contexts of the grammar presented in Example 17.1.1 and the parsing algorithm for LR(0) grammars.

u	*v*	Rule	Action
λ	*aabbbbcc*		shift
a	*abbbbcc*		shift
aa	*bbbbcc*		shift
aab	*bbbcc*		shift
aabb	*bbcc*		shift
aabbb	*bcc*		shift
aabbbb	*cc*		shift
aabbbbc	*c*	$B \rightarrow bc$	reduce
aabbbB	*c*		shift
aabbbBc	λ	$B \rightarrow bBc$	reduce

aabbB	λ	$A \rightarrow bB$	reduce
aabA	λ	$A \rightarrow abA$	reduce
aA	λ	$S \rightarrow aA$	reduce
S			

□

17.3 The LR(0) Machine

A rule may contain infinitely many LR(0) contexts. Moreover, there is no upper bound on the length of the contexts. These properties make it impossible to generate the complete set of LR(0) contexts for an arbitrary context-free grammar. The problem of dealing with infinite sets was avoided in LL(k) grammars by restricting the length of the lookahead strings. Unfortunately, the decision to reduce a string requires knowledge of the entire scanned string (the context). The LR(0) grammars G_1 and G_2 demonstrate this dependence.

The LR(0) contexts of the rules $A \rightarrow aAb$ and $A \rightarrow ab$ of G_1 form disjoint sets that satisfy the prefix conditions. If these sets are truncated at any length k, the string a^k will be an element of both of the truncated sets. The final two symbols of the context are required to discriminate between these reductions.

Rule		LR(0) Contexts
G_1:	$S \rightarrow A$	$\{A\}$
	$A \rightarrow aAa$	$\{a^iAa \mid i > 0\}$
	$A \rightarrow aAB$	$\{a^iAb \mid i > 0\}$
	$A \rightarrow ab$	$\{a^ib \mid i > 0\}$

One may be tempted to consider only fixed-length suffixes of contexts, since a reduction alters the suffix of the scanned string. The grammar G_2 exhibits the futility of this approach.

Rule		LR(0) Contexts
G_2:	$S \rightarrow A$	$\{A\}$
	$S \rightarrow bB$	$\{bB\}$
	$A \rightarrow aA$	$\{a^iA \mid i > 0\}$
	$A \rightarrow ab$	$\{a^ib \mid i > 0\}$
	$B \rightarrow aB$	$\{ba^iB \mid i > 0\}$
	$B \rightarrow ab$	$\{ba^ib \mid i > 0\}$

The sole difference between the LR(0) contexts of $A \rightarrow ab$ and $B \rightarrow ab$ is the first element of the string. A parser will be unable to discriminate between these rules if the selection process uses only fixed-length suffixes of the LR(0) contexts.

The grammars G_1 and G_2 demonstrate that the entire scanned string is required by the LR(0) parser to select the appropriate action. This does not imply that the complete set of LR(0) contexts is required. For a given grammar, a finite automaton can be constructed whose computations determine whether a string is a viable prefix of the grammar. The states of the machine are constructed from the rules of the grammar.

Definition 17.3.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. The **LR(0) items** of G are defined as follows:

i) If $A \rightarrow uv \in P$, then $A \rightarrow u.v$ is an LR(0) item.

ii) If $A \rightarrow \lambda \in P$, then $A \rightarrow .$ is an LR(0) item.

The LR(0) items are obtained from the rules of the grammar by placing the marker . in the right-hand side of a rule. An item $A \rightarrow u.$ is called a **complete item**. A rule whose right-hand side has length n generates $n + 1$ items, one for each possible position of the marker.

Definition 17.3.2

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. The **nondeterministic LR(0) machine** of G is an NFA-λ $M = (Q, V \cup \Sigma, \delta, q_0, Q)$, where Q is the set of LR(0) items augmented with the state q_0. The transition function is defined by

i) $\delta(q_0, \lambda) = \{S \rightarrow .w \mid S \rightarrow w \in P\}$

ii) $\delta(A \rightarrow u.av, a) = \{A \rightarrow ua.v\}$

iii) $\delta(A \rightarrow u.Bv, B) = \{A \rightarrow uB.v\}$

iv) $\delta(A \rightarrow u.Bv, \lambda) = \{B \rightarrow .w \mid B \rightarrow w \in P\}$.

The computations of the nondeterministic LR(0) machine M of a grammar G completely process strings that are viable prefixes of the grammar. All other computations halt prior to reading the entire input. Since all the states of M are accepting, M accepts precisely the viable prefixes of the original grammar. A computation of M records the progress made toward matching the right-hand side of a rule of G. The item $A \rightarrow u.v$ indicates that the string u has been scanned and the automaton is looking for the string v to complete the match.

The symbol following the marker in an item defines the arcs leaving a node. If the marker precedes a terminal, the only arc leaving the node is labeled by that terminal. Arcs labeled B or λ may leave a node containing an item of the form $A \rightarrow u.Bv$. To extend the match of the right-hand side of the rule, the machine is looking for a B. The node $A \rightarrow uB.v$ is entered if the parser reads B. It is also looking for strings that may produce

B. The variable B may be obtained by reduction using a B rule. Consequently, the parser is also looking for the right-hand side of a B rule. This is indicated by lambda transitions to the items $B \rightarrow .w$.

Definition 17.3.2, the LR(0) items, and the LR(0) contexts of the rules of the grammar G given below are used to demonstrate the recognition of viable prefixes by the associated NFA-λ.

Rule	LR(0) Items	LR(0) Contexts
$S \rightarrow AB$	$S \rightarrow .AB$	$\{AB\}$
	$S \rightarrow A.B$	
	$S \rightarrow AB.$	
$A \rightarrow Aa$	$A \rightarrow .Aa$	$\{Aa\}$
	$A \rightarrow A.a$	
	$A \rightarrow Aa.$	
$A \rightarrow a$	$A \rightarrow .a$	$\{a\}$
	$A \rightarrow a.$	
$B \rightarrow bBa$	$B \rightarrow .bBa$	$\{Ab^iBa \mid i > 0\}$
	$B \rightarrow b.Ba$	
	$B \rightarrow bB.a$	
	$B \rightarrow bBa.$	
$B \rightarrow ba$	$B \rightarrow .ba$	$\{Ab^iba \mid i \geq 0\}$
	$B \rightarrow b.a$	
	$B \rightarrow ba.$	

The NFA-λ in Figure 17.1 is the LR(0) machine of the grammar G. A string w is a prefix of a context of the rule $A \rightarrow uv$ if $A \rightarrow u.v \in \hat{\delta}(q_0, w)$. The computation $\hat{\delta}(q_0, A)$ of the LR(0) machine in Figure 17.1 halts in the states containing the items $A \rightarrow A.a$, $S \rightarrow A.B$, $B \rightarrow .bBa$, and $B \rightarrow .ba$. These are precisely the rules that have LR(0) contexts beginning with A. Similarly, the computation with input AbB indicates that AbB is a viable prefix of the rule $B \rightarrow bBa$ and no other.

The techniques presented in Chapter 6 can be used to construct an equivalent DFA from the nondeterministic LR(0) machine of G. This machine, the **deterministic LR(0) machine** of G, is given in Figure 17.2. The start state q_s of the deterministic machine is the lambda closure of the q_0, the start state of the nondeterministic machine. The state that represents failure, the empty set, has been omitted. When the computation obtained by processing the string u successfully terminates, u is an LR(0) context or a viable prefix. Algorithm 17.3.3 incorporates the LR(0) machine into the LR(0) parsing strategy.

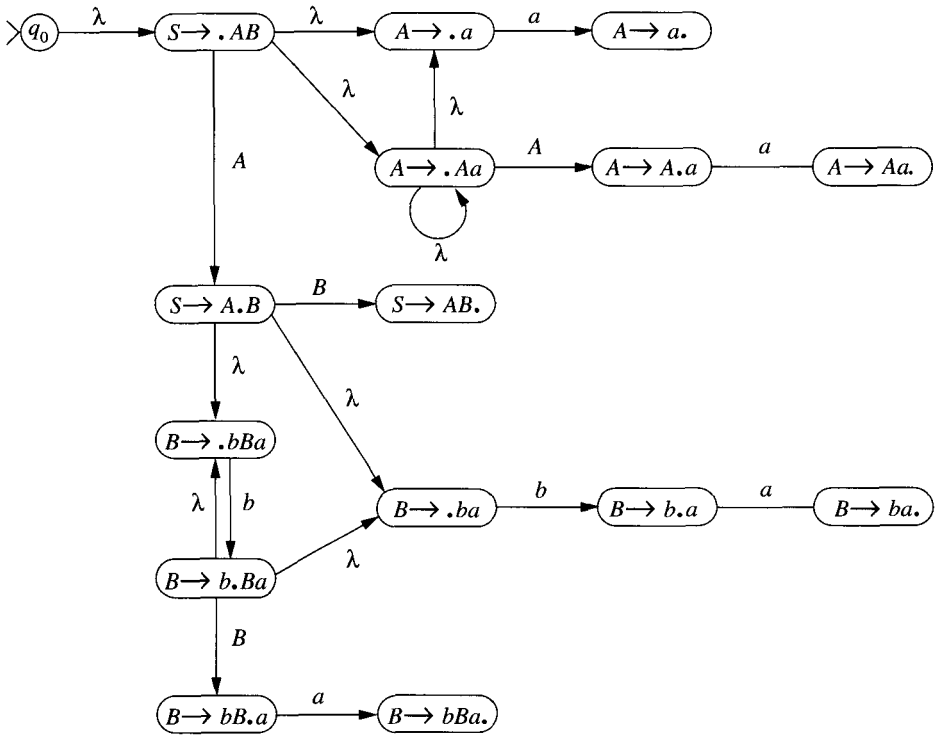


FIGURE 17.1 Nondeterministic LR(0) machine of G.

Algorithm 17.3.3
Parser Utilizing the Deterministic LR(0) Machine

input: LR(0) grammar G = (V, Σ, P, S)
string $p \in \Sigma^*$
deterministic LR(0) machine of G

1. $u := \lambda, v := p$
2. dead-end := *false*
3. **repeat**
   3.1. **if** $\hat{\delta}(q_s, u)$ contains $A \rightarrow w.$ where $u = xw$ **then** $u := xA$
   else if $\hat{\delta}(q_s, u)$ contains an item $A \rightarrow y.z$ and $v \neq \lambda$ **then** *shift*(u, v)
   else dead-end := *true*
   until $u = S$ **or** dead-end
4. **if** $u = S$ **then** accept **else** reject

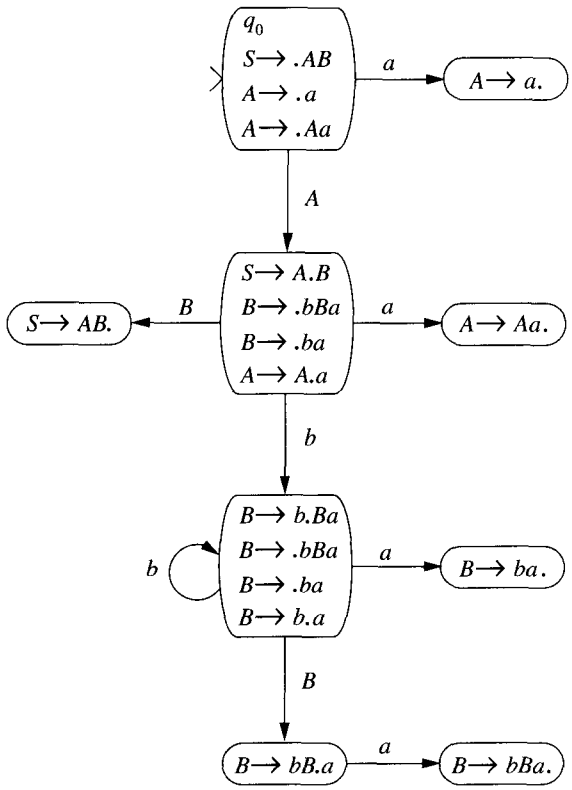


FIGURE 17.2 Deterministic LR(0) machine of G.

