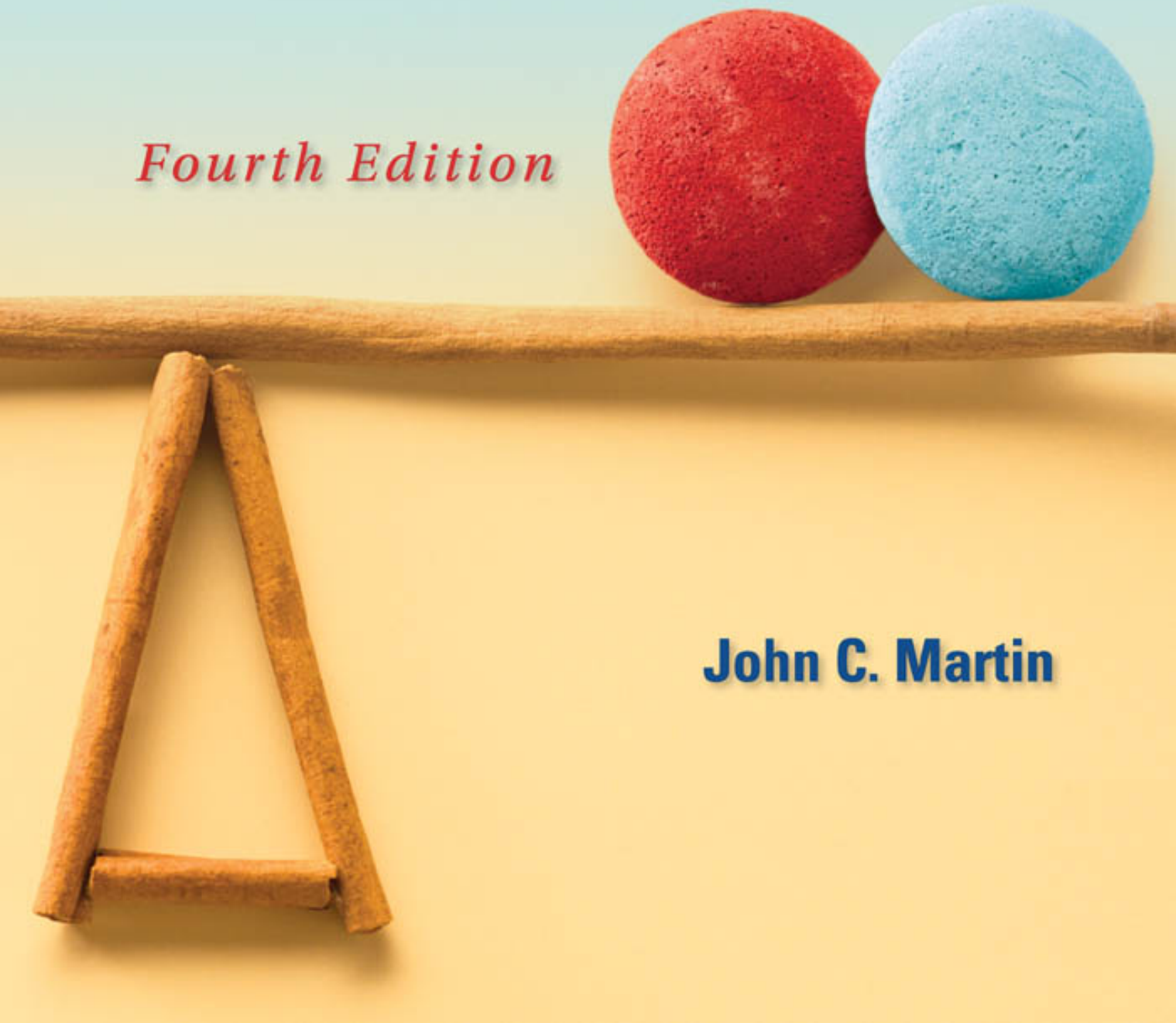
Introduction to Languages *and the* Theory of Computation

Fourth Edition



John C. Martin

Introduction to Languages and The Theory of Computation

Fourth Edition

John C. Martin
North Dakota State University

The McGraw-Hill Companies

INTRODUCTION TO LANGUAGES AND THE THEORY OF COMPUTATION, FOURTH EDITION

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 1 0 9 8 7 6 5 4 3 2 1 0

ISBN 978–0–07–319146–1
MHID 0–07–319146–9

Vice President & Editor-in-Chief: *Marty Lange*
Vice President, EDP: *Kimberly Meriwether David*
Global Publisher: *Raghothaman Srinivasan*
Director of Development: *Kristine Tibbetts*
Senior Marketing Manager: *Curt Reynolds*
Senior Project Manager: *Joyce Watters*
Senior Production Supervisor: *Laura Fuller*
Senior Media Project Manager: *Tammy Juran*
Design Coordinator: *Brenda A. Rolwes*
Cover Designer: *Studio Montage, St. Louis, Missouri*
(USE) Cover Image: © *Getty Images*
Compositor: *Laserwords Private Limited*
Typeface: *10/12 Times Roman*
Printer: *R. R. Donnelley*

All credits appearing on page or at the end of the book are considered to be an extension of the copyright page.

Library of Congress Cataloging-in-Publication Data

Martin, John C.
Introduction to languages and the theory of computation / John C. Martin.—4th ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-07-319146-1 (alk. paper)
1. Sequential machine theory. 2. Computable functions. I. Title.
QA267.5.S4M29 2010
511.3′5–dc22

2009040831

www.mhhe.com

To the memory of

Mary Helen Baldwin Martin, 1918–2008
D. Edna Brown, 1927–2007

and to

John C. Martin
Dennis S. Brown

CONTENTS

PREFACE

This book is an introduction to the theory of computation. After a chapter presenting the mathematical tools that will be used, the book examines models of computation and the associated languages, from the most elementary to the most general: finite automata and regular languages; context-free languages and pushdown automata; and Turing machines and recursively enumerable and recursive languages. There is a chapter on decision problems, reductions, and undecidability, one on the Kleene approach to computability, and a final one that introduces complexity and *NP*-completeness.

Specific changes from the third edition are described below. Probably the most noticeable difference is that this edition is shorter, with three fewer chapters and fewer pages. Chapters have generally been rewritten and reorganized rather than omitted. The reduction in length is a result not so much of leaving out topics as of trying to write and organize more efficiently. My overall approach continues to be to rely on the clarity and efficiency of appropriate mathematical language and to add informal explanations to ease the way, not to substitute for the mathematical language but to familiarize it and make it more accessible. Writing "more efficiently" has meant (among other things) limiting discussions and technical details to what is necessary for the understanding of an idea, and reorganizing or replacing examples so that each one contributes something not contributed by earlier ones.

In each chapter, there are several exercises or parts of exercises marked with a (†). These are problems for which a careful solution is likely to be less routine or to require a little more thought.

Previous editions of the text have been used at North Dakota State in a two-semester sequence required of undergraduate computer science majors. A one-semester course could cover a few essential topics from Chapter 1 and a substantial portion of the material on finite automata and regular languages, context-free languages and pushdown automata, and Turing machines. A course on Turing machines, computability, and complexity could cover Chapters 7–11.

As I was beginning to work on this edition, reviewers provided a number of thoughtful comments on both the third edition and a sample chapter of the new one. I appreciated the suggestions, which helped me in reorganizing the first few chapters and the last chapter and provided a few general guidelines that I have tried to keep in mind throughout. I believe the book is better as a result. Reviewers to whom I am particularly grateful are Philip Bernhard, Florida Institute of Technology; Albert M. K. Cheng, University of Houston; Vladimir Filkov, University of California-Davis; Mukkai S. Krishnamoorthy, Rensselaer Polytechnic University; Gopalan Nadathur, University of Minnesota; Prakash Panangaden, McGill University; Viera K. Proulx, Northeastern University; Sing-Ho Sze, Texas A&M University; and Shunichi Toida, Old Dominion University.

I have greatly enjoyed working with Melinda Bilecki again, and Raghu Srinivasan at McGraw-Hill has been very helpful and understanding. Many thanks to Michelle Gardner, of Laserwords Maine, for her attention to detail and her unfailing cheerfulness. Finally, one more thank-you to my long-suffering wife, Pippa.

What's New in This Edition

The text has been substantially rewritten, and only occasionally have passages from the third edition been left unchanged. Specific organizational changes include the following.

1. One introductory chapter, "Mathematical Tools and Techniques," replaces Chapters 1 and 2 of the third edition. Topics in discrete mathematics in the first few sections have been limited to those that are used directly in subsequent chapters. Chapter 2 in the third edition, on mathematical induction and recursive definitions, has been shortened and turned into the last two sections of Chapter 1. The discussion of induction emphasizes "structural induction" and is tied more directly to recursive definitions of sets, of which the definition of the set of natural numbers is a notable example. In this way, the overall unity of the various approaches to induction is clarified, and the approach is more consistent with subsequent applications in the text.
2. Three chapters on regular languages and finite automata have been shortened to two. Finite automata are now discussed first; the first of the two chapters begins with the model of computation and collects into one chapter the topics that depend on the devices rather than on features of regular expressions. Those features, along with the nondeterminism that simplifies the proof of Kleene's theorem, make up the other chapter. Real-life examples of both finite automata and regular expressions have been added to these chapters.
3. In the chapter introducing Turing machines, there is slightly less attention to the "programming" details of Turing machines and more emphasis on their role as a general model of computation. One way that Chapters 8 and 9 were shortened was to rely more on the Church-Turing thesis in the presentation of an algorithm rather than to describe in detail the construction of a Turing machine to carry it out.
4. The two chapters on computational complexity in the third edition have become one, the discussion focuses on time complexity, and the emphasis has been placed on polynomial-time decidability, the sets P and NP, and NP-completeness. A section has been added that characterizes NP in terms of polynomial-time verifiability, and an introductory example has been added to clarify the proof of the Cook-Levin theorem, in order to illustrate the idea of the proof.
5. In order to make the book more useful to students, a section has been added at the end that contains solutions to selected exercises. In some cases these are exercises representative of a general class of problems; in other cases the

solutions may suggest approaches or techniques that have not been discussed in the text. An exercise or part of an exercise for which a solution is provided will have the exercise number highlighted in the chapter.

PowerPoint slides accompanying the book will be available on the McGraw-Hill website at http://mhhe.com/martin, and solutions to most of the exercises will be available to authorized instructors. In addition, the book will be available in e-book format, as described in the paragraph below.

John C. Martin

Electronic Books

If you or your students are ready for an alternative version of the traditional textbook, McGraw-Hill has partnered with CourseSmart to bring you an innovative and inexpensive electronic textbook. Students can save up to 50% off the cost of a print book, reduce their impact on the environment, and gain access to powerful Web tools for learning, including full text search, notes and highlighting, and email tools for sharing notes between classmates. eBooks from McGraw-Hill are smart, interactive, searchable, and portable.

To review comp copies or to purchase an eBook, go to either www.CourseSmart.com <http://www.coursesmart.com/>.

Tegrity

Tegrity Campus is a service that makes class time available all the time by automatically capturing every lecture in a searchable format for students to review when they study and complete assignments. With a simple one-click start and stop process, you capture all computer screens and corresponding audio. Students replay any part of any class with easy-to-use browser-based viewing on a PC or Mac.

Educators know that the more students can see, hear, and experience class resources, the better they learn. With Tegrity Campus, students quickly recall key moments by using Tegrity Campus's unique search feature. This search helps students efficiently find what they need, when they need it, across an entire semester of class recordings. Help turn all your students' study time into learning moments immediately supported by your lecture.

To learn more about Tegrity, watch a 2-minute Flash demo at http://tegritycampus.mhhe.com

INTRODUCTION

Computers play such an important part in our lives that formulating a "theory of computation" threatens to be a huge project. To narrow it down, we adopt an approach that seems a little old-fashioned in its simplicity but still allows us to think systematically about what computers do. Here is the way we will think about a computer: It receives some input, in the form of a string of characters; it performs some sort of "computation"; and it gives us some output.

In the first part of this book, it's even simpler than that, because the questions we will be asking the computer can all be answered either yes or no. For example, we might submit an input string and ask, "Is it a legal algebraic expression?" At this point the computer is playing the role of a *language acceptor*. The language accepted is the set of strings to which the computer answers yes—in our example, the language of legal algebraic expressions. Accepting a language is approximately the same as solving a *decision problem*, by receiving a string that represents an *instance* of the problem and answering either yes or no. Many interesting computational problems can be formulated as decision problems, and we will continue to study them even after we get to models of computation that are capable of producing answers more complicated than yes or no.

