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 $f : X \rightarrow Y$. The element of Y assigned by the function f to an element $x \in X$ is denoted $f(x)$. The set X is called the **domain** of the function. The **range** of f is the subset of Y consisting of the members of Y that are assigned to elements of X. Thus the range of a function $f : X \rightarrow Y$ is the set $\{y \in Y \mid y = f(x) \text{ for some } x \in X\}$.

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

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

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

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.

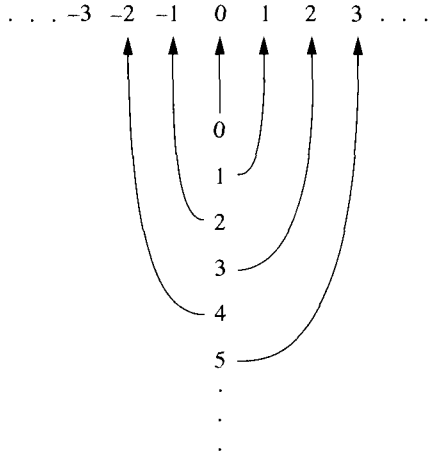
Definition 1.4.3

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

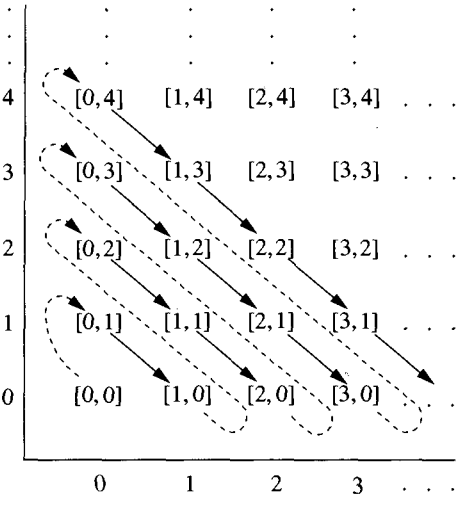


exhibits the countability of the set of integers. This correspondence is defined by the function

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

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

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

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,

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

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, $D = N_i$ for some i. Is the number i in the set D? By definition of D,

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

But since $D = N_i$, this becomes

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

We have shown that $i \in D$ if, and only if, $i \notin D$, 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.

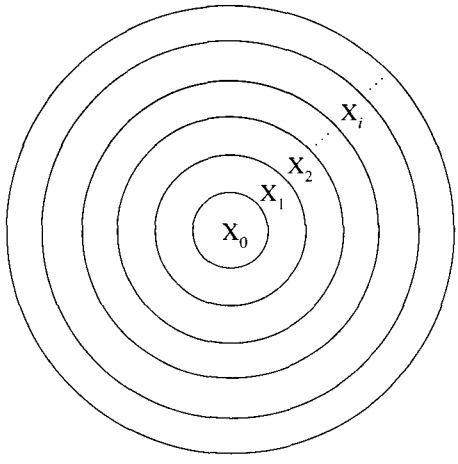
Following the construction given in Definition 1.5.2, the sum of any two natural numbers can be computed using 0 and s, the primitives used in the definition of the natural numbers. Example 1.5.1 traces the recursive computation of $3 + 2$.

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.

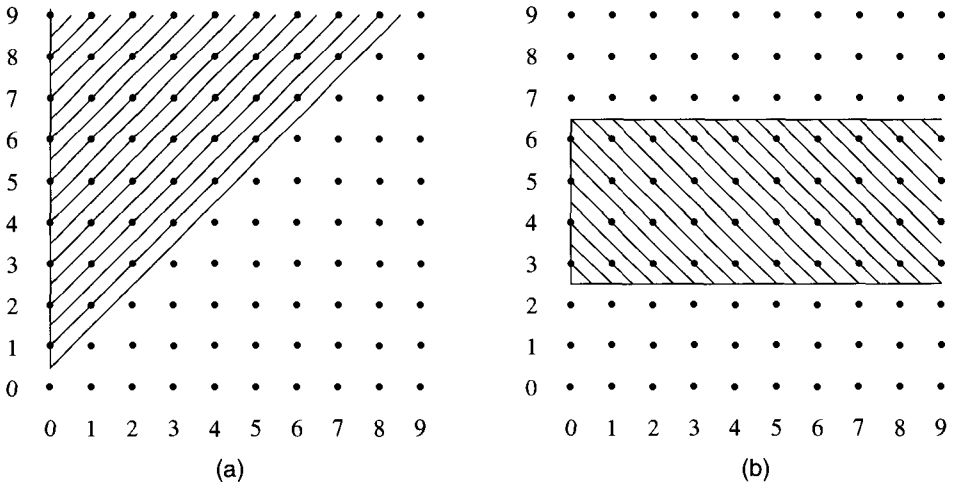


FIGURE 1.2 Relations on $\mathbf{N} \times \mathbf{N}$.

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.

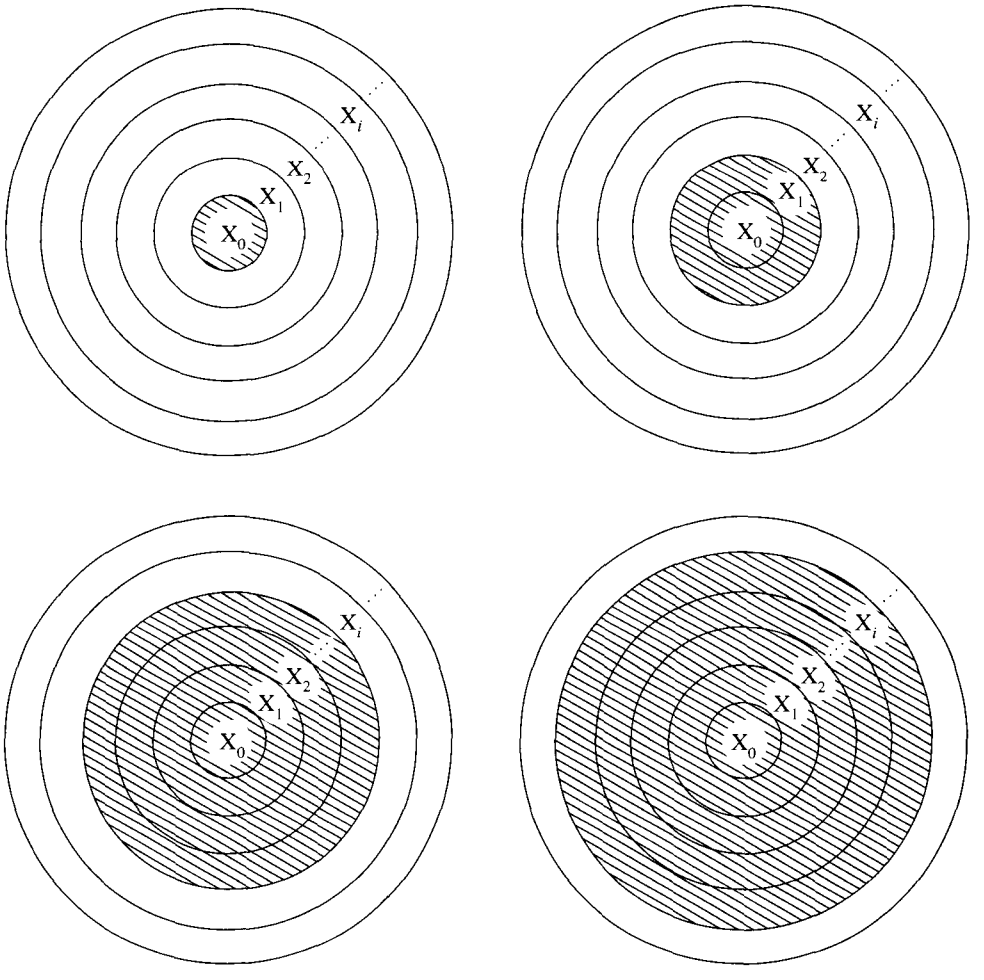


FIGURE 1.3 Principle of mathematical induction.

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

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

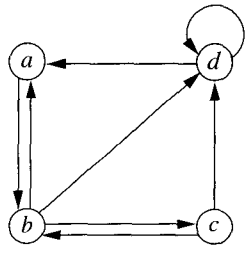
□

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 **N** contains the natural numbers, s is a unary function on **N**, and 0 is a distinguished element of **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

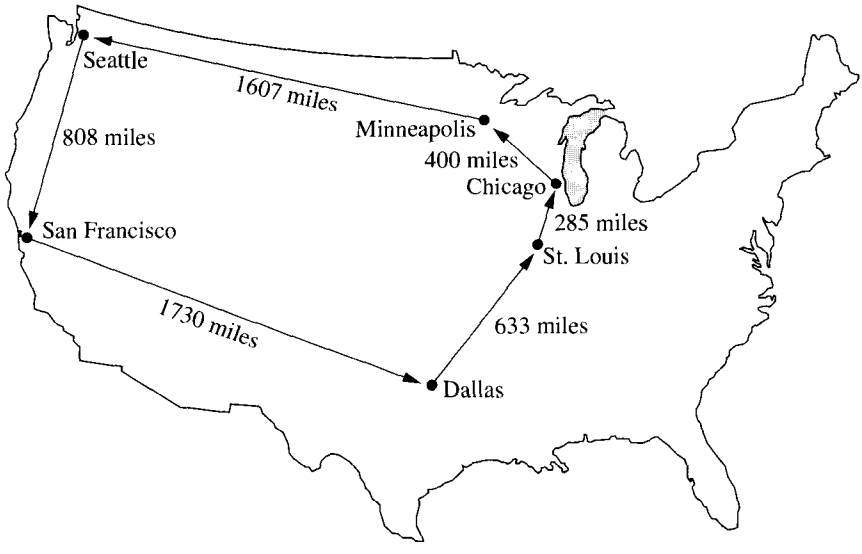


FIGURE 1.5 Labeled directed graph.

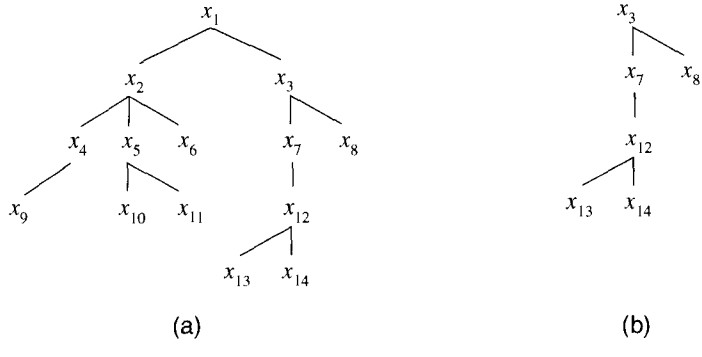


FIGURE 1.6 (a) Tree with root x_1. (b) Subtree generated by x_3.

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.

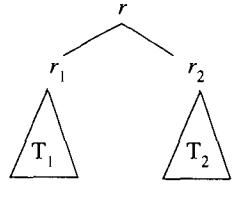
iii) Closure: T is a strictly binary tree only if it can be obtained from the basis elements by a finite number of applications of the construction given in the recursive step.

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 **N** to {0, 1}.
19. A total function f from **N** to **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 **N** to **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 **N** to **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

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

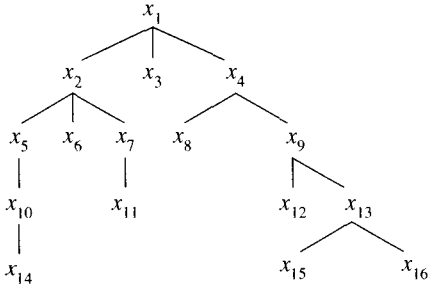
using the operator s.