Example 17.3.1

The string *aabbaa* is parsed using Algorithm 17.3.3 and the deterministic LR(0) machine in Figure 17.2. Upon processing the leading a, the machine enters the state $A \rightarrow a.$, specifying a reduction using the rule $A \rightarrow a$. Since $\hat{\delta}(q_s, A)$ does not contain a complete item, the parser shifts and constructs the string Aa. The computation $\hat{\delta}(q_s, Aa) = \{A \rightarrow Aa.\}$ indicates that Aa is an LR(0) context of $A \rightarrow Aa$ and that it is not a prefix of a context of any other rule. Having generated a complete item, the parser reduces the string using the rule $A \rightarrow Aa$. The shift and reduction cycle continues until the sentential form is reduced to the start symbol S.

u	v	Computation		Action
λ	$aabbaa$	$\hat{\delta}(q_s, \lambda) =$	$\{S \to .AB,$	shift
			$A \to .a,$	
			$A \to .Aa\}$	
a	$abbaa$	$\hat{\delta}(q_s, a) =$	$\{A \to a.\}$	reduce
A	$abbaa$	$\hat{\delta}(q_s, A) =$	$\{A \to A.a,$	shift
			$S \to A.B,$	
			$B \to .bBa,$	
			$B \to .ba\}$	
Aa	$bbaa$	$\hat{\delta}(q_s, Aa) =$	$\{A \to Aa.\}$	reduce
A	$bbaa$	$\hat{\delta}(q_s, A) =$	$\{A \to A.a,$	shift
			$S \to A.B,$	
			$B \to .bBa,$	
			$B \to .ba\}$	
Ab	baa	$\hat{\delta}(q_s, Ab) =$	$\{B \to .bBa,$	shift
			$B \to b.Ba,$	
			$B \to .ba,$	
			$B \to b.a\}$	
Abb	aa	$\hat{\delta}(q_s, Abb) =$	$\{B \to .bBa,$	shift
			$B \to b.Ba,$	
			$B \to .ba,$	
			$B \to b.a\}$	
$Abba$	a	$\hat{\delta}(q_s, Abba) =$	$\{B \to ba.\}$	reduce
AbB	a	$\hat{\delta}(q_s, AbB) =$	$\{B \to bB.a\}$	shift
$AbBa$	λ	$\hat{\delta}(q_s, AbBa) =$	$\{B \to bBa.\}$	reduce
AB	λ	$\hat{\delta}(q_s, AB) =$	$\{S \to AB.\}$	reduce
S				

□

17.4 Acceptance by the LR(0) Machine

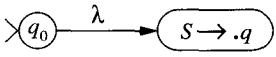
The LR(0) machine has been constructed to decide whether a string is a viable prefix of the grammar. Theorem 17.4.1 establishes that computations of the LR(0) machine provide the desired information.

Theorem 17.4.1

Let G be a context-free grammar and M the nondeterministic LR(0) machine of G. The LR(0) item $A \to u.v$ is in $\hat{\delta}(q_0, w)$ if, and only if, $w = pu$, where puv is an LR(0) context of $A \to uv$.

Proof Let $A \to u.v$ be an element of $\hat{\delta}(q_0, w)$. We prove, by induction on the number of transitions in the computation $\hat{\delta}(q_0, w)$, that wv is an LR(0) context of $A \to uv$.

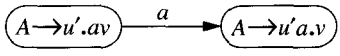
The basis consists of computations of length one. All such computations have the form



where $S \to q$ is a rule of the grammar. These computations process the input string $w = \lambda$. Setting $p = \lambda$, $u = \lambda$, and $v = q$ gives the desired decomposition of w.

Now let $\hat{\delta}(q_0, w)$ be a computation of length $k > 1$ with $A \to u.v$ in $\hat{\delta}(q_0, w)$. Isolating the final transition, we can write this computation as $\delta(\hat{\delta}(q_0, y), x)$, where $w = yx$ and $x \in V \cup \Sigma \cup \{\lambda\}$. The remainder of the proof is divided into three cases.

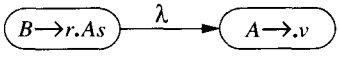
Case 1: $x = a \in \Sigma$: Then $u = u'a$. The final transition of the computation has the form



By the inductive hypothesis, $yu'av = wv$ is an LR(0) context of $A \to uv$.

Case 2: $x \in V$: The proof is similar to that of Case 1.

Case 3: $x = \lambda$: Then $y = w$ and the computation terminates at an item $A \to .v$. The final transition has the form



The inductive hypothesis implies that w can be written $w = pr$, where $prAs$ is an LR(0) context of $B \to rAs$. Thus there is a rightmost derivation

$$S \underset{R}{\overset{*}{\Rightarrow}} pBq \underset{R}{\Rightarrow} prAsq.$$

The application of $A \to v$ yields

$$S \underset{R}{\overset{*}{\Rightarrow}} pBq \underset{R}{\Rightarrow} prAsq \underset{R}{\Rightarrow} prvsq.$$

The final step of this derivation shows that $prv = wv$ is an LR(0) context of $A \to v$.

To establish the opposite implication we must show that $\hat{\delta}(q_0, pu)$ contains the item $A \to u.v$ whenever puv is an LR(0) context of a rule $A \to uv$. First we note that if $\hat{\delta}(q_0, p)$ contains $A \to .uv$, then $\hat{\delta}(q_0, pu)$ contains $A \to u.v$. This follows immediately from conditions (ii) and (iii) of Definition 17.3.2.

Since puv is an LR(0) context of $A \to uv$, there is a derivation

$$S \underset{R}{\overset{*}{\Rightarrow}} pAq \underset{R}{\Rightarrow} puvq.$$

We prove, by induction on the length of the derivation $S \underset{R}{\overset{*}{\Rightarrow}} pAq$, that $\hat{\delta}(q_0, p)$ contains $A \to .uv$. The basis consists of derivations $S \Rightarrow pAq$ of length one. The desired computation consists of traversing the lambda arc to $S \to .pAq$ followed by the arcs that process the string p. The computation is completed by following the lambda arc from $S \to p.Aq$ to $A \to .uv$.

Now consider a derivation in which the variable A is introduced on the kth rule application. A derivation of this form can be written

$$S \underset{R}{\overset{k-1}{\Longrightarrow}} xBy \underset{R}{\Rightarrow} xwAzy.$$

The inductive hypothesis asserts that $\hat{\delta}(q_0, x)$ contains the item $B \to .wAz$. Hence $B \to w.Az \in \hat{\delta}(q_0, xw)$. The lambda transition to $A \to .uv$ completes the computation. ■

The relationships in Lemma 17.4.2 between derivations in a context-free grammar and the items in the nodes of the deterministic LR(0) machine of the grammar follow from Theorem 17.4.1. The proof of Lemma 17.4.2 is left as an exercise. Recall that q_s is the start symbol of the deterministic machine.

Lemma 17.4.2

Let M be the deterministic LR(0) machine of a context-free grammar G. Assume $\hat{\delta}(q_s, w)$ contains an item $A \to u.Bv$.

i) If $B \overset{*}{\Rightarrow} \lambda$, then $\hat{\delta}(q_s, w)$ contains an item of the form $C \to .$ for some variable $C \in \mathrm{V}$.

ii) If $B \overset{*}{\Rightarrow} x \in \Sigma^+$, then there is an arc labeled by a terminal symbol leaving the node $\hat{\delta}(q_s, w)$ or $\hat{\delta}(q_s, w)$ contains an item of the form $C \to .$ for some variable $C \in \mathrm{V}$.

Lemma 17.4.3

Let M be the deterministic LR(0) machine of an LR(0) grammar G. Assume $\hat{\delta}(q_s, u)$ contains the complete item $A \to w.$. Then $\hat{\delta}(q_s, ua)$ is undefined for all terminal symbols $a \in \Sigma$.

Proof By Theorem 17.4.1, u is an LR(0) context of $A \to w$. Assume that $\hat{\delta}(q_s, ua)$ is defined for some terminal a. Then ua is a prefix of an LR(0) context of some rule $B \to y$. This implies that there is a derivation

$$S \underset{R}{\overset{*}{\Rightarrow}} pBv \Rightarrow pyv = uazv$$

with $z \in (V \cup \Sigma)^*$ and $v \in \Sigma^*$. Consider the possibilities for the string z. If $z \in \Sigma^*$, then uaz is an LR(0) context of the rule $B \to y$. If z is not a terminal string, then there is a terminal string derivable from z

$$z \underset{R}{\overset{*}{\Rightarrow}} rCs \Rightarrow rts \qquad r, s, t \in \Sigma^*,$$

where $C \to t$ is the final rule application in the derivation of the terminal string from z. Combining the derivations from S and z shows that $uart$ is an LR(0) context of $C \to t$. In either case, u is an LR(0) context and ua is a viable prefix. This contradicts the assumption that G is LR(0). ■

The previous results can be combined with Definition 17.1.2 to obtain a characterization of LR(0) grammars in terms of the structure of the deterministic LR(0) machine.

Theorem 17.4.4

Let G be a context-free grammar with a nonrecursive start symbol. G is LR(0) if, and only if, the extended transition function $\hat{\delta}$ of the deterministic LR(0) machine of G satisfies the following conditions:

i) If $\hat{\delta}(q_s, u)$ contains a complete item $A \to w.$ with $w \neq \lambda$, then $\hat{\delta}(q_s, u)$ contains no other items.

ii) If $\hat{\delta}(q_s, u)$ contains a complete item $A \to .$, then the marker is followed by a variable in all other items in $\hat{\delta}(q_s, u)$.

Proof First we show that a grammar G with nonrecursive start symbol is LR(0) when the extended transition function satisfies conditions (i) and (ii). Let u be an LR(0) context of the rule $A \to w$. Then $\hat{\delta}(q_s, uv)$ is defined only when v begins with a variable. Thus, for all strings $v \in \Sigma^*$, $uv \in \text{LR}(0) - \text{CONTEXT}(B \to x)$ implies $v = \lambda$, $B = A$, and $w = x$.

Conversely, let G be an LR(0) grammar and u an LR(0) context of the rule $A \to w$. By Theorem 17.4.1, $\hat{\delta}(q_s, u)$ contains the complete item $A \to w.$. The state $\hat{\delta}(q_s, u)$ does not contain any other complete items $B \to v.$ since this would imply that u is also an LR(0) context of $B \to v$. By Lemma 17.4.3, all arcs leaving $\hat{\delta}(q_s, u)$ must be labeled by variables.