If we restrict ourselves for the time being, then, to computations that are supposed to solve decision problems, or to accept languages, then we can adjust the level of complexity of our model in one of two ways. The first is to vary the problems we try to solve or the languages we try to accept, and to formulate a model appropriate to the level of the problem. Accepting the language of legal algebraic expressions turns out to be moderately difficult; it can't be done using the first model of computation we discuss, but we will get to it relatively early in the book. The second approach is to look at the computations themselves: to say at the outset how sophisticated the steps carried out by the computer are allowed to be, and to see what sorts of languages can be accepted as a result. Our first model, a *finite automaton*, is characterized by its lack of any auxiliary memory, and a language accepted by such a device can't require the acceptor to remember very much information during its computation.

A finite automaton proceeds by moving among a finite number of distinct states in response to input symbols. Whenever it reaches an *accepting* state, we think of it as giving a "yes" answer for the string of input symbols it has received so far. Languages that can be accepted by finite automata are regular languages; they can be described by either *regular expressions* or *regular grammars*, and generated by combining one-element languages using certain simple operations. One step up from a finite automaton is a *pushdown automaton*, and the languages these devices accept can be generated by more general grammars called *context-free* grammars. Context-free grammars can describe much of the syntax of high-level programming

languages, as well as related languages like legal algebraic expressions and balanced strings of parentheses. The most general model of computation we will study is the Turing machine, which can in principle carry out any algorithmic procedure. It is as powerful as any computer. Turing machines accept recursively enumerable languages, and one way of generating these is to use *unrestricted* grammars.

Turing machines do not represent the only general model of computation, and in Chapter 10 we consider Kleene's alternative approach to computability. The class of computable functions, which turn out to be the same as the Turing-computable ones, can be described by specifying a set of "initial" functions and a set of operations that can be applied to functions to produce new ones. In this way the computable functions can be characterized in terms of the operations that can actually be carried out algorithmically.

As powerful as the Turing machine model is potentially, it is not especially user-friendly, and a Turing machine leaves something to be desired as an actual computer. However, it can be used as a yardstick for comparing the inherent complexity of one solvable problem to that of another. A simple criterion involving the number of steps a Turing machine needs to solve a problem allows us to distinguish between problems that can be solved in a reasonable time and those that can't. At least, it allows us to distinguish between these two categories in principle; in practice it can be very difficult to determine which category a particular problem is in. In the last chapter, we discuss a famous open question in this area, and look at some of the ways the question has been approached.

The fact that these elements (abstract computing devices, languages, and various types of grammars) fit together so nicely into a theory is reason enough to study them—for people who enjoy theory. If you're not one of those people, or have not been up to now, here are several other reasons.

The algorithms that finite automata can execute, although simple by definition, are ideally suited for some computational problems—they might be the algorithms of choice, even if we have computers with lots of horsepower. We will see examples of these algorithms and the problems they can solve, and some of them are directly useful in computer science. Context-free grammars and pushdown automata are used in software form in compiler design and other eminently practical areas.

A model of computation that is inherently simple, such as a finite automaton, is one we can understand thoroughly and describe precisely, using appropriate mathematical notation. Having a firm grasp of the principles governing these devices makes it easier to understand the notation, which we can then apply to more complicated models of computation.

A Turing machine is simpler than any actual computer, because it is abstract. We can study it, and follow its computation, without becoming bogged down by hardware details or memory restrictions. A Turing machine is an implementation of an algorithm. Studying one in detail is equivalent to studying an algorithm, and studying them in general is a way of studying the algorithmic method. Having a precise model makes it possible to identify certain types of computations that Turing

machines cannot carry out. We said earlier that Turing machines accept recursively enumerable languages. These are not *all* languages, and Turing machines can't solve every problem. When we find a problem a finite automaton can't solve, we can look for a more powerful type of computer, but when we find a problem that can't be solved by a Turing machine (and we will discuss several examples of such "undecidable" problems), we have found a limitation of the algorithmic method.

CHAPTER

1

Mathematical Tools and Techniques

When we discuss formal languages and models of computation, the definitions will rely mostly on familiar mathematical objects (logical propositions and operators, sets, functions, and equivalence relations) and the discussion will use common mathematical techniques (elementary methods of proof, recursive definitions, and two or three versions of mathematical induction). This chapter lays out the tools we will be using, introduces notation and terminology, and presents examples that suggest directions we will follow later.

The topics in this chapter are all included in a typical beginning course in discrete mathematics, but you may be more familiar with some than with others. Even if you have had a discrete math course, you will probably find it helpful to review the first three sections. You may want to pay a little closer attention to the last three, in which many of the approaches that characterize the subjects in this course first start to show up.

1.1 | LOGIC AND PROOFS

In this first section, we consider some of the ingredients used to construct logical arguments. Logic involves *propositions*, which have *truth values*, either the value *true* or the value *false*. The propositions "$0 = 1$" and "peanut butter is a source of protein" have truth values *false* and *true*, respectively. When a simple proposition, which has no variables and is not constructed from other simpler propositions, is used in a logical argument, its truth value is the only information that is relevant.

A proposition involving a variable (a *free* variable, terminology we will explain shortly) may be true or false, depending on the value of the variable. If the domain, or set of possible values, is taken to be $\mathcal{N}$, the set of nonnegative integers, the proposition "$x - 1$ is prime" is true for the value $x = 8$ and false when $x = 10$.

Compound propositions are constructed from simpler ones using *logical connectives*. We will use five connectives, which are shown in the table below. In each case, p and q are assumed to be propositions.

Connective	Symbol	Typical Use	English Translation
conjunction	$\wedge$	$p \wedge q$	p and q
disjunction	$\vee$	$p \vee q$	p or q
negation	$\neg$	$\neg p$	not p
conditional	$\rightarrow$	$p \rightarrow q$	if p then q p only if q
biconditional	$\leftrightarrow$	$p \leftrightarrow q$	p if and only if q

Each of these connectives is defined by saying, for each possible combination of truth values of the propositions to which it is applied, what the truth value of the result is. The truth value of $\neg p$ is the opposite of the truth value of p. For the other four, the easiest way to present this information is to draw a *truth table* showing the four possible combinations of truth values for p and q.

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

Many of these entries don't require much discussion. The proposition $p \wedge q$ ("p and q") is true when both p and q are true and false in every other case. "p or q" is true if either or both of the two propositions p and q are true, and false only when they are both false.

The conditional proposition $p \rightarrow q$, "if p then q", is defined to be false when p is true and q is false; one way to understand why it is defined to be true in the other cases is to consider a proposition like

$$x < 1 \rightarrow x < 2$$

where the domain associated with the variable x is the set of natural numbers. It sounds reasonable to say that this proposition ought to be true, no matter what value is substituted for x, and you can see that there is no value of x that makes $x < 1$ true and $x < 2$ false. When $x = 0$, both $x < 1$ and $x < 2$ are true; when $x = 1$, $x < 1$ is false and $x < 2$ is true; and when $x = 2$, both $x < 1$ and $x < 2$ are false; therefore, the truth table we have drawn is the only possible one if we want this compound proposition to be true in every case.

In English, the word order in a conditional statement can be changed without changing the meaning. The proposition $p \rightarrow q$ can be read either "if p then q" or "q if p". In both cases, the "if" comes right before p. The other way to read $p \rightarrow q$, "p only if q", may seem confusing until you realize that "only if" and "if" mean different things. The English translation of the biconditional statement

$p \leftrightarrow q$ is a combination of "p if q" and "p only if q". The statement is true when the truth values of p and q are the same and false when they are different.

Once we have the truth tables for the five connectives, finding the truth values for an arbitrary compound proposition constructed using the five is a straightforward operation. We illustrate the process for the proposition

$$(p \vee q) \wedge \neg(p \rightarrow q)$$

We begin filling in the table below by entering the values for p and q in the two leftmost columns; if we wished, we could copy one of these columns for each occurrence of p or q in the expression. The order in which the remaining columns are filled in (shown at the top of the table) corresponds to the order in which the operations are carried out, which is determined to some extent by the way the expression is parenthesized.

		1	**4**	**3**	**2**
p	q	$(p \vee q)$	$\wedge$	$\neg$	$(p \rightarrow q)$
T	T	T	F	F	T
T	F	T	T	T	F
F	T	T	F	F	T
F	F	F	F	F	T

The first two columns to be computed are those corresponding to the subexpressions $p \vee q$ and $p \rightarrow q$. Column 3 is obtained by negating column 2, and the final result in column 4 is obtained by combining columns 1 and 3 using the $\wedge$ operation.

A *tautology* is a compound proposition that is true for every possible combination of truth values of its constituent propositions—in other words, true in every case. A *contradiction* is the opposite, a proposition that is false in every case. The proposition $p \vee \neg p$ is a tautology, and $p \wedge \neg p$ is a contradiction. The propositions p and $\neg p$ by themselves, of course, are neither.

According to the definition of the biconditional connective, $p \leftrightarrow q$ is true precisely when p and q have the same truth values. One type of tautology, therefore, is a proposition of the form $P \leftrightarrow Q$, where P and Q are compound propositions that are *logically equivalent*—i.e., have the same truth value in every possible case. Every proposition appearing in a formula can be replaced by any other logically equivalent proposition, because the truth value of the entire formula remains unchanged. We write $P \Leftrightarrow Q$ to mean that the compound propositions P and Q are logically equivalent. A related idea is *logical implication*. We write $P \Rightarrow Q$ to mean that in every case where P is true, Q is also true, and we describe this situation by saying that P logically implies Q.

The proposition $P \rightarrow Q$ and the assertion $P \Rightarrow Q$ look similar but are different kinds of things. $P \rightarrow Q$ is a proposition, just like P and Q, and has a truth value in each case. $P \Rightarrow Q$ is a "meta-statement", an assertion about the relationship between the two propositions P and Q. Because of the way we have defined the conditional, the similarity between them can be accounted for by observing

that $P \Rightarrow Q$ means $P \rightarrow Q$ is a tautology. In the same way, as we have already observed, $P \Leftrightarrow Q$ means that $P \leftrightarrow Q$ is a tautology.

There is a long list of logical identities that can be used to simplify compound propositions. We list just a few that are particularly useful; each can be verified by observing that the truth tables for the two equivalent statements are the same.

The commutative laws:	$p \vee q \Leftrightarrow q \vee p$
	$p \wedge q \Leftrightarrow q \wedge p$
The associative laws:	$p \vee (q \vee r) \Leftrightarrow (p \vee q) \vee r$
	$p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$
The distributive laws:	$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
	$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$
The De Morgan laws:	$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$
	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$

Here are three more involving the conditional and biconditional.

$$(p \rightarrow q) \Leftrightarrow (\neg p \vee q)$$
$$(p \rightarrow q) \Leftrightarrow (\neg q \rightarrow \neg p)$$
$$(p \leftrightarrow q) \Leftrightarrow ((p \rightarrow q) \wedge (q \rightarrow p))$$

The first and third provide ways of expressing $\rightarrow$ and $\leftrightarrow$ in terms of the three simpler connectives $\vee$, $\wedge$, and $\neg$. The second asserts that the conditional proposition $p \rightarrow q$ is equivalent to its *contrapositive*. The *converse* of $p \rightarrow q$ is $q \rightarrow p$, and these two propositions are not equivalent, as we suggested earlier in discussing *if* and *only if*.

We interpret a proposition such as "$x - 1$ is prime", which we considered earlier, as a statement about x, which may be true or false depending on the value of x. There are two ways of attaching a *logical quantifier* to the beginning of the proposition; we can use the universal quantifier "for every", or the existential quantifier "for some". We will write the resulting quantified statements as

$$\forall x(x - 1 \text{ is prime})$$
$$\exists x(x - 1 \text{ is prime})$$

In both cases, what we have is no longer a statement about x, which still appears but could be given another name without changing the meaning, and it no longer makes sense to substitute an arbitrary value for x. We say that x is no longer a free variable, but is bound to the quantifier. In effect, the statement has become a statement about the domain from which possible values may be chosen for x. If as before we take the domain to be the set $\mathcal{N}$ of nonnegative integers, the first statement is false, because "$x - 1$ is prime" is not true for every x in the domain (it is false when $x = 10$). The second statement, which is often read "there exists x such that $x - 1$ is prime", is true; for example, $8 - 1$ is prime.

An easy way to remember the notation for the two quantifiers is to think of $\forall$ as an upside-down A, for "all", and to think of $\exists$ as a backward E, for "exists". Notation for quantified statements sometimes varies; we use parentheses

in order to specify clearly the *scope* of the quantifier, which in our example is the statement "$x - 1$ is prime". If the quantified statement appears within a larger formula, then an appearance of x outside the scope of this quantifier means something different.