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 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

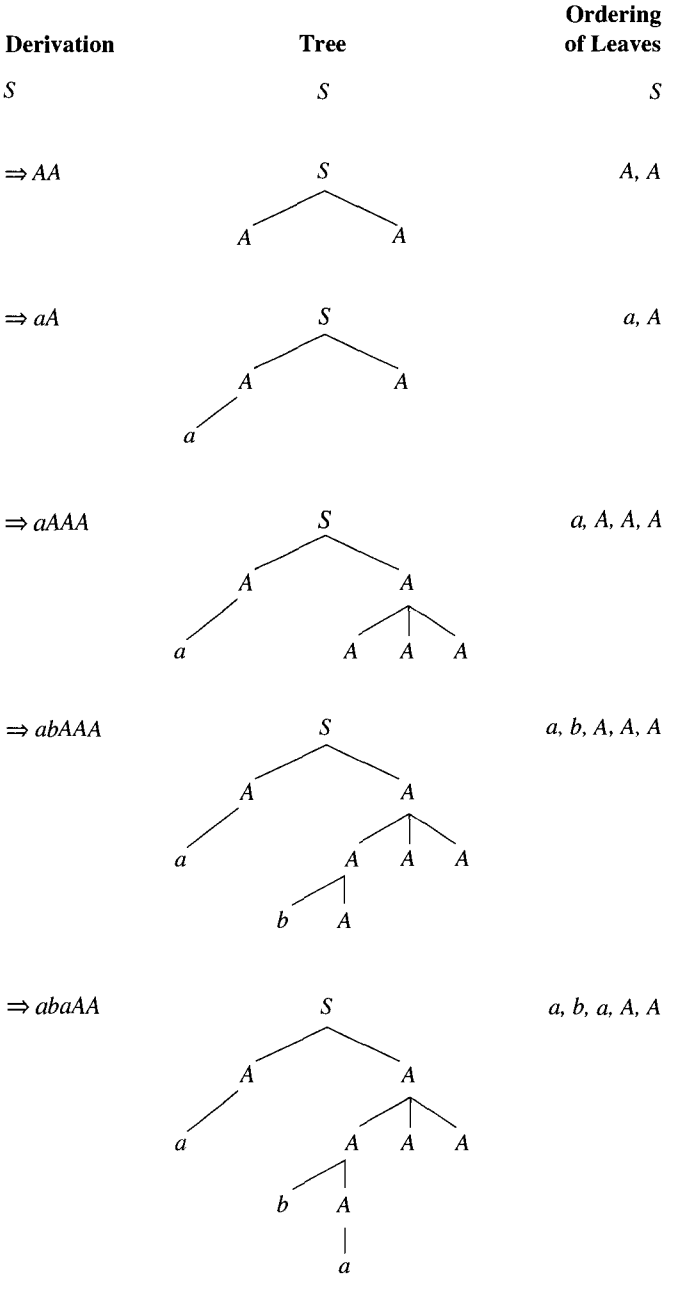


FIGURE 3.2 Construction of derivation tree.

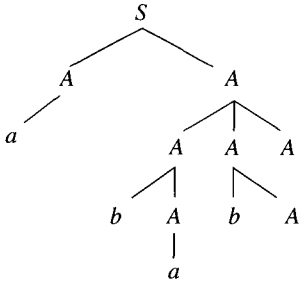
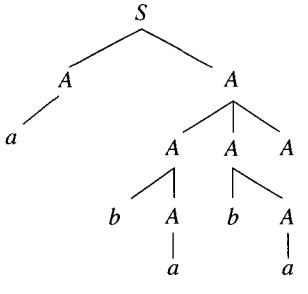
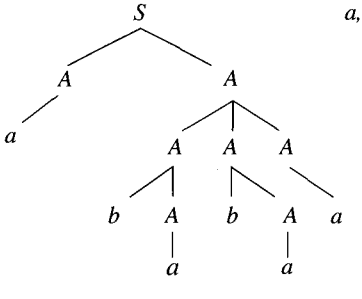
Derivation	Tree	Ordering of Leaves
$\Rightarrow ababAA$		a, b, a, b, A, A
$\Rightarrow ababaA$		a, b, a, b, a, A
$\Rightarrow ababaa$		a, b, a, b, a, a

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

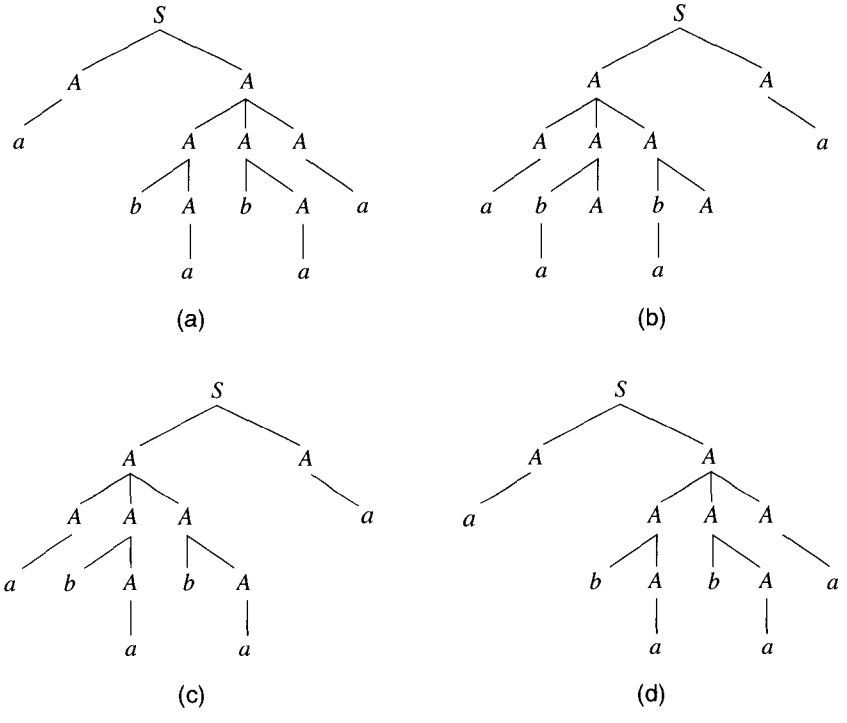


FIGURE 3.3 Trees corresponding to the derivations in Figure 3.1.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Both of these grammars generate the language a^+b^*. The A rules in G_1 provide the standard method of generating a nonnull string of a's. The use of the 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 $a^*ba^*ba^*$.

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

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

$$\begin{aligned} G:\ S &\to AASB \mid AAB \\ A &\to a \\ B &\to bbb \end{aligned}$$

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

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

Any string in 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 $w \in$ L(G), the inequality yields $n_a(w) \leq n_b(w)$ as desired. □

Example 3.4.4

The grammar

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

was constructed in Example 3.2.2 to generate the language $\text{L} = \{a^n b^m c^m d^{2n} \mid n \geq 0, m > 0\}$. We develop relationships among the variables and terminals that are sufficient to prove that L(G) $\subseteq$ L. The S and the A rules enforce the numeric relationships between the a's and d's and the b's and c's. In a derivation of G, the start symbol is removed by an application of the rule $S \rightarrow A$. The presence of an A guarantees that a b will eventually be generated. These observations lead to the following 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

$$\langle \textit{expression} \rangle \rightarrow \langle \textit{simple expression} \rangle$$

$$\begin{aligned} \langle \textit{simple expression} \rangle \rightarrow\ & \langle \textit{term} \rangle \mid \\ & \langle \textit{sign} \rangle\ \langle\ \textit{term} \rangle \mid \\ & \langle \textit{simple expression} \rangle\ \langle\ \textit{adding operator} \rangle\ \langle \textit{term} \rangle. \end{aligned}$$

The arithmetic operators for numeric computation are defined by

$$\begin{aligned} &\langle \textit{adding operator} \rangle \rightarrow + \mid - \\ &\langle \textit{multiplying operator} \rangle \rightarrow * \mid / \mid \textbf{div} \mid \textbf{mod}. \end{aligned}$$

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.

$$\langle \textit{term} \rangle \rightarrow \langle \textit{factor} \rangle \mid \langle\ \textit{term} \rangle\ \langle\ \textit{multiplying operator} \rangle\ \langle\ \textit{factor} \rangle$$

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.

$$\langle \textit{factor} \rangle \rightarrow \langle \textit{variable} \rangle \mid \langle \textit{unsigned constant} \rangle \mid (\langle \textit{expression} \rangle)$$

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 $x + 5$.

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

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

$$S \rightarrow abSc \mid A$$
$$A \rightarrow cAd \mid cd.$$

   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

$$S \rightarrow ASB \mid \lambda$$
$$A \rightarrow aAb \mid \lambda$$
$$B \rightarrow bBa \mid ba.$$

   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

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

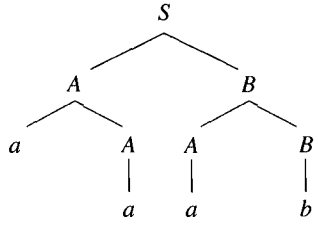
a) Give a leftmost derivation of $abbaab$.

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 $x1y$ from the variable $\langle variable \rangle$.
46. Construct a derivation of $(x1y)$ from $\langle expression \rangle$.
47. Construct a derivation for the expression $(x * y * 5)$ from the variable $\langle expression \rangle$.
48. For the not-faint-of-heart: Construct a derivation of $(x + y * (12 + z))$ from the variable $\langle expression \rangle$.
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) $N = \{w \in (V \cup \Sigma)^* \mid S \underset{L}{\overset{*}{\Rightarrow}} w\}$