Now assume that $\hat{\delta}(q_s, u)$ contains a complete item $A \to w.$ where $w \neq \lambda$. By Lemma 17.4.2, if there is an arc labeled by a variable with tail $\hat{\delta}(q_s, u)$, then $\hat{\delta}(q_s, u)$ contains a complete item $C \to .$ or $\hat{\delta}(q_s, u)$ has an arc labeled by a terminal leaving it. In the former case, u is an LR(0) context of both $A \to w$ and $C \to \lambda$, contradicting the assumption that G is LR(0). The latter possibility contradicts Lemma 17.4.3. Thus $A \to w.$ is the only item in $\hat{\delta}(q_s, u)$. ■

Intuitively we would like to say that a grammar is LR(0) if a state containing a complete item contains no other items. This condition is satisfied by all states containing complete items generated by nonnull rules. The previous theorem permits a state containing $A \to .$ to contain items in which the marker is followed by a variable. Consider the derivation using the rules $S \to aABc$, $A \to \lambda$, and $B \to b$.

$$S \underset{R}{\Rightarrow} aABc \underset{R}{\Rightarrow} aAbc \underset{R}{\Rightarrow} abc$$

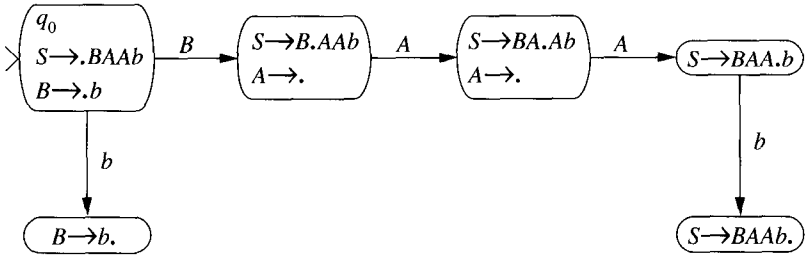
The string a is an LR(0) context of $A \rightarrow \lambda$ and a prefix of aAb, which is an LR(0) context of $B \rightarrow b$. The effect of reductions by lambda rules in an LR(0) parser is demonstrated in Example 17.4.1.

Example 17.4.1

The deterministic LR(0) machine for the grammar

$$\begin{aligned} \text{G: } & S \rightarrow BAAb \\ & A \rightarrow \lambda \\ & B \rightarrow b \end{aligned}$$

is given below. The analysis of the string bb is traced using the computations of the machine to specify the actions of the parser.



u	v	Computation		Action
λ	bb	$\hat{\delta}(q_s, \lambda) =$	$\{S \rightarrow .BAAb$ $B \rightarrow .b\}$	shift
b	b	$\hat{\delta}(q_s, b) =$	$\{B \rightarrow b.\}$	reduce
B	b	$\hat{\delta}(q_s, B) =$	$\{S \rightarrow B.AAb$ $A \rightarrow .\}$	reduce
BA	b	$\hat{\delta}(q_s, BA) =$	$\{S \rightarrow BA.Ab$ $A \rightarrow .\}$	reduce
BAA	b	$\hat{\delta}(q_s, BAA) =$	$\{S \rightarrow BAA.b\}$	shift
$BAAb$	λ	$\hat{\delta}(q_s, BAAb) =$	$\{S \rightarrow BAAb.\}$	reduce
S				

The parser reduces the sentential form with the rule $A \rightarrow \lambda$ whenever the LR(0) machine halts in a state containing the complete item $A \rightarrow .$. This reduction adds an A to the end of the currently scanned string. In the next iteration, the LR(0) machine follows

the arc labeled A to the subsequent state. An A is generated by a lambda reduction only when its presence adds to the prefix of an item being recognized. □

Theorem 17.4.4 establishes a procedure for deciding whether a grammar is LR(0). The process begins by constructing the deterministic LR(0) machine of the grammar. A grammar with a nonrecursive start symbol is LR(0) if the restrictions imposed by conditions (ii) and (iii) of Theorem 17.4.4 are satisfied by the LR(0) machine.

Example 17.4.2

The grammar AE augmented with the endmarker #

$$\begin{aligned} \text{AE: } & S \rightarrow A\# \\ & A \rightarrow A + T \mid T \\ & T \rightarrow b \mid (A) \end{aligned}$$

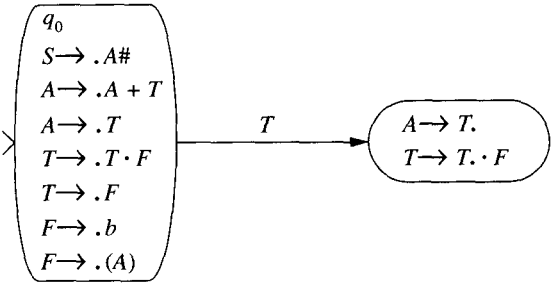
is LR(0). The deterministic LR(0) machine of AE is given in Figure 17.3. Since each of the states containing a complete item is a singleton set, the grammar is LR(0). □

Example 17.4.3

The grammar

$$\begin{aligned} & S \rightarrow A\# \\ & A \rightarrow A + T \mid T \\ & T \rightarrow F \cdot T \mid F \\ & F \rightarrow b \mid (A) \end{aligned}$$

is not LR(0). This grammar is obtained by adding the variable F (factor) to AE to generate multiplicative subexpressions. We show that this grammar is not LR(0) by constructing two states of the deterministic LR(0) machine.



The computation generated by processing T contains the complete item $A \rightarrow T.$ and the item $T \rightarrow T. \cdot F$. When the parser scans the string T, there are two possible courses of action: Reduce using $A \rightarrow T$ or shift in an attempt to construct the string $T \cdot F$. □

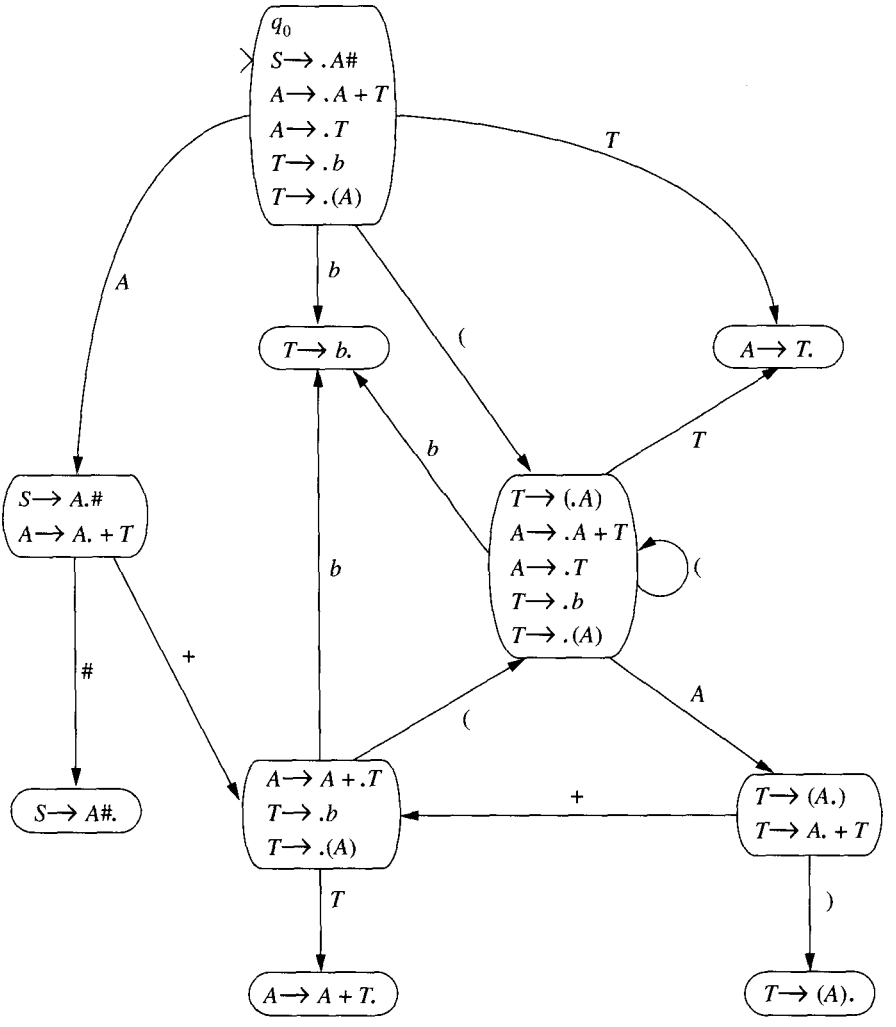


FIGURE 17.3 Deterministic LR(0) machine of AE with end marker.

17.5 LR(1) Grammars

The LR(0) conditions are generally too restrictive to construct grammars that define programming languages. In this section the LR parser is modified to utilize information obtained by looking beyond the substring that matches the right-hand side of the rule. The lookahead is limited to a single symbol. The definitions and algorithms, with obvious modifications, can be extended to utilize a lookahead of arbitrary length.

A grammar in which strings can be deterministically parsed using a one-symbol lookahead is called LR(1). The lookahead symbol is the symbol to the immediate right of the substring to be reduced by the parser. The decision to reduce with the rule $A \to w$ is made upon scanning a string of the form uwz, where $z \in \Sigma \cup \{\lambda\}$. Following the example of LR(0) grammars, a string uwz is called an **LR(1) context** if there is a derivation

$$S \underset{R}{\overset{*}{\Rightarrow}} uAv \underset{R}{\Rightarrow} uwv,$$

where z is the first symbol of v or the null string if $v = \lambda$. Since the derivation constructed by a bottom-up parser is rightmost, the lookahead symbol z is either a terminal symbol or the null string.

The role of the lookahead symbol in reducing the number of possibilities that must be examined by the parser is demonstrated by considering reductions in the grammar

$$\begin{aligned} \text{G: } S &\to A \mid Bc \\ A &\to aA \mid a \\ B &\to a \mid ab. \end{aligned}$$

When an LR(0) parser reads the symbol a, there are three possible actions:

i) Reduce with $A \to a$.

ii) Reduce with $B \to a$.

iii) Shift to obtain either aA or ab.

One-symbol lookahead is sufficient to determine the appropriate operation. The symbol underlined in each of the following derivations is the lookahead symbol when the initial a is scanned by the parser.

$$\begin{array}{llll} S \Rightarrow A & S \Rightarrow A & S \Rightarrow Bc & S \Rightarrow Bc \\ \Rightarrow \underline{a} & \Rightarrow a\underline{A} & \Rightarrow a\underline{c} & \Rightarrow a\underline{b}c \\ & \Rightarrow aa\underline{A} & & \\ & \Rightarrow a\underline{a}a & & \end{array}$$

In the preceding grammar, the action of the parser when reading an a is completely determined by the lookahead symbol.

String Scanned	Lookahead Symbol	Action
a	λ	reduce with $A \to a$
a	a	shift
a	b	shift
a	c	reduce with $B \to a$

The action of an LR(0) parser is determined by the result of a computation of the LR(0) machine of the grammar. An LR(1) parser incorporates the lookahead symbol into the decision procedure. An LR(1) item is an ordered pair consisting of an LR(0) item and a set containing the possible lookahead symbols.

Definition 17.5.1

Let G = (V, Σ, P, S) be a context-free grammar. The **LR(1) items** of G have the form

$$[A \rightarrow u.v, \{z_1, z_2, \ldots, z_n\}],$$

where $A \rightarrow uv \in P$ and $z_i \in \Sigma \cup \{\lambda\}$. The set $\{z_1, z_2, \ldots, z_n\}$ is the lookahead set of the LR(1) item.