We assume, unless explicitly stated otherwise, that in statements containing two or more quantifiers, the same domain is associated with all of them. Being able to understand statements of this sort requires paying particular attention to the scope of each quantifier. For example, the two statements

$$\forall x(\exists y((x < y))$$
$$\exists y(\forall x((x < y))$$

are superficially similar (the same variables are bound to the same quantifiers, and the inequalities are the same), but the statements do not express the same idea. The first says that for every x, there is a y that is larger. This is true if the domain in both cases is $\mathcal{N}$, for example. The second, on the other hand, says that there is a single y such that no matter what x is, x is smaller than y. This statement is false, for the domain $\mathcal{N}$ and every other domain of numbers, because if it were true, one of the values of x that would have to be smaller than y is y itself. The best way to explain the difference is to observe that in the first case the statement $\exists y(x < y)$ is within the scope of $\forall x$, so that the correct interpretation is "there exists y, which may depend on x".

Manipulating quantified statements often requires negating them. If it is not the case that for every x, $P(x)$, then there must be some value of x for which $P(x)$ is not true. Similarly, if there does not exist an x such that $P(x)$, then $P(x)$ must fail for every x. The general procedure for negating a quantifed statement is to reverse the quantifier (change $\forall$ to $\exists$, and vice versa) and move the negation inside the quantifier. $\neg(\forall x(P(x)))$ is the same as $\exists x(\neg P(x))$, and $\neg(\exists x(P(x)))$ is the same as $\forall x(\neg P(x))$. In order to negate a statement with several nested quantifiers, such as

$$\forall x(\exists y(\forall z(P(x, y, z))))$$

apply the general rule three times, moving from the outside in, so that the final result is

$$\exists x(\forall y(\exists z(\neg P(x, y, z))))$$

We have used "$\exists x(x - 1$ is prime)" as an example of a quantified statement. To conclude our discussion of quantifiers, we consider how to express the statement "x is prime" itself using quantifiers, where again the domain is the set $\mathcal{N}$. A prime is an integer greater than 1 whose only divisors are 1 and itself; the statement "x is prime" can be formulated as "$x > 1$, and for every k, if k is a divisor of x, then either k is 1 or k is x". Finally, the statement "k is a divisor of x" means that there is an integer m with $x = m * k$. Therefore, the statement we are looking for can be written

$$(x > 1) \wedge \forall k((\exists m(x = m * k)) \rightarrow (k = 1 \vee k = x))$$

A typical step in a *proof* is to derive a statement from initial assumptions and hypotheses, or from statements that have been derived previously, or from other generally accepted facts, using principles of logical reasoning. The more formal the proof, the stricter the criteria regarding what facts are "generally accepted", what principles of reasoning are allowed, and how carefully they are elaborated.

You will not learn how to write proofs just by reading this section, because it takes a lot of practice and experience, but we will illustrate a few basic proof techniques in the simple proofs that follow.

We will usually be trying to prove a statement, perhaps with a quantifier, involving a conditional proposition $p \rightarrow q$. The first example is a *direct* proof, in which we assume that p is true and derive q. We begin with the definitions of odd integers, which appear in this example, and even integers, which will appear in Example 1.3.

An integer n is odd if there exists an integer k so that $n = 2k + 1$.

An integer n is even if there exists an integer k so that $n = 2k$.

In Example 1.3, we will need the fact that every integer is either even or odd and no integer can be both (see Exercise 1.51).

EXAMPLE 1.1 The Product of Two Odd Integers Is Odd

To Prove: For every two integers a and b, if a and b are odd, then ab is odd.

■ **Proof**

The conditional statement can be restated as follows: If there exist integers i and j so that $a = 2i + 1$ and $b = 2j + 1$, then there exists an integer k so that $ab = 2k + 1$. Our proof will be *constructive*—not only will we show that there exists such an integer k, but we will demonstrate how to construct it. Assuming that $a = 2i + 1$ and $b = 2j + 1$, we have

$$\begin{aligned} ab &= (2i + 1)(2j + 1) \\ &= 4ij + 2i + 2j + 1 \\ &= 2(2ij + i + j) + 1 \end{aligned}$$

Therefore, if we let $k = 2ij + i + j$, we have the result we want, $ab = 2k + 1$.

An important point about this proof, or any proof of a statement that begins "for every", is that a "proof by example" is not sufficient. An example can constitute a proof of a statement that begins "there exists", and an example can disprove a statement beginning "for every", by serving as a counterexample, but the proof above makes no assumptions about a and b except that each is an odd integer.

Next we present examples illustrating two types of *indirect proofs*, proof by contrapositive and proof by contradiction.

Proof by Contrapositive — EXAMPLE 1.2

To Prove: For every three positive integers i, j, and n, if $ij = n$, then $i \leq \sqrt{n}$ or $j \leq \sqrt{n}$.

■ Proof

The conditional statement $p \rightarrow q$ inside the quantifier is logically equivalent to its contrapositive, and so we start by assuming that there exist values of i, j, and n such that

$$\text{not } (i \leq \sqrt{n} \text{ or } j \leq \sqrt{n})$$

According to the De Morgan law, this implies

$$\text{not } (i \leq \sqrt{n}) \text{ and not } (j \leq \sqrt{n})$$

which in turn implies $i > \sqrt{n}$ and $j > \sqrt{n}$. Therefore,

$$ij > \sqrt{n}\sqrt{n} = n$$

which implies that $ij \neq n$. We have constructed a direct proof of the contrapositive statement, which means that we have effectively proved the original statement.

For every proposition p, p is equivalent to the conditional proposition $true \rightarrow p$, whose contrapositive is $\neg p \rightarrow false$. A proof of p by contradiction means assuming that p is false and deriving a contradiction (i.e., deriving the statement *false*). The example we use to illustrate proof by contradiction is more than two thousand years old and was known to members of the Pythagorean school in Greece. It involves positive rational numbers: numbers of the form m/n, where m and n are positive integers.

Proof by Contradiction: The Square Root of 2 Is Irrational — EXAMPLE 1.3

To Prove: There are no positive integers m and n satisfying $m/n = \sqrt{2}$.

■ Proof

Suppose for the sake of contradiction that there are positive integers m and n with $m/n = \sqrt{2}$. Then by dividing both m and n by all the factors common to both, we obtain $p/q = \sqrt{2}$, for some positive integers p and q with no common factors. If $p/q = \sqrt{2}$, then $p = q\sqrt{2}$, and therefore $p^2 = 2q^2$. According to Example 1.1, since p^2 is even, p must be even; therefore, $p = 2r$ for some positive integer r, and $p^2 = 4r^2$. This implies that $2r^2 = q^2$, and the same argument we have just used for p also implies that q is even. Therefore, 2 is a common factor of p and q, and we have a contradiction of our previous statement that p and q have no common factors.

It is often necessary to use more than one proof technique within a single proof. Although the proof in the next example is not a proof by contradiction, that technique is used twice within it. The statement to be proved involves the factorial

of a positive integer n, which is denoted by $n!$ and is the product of all the positive integers less than or equal to n.

EXAMPLE 1.4 There Must Be a Prime Between n and $n!$

To Prove: For every integer $n > 2$, there is a prime p satisfying $n < p < n!$.

■ Proof

Because $n > 2$, the distinct integers n and 2 are two of the factors of $n!$. Therefore,

$$n! - 1 \geq 2n - 1 = n + n - 1 > n + 1 - 1 = n$$

The number $n! - 1$ has a prime factor p, which must satisfy $p \leq n! - 1 < n!$. Therefore, $p < n!$, which is one of the inequalities we need. To show the other one, suppose for the sake of contradiction that $p \leq n$. Then by the definition of factorial, p must be one of the factors of $n!$. However, p cannot be a factor of both $n!$ and $n! - 1$; if it were, it would be a factor of 1, their difference, and this is impossible because a prime must be bigger than 1. Therefore, the assumption that $p \leq n$ leads to a contradiction, and we may conclude that $n < p < n!$.

EXAMPLE 1.5 Proof by Cases

The last proof technique we will mention in this section is proof by cases. If P is a proposition we want to prove, and P_1 and P_2 are propositions, at least one of which must be true, then we can prove P by proving that P_1 implies P and P_2 implies P. This is sufficient because of the logical identities

$$\begin{aligned}(P_1 \rightarrow P) \wedge (P_2 \rightarrow P) &\Leftrightarrow (P_1 \vee P_2) \rightarrow P \\ &\Leftrightarrow \mathit{true} \rightarrow P \\ &\Leftrightarrow P\end{aligned}$$

which can be verified easily (saying that P_1 or P_2 must be true is the same as saying that $P_1 \vee P_2$ is equivalent to *true*).

The principle is the same if there are more than two cases. If we want to show the first distributive law

$$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$$

for example, then we must show that the truth values of the propositions on the left and right are the same, and there are eight cases, corresponding to the eight combinations of truth values for p, q, and r. An appropriate choice for P_1 is "p, q, and r are all true".

1.2 | SETS

A finite set can be described, at least in principle, by listing its elements. The formula

$$A = \{1, 2, 4, 8\}$$

says that A is the set whose elements are 1, 2, 4, and 8.

For infinite sets, and even for finite sets if they have more than just a few elements, ellipses (. . .) are sometimes used to describe how the elements might be listed:

$$B = \{0, 3, 6, 9, \ldots\}$$
$$C = \{13, 14, 15, \ldots, 71\}$$

A more reliable and often more informative way to describe sets like these is to give the property that characterizes their elements. The sets B and C could be described this way:

$$B = \{x \mid x \text{ is a nonnegative integer multiple of } 3\}$$
$$C = \{x \mid x \text{ is an integer and } 13 \leq x \leq 71\}$$

We would read the first formula "B is the set of all x such that x is a nonnegative integer multiple of 3". The expression before the vertical bar represents an arbitrary element of the set, and the statement after the vertical bar contains the conditions, or restrictions, that the expression must satisfy in order for it to represent a legal element of the set.

In these two examples, the "expression" is simply a variable, which we have arbitrarily named x. We often choose to include a little more information in the expression; for example,

$$B = \{3y \mid y \text{ is a nonnegative integer}\}$$

which we might read "B is the set of elements of the form $3y$, where y is a nonnegative integer". Two more examples of this approach are

$$D = \{\{x\} \mid x \text{ is an integer such that } x \geq 4\}$$
$$E = \{3i + 5j \mid i \text{ and } j \text{ are nonnegative integers}\}$$

Here D is a set of sets; three of its elements are $\{4\}$, $\{5\}$, and $\{6\}$. We could describe E using the formula

$$E = \{0, 3, 5, 6, 8, 9, 10, \ldots\}$$

but the first description of E is more informative, even if the other seems at first to be more straightforward.

For any set A, the statement that x is an element of A is written $x \in A$, and $x \notin A$ means x is not an element of A. We write $A \subseteq B$ to mean A is a *subset* of B, or that every element of A is an element of B; $A \not\subseteq B$ means that A is not a subset of B (there is at least one element of A that is not an element of B). Finally, the *empty set*, the set with no elements, is denoted by $\emptyset$.

A set is determined by its elements. For example, the sets $\{0, 1\}$ and $\{1, 0\}$ are the same, because both contain the elements 0 and 1 and no others; the set $\{0, 0, 1, 1, 1, 2\}$ is the same as $\{0, 1, 2\}$, because they both contain 0, 1, and 2 and no other elements (no matter how many times each element is written, it's the same element); and there is only one empty set, because once you've said that a set

contains no elements, you've described it completely. To show that two sets A and B are the same, we must show that A and B have exactly the same elements—i.e., that $A \subseteq B$ and $B \subseteq A$.

A few sets will come up frequently. We have used $\mathcal{N}$ in Section 1.1 to denote the set of *natural numbers*, or nonnegative integers; $\mathcal{Z}$ is the set of all integers, $\mathcal{R}$ the set of all real numbers, and $\mathcal{R}^+$ the set of nonnegative real numbers. The sets B and E above can be written more concisely as

$$B = \{3y \mid y \in \mathcal{N}\} \qquad E = \{3i + 5j \mid i, j \in \mathcal{N}\}$$

We sometimes relax the { expression | conditions } format slightly when we are describing a subset of another set, as in

$$C = \{x \in \mathcal{N} \mid 13 \leq x \leq 71\}$$

which we would read "C is the set of all x in $\mathcal{N}$ such that . . . "

For two sets A and B, we can define their *union* $A \cup B$, their *intersection* $A \cap B$, and their *difference* $A - B$, as follows:

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$
$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$
$$A - B = \{x \mid x \in A \text{ and } x \notin B\}$$

For example,

$$\{1, 2, 3, 5\} \cup \{2, 4, 6\} = \{1, 2, 3, 4, 5, 6\}$$
$$\{1, 2, 3, 5\} \cap \{2, 4, 6\} = \{2\}$$
$$\{1, 2, 3, 5\} - \{2, 4, 6\} = \{1, 3, 5\}$$

If we assume that A and B are both subsets of some "universal" set U, then we can consider the special case $U - A$, which is written A' and referred to as the *complement* of A.

$$A' = U - A = \{x \in U \mid x \notin A\}$$

We think of A' as "the set of everything that's not in A", but to be meaningful this requires context. The complement of $\{1, 2\}$ varies considerably, depending on whether the universal set is chosen to be $\mathcal{N}$, $\mathcal{Z}$, $\mathcal{R}$, or some other set.