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**

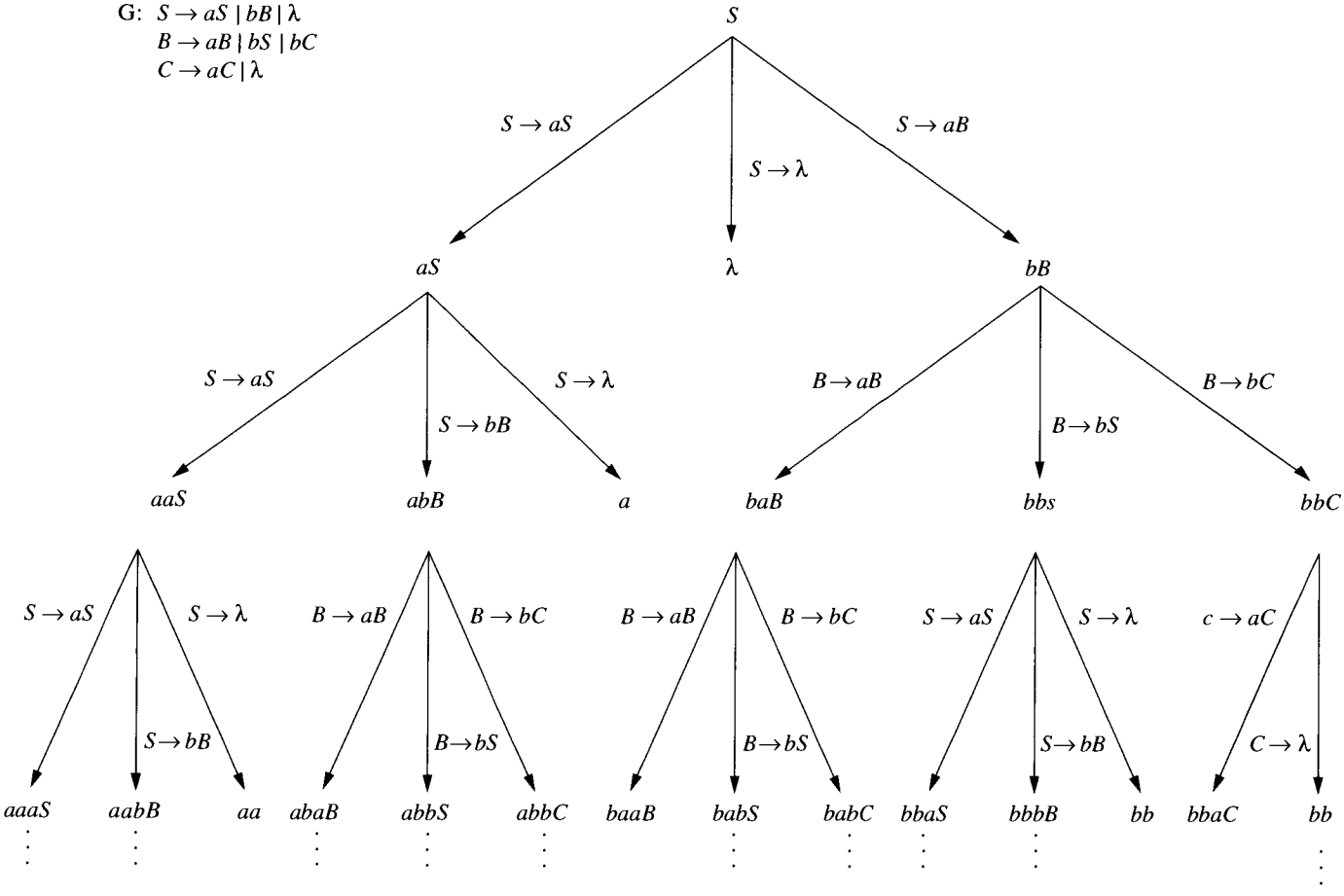
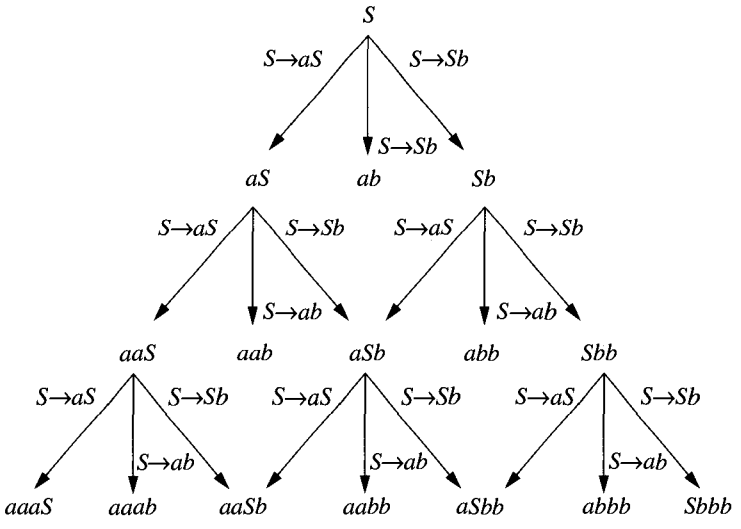
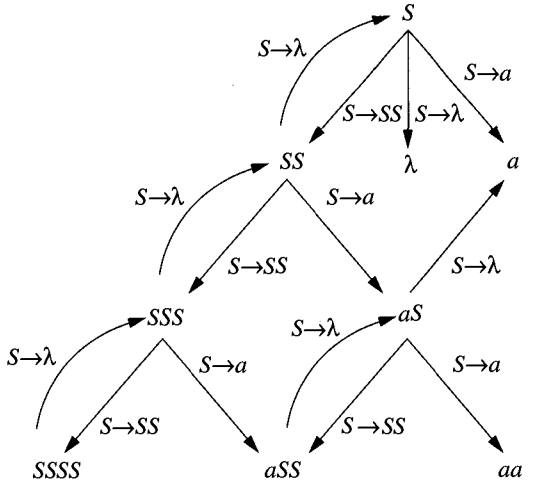


FIGURE 4.1 Graph of grammar G.



(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(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 G = (V, Σ, 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.

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

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.

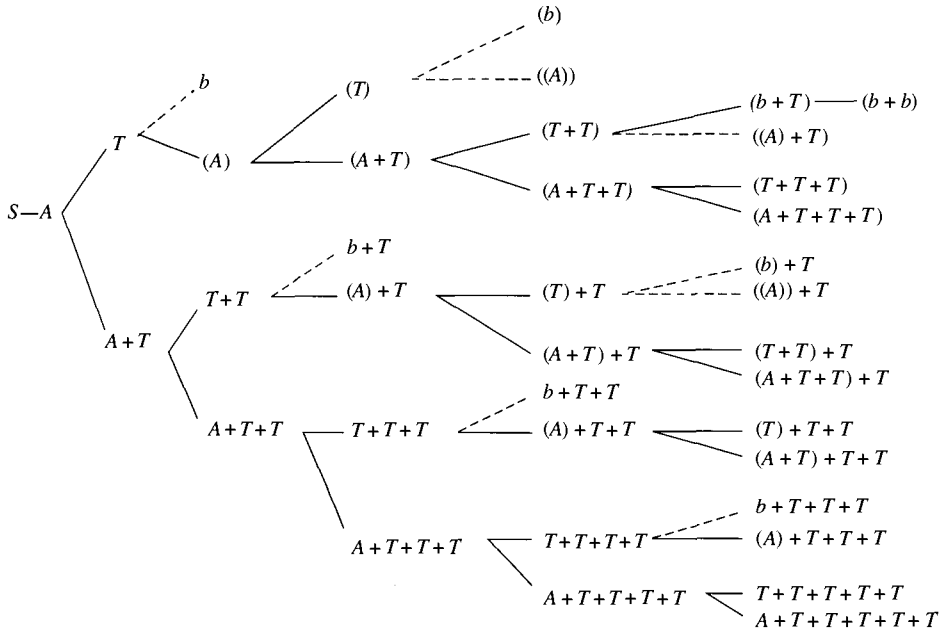


FIGURE 4.3 Breadth-first top-down parse of $(b + b)$.

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.

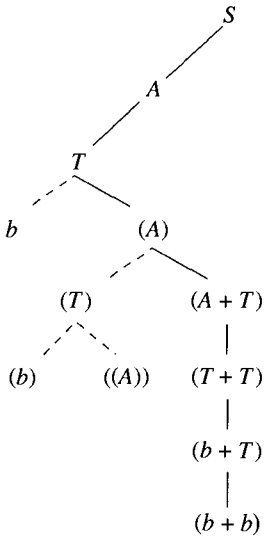
More complicated parsers may scan the input string several times. Multipass parsers may use other criteria, such as the total length of the derived string, to determine dead-ends. An initial pass can be used to obtain the length of the input string. A sentential form derived by the parser containing 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. $[q, i] := POP(\mathbf{S})$
   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, **Q**)
2. **repeat**
   q := *REMOVE*(**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, **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*(**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

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

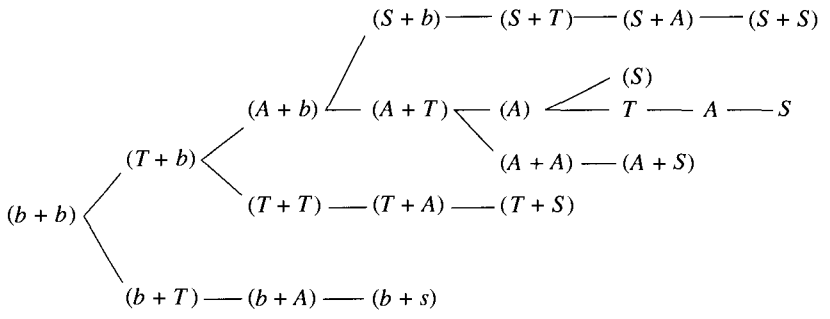


FIGURE 4.5 Breadth-first bottom-up parse of $(b + b)$.

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*(**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*(**S**)
3. **if** *EMPTY*(**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 $G = (V, \Sigma, P, S)$ be a context-free grammar. There is a grammar $G' = (V', \Sigma, P', S')$ 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:	G′:
$S \to aS \mid AB \mid AC$	$S' \to S$
$A \to aA \mid \lambda$	$S \to aS \mid AB \mid AC$
$B \to bB \mid bS$	$A \to aA \mid \lambda$
$C \to cC \mid \lambda$	$B \to bB \mid bS$
	$C \to cC \mid \lambda$

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 $S \rightarrow \lambda$ 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) $A \rightarrow \lambda$ 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 $S_L \rightarrow \lambda \in P_L$.

ii) Let $A \rightarrow w$ 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

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

is a rule of P_L.

iii) $A \rightarrow \lambda \in P_L$ 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 $S_L \rightarrow \lambda$ 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 $A \rightarrow w$ is a rule in P and, since $w \neq \lambda$, $A \rightarrow w$ 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

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

where $A_i \in V$ and $w_i \in \Sigma^*$. By Lemma 3.1.5, w can be written

$$w = w_1 p_1 w_2 p_2 \ldots w_k p_k w_{k+1},$$

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

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

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.

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

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 $A \rightarrow w \in P$ 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 $A \rightarrow w$ 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.

$$V_T = \text{TERM}$$

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

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

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 $G = (V, \Sigma, P, S)$

1. REACH := $\{S\}$
2. PREV := $\emptyset$
3. **repeat**
   3.1. NEW := REACH − PREV
   3.2. PREV := REACH
   3.3. **for** each variable $A \in$ 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 $G = (V, \Sigma, P, S)$ 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 $G = (V, \Sigma, P, S)$ be a context-free grammar. There is an algorithm to construct a context-free grammar G_U that satisfies

i) $L(G_U) = L(G)$.

ii) G_U has no useless symbols.

Proof The removal of useless symbols begins by building G_T from G. Algorithm 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 $G = (V, \Sigma, P, S)$ be a context-free grammar and let $A \in V$ be a directly left recursive variable in G. There is an algorithm to construct an equivalent grammar $G' = (V', \Sigma, P', S')$ in which A is not directly left recursive.

Proof We assume that the start symbol of G is nonrecursive, the only 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

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

Now the B rules must be transformed to the appropriate form. The rule $B \rightarrow AB$ must be replaced since the number of B is three and A, which occurs as the first symbol on the right-hand side, is two. Lemma 5.6.2 permits the leading A in the right-hand side of the rule $B \rightarrow AB$ to be replaced by the right-hand side of the A rules, producing

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

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

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

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

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

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.

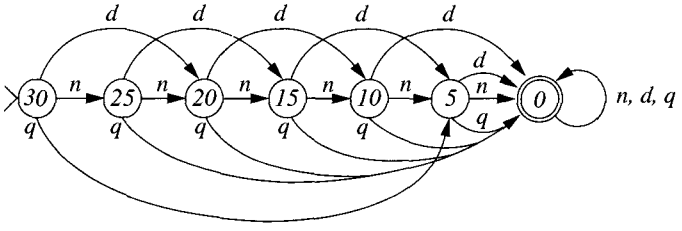


FIGURE 6.1 State diagram of newspaper vending machine.