The lookahead set of an item $[A \rightarrow u.v, \{z_1, \ldots, z_n\}]$ consists of the first symbol in the terminal strings y that follow uv in rightmost derivations.

$$S \underset{R}{\overset{*}{\Rightarrow}} xAy \underset{R}{\Rightarrow} xuvy$$

Since the S rules are nonrecursive, the only derivation terminated by a rule $S \rightarrow w$ is the derivation $S \Rightarrow w$. The null string follows w in this derivation. Consequently, the lookahead set of an S rule is always the singleton set $\{\lambda\}$.

As before, a complete item is an item in which the marker follows the entire right-hand side of the rule. The LR(1) machine, which specifies the actions of an LR(1) parser, is constructed from the LR(1) items of the grammar.

Definition 17.5.2

Let G = (V, Σ, P, S) be a context-free grammar. The **nondeterministic LR(1) machine** of G is an NFA-λ M = (Q, V ∪ Σ, δ, q_0, Q), where Q is a set of LR(1) items augmented with the state q_0. The transition function is defined by

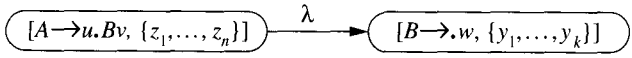
i) $\delta(q_0, \lambda) = [\{S \rightarrow .w, \{\lambda\}] \mid S \rightarrow w \in P\}$

ii) $\delta([A \rightarrow u.Bv, \{z_1, \ldots, z_n\}], B) = \{[A \rightarrow uB.v, \{z_1, \ldots, z_n\}]\}$

iii) $\delta([A \rightarrow u.av, \{z_1, \ldots, z_n\}], a) = \{[A \rightarrow ua.v, \{z_1, \ldots, z_n\}]\}$

iv) $\delta([A \rightarrow u.Bv, \{z_1, \ldots, z_n\}], \lambda) = \{[B \rightarrow .w, \{y_1, \ldots, y_k\}] \mid B \rightarrow w \in P$ where $y_i \in \text{FIRST}_1(vz_j)$ for some $j\}$.

If we disregard the lookahead sets, the transitions of the LR(1) machine defined in (i), (ii), and (iii) have the same form as those of the LR(0) machine. The LR(1) item $[A \rightarrow u.v, \{z_1, \ldots, z_n\}]$ indicates that the parser has scanned the string u and is attempting to find v to complete the match of the right-hand side of the rule. The transitions generated by conditions (ii) and (iii) represent intermediate steps in matching the right-hand side of a rule and do not alter the lookahead set. Condition (iv) introduces transitions of the form



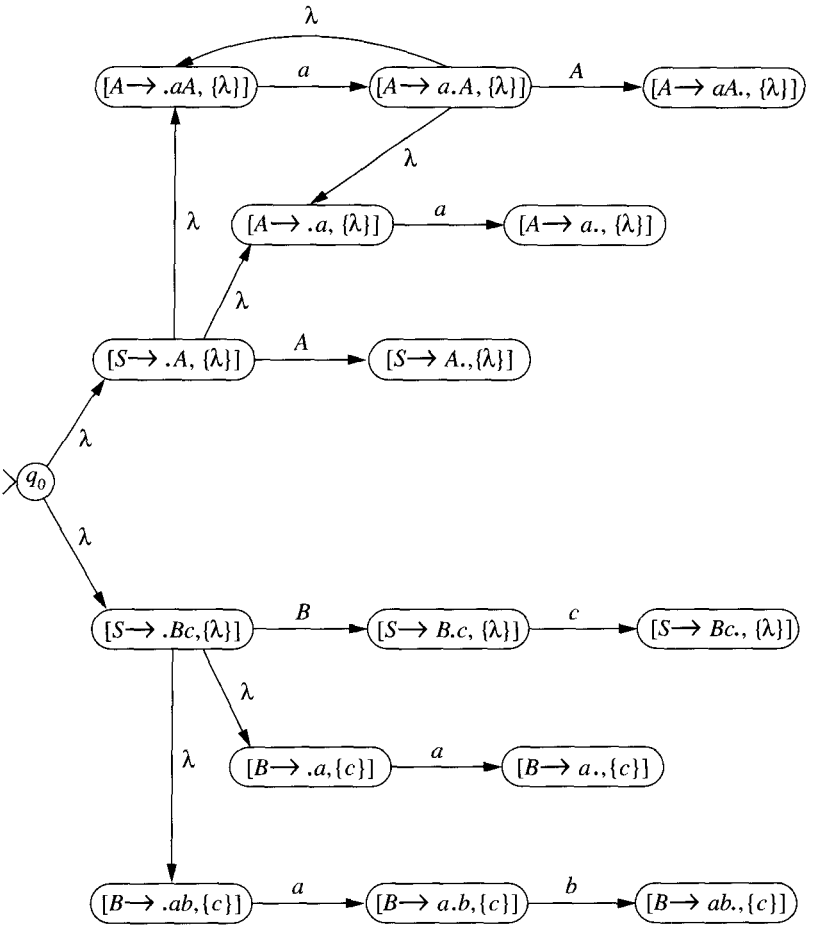


FIGURE 17.4 Nondeterministic LR(1) machine of G.

Following this arc, the LR(1) machine attempts to match the right-hand side of the rule $B \rightarrow w$. If the string w is found, a reduction of uwv produces $uB.v$, as desired. The lookahead set consists of the symbols that follow w, that is, the first terminal symbol in strings derived from v and the lookahead set $\{z_1, \ldots, z_n\}$ if $v \stackrel{*}{\Rightarrow} \lambda$.

A bottom-up parser may reduce the string uw to uA whenever $A \rightarrow w$ is a rule of the grammar. An LR(1) parser uses the lookahead set to decide whether to reduce or to shift when this occurs. If $\delta(q_0, uw)$ contains a complete item $[A \rightarrow w., \{z_1, \ldots, z_n\}]$, the string is reduced only if the lookahead symbol is in the set $\{z_1, \ldots, z_n\}$.

The state diagrams of the nondeterministic and deterministic LR(1) machines of the grammar G are given in Figures 17.4 and 17.5, respectively.

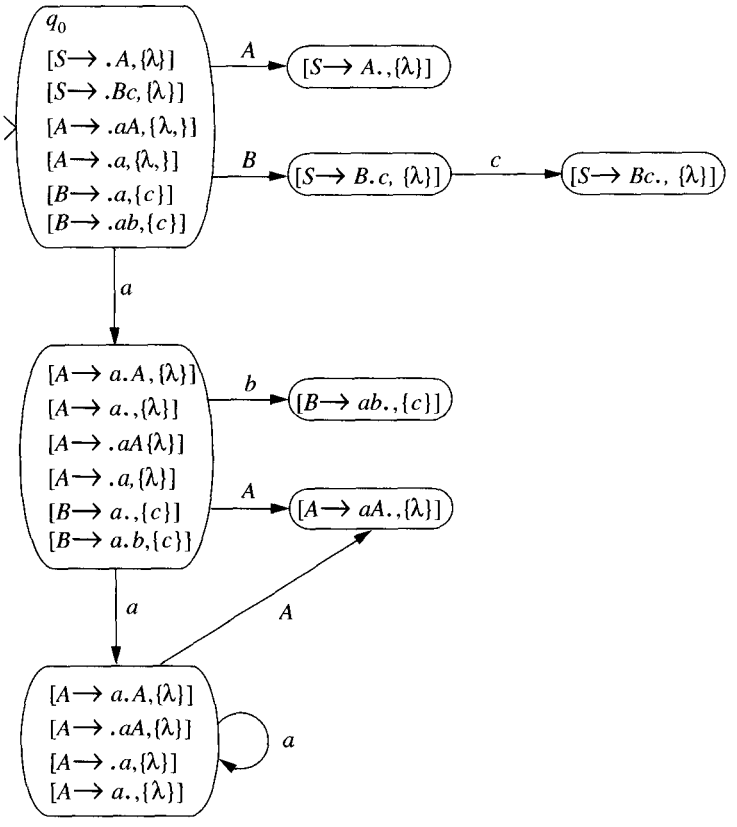


FIGURE 17.5 Deterministic LR(1) machine of G.

$$\begin{aligned} \text{G: } S &\to A \mid Bc \\ A &\to aA \mid a \\ B &\to a \mid ab \end{aligned}$$

A grammar is LR(1) if the actions of the parser are uniquely determined using a single lookahead symbol. The structure of the deterministic LR(1) machine can be used to define the LR(1) grammars.

Definition 17.5.3

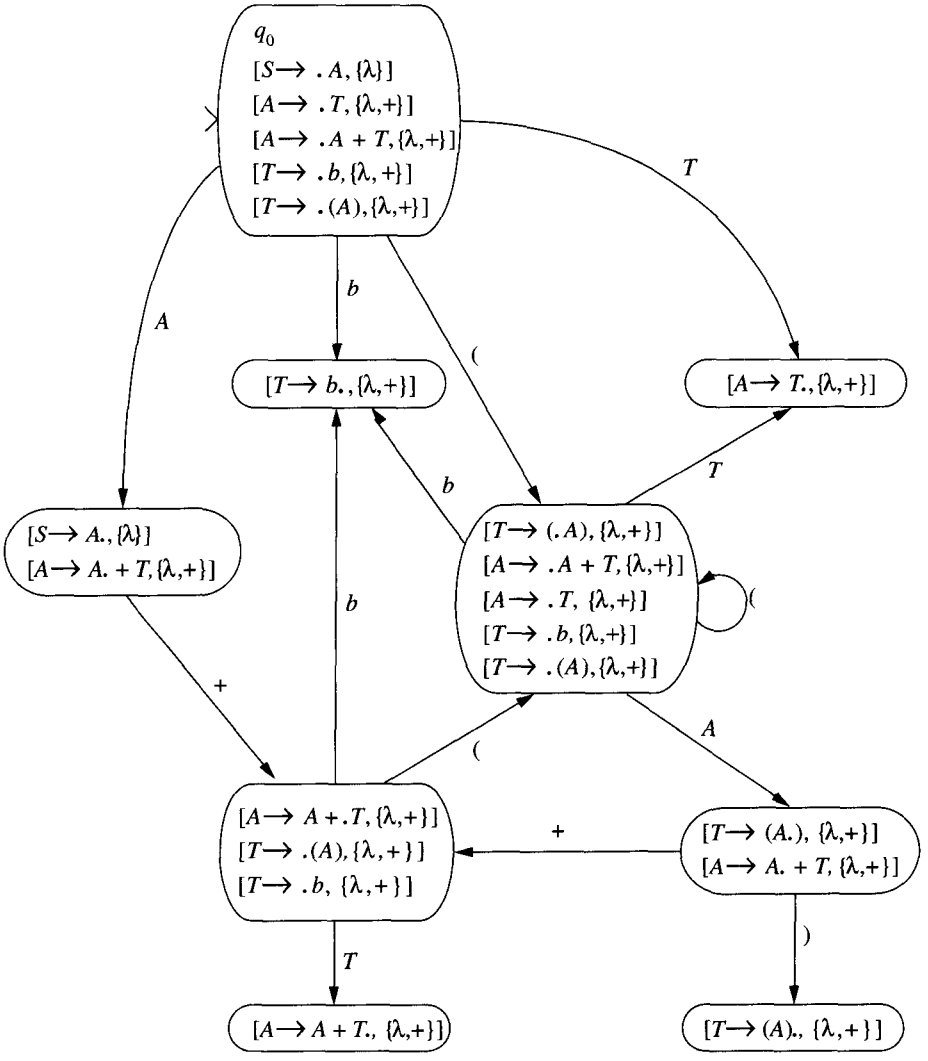
Let G be a context-free grammar with a nonrecursive start symbol. The grammar G is **LR(1)** if the extended transition function $\hat{\delta}$ of the deterministic LR(1) machine of G satisfies the following conditions:

i) If $\hat{\delta}(q_s, u)$ contains a complete item $[A \to w., \{z_1, \ldots, z_n\}]$ and $\hat{\delta}(q_s, u)$ contains an item $[B \to r.as, \{y_1, \ldots, y_k\}]$, then $a \neq z_i$ for all $1 \leq i \leq n$.

ii) If $\hat{\delta}(q_s, u)$ contains two complete items $[A \rightarrow w., \{z_1, \ldots, z_n\}]$ and $[B \rightarrow v., \{y_1, \ldots, y_k\}]$, then $y_i \neq z_j$ for all $1 \leq i \leq k$, $1 \leq j \leq n$.