If the intersection of two sets is the empty set, which means that the two sets have no elements in common, they are called *disjoint* sets. The sets in a collection of sets are *pairwise disjoint* if, for every two distinct ones A and B ("distinct" means not identical), A and B are disjoint. A *partition* of a set S is a collection of pairwise disjoint subsets of S whose union is S; we can think of a partition of S as a way of dividing S into non-overlapping subsets.

There are a number of useful "set identities", but they are closely analogous to the logical identities we discussed in Section 1.1, and as the following example demonstrates, they can be derived the same way.

EXAMPLE 1.6 The First De Morgan Law

There are two De Morgan laws for sets, just as there are for propositions; the first asserts that for every two sets A and B,

$$(A \cup B)' = A' \cap B'$$

We begin by noticing the resemblance between this formula and the logical identity

$$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$$

The resemblance is not just superficial. We defined the logical connectives such as $\wedge$ and $\vee$ by drawing truth tables, and we could define the set operations $\cap$ and $\cup$ by drawing *membership* tables, where T denotes membership and F nonmembership:

A	B	$A \cap B$	$A \cup B$
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

As you can see, the truth values in the two tables are identical to the truth values in the tables for $\wedge$ and $\vee$. We can therefore test a proposed set identity the same way we can test a proposed logical identity, by constructing tables for the two expressions being compared. When we do this for the expressions $(A \cup B)'$ and $A' \cap B'$, or for the propositions $\neg(p \vee q)$ and $\neg p \wedge \neg q$, by considering the four cases, we obtain identical values in each case. We may conclude that no matter what case x represents, $x \in (A \cup B)'$ if and only if $x \in A' \cap B'$, and the two sets are equal.

The associative law for unions, corresponding to the one for $\vee$, says that for arbitrary sets A, B, and C,

$$A \cup (B \cup C) = (A \cup B) \cup C$$

so that we can write $A \cup B \cup C$ without worrying about how to group the terms. It is easy to see from the definition of union that

$$A \cup B \cup C = \{x \mid x \text{ is an element of at least one of the sets } A, B, \text{ and } C\}$$

For the same reasons, we can consider unions of any number of sets and adopt notation to describe such unions. For example, if $A_0, A_1, A_2, \ldots$ are sets,

$$\bigcup\{A_i \mid 0 \le i \le n\} = \{x \mid x \in A_i \text{ for at least one } i \text{ with } 0 \le i \le n\}$$

$$\bigcup\{A_i \mid i \ge 0\} = \{x \mid x \in A_i \text{ for at least one } i \text{ with } i \ge 0\}$$

In Chapter 3 we will encounter the set

$$\bigcup \{\delta(p, \sigma) \mid p \in \delta^*(q, x)\}$$

In all three of these formulas, we have a set S of sets, and we are describing the union of all the sets in S. We do not need to know what the sets $\delta^*(q, x)$ and $\delta(p, \sigma)$ are to understand that

$$\bigcup\{\delta(p, \sigma) \mid p \in \delta^*(q, x)\} = \{x \mid x \in \delta(p, \sigma) \text{ for at least one element } p \text{ of } \delta^*(q, x)\}$$

If $\delta^*(q, x)$ were $\{r, s, t\}$, for example, we would have

$$\bigcup\{\delta(p, \sigma) \mid p \in \delta^*(q, x)\} = \delta(r, \sigma) \cup \delta(s, \sigma) \cup \delta(t, \sigma)$$

Sometimes the notation varies slightly. The two sets

$$\bigcup\{A_i \mid i \geq 0\} \quad \text{and} \quad \bigcup\{\delta(p, \sigma) \mid p \in \delta^*(q, x)\}$$

for example, might be written

$$\bigcup_{i=0}^{\infty} A_i \quad \text{and} \quad \bigcup_{p \in \delta^*(q,x)} \delta(p, \sigma)$$

respectively.

Because there is also an associative law for intersections, exactly the same notation can be used with $\cap$ instead of $\cup$.

For a set A, the set of all subsets of A is called the *power set* of A and written 2^A. The reason for the terminology and the notation is that if A is a finite set with n elements, then 2^A has exactly 2^n elements (see Example 1.23). For example,

$$2^{\{a,b,c\}} = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

This example illustrates the fact that the empty set is a subset of every set, and every set is a subset of itself.

One more set that can be constructed from two sets A and B is $A \times B$, their *Cartesian product*:

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$$

For example,

$$\{0, 1\} \times \{1, 2, 3\} = \{(0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3)\}$$

The elements of $A \times B$ are called *ordered* pairs, because $(a, b) = (c, d)$ if and only if $a = c$ and $b = d$; in particular, (a, b) and (b, a) are different unless a and b happen to be equal. More generally, $A_1 \times A_2 \times \cdots \times A_k$ is the set of all "ordered k-tuples" $(a_1, a_2, \ldots, a_k)$, where a_i is an element of A_i for each i.

1.3 FUNCTIONS AND EQUIVALENCE RELATIONS

If A and B are two sets (possibly equal), a *function* f from A to B is a rule that assigns to each element x of A an element $f(x)$ of B. (Later in this section we will mention a more precise definition, but for our purposes the informal "rule"

definition will be sufficient.) We write $f : A \to B$ to mean that f is a function from A to B.

Here are four examples:

1. The function $f : \mathcal{N} \to \mathcal{R}$ defined by the formula $f(x) = \sqrt{x}$. (In other words, for every $x \in \mathcal{N}$, $f(x) = \sqrt{x}$.)
2. The function $g : 2^{\mathcal{N}} \to 2^{\mathcal{N}}$ defined by the formula $g(A) = A \cup \{0\}$.
3. The function $u : 2^{\mathcal{N}} \times 2^{\mathcal{N}} \to 2^{\mathcal{N}}$ defined by the formula $u(S, T) = S \cup T$.
4. The function $i : \mathcal{N} \to \mathcal{Z}$ defined by

$$i(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ (-n-1)/2 & \text{if } n \text{ is odd} \end{cases}$$

For a function f from A to B, we call A the *domain* of f and B the *codomain* of f. The domain of a function f is the set of values x for which $f(x)$ is defined. We will say that two functions f and g are the same if and only if they have the same domain, they have the same codomain, and $f(x) = g(x)$ for every x in the domain.

In some later chapters it will be convenient to refer to a *partial* function f from A to B, one whose domain is a subset of A, so that f may be undefined at some elements of A. We will still write $f : A \to B$, but we will be careful to distinguish the set A from the domain of f, which may be a smaller set. When we speak of a *function* from A to B, without any qualification, we mean one with domain A, and we might emphasize this by calling it a *total* function.

If f is a function from A to B, a third set involved in the description of f is its *range*, which is the set

$$\{f(x) \mid x \in A\}$$

(a subset of the codomain B). The range of f is the set of elements of the codomain that are actually assigned by f to elements of the domain.

Definition 1.7 One-to-One and Onto Functions

A function $f : A \to B$ is *one-to-one* if f never assigns the same value to two different elements of its domain. It is *onto* if its range is the entire set B. A function from A to B that is both one-to-one and onto is called a *bijection* from A to B.

Another way to say that a function $f : A \to B$ is one-to-one is to say that for every $y \in B$, $y = f(x)$ for *at most* one $x \in A$, and another way to say that f is onto is to say that for every $y \in B$, $y = f(x)$ for *at least* one $x \in A$. Therefore, saying that f is a bijection from A to B means that every element y of the codomain B is $f(x)$ for *exactly* one $x \in A$. This allows us to define another function f^{-1} from B to A, by saying that for every $y \in B$, $f^{-1}(y)$ is the element $x \in A$ for which

$f(x) = y$. It is easy to check that this "inverse function" is also a bijection and satisfies these two properties: For every $x \in A$, and every $y \in B$,

$$f^{-1}(f(x)) = x \qquad f(f^{-1}(y)) = y$$

Of the four functions defined above, the function f from $\mathcal{N}$ to $\mathcal{R}$ is one-to-one but not onto, because a real number is the square root of at most one natural number and might not be the square root of any. The function g is not one-to-one, because for every subset A of $\mathcal{N}$ that doesn't contain 0, A and $A \cup \{0\}$ are distinct and $g(A) = g(A \cup \{0\})$. It is also not onto, because every element of the range of g is a set containing 0 and not every subset of $\mathcal{N}$ does. The function u is onto, because $u(A, A) = A$ for every $A \in 2^{\mathcal{N}}$, but not one-to-one, because for every $A \in 2^{\mathcal{N}}$, $u(A, \emptyset)$ is also A.

The formula for i seems more complicated, but looking at this partial tabulation of its values

x	0	1	2	3	4	5	6	...
$i(x)$	0	-1	1	-2	2	-3	3	...

makes it easy to see that i is both one-to-one and onto. No integer appears more than once in the list of values of i, and every integer appears once.

In the first part of this book, we will usually not be concerned with whether the functions we discuss are one-to-one or onto. The idea of a bijection between two sets, such as our function i, will be important in Chapter 8, when we discuss infinite sets with different sizes.

An *operation on a set A* is a function that assigns to elements of A, or perhaps to combinations of elements of A, other elements of A. We will be interested particularly in *binary* operations (functions from $A \times A$ to A) and *unary* operations (functions from A to A). The function u described above is an example of a binary operation on the set $2^{\mathcal{N}}$, and for every set S, both union and intersection are binary operations on 2^S. Familar binary operations on $\mathcal{N}$, or on $\mathcal{Z}$, include addition and multiplication, and subtraction is a binary operation on $\mathcal{Z}$. The complement operation is a unary operation on 2^S, for every set S, and negation is a unary operation on the set $\mathcal{Z}$. The notation adopted for some of these operations is different from the usual functional notation; we write $U \cup V$ rather than $\cup(U, V)$, and $a - b$ rather than $-(a, b)$.

For a unary operation or a binary operation on a set A, we say that a subset A_1 of A is *closed under the operation* if the result of applying the operation to elements of A_1 is an element of A_1. For example, if $A = 2^{\mathcal{N}}$, and A_1 is the set of all *nonempty* subsets of $\mathcal{N}$, then A_1 is closed under union (the union of two nonempty subsets of $\mathcal{N}$ is a nonempty subset of $\mathcal{N}$) but not under intersection. The set of all subsets of $\mathcal{N}$ with fewer than 100 elements is closed under intersection but not under union. If $A = \mathcal{N}$, and A_1 is the set of even natural numbers, then A_1 is closed under both addition and multiplication; the set of odd natural numbers is closed under multiplication but not under addition. We will return to this idea later in this chapter, when we discuss recursive definitions of sets.

We can think of a function f from a set A to a set B as establishing a relationship between elements of A and elements of B; every element $x \in A$ is "related" to exactly one element $y \in B$, namely, $y = f(x)$. A *relation* R from A to B may be more general, in that an element $x \in A$ may be related to no elements of B, to one element, or to more than one. We will use the notation aRb to mean that a is related to b with respect to the relation R. For example, if A is the set of people and B is the set of cities, we might consider the "has-lived-in" relation R from A to B: If $x \in A$ and $y \in B$, xRy means that x has lived in y. Some people have never lived in a city, some have lived in one city all their lives, and some have lived in several cities.

We've said that a function is a "rule"; exactly what is a relation?

Definition 1.8 A Relation from *A* to *B*, and a Relation on *A*

For two sets A and B, a relation from A to B is a subset of $A \times B$. A relation on the set A is a relation from A to A, or a subset of $A \times A$.

The statement "a is related to b with respect to R" can be expressed by either of the formulas aRb and $(a, b) \in R$. As we have already pointed out, a function f from A to B is simply a relation having the property that for every $x \in A$, there is exactly one $y \in B$ with $(x, y) \in f$. Of course, in this special case, a third way to write "x is related to y with respect to f" is the most common: $y = f(x)$.