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

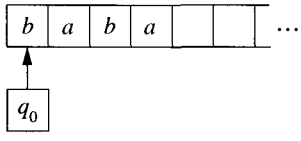
as the *start state*, F a subset of Q called the *final* or *accepting states*, and δ a total function from $Q \times \Sigma$ to Q known as the *transition function*.

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

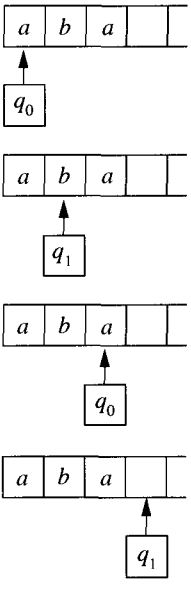


FIGURE 6.2 Computation in a DFA.

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, $L(M) = (a \cup b)^* bb(a \cup b)^*$.

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

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

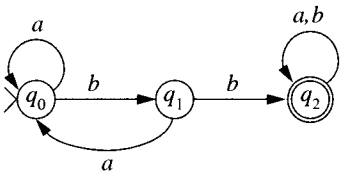
Let M = (Q, Σ, δ, q_0, F) be a DFA and let $w \in \Sigma^*$. Then w determines a unique path $\mathbf{p}_w$ in the state diagram of M and $\hat{\delta}(q_0, w) = q_w$.

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



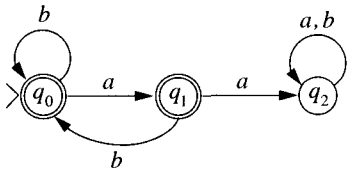
The states are used to record the number of consecutive b's processed. The state q_2 is entered when a substring bb is encountered. Once the machine enters q_2 the remainder of the input is processed, leaving the state unchanged. The computation of the DFA with input $ababb$ and the corresponding path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

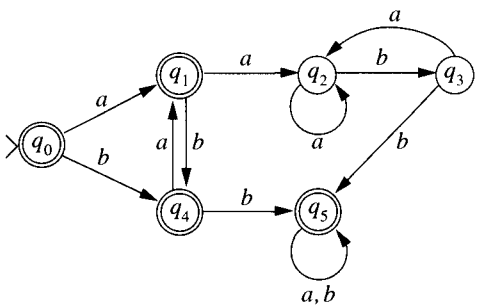
The DFA



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.

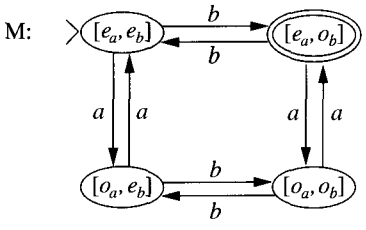


□

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

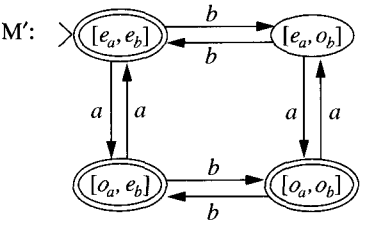
The DFA M accepts the language consisting of all strings over $\{a, b\}$ that contain an even number of a's and an odd number of b's.



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.

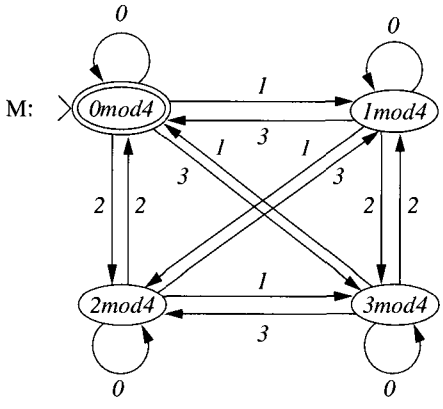
Theorem 6.3.3

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.



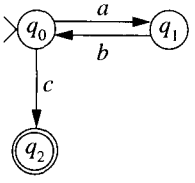
□

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.

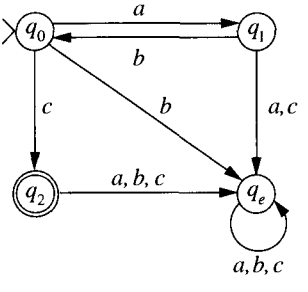


The computation with input $abcc$ is rejected since the machine is unable process the final c from state q_2. □

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

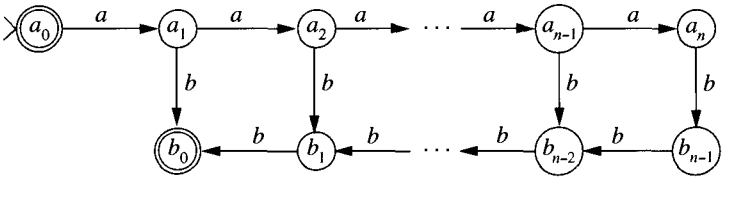
The DFA



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.

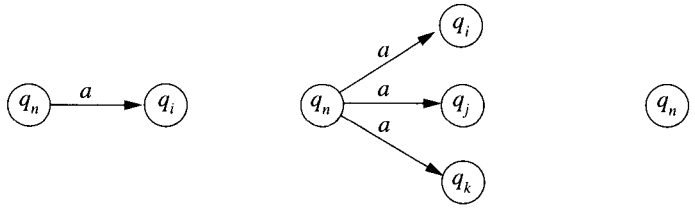


This technique cannot be extended to accept $\{a^i b^i \mid i \geq 0\}$ since an infinite number of states would be needed. In the next chapter we will show that this language is not accepted by any finite automaton. □

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, $L(M) = \{w \mid \text{there is a computation } [q_0, w] \vdash^* [q_i, \lambda] \text{ with } q_i \in F\}$.

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.

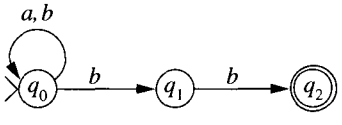
iv) F is the set of accepting nodes.

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

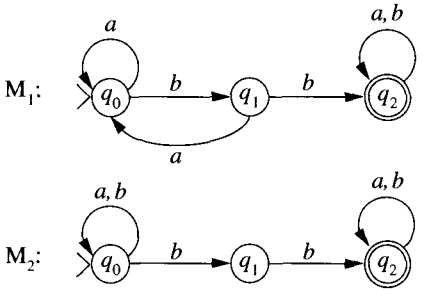
The state diagram for the NFA M from Example 6.4.1 is



Pictorially, it is clear that the language accepted by M is $(a \cup b)^*bb$. □

Example 6.4.3

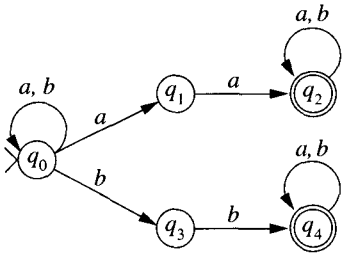
The state diagrams M_1 and M_2 define finite automata that accept $(a \cup b)^*bb(a \cup b)^*$.



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.

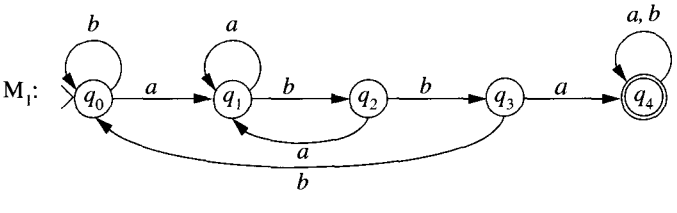


A path exhibiting the acceptance of a string reads the input in state q_0 until an occurrence of the substring aa or bb is encountered. At this point, the path branches to either q_1 or q_3, depending upon the substring. There are three distinct paths that exhibit the acceptance of the string $abaaabb$. □

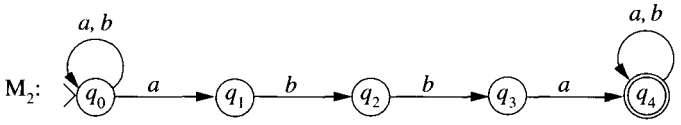
The flexibility permitted by the use of nondeterminism does not always simplify the problem of constructing a machine that accepts $L(M_1) \cup L(M_2)$ from the machines M_1 and M_2. This can be seen by attempting to construct an NFA that accepts the language of the DFA in Example 6.3.3.

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.

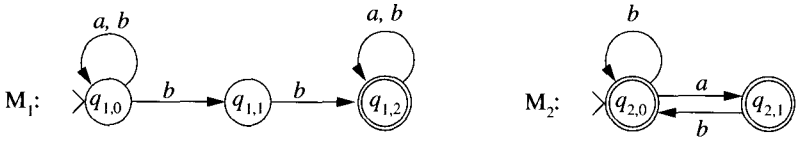
Lambda transitions represent another step away from the deterministic effective computations of a DFA. They do, however, provide a useful tool for the design of machines to accept complex languages.

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.

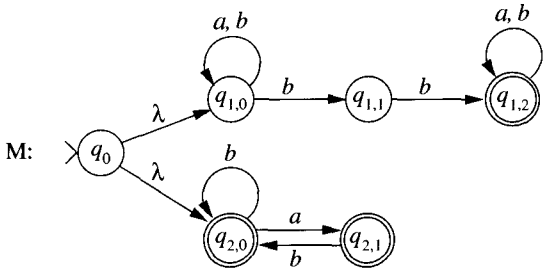
Lambda moves can be used to construct complex machines from simpler machines. Let M_1 and M_2 be the machines



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

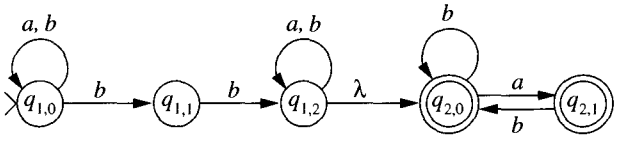
The language of the NFA-λ M is $L(M_1) \cup L(M_2)$.



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

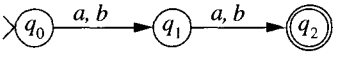
An NFA-λ that accepts $L(M_1)L(M_2)$, the concatenation of the languages of M_1 and M_2, is constructed by joining the two machines with a lambda arc.



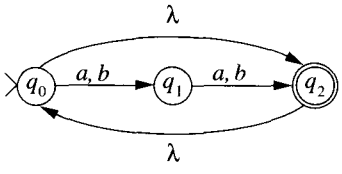
An input string is accepted only if it consists of a string from $L(M_1)$ concatenated with one from $L(M_2)$. The lambda transition allows the computation to enter M_2 whenever a prefix of the input string is accepted by M_1. □

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

$$\delta(q_0', \lambda) = \{q_0\}$$

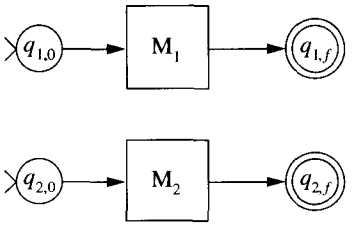
$$\delta(q_i, \lambda) = \{q_f\} \text{ for every } q_i \in F$$

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'. ■

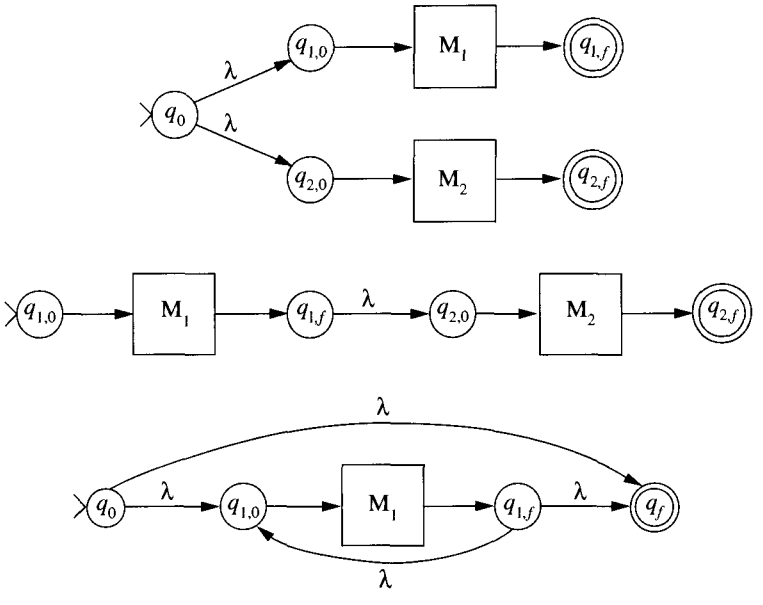
Theorem 6.5.3

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

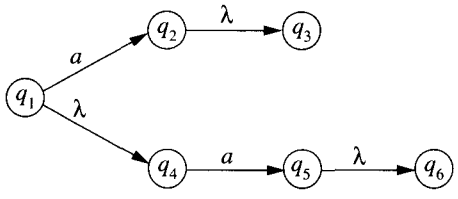


FIGURE 6.3 Paths with lambda transitions.