Example 17.5.1

The deterministic LR(1) machine is constructed for the grammar AE.



The state containing the complete item $S \rightarrow A.$ also contains $A \rightarrow A. + T$. It follows that AE is not LR(0). Upon entering this state, the LR(1) parser halts unsuccessfully unless the lookahead symbol is $+$ or the null string. In the latter case, the entire input string has

been read and a reduction with the rule $S \rightarrow A$ is specified. When the lookahead symbol is +, the parser shifts in an attempt to construct the string $A + T$. □

The action of a parser for an LR(1) grammar upon scanning the string u is selected by the result of the computation $\hat{\delta}(q_s, u)$. Algorithm 17.5.4 gives a deterministic algorithm for parsing an LR(1) grammar.

Algorithm 17.5.4
Parser for an LR(1) Grammar

input: LR(1) grammar G = (V, Σ, P, S)
string $p \in \Sigma^*$
deterministic LR(1) machine of G

1. Let $p = zv$ where $z \in \Sigma \cup \{\lambda\}$ and $v \in \Sigma^*$
   (z is the lookahead symbol, v the remainder of the input)
2. $u := \lambda$
3. dead-end := *false*
4. **repeat**
   4.1. **if** $\hat{\delta}(q_s, u)$ contains $[A \rightarrow w., \{z_1, \ldots, z_n\}]$
   where $u = xw$ and $z = z_i$ for some $1 \leq i \leq n$ **then** $u := xA$
   else if $z \neq \lambda$ **and** $\hat{\delta}(q_s, u)$ contains an item $A \rightarrow p.zq$ **then**
   (shift and obtain new lookahead symbol)
   4.1.1. $u := uz$
   4.1.2. Let $v = zv'$ where $z \in \Sigma \cup \{\lambda\}$ and $v' \in \Sigma^*$
   4.1.3. $v := v'$
   end if
   else dead-end := *true*
   until $u = S$ **or** dead-end
5. **if** $u = S$ **then** accept **else** reject

For an LR(1) grammar, the structure of the LR(1) machine ensures that the action specified in step 4.1 is unique. When a state contains more than one complete item, the lookahead symbol specifies the appropriate operation.

Example 17.5.2

Algorithm 17.5.4 and the deterministic LR(1) machine in Figure 17.5 are used to parse the strings aaa and ac using the grammar

$$
\begin{aligned}
G:\ & S \to A \mid Bc \\
& A \to aA \mid a \\
& B \to a \mid ab.
\end{aligned}
$$

u	z	v	**Computation**		**Action**
λ	a	aa	$\hat{\delta}(q_s, \lambda) =$	$\{[S \to .A, \{\lambda\}],$ $[S \to .Bc, \{\lambda\}],$ $[A \to .aA, \{\lambda\}],$ $[A \to .a, \{\lambda\}],$ $[B \to .a\ \{c\}],$ $[B \to .ab\ \{c\}]\}$	shift
a	a	a	$\hat{\delta}(q_s, a) =$	$\{[A \to a.A, \{\lambda\}],$ $[A \to a., \{\lambda\}],$ $[A \to .aA, \{\lambda\}],$ $[A \to .a, \{\lambda\}],$ $[B \to a., \{c\}],$ $[B \to a.b, \{c\}]\}$	shift
aa	a	λ	$\hat{\delta}(q_s, aa) =$	$\{[A \to a.A, \{\lambda\}],$ $[A \to .aA, \{\lambda\}],$ $[A \to .a, \{\lambda\}],$ $[A \to a., \{\lambda\}]\}$	shift
aaa	λ	λ	$\hat{\delta}(q_s, aaa) =$	$\{[A \to a.A, \{\lambda\}],$ $[A \to .aA, \{\lambda\}],$ $[A \to .a, \{\lambda\}],$ $[A \to a., \{\lambda\}]\}$	reduce
aaA	λ	λ	$\hat{\delta}(q_s, aaA) =$	$\{[A \to aA., \{\lambda\}]\}$	reduce
aA	λ	λ	$\hat{\delta}(q_s, aA) =$	$\{[A \to aA., \{\lambda\}]\}$	reduce
A	λ	λ	$\hat{\delta}(q_s, A) =$	$\{[S \to A., \{\lambda\}]\}$	reduce
S					

u	z	v	Computation		Action
λ	a	c	$\hat{\delta}(q_s, \lambda) =$	$\{[S \to .A, \{\lambda\}],$ $[S \to .Bc, \{\lambda\}],$ $[A \to .aA, \{\lambda\}],$ $[A \to .a, \{\lambda\}],$ $[B \to .a\ \{c\}],$ $[B \to .ab\ \{c\}]\}$	shift
a	c	λ	$\hat{\delta}(q_s, a) =$	$\{[A \to a.A, \{\lambda\}],$ $[A \to a., \{\lambda\}],$ $[A \to .aA, \{\lambda\}],$ $[A \to .a, \{\lambda\}],$ $[B \to a., \{c\}],$ $[B \to a.b, \{c\}]\}$	reduce
B	c	λ	$\hat{\delta}(q_s, B) =$	$\{[S \to B.c, \{\lambda\}]\}$	shift
Bc	λ	λ	$\hat{\delta}(q_s, Bc) =$	$\{[S \to Bc., \{\lambda\}]\}$	reduce
S					

□

Exercises

1. Give the LR(0) contexts for the rules of the grammars below. Build the nondeterministic LR(0) machine. Use this to construct the deterministic LR(0) machine. Is the grammar LR(0)?

   a) $S \to AB$
   $A \to aA \mid b$
   $B \to bB \mid a$

   b) $S \to Ac$
   $A \to BA \mid \lambda$
   $B \to aB \mid b$

   c) $S \to A$
   $A \to aAb \mid bAa \mid \lambda$

   d) $S \to aA \mid AB$
   $A \to aAb \mid b$
   $B \to ab \mid b$

e) $S \to BA \mid bAB$
$A \to aA \mid \lambda$
$B \to Bb \mid b$

f) $S \to A \mid aB$
$A \to BC \mid \lambda$
$B \to Bb \mid C$
$C \to Cc \mid c$

2. Build the deterministic LR(0) machine for the grammar

$$S \to aAb \mid aB$$
$$A \to Aa \mid \lambda$$
$$B \to Ac.$$

Use the technique presented in Example 17.3.1 to trace the parse of the strings $aaab$ and ac.

3. Show that the grammar AE without an endmarker is not LR(0).
4. Prove Lemma 17.4.2.
5. Prove that an LR(0) grammar is unambiguous.
6. Define the LR(k) contexts of a rule $A \to w$.
7. For each of the grammars defined below, construct the nondeterministic and deterministic LR(1) machines. Is the grammar LR(1)?

a) $S \to Ac$
$A \to BA \mid \lambda$
$B \to aB \mid b$

b) $S \to A$
$A \to AaAb \mid \lambda$

c) $S \to A$
$A \to aAb \mid B$
$B \to Bb \mid b$

d) $S \to A$
$A \to BB$
$B \to aB \mid b$

e) $S \to A$
$A \to AAa \mid AAb \mid c$

8. Construct the LR(1) machine for the grammar introduced in Example 17.4.3. Is this grammar LR(1)?
9. Parse the strings below using the LR(1) parser and the grammar AE. Trace the actions of the parser using the format of Example 17.5.2. The deterministic LR(1) machine of AE is given in Example 17.5.1.

a) $b + b$

b) (b)

c) $b + +b$

Bibliographic Notes

LR grammars were introduced by Knuth [1965]. The number of states and transitions in the LR machine made the use of LR techniques impractical for parsers of computer languages. Korenjak [1969] and De Remer [1969, 1971] developed simplifications that eliminated these difficulties. The latter works introduced the SLR (simple LR) and LALR (lookahead LR) grammars.

The relationships between the class of LR(k) grammars and other classes of grammars that can be deterministically parsed, including the LL(k) grammars, are developed in Aho and Ullman [1972, 1973]. Additional information on syntax analysis using LR parsers can be found in the compiler design texts mentioned in the bibliographic notes of Chapter 16.

APPENDIX I

Index of Notation

Symbol	Page	Interpretation
$\in$	8	is an element of
$\notin$	8	is not an element of
$\{x \mid \ldots\}$	8	the set of x such that . . .
$\mathbf{N}$	8	the set of natural numbers
$\emptyset$	8	empty set
$\subseteq$	9	is a subset of
$\mathcal{P}(\mathrm{X})$	9	power set of X
$\cup$	9	union
$\cap$	9	intersection
$-$	9	X − Y: set difference
$\overline{\mathrm{X}}$	10	complement
$\times$	11	X × Y: Cartesian product
$[x, y]$	11	ordered pair
$f: \mathrm{X} \to \mathrm{Y}$	11	f is a function from X to Y
$f(x)$	11	value assigned to x by the function f
$f(x) \uparrow$	13	$f(x)$ is undefined

Symbol	Page	Interpretation
$f(x)\downarrow$	13	$f(x)$ is defined
div	13	integer division
$\equiv$	14	equivalence relation
$[\,]_{\equiv}$	14	equivalence class
card	15	cardinality
s	20, 355	successor function
$\sum_{i=n}^{m}$	26, 390	bounded summation
!	27, 384	factorial
$\dot{-}$	34	proper subtraction
λ	38	null sting
Σ^*	38	set of strings over Σ
length	38	length of a string
u^R	40	reversal of u
XY	42	concatenation of sets X and Y
X^i	42	concatenation of X with itself i times
X^*	43	strings over X
X^+	43	nonnull strings over X
∞	43	infinity
$\emptyset$	45	regular expression for the empty set
λ	45	regular expression for the null string
a	45	regular expression for the symbol a
$\cup$	45	regular expression union operation
$\rightarrow$	55, 297	rule of a grammar
$\Rightarrow$	57, 298	is derivable by one rule application
$\overset{*}{\Rightarrow}$	59, 298	is derivable from
$\overset{+}{\Rightarrow}$	59	is derivable by one or more rule applications
$\overset{n}{\Rightarrow}$	59	is derivable by n rule applications
L(G)	60	language of the grammar G
$n_x(u)$	73	number of occurrences of x in u