In the has-lived-in example above, the statement "Sally has lived in Atlanta" seems easier to understand than the statement "(Sally, Atlanta)$\in R$", but this is just a question of notation. If we understand what R is, the two statements say the same thing. In this book, we will be interested primarily in relations on a set, especially ones that satisfy the three properties in the next definition.

Definition 1.9 Equivalence Relations

A relation R on a set A is an *equivalence relation* if it satisfies these three properties.

1. R is *reflexive*: for every $x \in A$, xRx.
2. R is *symmetric*: for every x and every y in A, if xRy, then yRx.
3. R is *transitive*: for every x, every y, and every z in A, if xRy and yRz, then xRz.

If R is an equivalence relation on A, we often say "x is equivalent to y" instead of "x is related to y". Examples of relations that do not satisfy all three properties can be found in the exercises. Here we present three simple examples of equivalence relations.

EXAMPLE 1.10 The Equality Relation

We can consider the relation of equality on every set A, and the formula $x = y$ expresses the fact that (x, y) is an element of the relation. The properties of reflexivity, symmetry, and transitivity are familiar properties of equality: Every element of A is equal to itself; for every x and y in A, if $x = y$, then $y = x$; and for every x, y, and z, if $x = y$ and $y = z$, then $x = z$. This relation is the prototypical equivalence relation, and the three properties are no more than what we would expect of any relation we described as one of *equivalence*.

EXAMPLE 1.11 The Relation on A Containing All Ordered Pairs

On every set A, we can also consider the relation $R = A \times A$. Every possible ordered pair of elements of A is in the relation—every element of A is related to every other element, including itself. This relation is also clearly an equivalence relation; no statement of the form "(under certain conditions) xRy" can possibly fail if xRy for every x and every y.

EXAMPLE 1.12 The Relation of Congruence Mod n on $\mathcal{N}$

We consider the set $\mathcal{N}$ of natural numbers, and, for some positive integer n, the relation R on $\mathcal{N}$ defined as follows: for every x and y in $\mathcal{N}$,

$$xRy \text{ if there is an integer } k \text{ so that } x - y = kn$$

In this case we write $x \equiv_n y$ to mean xRy. Checking that the three properties are satisfied requires a little more work this time, but not much. The relation is reflexive, because for every $x \in \mathcal{N}$, $x - x = 0 * n$. It is symmetric, because for every x and every y in $\mathcal{N}$, if $x - y = kn$, then $y - x = (-k)n$. Finally, it is transitive, because if $x - y = kn$ and $y - z = jn$, then

$$x - z = (x - y) + (y - z) = kn + jn = (k + j)n$$

One way to understand an equivalence relation R on a set A is to consider, for each $x \in A$, the subset $[x]_R$ of A containing all the elements equivalent to x. Because an equivalence relation is reflexive, one of these elements is x itself, and we can refer to the set $[x]_R$ as *the equivalence class containing* x.

Definition 1.13 The Equivalence Class Containing x

For an equivalence relation R on a set A, and an element $x \in A$, the equivalence class containing x is

$$[x]_R = \{y \in A \mid yRx\}$$

If there is no doubt about which equivalence relation we are using, we will drop the subscript and write $[x]$.

The phrase "the equivalence class containing x" is not misleading: For every $x \in A$, we have already seen that $x \in [x]$, and we can also check that x belongs to only one equivalence class. Suppose that $x, y \in A$ and $x \in [y]$, so that xRy; we show that $[x] = [y]$. Let z be an arbitrary element of $[x]$, so that zRx. Because zRx, xRy, and R is transitive, it follows that zRy; therefore, $[x] \subseteq [y]$. For the other inclusion we observe that if $x \in [y]$, then $y \in [x]$ because R is symmetric, and the same argument with x and y switched shows that $[y] \subseteq [x]$.

These conclusions are summarized by Theorem 1.14.

Theorem 1.14

If R is an equivalence relation on a set A, the equivalence classes with respect to R form a partition of A, and two elements of A are equivalent if and only if they are elements of the same equivalence class.

Example 1.10 illustrates the extreme case in which every equivalence class contains just one element, and Example 1.11 illustrates the other extreme, in which the single equivalence class A contains all the elements. In the case of congruence mod n for a number $n > 1$, some but not all of the elements of $\mathcal{N}$ other than x are in $[x]$; the set $[x]$ contains all natural numbers that differ from x by a multiple of n.

For an arbitrary equivalence relation R on a set A, knowing the partition determined by R is enough to describe the relation completely. In fact, if we begin with a partition of A, then the relation R on A that is defined by the last statement of Theorem 1.1 (two elements x and y are related if and only if x and y are in the same subset of the partition) is an equivalence relation whose equivalence classes are precisely the subsets of the partition. Specifying a subset of $A \times A$ and specifying a partition on A are two ways of conveying the same information.

Finally, if R is an equivalence relation on A and $S = [x]$, it follows from Theorem 1.14 that every two elements of S are equivalent and no element of S is equivalent to an element not in S. On the other hand, if S is a nonempty subset of A, knowing that S satisfies these two properties allows us to say that S is an equivalence class, even if we don't start out with any particular x satisfying $S = [x]$. If x is an arbitrary element of S, every element of S belongs to $[x]$, because it is equivalent to x; and every element of $[x]$ belongs to S, because otherwise the element x of S would be equivalent to some element not in S. Therefore, for every $x \in S$, $S = [x]$.

1.4 | LANGUAGES

Familar languages include programming languages such as Java and natural languages like English, as well as unofficial "dialects" with specialized vocabularies, such as the language used in legal documents or the language of mathematics. In this book we use the word "language" more generally, taking a language to be any set of strings over an alphabet of symbols. In applying this definition to English,

we might take the individual strings to be English words, but it is more common to consider English sentences, for which many grammar rules have been developed. In the case of a language like Java, a string must satisfy certain rules in order to be a legal statement, and a sequence of statements must satisfy certain rules in order to be a legal program.

Many of the languages we study initially will be much simpler. They might involve alphabets with just one or two symbols, and perhaps just one or two basic patterns to which all the strings must conform. The main purpose of this section is to present some notation and terminology involving strings and languages that will be used throughout the book.

An *alphabet* is a finite set of symbols, such as $\{a, b\}$ or $\{0, 1\}$ or $\{A, B, C, \ldots, Z\}$. We will usually use the Greek letter Σ to denote the alphabet. A *string* over Σ is a finite sequence of symbols in Σ. For a string x, $|x|$ stands for the *length* (the number of symbols) of x. In addition, for a string x over Σ and an element $\sigma \in \Sigma$,

$$n_\sigma(x) = \text{ the number of occurrences of the symbol } \sigma \text{ in the string } x$$

The *null string* Λ is a string over Σ, no matter what the alphabet Σ is. By definition, $|\Lambda| = 0$.

The set of all strings over Σ will be written Σ^*. For the alphabet $\{a, b\}$, we have

$$\{a, b\}^* = \{\Lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$$

Here we have listed the strings in *canonical* order, the order in which shorter strings precede longer strings and strings of the same length appear alphabetically. Canonical order is different from *lexicographic*, or strictly alphabetical order, in which aa precedes b. An essential difference is that canonical order can be described by making a single list of strings that includes every element of Σ^* exactly once. If we wanted to describe an algorithm that did something with each string in $\{a, b\}^*$, it would make sense to say, "Consider the strings in canonical order, and for each one, ..." (see, for example, Section 8.2). If an algorithm were to "consider the strings of $\{a, b\}^*$ in lexicographic order", it would have to start by considering Λ, a, aa, aaa, ..., and it would never get around to considering the string b.

A language over Σ is a subset of Σ^*. Here are a few examples of languages over $\{a, b\}$:

1. The empty language $\emptyset$.
2. $\{\Lambda, a, aab\}$, another finite language.
3. The language *Pal* of palindromes over $\{a, b\}$ (strings such as aba or $baab$ that are unchanged when the order of the symbols is reversed).
4. $\{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$.
5. $\{x \in \{a, b\}^* \mid |x| \geq 2 \text{ and } x \text{ begins and ends with } b\}$.

The null string Λ is always an element of Σ^*, but other languages over Σ may or may not contain it; of these five examples, only the second and third do.

Here are a few real-world languages, in some cases involving larger alphabets.

6. The language of legal Java identifiers.
7. The language *Expr* of legal algebraic expressions involving the identifier a, the binary operations $+$ and $*$, and parentheses. Some of the strings in the language are a, $a + a * a$, and $(a + a * (a + a))$.
8. The language *Balanced* of balanced strings of parentheses (strings containing the occurrences of parentheses in some legal algebraic expression). Some elements are Λ, $()(())$, and $(((((\,)))))$.
9. The language of numeric "literals" in Java, such as -41, 0.03, and 5.0E$-$3.
10. The language of legal Java programs. Here the alphabet would include upper- and lowercase alphabetic symbols, numerical digits, blank spaces, and punctuation and other special symbols.

The basic operation on strings is *concatenation*. If x and y are two strings over an alphabet, the concatenation of x and y is written xy and consists of the symbols of x followed by those of y. If $x = ab$ and $y = bab$, for example, then $xy = abbab$ and $yx = babab$. When we concatenate the null string Λ with another string, the result is just the other string (for every string x, $x\Lambda = \Lambda x = x$); and for every x, if one of the formulas $xy = x$ or $yx = x$ is true for some string y, then $y = \Lambda$. In general, for two strings x and y, $|xy| = |x| + |y|$.

Concatenation is an associative operation; that is, $(xy)z = x(yz)$, for all possible strings x, y, and z. This allows us to write xyz without specifying how the factors are grouped.

If s is a string and $s = tuv$ for three strings t, u, and v, then t is a *prefix* of s, v is a *suffix* of s, and u is a *substring* of s. Because one or both of t and u might be Λ, prefixes and suffixes are special cases of substrings. The string Λ is a prefix of every string, a suffix of every string, and a substring of every string, and every string is a prefix, a suffix, and a substring of itself.

Languages are sets, and so one way of constructing new languages from existing ones is to use set operations. For two languages L_1 and L_2 over the alphabet Σ, $L_1 \cup L_2$, $L_1 \cap L_2$, and $L_1 - L_2$ are also languages over Σ. If $L \subseteq \Sigma^*$, then by the complement of L we will mean $\Sigma^* - L$. This is potentially confusing, because if L is a language over Σ, then L can be interpreted as a language over any larger alphabet, but it will usually be clear what alphabet we are referring to.

We can also use the string operation of concatenation to construct new languages. If L_1 and L_2 are both languages over Σ, the concatenation of L_1 and L_2 is the language

$$L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$$

For example, $\{a, aa\}\{\Lambda, b, ab\} = \{a, ab, aab, aa, aaab\}$. Because $\Lambda x = x\Lambda$ for every string x, we have

$$\{\Lambda\}L = L\{\Lambda\} = L$$

for every language L.

The language $L = \Sigma^*$, for example, satisfies the formula $LL = L$, and so the formula $LL_1 = L$ does not always imply that $L_1 = \{\Lambda\}$. However, if L_1 is

a language such that $LL_1 = L$ for *every* language L, or if $L_1L = L$ for every language L, then $L_1 = \{\Lambda\}$.

At this point we can adopt "exponential" notation for the concatenation of k copies of a single symbol a, a single string x, or a single language L. If $k > 0$, then $a^k = aa \ldots a$, where there are k occurrences of a, and similarly for x^k and L^k. In the special case where L is simply the alphabet Σ (which can be interpreted as a set of strings of length 1), $\Sigma^k = \{x \in \Sigma^* \mid |x| = k\}$.

We also want the exponential notation to make sense if $k = 0$, and the correct definition requires a little care. It is desirable to have the formulas

$$a^i a^j = a^{i+j} \qquad x^i x^j = x^{i+j} \qquad L^i L^j = L^{i+j}$$

where a, x, and L are an alphabet symbol, a string, and a language, respectively. In the case $i = 0$, the first two formulas require that we define a^0 and x^0 to be Λ, and the last formula requires that L^0 be $\{\Lambda\}$.

Finally, for a language L over an alphabet Σ, we use the notation L^* to denote the language of all strings that can be obtained by concatenating zero or more strings in L. This operation on a language L is known as the *Kleene star*, or Kleene closure, after the mathematician Stephen Kleene. The notation L^* is consistent with the earlier notation Σ^*, which we can describe as the set of strings obtainable by concatenating zero or more strings of length 1 over Σ. L^* can be defined by the formula

$$L^* = \bigcup \{L^k \mid k \in \mathcal{N}\}$$

Because we have defined L^0 to be $\{\Lambda\}$, "concatenating zero strings in L" produces the null string, and $\Lambda \in L^*$, no matter what the language L is.