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

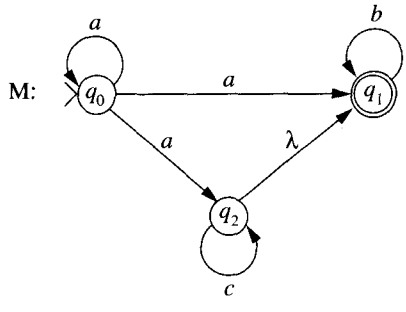
$$t(q_i, a) = \bigcup_{q_j \in \lambda\text{-}closure(q_i)} \lambda\text{-}closure(\delta(q_j, a)),$$

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

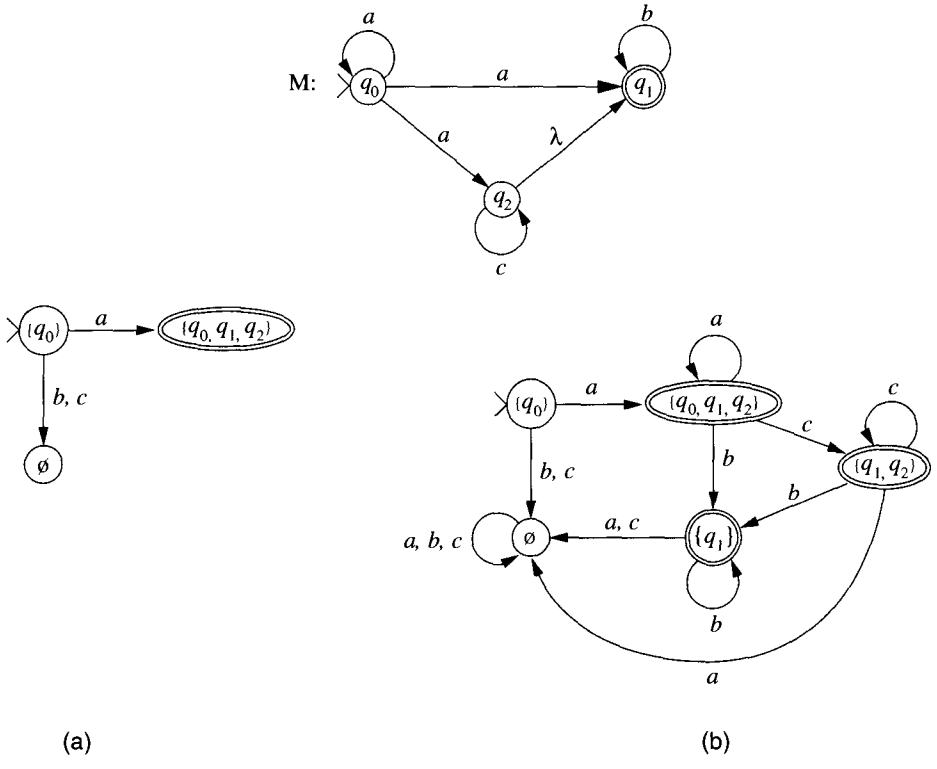


FIGURE 6.4 Construction of equivalent deterministic automaton.

$t(q_0, a)$, $t(q_1, a)$, 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.

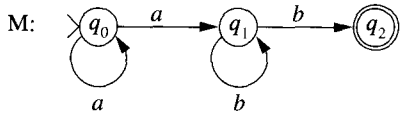
Conversely, let w be a string accepted by DM. Then Q_w contains an accepting state of M. The construction of Q_w guarantees the existence of a computation in M that processes w and terminates in that accepting state. These observations provide the justification for Corollary 6.6.5.

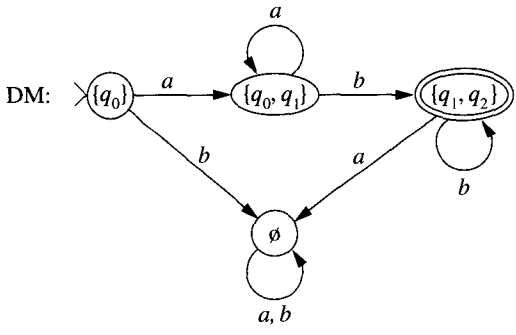
Corollary 6.6.5

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.

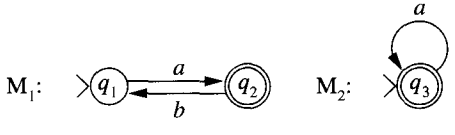




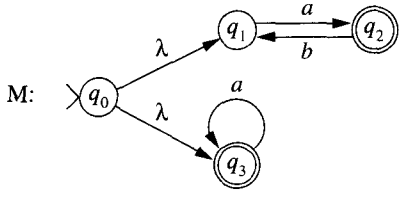
□

Example 6.6.3

The machines M_1 and M_2 accept $a(ba)^*$ and a^* respectively.



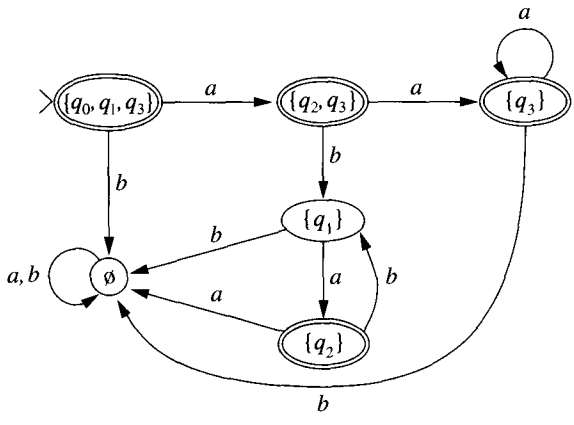
Using lambda arcs to connect a new start state to the start states of the original machines creates an NFA-λ M that accepts $a(ba)^* \cup a^*$.



The input transition function for M is

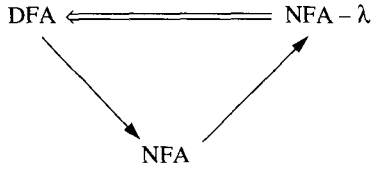
t	a	b
q_0	$\{q_2, q_3\}$	$\emptyset$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\{q_1\}$
q_3	$\{q_3\}$	$\emptyset$

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.

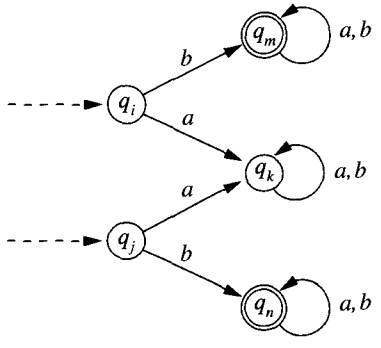
6.7 DFA Minimization

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 $u \in \Sigma^*$.

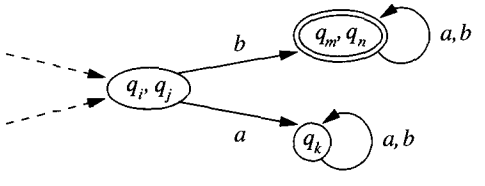
Two states that are equivalent are called *indistinguishable*. The binary relation over Q defined by indistinguishability of states is an equivalence relation, that is, the relation is reflexive, symmetric, and transitive. Two states that are not equivalent are said to be *distinquishable*. States q_i 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.

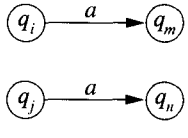
Algorithm 6.7.2
Determination of Equivalent States of DFA

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.

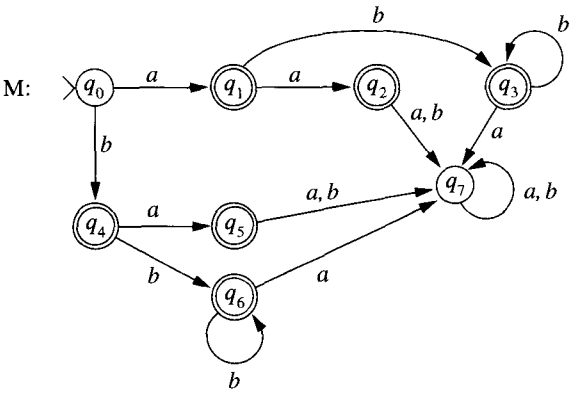
Now assume that every pair of states distinguishable by a string of length k or less is marked by the algorithm. Let q_i and q_j be states for which the shortest distinguishing string u has length $k + 1$. Then u can be written av and the computations with input u have the form $\hat{\delta}(q_i, u) = \hat{\delta}(q_i, av) = \hat{\delta}(q_m, v) = q_s$ 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

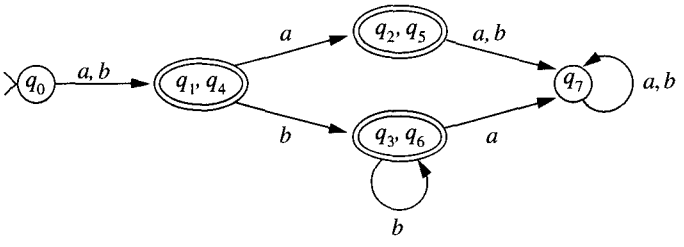


that accepts the language $(a \cup b)(a \cup b^*)$.

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

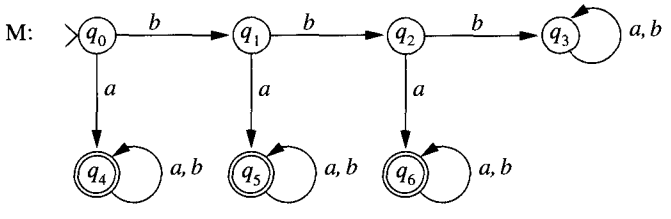
After each pair of indices is examined, [1,4], [2,5] and [3,6] are left as equivalent pairs of states. Merging these states produces the minimal state DFA M′ that accepts $(a \cup b)(a \cup b^*)$.