Symbol	Page	Interpretation
$\underset{L}{\Rightarrow}$	89	leftmost rule application
$\underset{R}{\Rightarrow}$	89	rightmost rule application
$g(\mathrm{G})$	92	graph of the grammar G
shift	105, 517	shift function
δ	158, 168, 228, 260	transition function
L(M)	159, 169, 230, 263	language of the machine M
$\vdash$	160, 230, 261	yields by one transition
$\vdash^*$	160, 230, 261	yields by zero or more transitions
$\hat{\delta}$	161, 192	extended transition function
λ-*closure*	176	lambda closure function
B	259	blank tape symbol
lo	288	lexicographical ordering
$\mathfrak{P}$	344	property of recursively enumerable languages
z	355	zero function
e	356	empty function
$p_i^{(k)}$	356	k-variable projection function
id	356	identity function
$\circ$	364	composition
$c_i^{(k)}$	367	k-variable constant function
$\lfloor x \rfloor$	376	greatest integer less than or equal to x
$\prod_{i=0}^{n}$	390	bounded product
$\mu z^{y}[p]$	393	bounded minimalization
$pn(i)$	396	ith prime function
gn_k	397	k-variable Gödel numbering function
$dec(i, x)$	398	decoding function
gn_f	398	bounded Gödel numbering function
$\mu z[p]$	404	unbounded minimalization
tr_{M}	408	Turing machine trace function

Symbol	Page	Interpretation
tc_M	426	time complexity function
$\lceil x \rceil$	432	least integer greater than or equal to x
$O(f)$	434	order of the function f
sc_M	448	space complexity function
$\mathcal{P}$	454	polynomial languages
$\mathcal{NP}$	457	nondeterministically polynomial languages
$LA(A)$	490	lookahead set of variable A
$LA(A \to w)$	490	lookahead set of the rule $A \to w$
$trunc_k$	492	length-k truncation function
$FIRST_k(u)$	494	$FIRST_k$ set of the string u
$FOLLOW_k(A)$	495	$FOLLOW_k$ set of the variable A

APPENDIX II

The Greek Alphabet

Uppercase	Lowercase	Name
A	α	alpha
B	β	beta
Γ	γ	gamma
Δ	δ	delta
E	ϵ	epsilon
Z	ζ	zeta
H	η	eta
Θ	θ	theta
I	ι	iota
K	κ	kappa
Λ	λ	lambda
M	μ	mu
N	ν	nu
Ξ	ξ	xi
O	o	omicron
Π	π	pi
P	ρ	rho
Σ	σ	sigma
T	τ	tau
Υ	υ	upsilon
Φ	ϕ	phi
X	χ	chi
Ψ	ψ	psi
Ω	ω	omega

APPENDIX III

Backus-Naur Definition of Pascal

The programming language Pascal was developed by Niklaus Wirth in the late 1960s. The language was defined using the notation known as the Backus-Naur form (BNF). The metasymbol $\{u\}$ denotes zero or more repetitions of the string inside the brackets. Thus the BNF rule $A \rightarrow \{u\}$ is represented in a context-free grammar by the rules $A \rightarrow uA \mid \lambda$. The variables in the BNF definition are enclosed in $\langle\ \rangle$. The null string is represented by $\langle empty \rangle$. The BNF rule and its context-free counterpart are given to illustrate the conversion from one notation to the other.

BNF definition:

$$\langle unsigned\ integer \rangle \rightarrow \langle digit \rangle\ \{\langle digit \rangle\}$$

Context-free equivalent:

$$\langle unsigned\ integer \rangle \rightarrow \langle digit \rangle \mid \langle digits \rangle$$

$$\langle digits \rangle \rightarrow \langle digit \rangle\ \langle digits \rangle \mid \lambda$$

The context-free rules can be simplified to

$$\langle unsigned\ integer \rangle \rightarrow \langle digit \rangle\ \langle unsigned\ integer \rangle \mid \langle digit \rangle$$

The start symbol of the BNF definition of Pascal is the variable $\langle program \rangle$. A syntactically correct Pascal program is a string that can be obtained by a derivation initiated with the rule $\langle program \rangle \rightarrow \langle program\ heading \rangle; \langle program\ block \rangle$.

Reprinted from Kathleen Jensen and Niklaus Wirth, *Pascal: User Manual and Report*, 2d ed., Springer Verlag, New York, 1974.

⟨*program*⟩ → ⟨*program heading*⟩ ; ⟨*program block*⟩

⟨*program block*⟩ → ⟨*block*⟩

⟨*program heading*⟩ → **program** ⟨*identifier*⟩ (⟨*file identifier*⟩ {, ⟨*file identifier*⟩}) ;

⟨*file identifier*⟩ → ⟨*identifier*⟩

⟨*identifier*⟩ → ⟨*letter*⟩ {⟨*letter or digit*⟩}

⟨*letter or digit*⟩ → ⟨*letter*⟩ | ⟨*digit*⟩

⟨*letter*⟩ → a | b | . . . | z

⟨*digit*⟩ → 0 | 1 | . . . | 9

⟨*block*⟩ → ⟨*label declaration part*⟩ ⟨*constant definition part*⟩ ⟨*type definition part*⟩
⟨*variable declaration part*⟩ ⟨*procedure and function declaration part*⟩
⟨*statement part*⟩

⟨*label declaration part*⟩ → ⟨*empty*⟩ | **label** ⟨*label*⟩ {, ⟨*label*⟩} ;

⟨*label*⟩ → ⟨*unsigned integer*⟩

⟨*constant definition part*⟩ → ⟨*empty*⟩ |
const ⟨*constant definition*⟩ {; ⟨*constant definition*⟩} ;

⟨*constant definition*⟩ → ⟨*identifier*⟩ = ⟨*constant*⟩

⟨*constant*⟩ → ⟨*unsigned number*⟩ | ⟨*sign*⟩ ⟨*unsigned number*⟩ | ⟨*constant identifier*⟩ |
⟨*sign*⟩ ⟨*constant identifier*⟩ | ⟨*string*⟩

⟨*unsigned number*⟩ → ⟨*unsigned integer*⟩ | ⟨*unsigned real*⟩

⟨*unsigned integer*⟩ → ⟨*digit*⟩ {⟨*digit*⟩}

⟨*unsigned real*⟩ → ⟨*unsigned integer*⟩ . ⟨*digit*⟩ {⟨*digit*⟩} |
⟨*unsigned integer*⟩ . ⟨*digit*⟩ {⟨*digit*⟩} E ⟨*scale factor*⟩ |
⟨*unsigned integer*⟩ E ⟨*scale factor*⟩

⟨*scale factor*⟩ → ⟨*unsigned integer*⟩ | ⟨*sign*⟩ ⟨*unsigned integer*⟩

⟨*sign*⟩ → + | −

⟨*constant identifier*⟩ → ⟨*identifier*⟩

⟨*string*⟩ → '⟨*character*⟩ {⟨*character*⟩}'

⟨*type definition part*⟩ → ⟨*empty*⟩ | **type** ⟨*type definition*⟩ {; ⟨*type definition*⟩} ;

⟨*type definition*⟩ → ⟨*identifier*⟩ = ⟨*type*⟩

⟨*type*⟩ → ⟨*simple type*⟩ | ⟨*structured type*⟩ | ⟨*pointer type*⟩

⟨*simple type*⟩ → ⟨*scalar type*⟩ | ⟨*subrange type*⟩ | ⟨*type identifier*⟩

⟨*scalar type*⟩ → ⟨*identifier*⟩ {, ⟨*identifier*⟩})

⟨*subrange type*⟩ → ⟨*constant*⟩ . . ⟨*constant*⟩

⟨*type identifier*⟩ → ⟨*identifier*⟩

⟨*structured type*⟩ → ⟨*unpacked structured type*⟩ | **packed** ⟨*unpacked structured type*⟩

⟨*unpacked structured type*⟩ → ⟨*array type*⟩ | ⟨*record type*⟩ | ⟨*set type*⟩ | ⟨*file type*⟩

⟨*array type*⟩ → **array** [⟨*index type*⟩ {, ⟨*index type*⟩}] **of** ⟨*component type*⟩

⟨*index type*⟩ → ⟨*simple type*⟩

⟨*component type*⟩ → ⟨*type*⟩

⟨*record type*⟩ → **record** ⟨*field list*⟩ **end**

⟨*field list*⟩ → ⟨*fixed part*⟩ | ⟨*fixed part*⟩ ; ⟨*variant part*⟩ | ⟨*variant part*⟩

⟨*fixed part*⟩ → ⟨*record section*⟩ {; ⟨*record section*⟩}

⟨*record section*⟩ → ⟨*field identifier*⟩ {, ⟨*field identifier*⟩} : ⟨*type*⟩ | ⟨*empty*⟩

⟨*variant part*⟩ → **case** ⟨*tag field*⟩ ⟨*type identifier*⟩ **of** ⟨*variant*⟩ {; ⟨*variant*⟩}

⟨*tag field*⟩ → ⟨*field identifier*⟩ : | ⟨*empty*⟩

⟨*variant*⟩ → ⟨*case label list*⟩ : (⟨*field list*⟩) | ⟨*empty*⟩

⟨*case label list*⟩ → ⟨*case label*⟩ {, ⟨*case label*⟩}

⟨*case label*⟩ → ⟨*constant*⟩

⟨*set type*⟩ → **set of** ⟨*base type*⟩

⟨*base type*⟩ → ⟨*simple type*⟩

⟨*file type*⟩ → **file of** ⟨*type*⟩

⟨*pointer type*⟩ → ↑ ⟨*type identifier*⟩

⟨*variable declaration part*⟩ → ⟨*empty*⟩ |
var ⟨*variable declaration*⟩ {; ⟨*variable declaration*⟩} ;

⟨*variable declaration*⟩ → ⟨*identifier*⟩ {, ⟨*identifier*⟩} : ⟨*type*⟩

⟨*procedure and function declaration part*⟩ → {⟨*procedure or function declaration*⟩ ;}

⟨*procedure or function declaration*⟩ → ⟨*procedure declaration*⟩ |
⟨*function declaration*⟩

⟨*procedure declaration*⟩ → ⟨*procedure heading*⟩ ⟨*block*⟩

⟨*procedure heading*⟩ → **procedure** ⟨*identifier*⟩ ; |
procedure ⟨*identifier*⟩ (⟨*formal parameter section*⟩
{; ⟨*formal parameter section*⟩}) ;

⟨*formal parameter section*⟩ → ⟨*parameter group*⟩ | **var** ⟨*parameter group*⟩ |
function ⟨*parameter group*⟩ |
procedure ⟨*identifier*⟩ {, ⟨*identifier*⟩}

⟨*parameter group*⟩ → ⟨*identifier*⟩ {, ⟨*identifier*⟩} : ⟨*type identifier*⟩

⟨*function declaration*⟩ → ⟨*function heading*⟩ ⟨*block*⟩

⟨*function heading*⟩ → **function** ⟨*identifier*⟩ : ⟨*result type*⟩ ; |
function ⟨*identifier*⟩ (⟨*formal parameter section*⟩
{; ⟨*formal parameter section*⟩}) : ⟨*result type*⟩ ;

⟨*result type*⟩ → ⟨*type identifier*⟩

⟨*statement part*⟩ → ⟨*compound statement*⟩

⟨*statement*⟩ → ⟨*unlabeled statement*⟩ | ⟨*label*⟩ : ⟨*unlabeled statement*⟩

⟨*unlabeled statement*⟩ → ⟨*simple statement*⟩ | ⟨*structured statement*⟩

⟨*simple statement*⟩ → ⟨*assignment statement*⟩ | ⟨*procedure statement*⟩ |
⟨*go to statement*⟩ | ⟨*empty statement*⟩