When we describe languages using formulas that contain the union, concatenation, and Kleene L^* operations, we will use precedence rules similar to the algebraic rules you are accustomed to. The formula $L_1 \cup L_2L_3^*$, for example, means $L_1 \cup (L_2(L_3^*))$; of the three operations, the highest-precedence operation is *, next-highest is concatenation, and lowest is union. The expressions $(L_1 \cup L_2)L_3^*$, $L_1 \cup (L_2L_3)^*$, and $(L_1 \cup L_2L_3)^*$ all refer to different languages.

Strings, by definition, are finite (have only a finite number of symbols). Almost all interesting languages are infinite sets of strings, and in order to use the languages we must be able to provide precise finite descriptions. There are at least two general approaches to doing this, although there is not always a clear line separating them. If we write

$$L_1 = \{ab, bab\}^* \cup \{b\}\{ba\}^*\{ab\}^*$$

we have described the language L_1 by providing a formula showing the possible ways of generating an element: either concatenating an arbitrary number of strings, each of which is either ab or bab, or concatenating a single b with an arbitrary number of copies of ba and then an arbitrary number of copies of ab. The fourth example in our list above is the language

$$L_2 = \{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$$

which we have described by giving a property that characterizes the elements. For every string $x \in \{a, b\}^*$, we can test whether x is in L_2 by testing whether the condition is satisfied.

In this book we will study notational schemes that make it easy to describe how languages can be *generated*, and we will study various types of algorithms, of increasing complexity, for *recognizing*, or *accepting*, strings in certain languages. In the second approach, we will often identify an algorithm with an abstract machine that can carry it out; a precise description of the algorithm or the machine will effectively give us a precise way of specifying the language.

1.5 | RECURSIVE DEFINITIONS

As you know, recursion is a technique that is often useful in writing computer programs. In this section we will consider recursion as a tool for defining sets: primarily, sets of numbers, sets of strings, and sets of sets (of numbers or strings).

A recursive definition of a set begins with a *basis* statement that specifies one or more elements in the set. The *recursive* part of the definition involves one or more operations that can be applied to elements already known to be in the set, so as to produce new elements of the set.

As a way of defining a set, this approach has a number of potential advantages: Often it allows very concise definitions; because of the algorithmic nature of a typical recursive definition, one can often see more easily how, or why, a particular object is an element of the set being defined; and it provides a natural way of defining functions on the set, as well as a natural way of proving that some condition or property is satisfied by every element of the set.

The Set of Natural Numbers **EXAMPLE 1.15**

The prototypical example of recursive definition is the axiomatic definition of the set $\mathcal{N}$ of natural numbers. We assume that 0 is a natural number and that we have a "successor" operation, which, for each natural number n, gives us another one that is the successor of n and can be written $n + 1$. We might write the definition this way:

1. $0 \in \mathcal{N}$.
2. For every $n \in \mathcal{N}$, $n + 1 \in \mathcal{N}$.
3. Every element of $\mathcal{N}$ can be obtained by using statement 1 or statement 2.

In order to obtain an element of $\mathcal{N}$, we use statement 1 once and statement 2 a finite number of times (zero or more). To obtain the natural number 7, for example, we use statement 1 to obtain 0; then statement 2 with $n = 0$ to obtain 1; then statement 2 with $n = 1$ to obtain 2; ... ; and finally, statement 2 with $n = 6$ to obtain 7.

We can summarize the first two statements by saying that $\mathcal{N}$ contains 0 and is *closed under the successor operation* (the operation of adding 1).

There are other sets of numbers that contain 0 and are closed under the successor operation: the set of all real numbers, for example, or the set of all fractions. The third

statement in the definition is supposed to make it clear that the set we are defining is the one containing *only* the numbers obtained by using statement 1 once and statement 2 a finite number of times. In other words, $\mathcal{N}$ is the *smallest* set of numbers that contains 0 and is closed under the successor operation: $\mathcal{N}$ is a subset of every other such set.

In the remaining examples in this section we will omit the statement corresponding to statement 3 in this example, but whenever we define a set recursively, we will assume that a statement like this one is in effect, whether or not it is stated explicitly.

Just as a recursive procedure in a computer program must have an "escape hatch" to avoid calling itself forever, a recursive definition like the one above must have a basis statement that provides us with at least one element of the set. The recursive statement, that $n + 1 \in \mathcal{N}$ for every $n \in \mathcal{N}$, works in combination with the basis statement to give us all the remaining elements of the set.

EXAMPLE 1.16 Recursive Definitions of Other Subsets of $\mathcal{N}$

If we use the definition in Example 1.15, but with a different value specified in the basis statement:

1. $15 \in A$.
2. For every $n \in A$, $n + 1 \in A$.

then the set A that has been defined is the set of natural numbers greater than or equal to 15.

If we leave the basis statement the way it was in Example 1.15 but change the "successor" operation by changing $n + 1$ to $n + 7$, we get a definition of the set of all natural numbers that are multiples of 7.

Here is a definition of a subset B of $\mathcal{N}$:

1. $1 \in B$.
2. For every $n \in B$, $2 * n \in B$.
3. For every $n \in B$, $5 * n \in B$.

The set B is the smallest set of numbers that contains 1 and is closed under multiplication by 2 and 5. Starting with the number 1, we can obtain 2, 4, 8, ... by repeated applications of statement 2, and we can obtain 5, 25, 125, ... by using statement 3. By using both statements 2 and 3, we can obtain numbers such as $2 * 5$, $4 * 5$, and $2 * 25$. It is not hard to convince yourself that B is the set

$$B = \{2^i * 5^j \mid i, j \in \mathcal{N}\}$$

EXAMPLE 1.17 Recursive Definitions of $\{a,b\}^*$

Although we use $\Sigma = \{a, b\}$ in this example, it will be easy to see how to modify the definition so that it uses another alphabet. Our recursive definition of $\mathcal{N}$ started with the natural number 0, and the recursive statement allowed us to take an arbitrary n and obtain a natural number 1 bigger. An analogous recursive definition of $\{a, b\}^*$ begins with the string of length 0 and says how to take an arbitrary string x and obtain strings of length $|x| + 1$.

1. $\Lambda \in \{a, b\}^*$.
2. For every $x \in \{a, b\}^*$, both xa and xb are in $\{a, b\}^*$.

To obtain a string z of length k, we start with Λ and obtain longer and longer prefixes of z by using the second statement k times, each time concatenating the next symbol onto the right end of the current prefix. A recursive definition that used ax and bx in statement 2 instead of xa and xb would work just as well; in that case we would produce longer and longer suffixes of z by adding each symbol to the left end of the current suffix.

Recursive Definitions of Two Other Languages over {*a*,*b*}

EXAMPLE 1.18

We let *AnBn* be the language

$$AnBn = \{a^n b^n \mid n \in \mathcal{N}\}$$

and *Pal* the language introduced in Section 1.4 of all *palindromes* over $\{a, b\}$; a palindrome is a string that is unchanged when the order of the symbols is reversed.

The shortest string in *AnBn* is Λ, and if we have an element $a^i b^i$ of length $2i$, the way to get one of length $2i + 2$ is to add a at the beginning and b at the end. Therefore, a recursive definition of *AnBn* is:

1. $\Lambda \in AnBn$.
2. For every $x \in AnBn$, $axb \in AnBn$.

It is only slightly harder to find a recursive definition of *Pal*. The length of a palindrome can be even or odd. The shortest one of even length is Λ, and the two shortest ones of odd length are a and b. For every palindrome x, a longer one can be obtained by adding the same symbol at both the beginning and the end of x, and every palindrome of length at least 2 can be obtained from a shorter one this way. The recursive definition is therefore

1. Λ, a, and b are elements of *Pal*.
2. For every $x \in Pal$, axa and bxb are in *Pal*.

Both *AnBn* and *Pal* will come up again, in part because they illustrate in a very simple way some of the limitations of the first type of abstract computing device we will consider.

Algebraic Expressions and Balanced Strings of Parentheses

EXAMPLE 1.19

As in Section 1.4, we let *Expr* stand for the language of legal algebraic expressions, where for simplicity we restrict ourselves to two binary operators, $+$ and $*$, a single identifier a, and left and right parentheses. Real-life expressions can be considerably more complicated because they can have additional operators, multisymbol identifiers, and numeric literals of various types; however, two operators are enough to illustrate the basic principles, and the other features can easily be added by substituting more general subexpressions for the identifier a.

Expressions can be illegal for "local" reasons, such as illegal symbol-pairs, or because of global problems involving mismatched parentheses. Explicitly prohibiting all the features

we want to consider illegal is possible but is tedious. A recursive definition, on the other hand, makes things simple. The simplest algebraic expression consists of a single a, and any other one is obtained by combining two subexpressions using $+$ or $*$ or by parenthesizing a single subexpression.

1. $a \in Expr$.
2. For every x and every y in $Expr$, $x + y$ and $x * y$ are in $Expr$.
3. For every $x \in Expr$, $(x) \in Expr$.

The expression $(a + a * (a + a))$, for example, can be obtained as follows:

$a \in Expr$, by statement 1.
$a + a \in Expr$, by statement 2, where x and y are both a.
$(a + a) \in Expr$, by statement 3, where $x = a + a$.
$a * (a + a) \in Expr$, by statement 2, where $x = a$ and $y = (a + a)$.
$a + a * (a + a) \in Expr$, by statement 2, where $x = a$ and $y = a * (a + a)$.
$(a + a * (a + a)) \in Expr$, by statement 3, where $x = a + a * (a + a)$.

It might have occurred to you that there is a shorter derivation of this string. In the fourth line, because we have already obtained both $a + a$ and $(a + a)$, we could have said

$$a + a * (a + a) \in Expr, \text{ by statement 2, where } x = a + a \text{ and } y = (a + a).$$

The longer derivation takes into account the normal rules of precedence, under which $a + a * (a + a)$ is interpreted as the sum of a and $a * (a + a)$, rather than as the product of $a + a$ and $(a + a)$. The recursive definition addresses only the strings that are in the language, not what they mean or how they should be interpreted. We will discuss this issue in more detail in Chapter 4.

Now we try to find a recursive definition for *Balanced*, the language of balanced strings of parentheses. We can think of balanced strings as the strings of parentheses that can occur within strings in the language *Expr*. The string a has no parentheses; and the two ways of forming new balanced strings from existing balanced strings are to concatenate two of them (because two strings in *Expr* can be concatenated, with either $+$ or $*$ in between), or to parenthesize one of them (because a string in *Expr* can be parenthesized).

1. $\Lambda \in Balanced$.
2. For every x and every y in *Balanced*, $xy \in Balanced$.
3. For every $x \in Balanced$, $(x) \in Balanced$.

In order to use the "closed-under" terminology to paraphrase the recursive definitions of *Expr* and *Balanced*, it helps to introduce a little notation. If we define operations $\circ$, $\bullet$, and $\diamond$ by saying $x \circ y = x + y$, $x \bullet y = x * y$, and $\diamond(x) = (x)$, then we can say that *Expr* is the smallest language that contains the string a and is closed under the operations $\circ$, $\bullet$, and $\diamond$. (This is confusing. We normally think of $+$ and $*$ as "operations", but addition and multiplication are operations on sets of numbers, not sets of strings. In this discussion $+$ and $*$ are simply alphabet symbols, and it would be incorrect to say that *Expr* is closed under addition and multiplication.) Along the same line, if we describe the

operation of enclosing a string within parentheses as "parenthesization", we can say that *Balanced* is the smallest language that contains Λ and is closed under concatenation and parenthesization.

A Recursive Definition of a Set of Languages over $\{a,b\}^*$ — EXAMPLE 1.20

We denote by $\mathcal{F}$ the subset of $2^{\{a,b\}^*}$ (the set of languages over $\{a, b\}$) defined as follows:

1. $\emptyset$, $\{\Lambda\}$, $\{a\}$, and $\{b\}$ are elements of $\mathcal{F}$.
2. For every L_1 and every L_2 in $\mathcal{F}$, $L_1 \cup L_2 \in \mathcal{F}$.
3. For every L_1 and every L_2 in $\mathcal{F}$, $L_1 L_2 \in \mathcal{F}$.

$\mathcal{F}$ is the smallest set of languages that contains the languages $\emptyset$, $\{\Lambda\}$, $\{a\}$, and $\{b\}$ and is closed under the operations of union and concatenation.

Some elements of $\mathcal{F}$, in addition to the four from statement 1, are $\{a, b\}$, $\{ab\}$, $\{a, b, ab\}$, $\{aba, abb, abab\}$, and $\{aa, ab, aab, ba, bb, bab\}$. The first of these is the union of $\{a\}$ and $\{b\}$, the second is the concatenation of $\{a\}$ and $\{b\}$, the third is the union of the first and second, the fourth is the concatenation of the second and third, and the fifth is the concatenation of the first and third.