□

Example 6.7.2

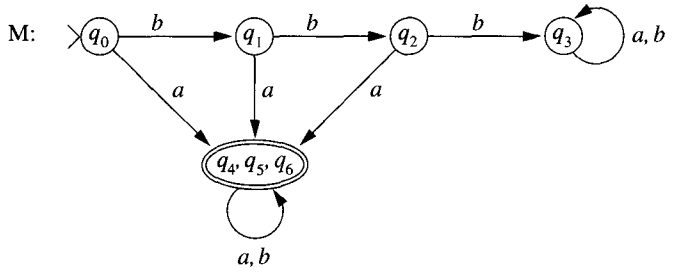
Minimizing the DFA M illustrates the recursive marking of states by the call to *DIST*. The language of M is $a(a \cup b)^* \cup ba(a \cup b)^* \cup bba(a \cup b)^*$.



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

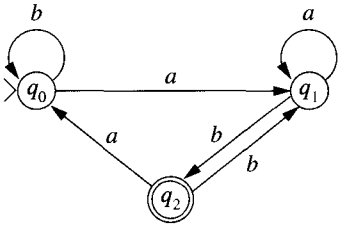
a) Give the state diagram of M.

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?

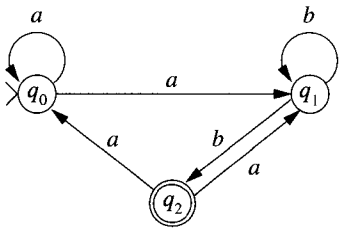
c) Give a regular expression for L(M).

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 $\hat{\delta} = \hat{\delta}'$.

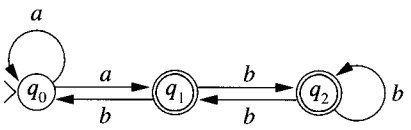
For Exercises 5 through 15, build a DFA that accepts the described language.

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.

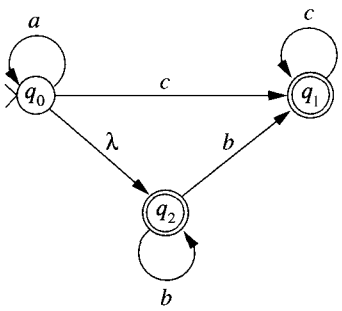
25. The set of strings over $\{a, b\}$ that contain an even number of substrings ba.

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



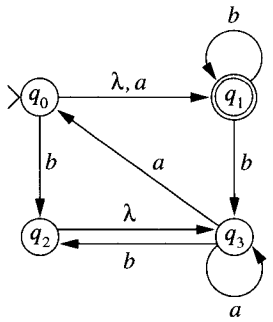
a) Compute λ-$closure(q_i)$ for $i = 0, 1, 2$.

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).

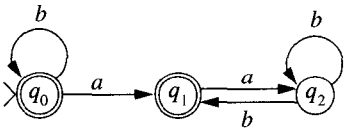
31. Give a recursive definition of the extended transition function $\hat{\delta}$ of an NFA-λ. The value $\hat{\delta}(q_i, w)$ is the set of states that can be reached by computations that begin at node q_i and completely process the string w.

32. Use Algorithm 6.6.3 to construct the state diagram of a DFA equivalent to the NFA in Example 6.5.2.

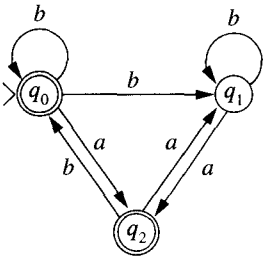
33. Use Algorithm 6.6.3 to construct the state diagram of a DFA equivalent to the NFA in Exercise 17.

34. For each of the following NFAs, use Algorithm 6.6.3 to construct the state diagram of an equivalent DFA.

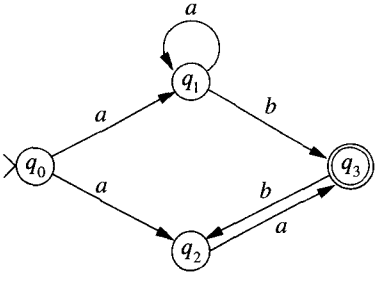
a)



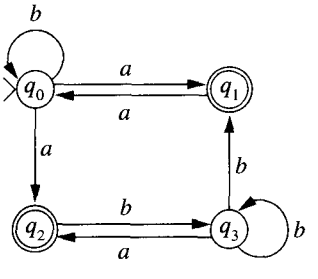
b)



c)

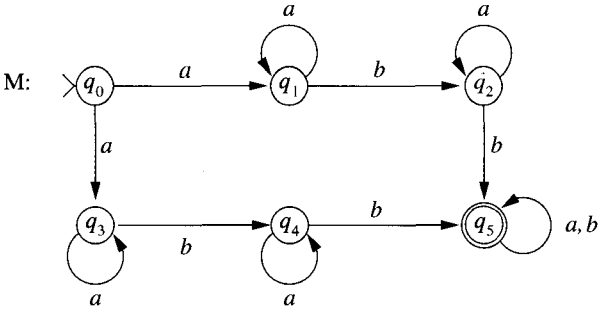


d)



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

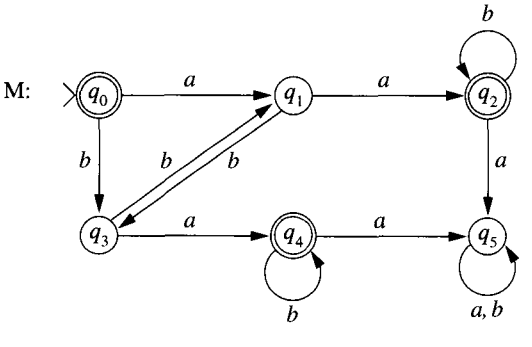


a) Trace the actions of Algorithm 6.7.2 to determine the equivalent states of M. Give the values of $D[i, j]$ and $S[i, j]$ computed by the algorithm.

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



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 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 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 G = (V, Σ, 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

$$\begin{aligned} &\text{F}(S) := \text{F}(S) \cup \mathit{trunc}_2(F'(A)\{\#\}\{\#\}) \\ &\text{F}(A) := \text{F}(A) \cup \mathit{trunc}_2(\{a\}\text{F}'(A)\{d\}) \cup \mathit{trunc}_2(\text{F}'(B)\text{F}'(C)) \\ &\text{F}(B) := \text{F}(B) \cup \mathit{trunc}_2(\{b\}\text{F}'(B)\{c\}) \\ &\text{F}(C) := \text{F}(C) \cup \mathit{trunc}_2(\{a\}\{c\}\text{F}'(C)) \cup \mathit{trunc}_2(\{a\}\{d\}) \end{aligned}$$

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 = (V, Σ, P, S) be a context-free grammar. Algorithm 16.4.1 generates the sets FIRST$_k(A)$, for every variable $A \in$ 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) $F(A) = FIRST_k(A)$: First we prove that $F(A) \subseteq FIRST_k(A)$ for all variables $A \in V$. To accomplish this we show that $F(A) \subseteq FIRST_k(A)$ at the beginning of each iteration of the repeat-until loop. By inspection, this inclusion holds prior to the first iteration. Assume $F(A) \subseteq FIRST_k(A)$ for all variables A after m iterations of the loop.

During the $m + 1$st iteration, the only additions to $F(A)$ come from assignment statements of the form

$$F(A) := F(A) \cup trunc_k(F'(u_1)F'(u_2) \ldots F'(u_n)),$$

where $A \to u_1u_2 \ldots u_n$ is a rule of G. By the inductive hypothesis, each of the sets $F'(u_i)$ is the subset of $FIRST_k(u_i)$. If u is added to $F(A)$ on the iteration then

$$\begin{aligned} u &\in trunc_k(F'(u_1)F'(u_2) \ldots F'(u_n)) \\ &\subseteq trunc_k(FIRST_k(u_1)FIRST_k(u_2) \ldots FIRST_k(u_n)) \\ &= FIRST_k(u_1u_2 \ldots u_n) \\ &\subseteq FIRST_k(A) \end{aligned}$$

and $u \in FIRST_k(A)$. The final two steps follow from parts 4 and 5 of Lemma 16.2.2.

We now show that $FIRST_k(A) \subseteq F(A)$ upon completion of the loop. Let $F_m(A)$ be the value of the set $F(A)$ 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 $F_m(A)$ for some $m > j$ then it is in $F_j(A)$. This follows since the sets $F(A)$ and $F'(A)$ would be identical for all iterations of the loop past iteration j. We will show that $FIRST_k(A) \subseteq F_j(A)$.

Let x be a string in $FIRST_k(A)$. Then there is a derivation $A \stackrel{m}{\Rightarrow} w$, where $w \in \sum^*$ and x is the prefix of w of length k. We show that $x \in F_m(A)$. 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 $A \to w \in P$, then x is added to $F_1(A)$.

Assume that $trunc_k(\{w \mid A \stackrel{m}{\Rightarrow} w \in \Sigma^*\}) \subseteq F_m(A)$ for all variables A in V. Let $x \in trunc_k(\{w \mid A \stackrel{m+1}{\Longrightarrow} w \in \Sigma^*\})$; that is, x is a prefix of terminal string derivable from A by $m + 1$ rule applications. We will show that $x \in F_{m+1}(A)$. The derivation of w can be written

$$A \Rightarrow u_1u_2 \ldots u_n \stackrel{m}{\Rightarrow} x_1x_2 \ldots x_n = w,$$

where $u_i \in V \cup \Sigma$ and $u_i \stackrel{*}{\Rightarrow} x_i$. Clearly, each subderivation $u_i \stackrel{*}{\Rightarrow} x_i$ has length less than $m + 1$. By the inductive hypothesis, the string obtained by truncating x_i at length k is in $F_m(u_i)$.

On the $m + 1$st iteration, $F_{m+1}(A)$ is augmented with the set

$$trunc_k(F'_{m+1}(u_1) \ldots F'_{m+1}(u_n)) = trunc_k(F_m(u_1) \ldots F_m(u_n)).$$

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
$S \rightarrow A\#\#$	$\text{FL}(A) := \text{FL}(A) \cup \mathit{trunc}_2(\{\#\#\}\text{FL}'(S))$
$A \rightarrow aAd$	$\text{FL}(A) := \text{FL}(A) \cup \mathit{trunc}_2(\{d\}\text{FL}'(A))$
$A \rightarrow BC$	$\text{FL}(C) := \text{FL}(C) \cup \text{FL}'(A)$
	$\text{FL}(B) := \text{FL}(B) \cup \mathit{trunc}_2(\text{FIRST}_2(C)\text{FL}'(A))$
	$= \text{FL}(B) \cup \mathit{trunc}_2(\{ad, ac\}\text{FL}'(A))$
$B \rightarrow bBc$	$\text{FL}(B) := \text{FL}(B) \cup \mathit{trunc}_2(\{c\}\text{FL}'(B))$

The rule $C \rightarrow acC$ 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

	$\text{FL}(S)$	$\text{FL}(A)$	$\text{FL}(B)$	$\text{FL}(C)$
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.

$$\text{LA}_2(S \rightarrow A\#\#) = \{ad, bc, aa, ab, bb, ac\}$$

$$\text{LA}_2(A \rightarrow aAd) = \{aa, ab\}$$

$$\text{LA}_2(A \rightarrow BC) = \{bc, bb, ad, ac\}$$

$$\text{LA}_2(B \rightarrow bBc) = \{bb, bc\}$$

$$\text{LA}_2(B \rightarrow \lambda) = \{ad, ac, ca, cc\}$$

$$\text{LA}_2(C \rightarrow acC) = \{ac\}$$

$$\text{LA}_2(C \rightarrow ad) = \{ad\}$$

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.