⟨*assignment statement*⟩ → ⟨*variable*⟩ := ⟨*expression*⟩ |
⟨*function identifier*⟩ := ⟨*expression*⟩

⟨*variable*⟩ → ⟨*entire variable*⟩ | ⟨*component variable*⟩ | ⟨*referenced variable*⟩

⟨*entire variable*⟩ → ⟨*variable identifier*⟩

⟨*variable identifier*⟩ → ⟨*identifier*⟩

⟨*component variable*⟩ → ⟨*indexed variable*⟩ | ⟨*field designator*⟩ | ⟨*file buffer*⟩

⟨*indexed variable*⟩ → ⟨*array variable*⟩ [⟨*expression*⟩ {, ⟨*expression*⟩}]

⟨*array variable*⟩ → ⟨*variable*⟩

⟨*field designator*⟩ → ⟨*record variable*⟩ . ⟨*field identifier*⟩

⟨*record variable*⟩ → ⟨*variable*⟩

⟨*field identifier*⟩ → ⟨*identifier*⟩

⟨*file buffer*⟩ → ⟨*file variable*⟩ ↑

⟨*file variable*⟩ → ⟨*variable*⟩

⟨*referenced variable*⟩ → ⟨*pointer variable*⟩ ↑

⟨*pointer variable*⟩ → ⟨*variable*⟩

⟨*expression*⟩ → ⟨*simple expression*⟩ | ⟨*simple expression*⟩ ⟨*relational operator*⟩
⟨*simple expression*⟩

⟨*relational operator*⟩ → = | <> | < | <= | >= | > | **in**

⟨*simple expression*⟩ → ⟨*term*⟩ | ⟨*sign*⟩ ⟨*term*⟩ |
⟨*simple expression*⟩ ⟨*adding operator*⟩ ⟨*term*⟩

⟨*adding operator*⟩ → + | − | **or**

⟨*term*⟩ → ⟨*factor*⟩ | ⟨*term*⟩ ⟨*multiplying operator*⟩ ⟨*factor*⟩

⟨*multiplying operator*⟩ → * | / | **div** | **mod** | **and**

⟨*factor*⟩ → ⟨*variable*⟩ | ⟨*unsigned constant*⟩ | (⟨*expression*⟩) | ⟨*function designator*⟩ |
⟨*set*⟩ | **not** ⟨*factor*⟩

⟨*unsigned constant*⟩ → ⟨*unsigned number*⟩ | ⟨*string*⟩ | ⟨*constant identifier*⟩ | **nil**

⟨*function designator*⟩ → ⟨*function identifier*⟩ |
⟨*function identifier*⟩ (⟨*actual parameter*⟩ {, ⟨*actual parameter*⟩})

⟨*function identifier*⟩ → ⟨*identifier*⟩

⟨*set*⟩ → [⟨*element list*⟩]

⟨*element list*⟩ → ⟨*element*⟩ {, ⟨*element*⟩} | ⟨*empty*⟩

⟨*element*⟩ → ⟨*expression*⟩ | ⟨*expression*⟩ .. ⟨*expression*⟩

⟨*procedure statement*⟩ → ⟨*procedure identifier*⟩ |
⟨*procedure identifier*⟩ (⟨*actual parameter*⟩
{, ⟨*actual parameter*⟩})

⟨*procedure identifier*⟩ → ⟨*identifier*⟩

⟨*actual parameter*⟩ → ⟨*expression*⟩ | ⟨*variable*⟩ |
⟨*procedure identifier*⟩ | ⟨*functional identifier*⟩

⟨*go to statement*⟩ → **goto** ⟨*label*⟩

⟨*empty statement*⟩ → ⟨*empty*⟩

⟨*empty*⟩ → λ

⟨*structured statement*⟩ → ⟨*compound statement*⟩ | ⟨*conditional statement*⟩ |
⟨*repetitive statement*⟩ | ⟨*with statement*⟩

⟨*compound statement*⟩ → **begin** ⟨*statement*⟩ {; ⟨*statement*⟩} **end**

⟨*conditional statement*⟩ → ⟨*if statement*⟩ | ⟨*case statement*⟩

⟨*if statement*⟩ → **if** ⟨*expression*⟩ **then** ⟨*statement*⟩ |
if ⟨*expression*⟩ **then** ⟨*statement*⟩ **else** ⟨*statement*⟩

⟨*case statement*⟩ → **case** ⟨*expression*⟩ **of** ⟨*case list element*⟩ {; ⟨*case list element*⟩} **end**

⟨*case list element*⟩ → ⟨*case label list*⟩ : ⟨*statement*⟩ | ⟨*empty*⟩

⟨*case label list*⟩ → ⟨*case label*⟩ {, ⟨*case label*⟩}

⟨*repetitive statement*⟩ → ⟨*while statement*⟩ | ⟨*repeat statement*⟩ | ⟨*for statement*⟩

⟨*while statement*⟩ → **while** ⟨*expression*⟩ **do** ⟨*statement*⟩

⟨*repeat statement*⟩ → **repeat** ⟨*statement*⟩ {; ⟨*statement*⟩} **until** ⟨*expression*⟩

⟨*for statement*⟩ → **for** ⟨*control variable*⟩ := ⟨*for list*⟩ **do** ⟨*statement*⟩

⟨*for list*⟩ → ⟨*initial value*⟩ **to** ⟨*final value*⟩ | ⟨*initial value*⟩ **downto** ⟨*final value*⟩

⟨*control variable*⟩ → ⟨*identifier*⟩

⟨*initial value*⟩ → ⟨*expression*⟩

⟨*final value*⟩ → ⟨*expression*⟩

⟨*with statement*⟩ → **with** ⟨*record variable list*⟩ **do** ⟨*statement*⟩

⟨*record variable list*⟩ → ⟨*record variable*⟩ {, ⟨*record variable*⟩}

Bibliography

Ackermann, W. [1928], "Zum Hilbertschen Aufbau der reellen Zahlen," *Mathematische Annalen,* 99, pp. 118–133.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1974], *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA.

Aho, A. V., R. Sethi, and J. D. Ullman [1986], *Compilers: Principles, Techniques and Tools,* Addison-Wesley, Reading, MA.

Aho, A. V., and J. D. Ullman [1972], *The Theory of Parsing, Translation and Compilation,* Vol. I: *Parsing,* Prentice-Hall, Englewood Cliffs, NJ.

Aho, A. V., and J. D. Ullman [1973], *The Theory of Parsing, Translation and Compilation,* Vol. II: *Compiling,* Prentice-Hall, Englewood Cliffs, NJ.

Backhouse, R. C. [1979], *Syntax of Programming Languages: Theory and Practice,* Prentice-Hall, Englewood Cliffs, NJ.

Backus, J. W. [1959], "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference," *Proceedings of the International Conference on Information Processing,* pp. 125–132.

Bar-Hillel, Y., M. Perles, and E. Shamir [1961], "On formal properties of simple phrase-structure grammars," *Zeitschrift für Phonetik, Sprachwissenschaft, und Kommunikationsforschung,* 14, pp. 143–177.

Barrett, W. A., R. M. Bates, D. A. Gustafson, and J. D. Couch [1986], *Compiler Construction: Theory and Practice,* Science Research Associates, Chicago, IL.

Bavel, Z. [1983], *Introduction to the Theory of Automata,* Reston Publishing Co., Reston, VA.

Blum, M. [1967], "A machine independent theory of the complexity of recursive functions," *J. ACM,* 14, pp. 322–336.

Bobrow, L. S., and M. A. Arbib [1974], *Discrete Mathematics: Applied Algebra for Computer and Information Science,* Saunders, Philadelphia, PA.

Bondy, J. A., and U. S. R. Murty [1977], *Graph Theory with Applications,* Elsevier, New York.

Brainerd, W. S., and L. H. Landweber [1974], *Theory of Computation,* Wiley, New York.

Busacker, R. G., and T. L. Saaty [1965], *Finite Graphs and Networks: An Introduction with Applications,* McGraw-Hill, New York.

Cantor, D. C. [1962], "On the ambiguity problems of Backus systems," *J. ACM,* 9, pp. 477–479.

Cantor, G. [1947], *Contributions to the Foundations of the Theory of Transfinite Numbers* (reprint), Dover, New York.

Chomsky, N. [1956], "Three models for the description of languages," *IRE Transactions on Information Theory,* 2, pp. 113–124.

Chomsky, N. [1959], "On certain formal properties of grammars," *Information and Control,* 2, pp. 137–167.

Chomsky, N. [1962], "Context-free grammar and pushdown storage," *Quarterly Progress Report* 65, M.I.T. Research Laboratory in Electronics, pp. 187–194.

Chomsky, N., and G. A. Miller [1958], "Finite state languages," *Information and Control,* 1, pp. 91–112.

Chomsky, N., and M. P. Schutzenberger [1963], "The algebraic theory of context free languages," in *Computer Programming and Formal Systems,* North-Holland, Amsterdam, pp. 118–161.

Church, A. [1936], "An unsolvable problem of elementary number theory," *American Journal of Mathematics,* 58, pp. 345–363.

Church, A. [1941], "The calculi of lambda-conversion," *Annals of Mathematics Studies,* 6, Princeton University Press, Princeton, NJ.

Cobham, A. [1964], "The intrinsic computational difficulty of functions," *Proceedings of the 1964 Congress for Logic, Mathematics and Philosophy of Science,* North-Holland, New York, pp. 24–30.

Cook, S. A. [1971], "The complexity of theorem proving procedures," *Proceedings of the Third Annual ACM Symposium on the Theory of Computing,* Association for Computing Machinery, New York, pp. 151–158.

Davis, M. D. [1965], *The Undecidable,* Raven Press, Hewlett, NY.

Davis, M. D., and E. J. Weyuker [1983], *Computability, Complexity and Languages: Fundamentals of Theoretical Computer Science,* Academic Press, New York.

Denning, P. J., J. B. Dennis, and J. E. Qualitz [1978], *Machines, Languages and Computation,* Prentice-Hall, Englewood Cliffs, NJ.

De Remer, F. L. [1969], "Generating parsers for BNF grammars," *Proceedings of the 1969 Fall Joint Computer Conference,* AFIPS Press, Montvale, NJ, pp. 793–799.

De Remer, F. L. [1971], "Simple LR(k) grammars," *Comm. ACM,* 14, pp. 453–460.

Edmonds, J. [1965], "Paths, trees and flowers," *Canadian Journal of Mathematics,* 3, pp. 449–467.

Evey, J. [1963], "Application of pushdown store machines," *Proceedings of the 1963 Fall Joint Computer Science Conference,* AFIPS Press, pp. 215–217.

Floyd, R. W. [1962], "On ambiguity in phrase structure languages," *Comm. ACM,* 5, pp. 526–534.

Floyd, R. W. [1964], *New Proofs and Old Theorems in Logic and Formal Linguistics,* Computer Associates, Inc., Wakefield, MA.

Foster, J. M. [1968], "A syntax improving program," *Computer J.,* 11, pp. 31–34.

Garey, M. R., and D. S. Johnson [1979], *Computers and Intractability: A Guide to the Theory of NP-completeness,* Freeman, New York.

Gersting, J. L. [1982], *Mathematical Structures for Computer Science,* W. H. Freeman, San Francisco, CA.

Ginsburg, S. [1966], *The Mathematical Theory of Context-Free Languages,* McGraw-Hill, New York.

Ginsburg, S., and H. G. Rice [1962], "Two families of languages related to ALGOL," *J. ACM,* 9, pp. 350–371.