Can you think of any languages over $\{a, b\}$ that are not in $\mathcal{F}$? For every string $x \in \{a, b\}^*$, the language $\{x\}$ can be obtained by concatenating $|x|$ copies of $\{a\}$ or $\{b\}$, and every set $\{x_1, x_2, \ldots, x_k\}$ of strings can be obtained by taking the union of the languages $\{x_i\}$. What could be missing?

This recursive definition is perhaps the first one in which we must remember that elements in the set we are defining are obtained by using the basis statement and one or more of the recursive statements a *finite* number of times. In the previous examples, it wouldn't have made sense to consider anything else, because natural numbers cannot be infinite, and in this book we never consider strings of infinite length. It makes sense to talk about infinite languages over $\{a, b\}$, but none of them is in $\mathcal{F}$. Statement 3 in the definition of $\mathcal{N}$ in Example 1.15 says every element of $\mathcal{N}$ *can be obtained* by using the first two statements—can be obtained, for example, by someone with a pencil and paper who is applying the first two statements in the definition in real time. For a language L to be in $\mathcal{F}$, there must be a sequence of steps, each of which involves statements in the definition, that this person could actually carry out to produce L: There must be languages L_0, L_1, L_2, $\ldots$, L_n so that L_0 is obtained from the basis statement of the definition; for each $i > 0$, L_i is either also obtained from the basis statement or obtained from two earlier L_j's using union or concatenation; and $L_n = L$. The conclusion in this example is that the set $\mathcal{F}$ is the set of all *finite* languages over $\{a, b\}$.

One final observation about certain recursive definitions will be useful in Chapter 4 and a few other places. Sometimes, although not in any of the examples so far in this section, a *finite* set can be described most easily by a recursive definition. In this case, we can take advantage of the algorithmic nature of these definitions to formulate an algorithm for obtaining the set.

EXAMPLE 1.21 The Set of Cities Reachable from City s

Suppose that C is a finite set of cities, and the relation R is defined on C by saying that for cities c and d in C, cRd if there is a nonstop commercial flight from c to d. For a particular city $s \in C$, we would like to determine the subset $r(s)$ of C containing the cities that can be reached from s, by taking zero or more nonstop flights. Then it is easy to see that the set $r(s)$ can be described by the following recursive definition.

1. $s \in r(s)$.
2. For every $c \in r(s)$, and every $d \in C$ for which cRd, $d \in r(s)$.

Starting with s, by the time we have considered every sequence of steps in which the second statement is used n times, we have obtained all the cities that can be reached from s by taking n or fewer nonstop flights. The set C is finite, and so the set $r(s)$ is finite. If $r(S)$ has N elements, then it is easy to see that by using the second statement $N - 1$ times we can find every element of $r(s)$. However, we may not need that many steps. If after n steps we have the set $r_n(s)$ of cities that can be reached from s in n or fewer steps, and $r_{n+1}(s)$ turns out to be the same set (with no additional cities), then further iterations will not add any more cities, and $r(s) = r_n(s)$. The conclusion is that we can obtain $r(s)$ using the following algorithm.

$r_0(s) = \{s\}$
$n = 0$
repeat
    $n = n + 1$
    $r_n(s) = r_{n-1}(s) \cup \{d \in C \mid cRd \text{ for some } c \in r_{n-1}(s)\}$
until $r_n(s) = r_{n-1}(s)$
$r(s) = r_n(s)$

In the same way, if we have a finite set C and a recursive definition of a subset S of C, then even if we don't know how many elements C has, we can translate our definition into an algorithm that is guaranteed to terminate and to produce the set S.

In general, if R is a relation on an arbitrary set A, we can use a recursive definition similar to the one above to obtain the *transitive closure* of R, which can be described as the smallest transitive relation containing R.

1.6 | STRUCTURAL INDUCTION

In the previous section we found a recursive definition for a language *Expr* of simple algebraic expressions. Here it is again, with the operator notation we introduced.

1. $a \in Expr$.
2. For every x and every y in *Expr*, $x \circ y$ and $x \bullet y$ are in *Expr*.
3. For every $x \in Expr$, $\diamond(x) \in Expr$.

(By definition, if x and y are elements of *Expr*, $x \circ y = x + y$, $x \bullet y = x * y$, and $\diamond(x) = (x)$.)

Suppose we want to prove that every string x in *Expr* satisfies the statement $P(x)$. (Two possibilities for $P(x)$ are the statements "x has equal numbers of left and right parentheses" and "x has an odd number of symbols".) Suppose also that the recursive definition of *Expr* provides all the information that we have about the language. How can we do it?

The principle of *structural induction* says that in order to show that $P(x)$ is true for every $x \in Expr$, it is sufficient to show:

1. $P(a)$ is true.
2. For every x and every y in *Expr*, if $P(x)$ and $P(y)$ are true, then $P(x \circ y)$ and $P(x \bullet y)$ are true.
3. For every $x \in Expr$, if $P(x)$ is true, then $P(\diamond(x))$ is true.

It's not hard to believe that this principle is correct. If the element a of *Expr* that we start with has the property we want, and if all the operations we can use to get new elements *preserve* the property (that is, when they are applied to elements having the property, they produce elements having the property), then there is no way we can ever use the definition to produce an element of *Expr* that does not have the property.

Another way to understand the principle is to use our paraphrase of the recursive definition of *Expr*. Suppose we denote by L_P the language of all strings satisfying P. Then saying every string in *Expr* satisfies P is the same as saying that $Expr \subseteq L_P$. If *Expr* is indeed the smallest language that contains a and is closed under the operations $\circ$, $\bullet$, and $\diamond$, then *Expr* is a subset of every language that has these properties, and so it is enough to show that L_P itself has them—i.e., L_P contains a and is closed under the three operations. And this is just what the principle of structural induction says.

The feature to notice in the statement of the principle is the close resemblance of the statements 1–3 to the recursive definition of *Expr*. The outline of the proof is provided by the structure of the definition. We illustrate the technique of structural induction by taking P to be the second of the two properties mentioned above and proving that every element of *Expr* satisfies it.

A Proof by Structural Induction That Every Element of *Expr* Has Odd Length

EXAMPLE 1.22

To simplify things slightly, we will combine statements 2 and 3 of our first definition into a single statement, as follows:

1. $a \in Expr$.
2. For every x and every y in *Expr*, $x + y$, $x * y$, and (x) are in *Expr*.

The corresponding statements that we will establish in our proof are these:

1. $|a|$ is odd.
2. For every x and y in *Expr*, if $|x|$ and $|y|$ are odd, then $|x + y|$, $|x * y|$, and $|(x)|$ are odd.

The *basis* statement of the proof is the statement that $|a|$ is odd, which corresponds to the basis statement $a \in Expr$ in the recursive definition of *Expr*. When we prove the conditional statement, in the *induction step* of the proof, we will assume that x and y are elements of *Expr* and that $|x|$ and $|y|$ are odd. We refer to this assumption as the *induction hypothesis*. We make no other assumptions about x and y; they are arbitrary elements of *Expr*. This is confusing at first, because it seems as though we are assuming what we're trying to prove (that for arbitrary elements x and y, $|x|$ and $|y|$ are odd). It's important to say it carefully. We are not assuming that $|x|$ and $|y|$ are odd for every x and y in *Expr*. Rather, we are considering two arbitrary elements x and y and assuming that the lengths of those two strings are odd, in order to prove the conditional statement

If $|x|$ and $|y|$ are odd, then $|x \circ y|$, $|x \bullet y|$ and $|\diamond (x)|$ are odd.

Each time we present a proof by structural induction, we will be careful to state explicitly what we are trying to do in each step. We say first what we are setting out to prove, or what the ultimate objective is; second, what the statement is that needs to be proved in the basis step, and why it is true; third, what the induction hypothesis is; fourth, what we are trying to prove in the induction step; and finally, what the steps of that proof are. Surprisingly often, if we are able to do the first four things correctly and precisely, the proof of the induction step turns out to be very easy.

Here is our proof.

To Prove: For every element x of *Expr*, $|x|$ is odd.

Basis step. We wish to show that $|a|$ is odd. This is true because $|a| = 1$.

Induction hypothesis. x and y are in *Expr*, and $|x|$ and $|y|$ are odd.

Statement to be proved in the induction step. $|x + y|$, $|x * y|$, and $|(x)|$ are odd.

Proof of induction step. The numbers $|x + y|$ and $|x * y|$ are both $|x| + |y| + 1$, because the symbols of $x + y$ include those in x, those in y, and the additional "operator" symbol. The number $|(x)|$ is $|x| + 2$, because two parentheses have been added to the symbols of x. The first number is odd because the induction hypothesis implies that it is the sum of two odd numbers plus 1, and the second number is odd because the induction hypothesis implies that it is an odd number plus 2.

EXAMPLE 1.23 Mathematical Induction

Very often, the easiest way to prove a statement of the form "For every integer $n \geq n_0$, $P(n)$" is to apply the principle of structural induction, using the recursive definition given in Example 1.11 of the subset $\{n \in \mathcal{N} \mid n \geq n_0\}$. Such a proof is referred to as a proof by mathematical induction, or simply a proof by induction. The statement $P(n)$ might be a

numerical fact or algebraic formula involving n, but in our subject it could just as easily be a statement about a set with n elements, or a string of length n, or a sequence of n steps.

The basis step is to establish the statement $P(n)$ for n_0, the smallest number in the set. The induction hypothesis is the assumption that k is a number in the set and that $P(n)$ is true when $n = k$, or that $P(k)$ is true; and the induction step is to show using this assumption that $P(k + 1)$ is true. Here is an example, in which $n_0 = 0$, so that the set is simply $\mathcal{N}$.

To prove: For every $n \in \mathcal{N}$, and every set A with n elements, 2^A has exactly 2^n elements.

Basis step. The statement to be proved is that for every set A with 0 elements, 2^A has $2^0 = 1$ element. This is true because the only set with no elements is $\emptyset$, and $2^\emptyset = \{\emptyset\}$, which has one element.

Induction hypothesis. $k \in \mathcal{N}$ and for every set A with k elements, 2^A has 2^k elements.

Statement to be proved in the induction step. For every set A with $k + 1$ elements, 2^A has 2^{k+1} elements.

Proof of induction step. If A has $k + 1$ elements, then because $k \geq 0$, it must have at least one. Let a be an element of A. Then $A - \{a\}$ has k elements. By the induction hypothesis, $A - \{k\}$ has exactly 2^k subsets, and so A has exactly 2^k subsets that do not contain a. Every subset B of A that contains a can be written $B = B_1 \cup \{a\}$, where B_1 is a subset of A that doesn't contain a, and for two different subsets containing a, the corresponding subsets not containing a are also different; therefore, there are precisely as many subsets of A that contain a as there are subsets that do not. It follows that the total number of subsets is $2 * 2^k = 2^{k+1}$.

Strong Induction

EXAMPLE 1.24

We present another proof by mathematical induction, to show that every positive integer 2 or larger can be factored into prime factors. The proof will illustrate a variant of mathematical induction that is useful in situations where the ordinary induction hypothesis is not quite sufficient.

To prove: For every natural number $n \geq 2$, n is either a prime or a product of two or more primes.

For reasons that will be clear very soon, we modify the statement to be proved, in a way that makes it seem like a stronger statement.

To prove: For every natural number $n \geq 2$, every natural number m satisfying $2 \leq m \leq n$ is either a prime or a product of two or more primes.

Basis step. When $n = 2$, the modified statement is that every number m satisfying $2 \leq m \leq 2$ is either a prime or a product of two or more primes. Of course the only such number m is 2, and the statement is true because 2 is prime.

Induction hypothesis. $k \geq 2$, and for every m satisfying $2 \leq m \leq k$, m is either a prime or a product of primes.

Statement to be proved in the induction step. For every m satisfying $2 \leq m \leq k + 1$, m is either prime or a product of primes.

Proof of induction step. For every m with $2 \leq m \leq k$, we already have the conclusion we want, from the induction hypothesis. The only additional statement we need to prove is that $k + 1$ is either prime or a product of primes.

If $k + 1$ is prime, then the statement we want is true. Otherwise, by the definition of a prime, $k + 1 = r * s$, for some positive integers r and s, neither of which is 1 or $k + 1$. It follows that $2 \leq r \leq k$ and $2 \leq s \leq k$, and so the induction hypothesis implies that each of the two is either prime or a product of primes. We may conclude that their product $k + 1$ is the product of two or more primes.