	$\mathbf{F}(S)$	$\mathbf{F}(A)$	$\mathbf{F}(B)$	$\mathbf{F}(Z)$	$\mathbf{F}(Y)$	$\mathbf{F}(T)$
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.

$$\text{LA}_1(S \rightarrow A\#) = \{b, (\}$$
$$\text{LA}_1(A \rightarrow TB) = \{b, (\}$$
$$\text{LA}_1(B \rightarrow Z) = \{+\}$$
$$\text{LA}_1(B \rightarrow \lambda) = \{\#,)\}$$
$$\text{LA}_1(Z \rightarrow +TY) = \{+\}$$
$$\text{LA}_1(Z \rightarrow \lambda) = \{\#,)\}$$
$$\text{LA}_1(Y \rightarrow Z) = \{+\}$$
$$\text{LA}_1(Y \rightarrow \lambda) = \{\#,)\}$$
$$\text{LA}_1(T \rightarrow b) = \{b\}$$
$$\text{LA}_1(T \rightarrow (A)) = \{(\}$$

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

$$G_1: \; S \to Aabd \mid cAbcd$$

$$A \to a \mid b \mid \lambda$$

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.

$$LA(B \to ab) = ab(ab)^*cd \cup abb(ab)^*cd$$
$$LA(B \to a) = a(ab)^*cd \cup ab(ab)^*cd$$

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) $S \rightarrow aAB\#$
$A \rightarrow a \mid \lambda$
$B \rightarrow b \mid \lambda$

b) $S \rightarrow AB\#$
$A \rightarrow aAb \mid B$
$B \rightarrow aBc \mid \lambda$

c) $S \rightarrow ABC\#$
$A \rightarrow aA \mid \lambda$
$B \rightarrow bBc \mid \lambda$
$C \rightarrow cA \mid dB \mid \lambda$

d) $S \rightarrow aAd\#$
$A \rightarrow BCD$
$B \rightarrow bB \mid \lambda$
$C \rightarrow cC \mid \lambda$
$D \rightarrow bD \mid \lambda$

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) $S \rightarrow ABC\#\#$
$A \rightarrow aA \mid a$
$B \rightarrow bB \mid \lambda$
$C \rightarrow cC \mid a \mid b \mid c$

b) $S \rightarrow A\#\#$
$A \rightarrow bBA \mid BcAa \mid \lambda$
$B \rightarrow acB \mid b$

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) $S \rightarrow aSb \mid A$
$A \rightarrow aAc \mid \lambda$

b) $S \rightarrow A \mid B$
$A \rightarrow aAb \mid ab$
$B \rightarrow aBc \mid ac$

c) $S \rightarrow A$
$A \rightarrow aAb \mid B$
$B \rightarrow aB \mid a$

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

$$S \Rightarrow aA \overset{i}{\Rightarrow} a(ab)^iA \Rightarrow a(ab)^ibB \overset{j}{\Rightarrow} a(ab)^ibb^jBc^j \Rightarrow a(ab)^ibb^jbcc^j,$$

where $i, j \geq 0$. Derivations beginning with $S \rightarrow bB$ can be written

$$S \Rightarrow bB \overset{i}{\Rightarrow} bb^iBc^i \Rightarrow bb^ibcc^i.$$

The LR(0) contexts can be obtained from the sentential forms generated in the preceding derivations.

Rule	LR(0) Contexts
$S \rightarrow aA$	$\{aA\}$
$S \rightarrow bB$	$\{bB\}$
$A \rightarrow abA$	$\{a(ab)^iA \mid i > 0\}$
$A \rightarrow bB$	$\{a(ab)^ibB \mid i \geq 0\}$
$B \rightarrow bBc$	$\{a(ab)^ibb^jBc, bb^jBc \mid i \geq 0, j > 0\}$
$B \rightarrow bc$	$\{a(ab)^ibb^jc, bb^jc \mid i \geq 0, j > 0\}$

□

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.

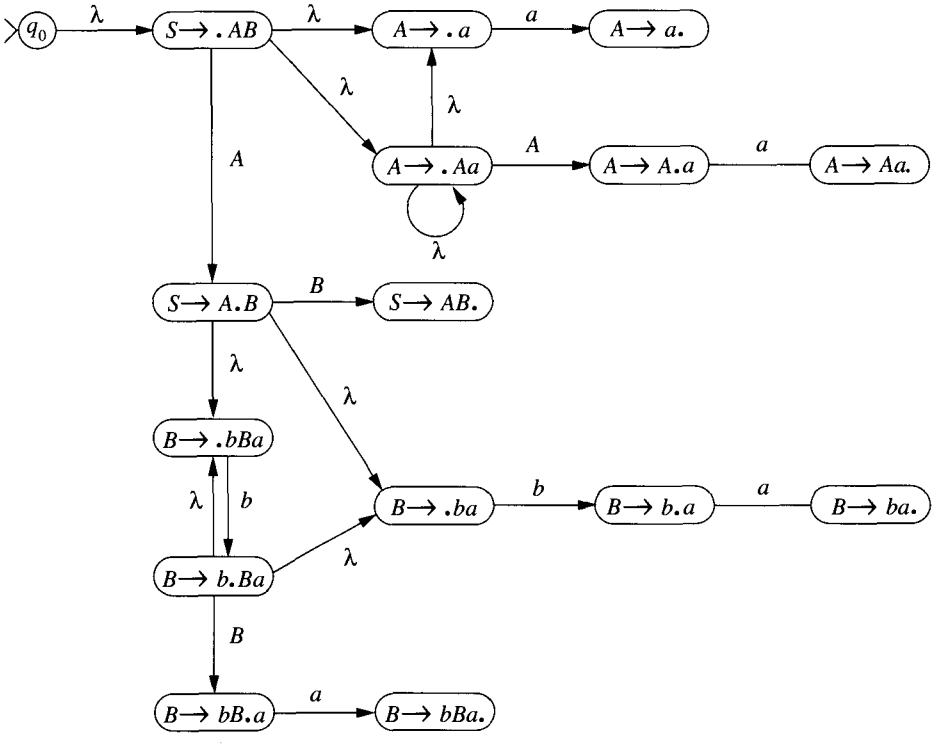


FIGURE 17.1 Nondeterministic LR(0) machine of G.

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

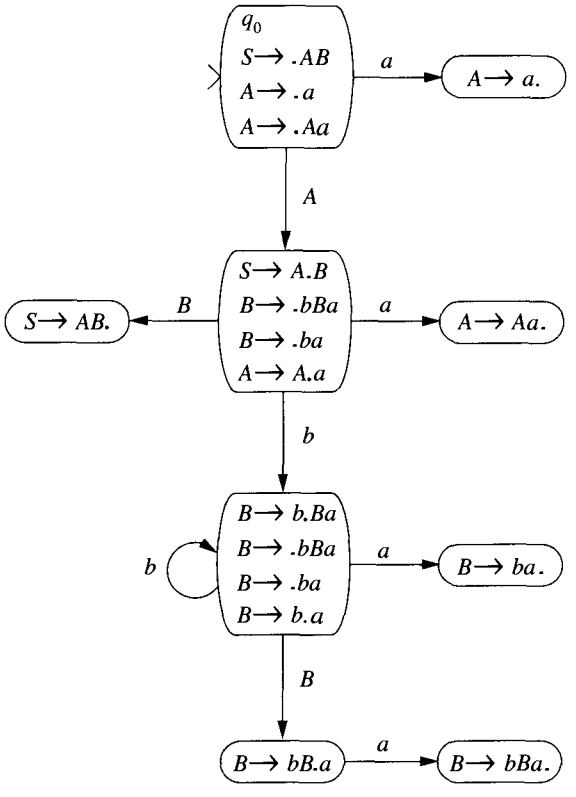


FIGURE 17.2 Deterministic LR(0) machine of G.

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

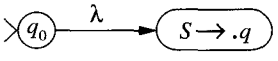
The LR(0) machine has been constructed to decide whether a string is a viable prefix of the grammar. Theorem 17.4.1 establishes that computations of the LR(0) machine provide the desired information.

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 $A \to uv$.

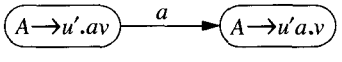
The basis consists of computations of length one. All such computations have the form



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.

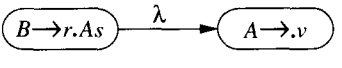
Case 1: $x = a \in \Sigma$: Then $u = u'a$. The final transition of the computation has the form



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 $C \in V$.

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 $C \in V$.

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 $B \to b$.

$$S \underset{R}{\Rightarrow} aABc \underset{R}{\Rightarrow} aAbc \underset{R}{\Rightarrow} abc$$

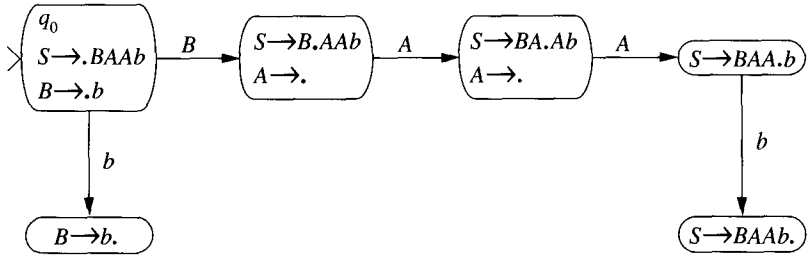
The string a is an LR(0) context of $A \rightarrow \lambda$ and a prefix of aAb, which is an LR(0) context of $B \rightarrow b$. The effect of reductions by lambda rules in an LR(0) parser is demonstrated in Example 17.4.1.

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 #

$$\begin{aligned} \text{AE: } & S \rightarrow A\# \\ & A \rightarrow A + T \mid T \\ & T \rightarrow b \mid (A) \end{aligned}$$

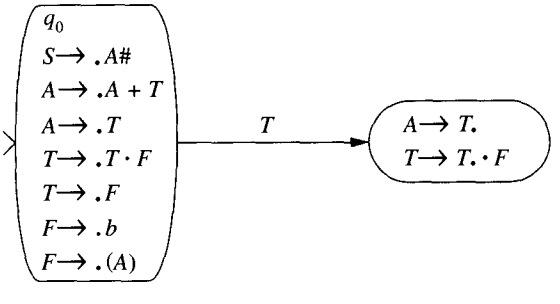
is LR(0). The deterministic LR(0) machine of AE is given in Figure 17.3. Since each of the states containing a complete item is a singleton set, the grammar is LR(0). □