Ginsburg, S., and G. F. Rose [1963a], "Some recursively unsolvable problems in ALGOL-like languages," *J. ACM,* 10, pp. 29–47.

Ginsburg, S., and G. F. Rose [1963b], "Operations which preserve definability in languages," *J. ACM,* 10, pp. 175–195.

Ginsburg, S., and J. S. Ullian [1966a], "Ambiguity in context-free languages," *J. ACM,* 13, pp. 62–89.

Ginsburg, S., and J. S. Ullian [1966b], "Preservation of unambiguity and inherent ambiguity in context-free languages," *J. ACM,* 13, pp. 364–368.

Gödel, K. [1931], "Uber formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I," *Monatschefte für Mathematik und Physik,* 38, pp. 173–198. (English translation in Davis [1965].)

Greibach, S. [1965], "A new normal form theorem for context-free phrase structure grammars," *J. ACM,* 12, pp. 42–52.

Halmos, P. R. [1974], "Naive Set Theory," Springer-Verlag, New York.

Harrison, M. A. [1978], *Introduction to Formal Language Theory,* Addison-Wesley, Reading, MA.

Hartmanis, J., and J. E. Hopcroft [1971], "An overview of the theory of computational complexity," *J. ACM,* 18, pp. 444–475.

Hennie, F. C. [1977], *Introduction to Computability,* Addison-Wesley, Reading, MA.

Hermes, H. [1965], *Enumerability, Decidability, Computability,* Academic Press, New York.

Hopcroft, J. E. [1971], "An $n \log n$ algorithm for minimizing the states in a finite automaton," in *The Theory of Machines and Computation,* ed. by Z. Kohavi, Academic Press, New York, pp. 189–196.

Hopcroft, J. E., and J. D. Ullman [1979], *Introduction to Automata Theory, Languages and Computation,* Addison-Wesley, Reading, MA.

Jensen, K., and N. Wirth [1974], *Pascal: User Manual and Report,* 2d ed., Springer-Verlag, New York.

Johnsonbaugh, R. [1984], *Discrete Mathematics,* Macmillan, New York.

Karp, R. M. [1972], "Reducibility among combinatorial problems," in *Complexity of Computer Computations,* Plenum Press, New York, pp. 85–104.

Karp, R. M. [1986], "Combinatorics, complexity and randomness," *Comm. ACM,* 29, no. 2, pp. 98–109.

Kleene, S. C. [1936], "General recursive functions of natural numbers," *Mathematische Annalen,* 112, pp. 727–742.

Kleene, S. C. [1952], *Introduction to Metamathematics,* Van Nostrand, Princeton, NJ.

Kleene, S. C. [1956], "Representation of events in nerve nets and finite automata," in *Automata Studies,* ed. by C. E. Shannon and J. McCarthy, Princeton University Press, Princeton, NJ., pp. 3–42.

Knuth, D. E. [1965], "On the translation of languages from left to right," *Information and Control,* 8, pp. 607–639.

Knuth, D. E. [1968], *The Art of Computer Programming:* Vol. 1: *Fundamental Algorithms,* Addison-Wesley, Reading, MA.

Kolman, B., and R. C. Busby [1984], *Discrete Mathematical Structures for Computer Science,* Prentice-Hall, Englewood Cliffs, NJ.

Korenjak, A. J. [1969], "A practical method for constructing LR(k) processors," *Comm. ACM,* 12, pp. 613–623.

Kurki-Suonio, R. [1969], "Notes on top-down languages," *BIT,* 9, pp. 225–238.

Kuroda, S. Y. [1964], "Classes of languages and linear-bounded automata," *Information and Control,* 7, pp. 207–223.

Ladner, R. E. [1975], "On the structure of polynomial time reducibility," *Journal of the ACM,* 22, pp. 155–171.

Landweber, P. S. [1963], "Three theorems of phrase structure grammars of type 1," *Information and Control,* 6, pp. 131–136.

Lewis, H. R., and C. H. Papadimitriou [1981], *Elements of the Theory of Computation,* Prentice-Hall, Englewood Cliffs, NJ.

Lewis, P. M., II, D. J. Rosenkrantz, and R. E. Stearns [1976], *Compiler Design Theory,* Addison-Wesley, Reading, MA.

Lewis, P. M., II, and R. E. Stearns [1968], "Syntax directed transduction," *J. ACM,* 15, pp. 465–488.

Machtey, M., and P. R. Young [1978], *An Introduction to the General Theory of Algorithms,* Elsevier North-Holland, New York.

Markov, A. A. [1961], *Theory of Algorithms,* Israel Program for Scientific Translations, Jerusalem.

McNaughton, R., and H. Yamada [1960], "Regular expressions and state graphs for automata," *IEEE Transactions on Electronic Computers,* 9, pp. 39–47.

Mealy, G. H. [1955], "A method for synthesizing sequential circuits," *Bell System Technical Journal,* 34, pp. 1045–1079.

Minsky, M. L. [1967], *Computation: Finite and Infinite Machines,* Prentice-Hall, Englewood Cliffs, NJ.

Moore, E. F. [1956], "Gendanken-experiments on sequential machines," in *Automata Studies,* ed. by C. E. Shannon and J. McCarthy, Princeton University Press, Princeton, NJ, pp. 129–153.

Myhill, J. [1957], "Finite automata and the representation of events," WADD Technical Report 57-624, pp. 129–153, Wright Patternson Air Force Base, Ohio.

Myhill, J. [1960], "Linear bounded automata," WADD Technical Note 60-165, Wright Patterson Air Force Base, Ohio.

Naur, P., ed. [1963], "Revised report on the algorithmic language ALGOL 60," *Comm. ACM,* 6, pp. 1–17.

Nerode, A. [1958], "Linear automaton transformations," *Proc. AMS,* 9, pp. 541–544.

Oettinger, A. G. [1961], "Automatic syntax analysis and the pushdown store," *Proceedings on Symposia on Applied Mathematics,* 12, American Mathematical Society, Providence, RI, pp. 104–129.

Ogden, W. G. [1968], "A helpful result for proving inherent ambiguity," *Mathematical Systems Theory,* 2, pp. 191–194.

Ore, O. [1963], *Graphs and Their Uses,* Random House, New York.

Papadimitriou, C. H. [1994], *Computational Complexity*, Addison-Wesley, Reading, MA.

Parikh, R. J. [1966], "On context-free languages," *J. ACM,* 13, pp. 570–581.

Péter, R. [1967], *Recursive Functions,* Academic Press, New York.

Post, E. L. [1936], "Finite combinatory processes—formulation I," *Journal of Symbolic Logic,* 1, pp. 103–105.

Post, E. L. [1946], "A variant of a recursively unsolvable problem," *Bulletin of the American Mathematical Society,* 52, pp. 264–268.

Post, E. L. [1947], "Recursive unsolvability of a problem of Thue," *Journal of Symbolic Logic,* 12, pp. 1–11.

Pratt, V. [1975], "Every prime has a succinct certificate," *SIAM Journal of Computation,* 4, pp. 214–220.

Pyster, A. B. [1980], *Compiler Design and Construction,* Van Nostrand Reinhold, New York.

Rabin, M. O., and D. Scott [1959], "Finite automata and their decision problems," *IBM J. Res.,* 3, pp. 115–125.

Rawlins, G. E. O. [1992], *Compared to What? An Introduction to the Analysis of Algorithms,* Computer Science Press, New York.

Rice, H. G. [1953], "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society,* 89, pp. 25–29.

Rice, H. G. [1956], "On completely recursively enumerable classes and their key arrays," *Journal of Symbolic Logic,* 21, pp. 304–341.

Rogers, H., Jr. [1967], *Theory of Recursive Functions and Effective Computation,* McGraw-Hill, New York.

Rosenkrantz, D. J., and R. E. Stearns [1970], "Properties of deterministic top-down grammars," *Information and Control,* 17, pp. 226–256.

Sahni, S. [1981], *Concepts of Discrete Mathematics,* Camelot, Fridley, MN.

Salomaa, A. [1966], "Two complete axiom systems for the algebra of regular events," *J. ACM,* 13, pp. 156–199.

Salomaa, A. [1973], *Formal Languages,* Academic Press, New York.

Scheinberg, S. [1960], "Note on the Boolean properties of context-free languages," *Information and Control,* 3, pp. 372–375.

Schutzenberger, M. P. [1963], "On context-free languages and pushdown automata," *Information and Control,* 6, pp. 246–264.

Sheperdson, J. C. [1959], "The reduction of two-way automata to one-way automata," *IBM J. Res.,* 3, pp. 198–200.

Soisalon-Soininen, E., and E. Ukkonen [1979], "A method for transforming grammars into LL(k) form," *Acta Informatica,* 12, pp. 339–369.

Stearns, R. E. [1971], "Deterministic top-down parsing," *Proceedings of the Fifth Annual Princeton Conference of Information Sciences and Systems,* pp. 182–188.

Stoll, R. [1963], *Set Theory and Logic,* W. H. Freeman, San Francisco, CA.

Thue, A. [1914], "Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln," *Skrifter utgit av Videnskappsselskapet i Kristiana,* I., Matematisk-naturvidenskabelig klasse 10.

Tremblay, J. P., and R. Manohar [1975], *Discrete Mathematical Structures with Applications to Computer Science,* McGraw-Hill, New York.

Turing, A. M. [1936], "On computable numbers with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society,* 2, no. 42, pp. 230–265; no. 43, pp. 544–546.

Wand, M. [1980], *Induction, Recursion and Programming,* North-Holland, New York.

Wilson, R. J. [1985], *Introduction to Graph Theory,* 3d ed., American Elsevier, New York.

Wirth, N. [1971], "The programming language Pascal," *Acta Informatica,* 1, pp. 35–63.

Wood, D. [1969], "The theory of left factored languages," *Computer Journal,* 12, pp. 349–356.

Subject Index

Languages *and* Machines, *Second Edition*

Thomas A. Sudkamp

Languages and Machines gives a mathematically sound presentation of the theory of computing at the junior and senior level, and is an invaluable tool for scientists investigating the theoretical foundations of computer science. Topics covered include the theory of formal languages and automata, computability, computational complexity, and deterministic parsing of context-free languages.

No special mathematical prerequisites are assumed; the theoretical concepts and associated mathematics are made accessible by a "learn as you go" approach that develops an intuitive understanding of the concepts through numerous examples and illustrations. *Languages and Machines* examines the languages of the Chomsky hierarchy, the grammars that generate them, and the finite automata that accept them. Sections on the Church–Turing thesis and computability theory further examine the development of abstract machines. Computational complexity and NP-completeness are introduced by analyzing the computations of Turing machines. Parsing with LL and LR grammars is included to emphasize language definition and to provide the groundwork for the study of compiler design.

Features New to this Edition:

- DFA minimization
- Rice's Theorem
- Increased coverage of computational complexity
- Additional examples throughout
- Over 150 additional exercises

About the Author:

Thomas A. Sudkamp holds a Ph.D. in mathematics from the University of Notre Dame. Professor Sudkamp worked extensively in industry and for the Air Force before joining the faculty at Wright State University, where he has taught for over 10 years.

Access the latest information about Addison-Wesley titles from our World Wide Web page:
http://www.aw.com/cseng/

ADDISON-WESLEY

Addison-Wesley is an imprint
of Addison Wesley Longman

90000
9 780201 821369
ISBN 0-201-82136-2