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 $T \cdot F$. □

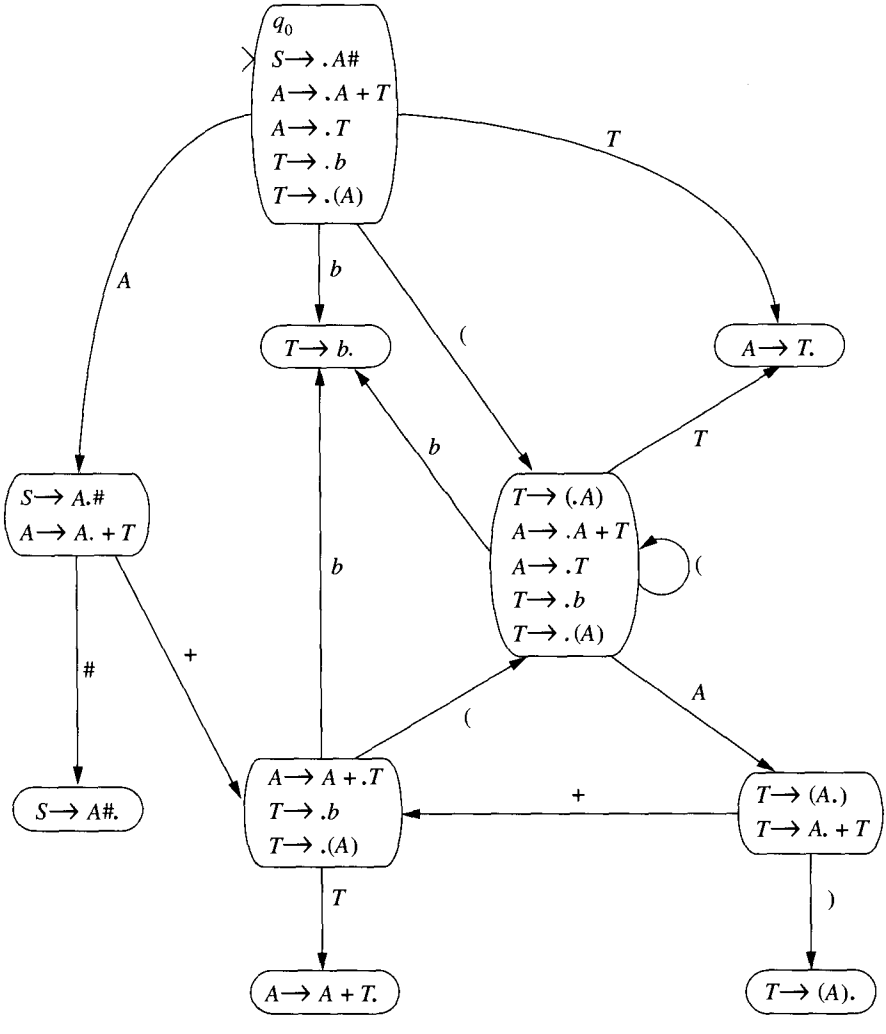


FIGURE 17.3 Deterministic LR(0) machine of AE with end marker.

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

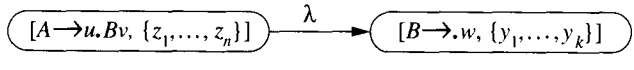
i) $\delta(q_0, \lambda) = [\{S \rightarrow .w, \{\lambda\}] \mid S \rightarrow w \in P\}$

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



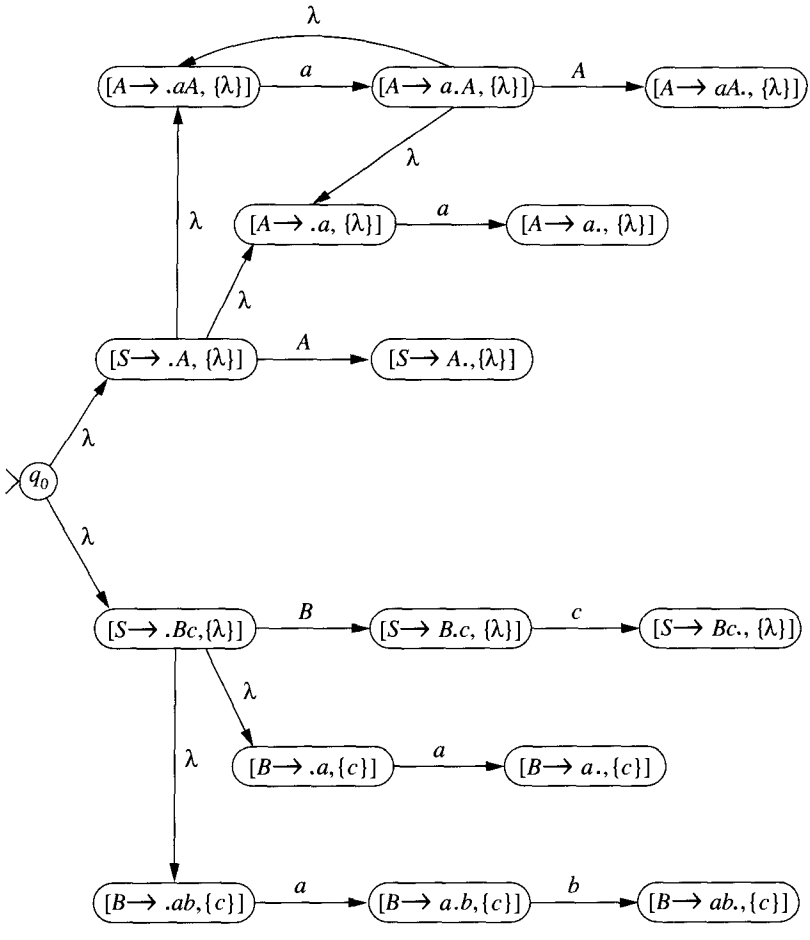


FIGURE 17.4 Nondeterministic LR(1) machine of G.

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.

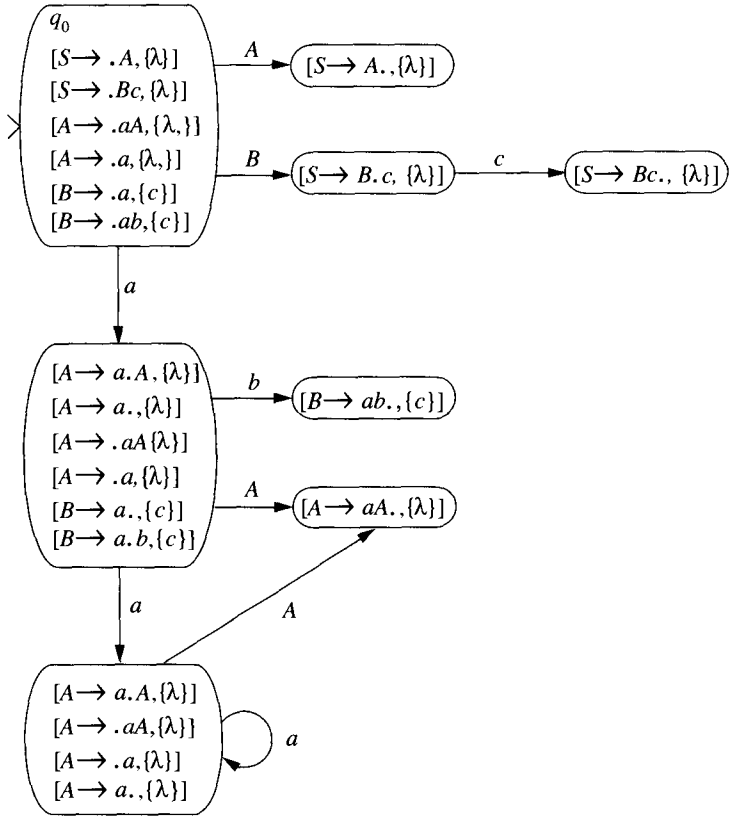


FIGURE 17.5 Deterministic LR(1) machine of G.

$$\begin{aligned} \text{G: } S &\to A \mid Bc \\ A &\to aA \mid a \\ B &\to a \mid ab \end{aligned}$$

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

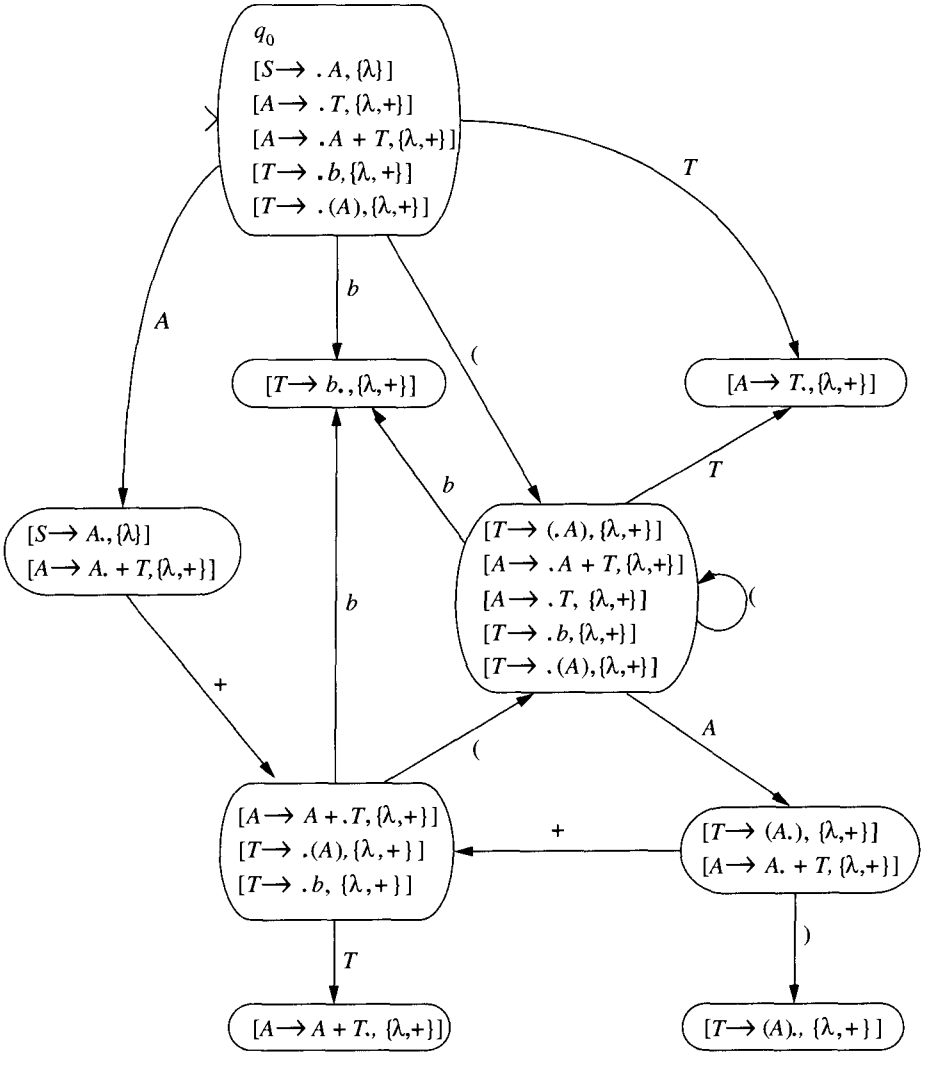
Let G be a context-free grammar with a nonrecursive start symbol. The grammar G is **LR(1)** if the extended transition function $\hat{\delta}$ of the deterministic LR(1) machine of G satisfies the following conditions:

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

$$
\begin{aligned}
\text{G: } & S \to A \mid Bc \\
& A \to aA \mid a \\
& B \to a \mid ab.
\end{aligned}
$$

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	$\hat{\delta}(q_s, \lambda) =$	$\{[S \to .A, \{\lambda\}],$	shift
				$[S \to .Bc, \{\lambda\}],$	
				$[A \to .aA, \{\lambda\}],$	
				$[A \to .a, \{\lambda\}],$	
				$[B \to .a\ \{c\}],$	
				$[B \to .ab\ \{c\}]\}$	
a	c	λ	$\hat{\delta}(q_s, a) =$	$\{[A \to a.A, \{\lambda\}],$	reduce
				$[A \to a., \{\lambda\}],$	
				$[A \to .aA, \{\lambda\}],$	
				$[A \to .a, \{\lambda\}],$	
				$[B \to a., \{c\}],$	
				$[B \to a.b, \{c\}]\}$	
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

9 780201 821369 90000
ISBN 0-201-82136-2