As we observed in the proof, the basis step and the statement to be proved in the induction step were not changed at all as a result of modifying the original statement. The purpose of the modification is simply to allow us to use the stronger induction hypothesis: not only that the statement $P(n)$ is true when $n = k$, but that it is true for every n satisfying $2 \leq n \leq k$. This was not necessary in Example 1.23, but often you will find when you reach the proof of the induction step that the weaker hypothesis doesn't provide enough information. In this example, it tells us that k is a prime or a product of primes—but we need to know that the numbers r and s have this property, and neither of these is k.

Now that we have finished this example, you don't have to modify the statement to be proved when you encounter another situation where the stronger induction hypothesis is necessary; declaring that you are using *strong induction* allows you to assume it.

In Example 1.19 we gave a recursive definition of the language *Balanced*, the set of balanced strings of parentheses. When you construct an algebraic expression or an expression in a computer program, and you need to end up with a balanced string of parentheses, you might check your work using an algorithm like the following. Go through the string from left to right, and keep track of the number of excess left parentheses; start at 0, add 1 each time you hit a left parenthesis, and subtract 1 each time you hit a right parenthesis; if the number is 0 when you reach the end and has never dropped below 0 along the way, the string is balanced. We mentioned at the beginning of this section that strings in *Expr* or *Balanced* have equal numbers of left and right parentheses; strengthening the condition by requiring that no prefix have more right parentheses than left produces a condition that characterizes balanced strings and explains why this algorithm works.

EXAMPLE 1.25 Another Characterization of Balanced Strings of Parentheses

The language *Balanced* was defined as follows in Example 1.19.

1. $\Lambda \in$ *Balanced*.
2. For every x and every y in *Balanced*, both xy and (x) are elements of *Balanced*.

We wish to show that a string x belongs to this language if and only if the statement $B(x)$ is true:

> $B(x)$: x contains equal numbers of left and right parentheses, and no prefix of x contains more right than left.

For the first part, showing that every string x in *Balanced* makes the condition $B(x)$ true, we can use structural induction, because we have a recursive definition of *Balanced*. The basis step is to show that $B(\Lambda)$ is true, and it is easy to see that it is. The induction hypothesis is that x and y are two strings in *Balanced* for which $B(x)$ and $B(y)$ are true, and the statement to be proved in the induction step is that $B(xy)$ and $B((x))$ are both true. We will show the first statement, and the second is at least as simple.

Because x and y both have equal numbers of left and right parentheses, the string xy does also. If z is a prefix of xy, then either z is a prefix of x or $z = xw$ for some prefix w of y. In the first case, the induction hypothesis tells us $B(x)$ is true, which implies that z cannot have more right parentheses than left. In the second case, the induction hypothesis tells us that x has equal numbers of left and right parentheses and that w cannot have more right than left; therefore, xw cannot have more right than left.

For the second part of the proof, we can't use structural induction based on the recursive definition of *Balanced*, because we're trying to prove that *every* string of parentheses, not just every string in *Balanced*, satisfies some property. There is not a lot to be gained by trying to use structural induction based on a recursive definition of the set of strings of parentheses, and instead we choose strong induction, rewriting the statement so that it involves an integer explicitly:

To prove: For every $n \in \mathcal{N}$, if x is a string of parentheses so that $|x| = n$ and $B(x)$ is true, then $x \in$ *Balanced*.

> **Basis step.** The statement in the basis step is that if $|x| = 0$ and $B(x)$, then $x \in$ *Balanced*. We have more assumptions than we need; if $|x| = 0$, then $x = \Lambda$, and so $x \in$ *Balanced* because of statement 1 in the definition of *Balanced*.
>
> **Induction hypothesis.** $k \in \mathcal{N}$, and for every string x of parentheses, if $|x| \leq k$ and $B(x)$, then $x \in$ *Balanced*. (Writing "for every string x of parentheses, if $|x| \leq k$" says the same thing as "for every $m \leq k$, and every string x of parentheses with $|x| = m$" but involves one fewer variable and sounds a little simpler.)
>
> **Statement to be proved in induction step.** For every string x of parentheses, if $|x| = k + 1$ and $B(x)$, then $x \in$ *Balanced*.
>
> **Proof of induction step.** We suppose that x is a string of parentheses with $|x| = k + 1$ and $B(x)$. We must show that $x \in$ *Balanced*, which means that x can be obtained from statement 1 or statement 2 in the definition. Since $|x| > 0$, statement 1 won't help; we must show x can be obtained from statement 2. The trick here is to look at the two cases in statement 2 and work backward.
>
> If we want to show that $x = yz$ for two shorter strings y and z in *Balanced*, the way to do it is to show that $x = yz$ for two shorter strings y and z satisfying $B(y)$ and $B(z)$; for then the induction hypothesis will tell us that y and z are in *Balanced*. However, this may not be possible, because the statements $B(y)$ and $B(z)$ require that

y and z both have equal numbers of left and right parentheses. The string ((())), for example, cannot be expressed as a concatenation like this. We must show that these other strings can be obtained from statement 2.

It seems reasonable, then, to consider two cases. Suppose first that $x = yz$, where y and z are both shorter than x and have equal numbers of left and right parentheses. No prefix of y can have more right parentheses than left, because every prefix of y is a prefix of x and $B(x)$ is true. Because y has equal numbers of left and right, and because no prefix of x can have more right than left, no prefix of z can have more right than left. Therefore, both the statements $B(y)$ and $B(z)$ are true. Since $|y| \leq k$ and $|z| \leq k$, we can apply the induction hypothesis to both strings and conclude that y and z are both elements of *Balanced*. It follows from statement 2 of the definition that $x = yz$ is also.

In the other case, we assume that $x = (y)$ for some string y of parentheses and that x cannot be written as a concatenation of shorter strings with equal numbers of left and right parentheses. This second assumption is useful, because it tells us that no prefix of y can have more right parentheses than left. (If some prefix did, then some prefix y_1 of y would have exactly one more right than left, which would mean that the prefix (y_1 of x had equal numbers; but this would contradict the assumption.) The string y has equal numbers of left and right parentheses, because x does, and so the statement $B(y)$ is true. Therefore, by the induction hypothesis, $y \in$ *Balanced*, and it follows from statement 2 that $x \in$ *Balanced*.

EXAMPLE 1.26 Sometimes Making a Statement Stronger Makes It Easier to Prove

In this example we return to the recursive definition of *Expr* that we have already used in Example 1.22 to prove a simple property of algebraic expressions. Suppose we want to prove now that no string in *Expr* can contain the substring ++.

In the basis step we observe that a does not contain this substring. If we assume in the induction hypothesis that neither x nor y contains it, then it is easy to conclude that $x * y$ and (x) also don't. Trying to prove that $x + y$ doesn't, however, presents a problem. Neither x nor y contains ++ as a substring, but if x ended with + or y started with +, then ++ would occur in the concatenation. The solution is to prove the stronger statement that for every $x \in$ *Expr*, x doesn't begin or end with + *and* doesn't contain the substring ++. In both the basis step and the induction step, there will be a few more things to prove, but they are not difficult and the induction hypothesis now contains all the information we need to carry out the proof.

EXAMPLE 1.27 Defining Functions on Sets Defined Recursively

A recursive definition of a set suggests a way to prove things about elements of the set. By the same principle, it offers a way of defining a function at every element of the set. In the case of the natural numbers, for example, if we define a function at 0, and if for every natural number n we say what $f(n + 1)$ is, assuming that we know what $f(n)$ is, then

we have effectively defined the function at every element of $\mathcal{N}$. There are many familiar functions commonly defined this way, such as the factorial function f:

$$f(0) = 1; \text{ for every } n \in \mathcal{N}, f(n+1) = (n+1) * f(n)$$

and the function $u : \mathcal{N} \to 2^A$ defined by

$$u(0) = S_0; \text{ for every } n \in \mathcal{N}, u(n+1) = u(n) \cup S_{n+1}$$

where $S_0, S_1, \ldots$ are assumed to be subsets of A. Writing nonrecursive definitions of these functions is also common:

$$f(n) = n * (n-1) * (n-2) * \cdots * 2 * 1$$
$$u(n) = S_0 \cup S_1 \cup S_2 \cup \cdots \cup S_n$$

In the second case, we could avoid "..." by writing

$$u(n) = \bigcup_{i=0}^{n} S_i = \bigcup \{S_i \mid 0 \leq i \leq n\}$$

although the recursive definition also provides concise definitions of both these notations.

It is easy to see that definitions like these are particularly well suited for induction proofs of properties of the corresponding functions, and the exercises contain a few examples. We consider a familiar function $r : \{a, b\}^* \to \{a, b\}^*$ that can be defined recursively by referring to the recursive definition of $\{a, b\}^*$ in Example 1.17.

$$r(\Lambda) = \Lambda; \text{ for every } x \in \{a, b\}^*, r(xa) = ar(x) \text{ and } r(xb) = br(x).$$

If it is not obvious what the function r is, you can see after using the definition to compute $r(aaba)$, for example,

$$r(aaba) = ar(aab) = abr(aa) = abar(a) = abar(\Lambda a) = abaar(\Lambda) = abaa\Lambda = abaa$$

that it is the function that reverses the order of the symbols of a string. We will often use the notation x^r instead of $r(x)$, in this example as well as several places where this function makes an appearance later.

To illustrate the close relationship between the recursive definition of $\{a, b\}^*$, the recursive definition of r, and the principle of structural induction, we prove the following fact about the reverse function.

$$\text{For every } x \text{ and every } y \text{ in } \{a, b\}^*, (xy)^r = y^r x^r$$

In planning the proof, we are immediately faced with a potential problem, because the statement has the form "for every x and every y, ..." rather than the simpler form "for every x, ..." that we have considered before. The first step in resolving this issue is to realize that the quantifiers are nested; we can write the statement in the form $\forall x(P(x))$, where $P(x)$ is itself a quantified statement, $\forall y(\ldots)$, and so we can attempt to use structural induction on x.

In fact, although the principle here is reasonable, you will discover if you try this approach that it doesn't work. It will be easier to see why after we have completed the proof using the approach that does work.

The phrase "for every x and every y" means the same thing as "for every y and every x", and now the corresponding formula looks like $\forall y(\ldots)$. As we will see, this turns out to be better because of the order in which x and y appear in the expression $r(xy)$.

To prove: For every y in $\{a, b\}^*$, $P(y)$ is true, where $P(y)$ is the statement "for every $x \in \{a, b\}^*$, $(xy)^r = y^r x^r$".

Basis step. The statement to be proved in the basis step is this: For every $x \in \{a, b\}^*$, $(x\Lambda)^r = \Lambda^r x^r$. This statement is true, because for every x, $x\Lambda = x$; Λ^r is defined to be Λ; and $\Lambda x^r = x^r$.

Induction hypothesis. $y \in \{a, b\}^*$, and for every $x \in \{a, b\}^*$, $(xy)^r = y^r x^r$.

Statement to be proved in induction step. For every $x \in \{a, b\}^*$, $(x(ya))^r = (ya)^r x^r$ and $(x(yb))^r = (yb)^r x^r$.

Proof of induction step. We will prove the first part of the statement, and the proof in the second part is almost identical. The tools that we have available are the recursive definition of $\{a, b\}^*$, the recursive definition of r, and the induction hypothesis; all we have to do is decide which one to use when, and avoid getting lost in the notation.

$$
\begin{aligned}
(x(ya))^r &= ((xy)a)^r && \text{(because concatenation is associative—i.e., } x(ya) = (xy)a\text{)} \\
&= a(xy)^r && \text{(by the second part of the definition of } r\text{, with } xy \text{ instead of } x\text{)} \\
&= a(y^r x^r) && \text{(by the induction hypothesis)} \\
&= (ay^r)x^r && \text{(because concatenation is associative)} \\
&= (ya)^r x^r && \text{(by the second part of the definition of } r\text{, with } y \text{ instead of } x\text{)}
\end{aligned}
$$

Now you can see why the first approach wouldn't have worked. Using induction on x, we would start out with $((xa)y)^r$. We can rewrite this as $(x(ay))^r$, and the induction hypothesis in this version would allow us to rewrite it again as $(ay)^r x^r$. But the a is at the wrong end of the string ay, and the definition of r gives us no way to proceed further.

EXERCISES

1.1. In each case below, construct a truth table for the statement and find another statement with at most one operator ($\vee$, $\wedge$, $\neg$, or $\rightarrow$) that is logically equivalent.

a. $(p \rightarrow q) \wedge (p \rightarrow \neg q)$

b. $p \vee (p \rightarrow q)$

c. $p \wedge (p \rightarrow q)$

d. $(p \rightarrow q) \wedge (\neg p \rightarrow q)$

e. $p \leftrightarrow (p \leftrightarrow q)$

f. $q \wedge (p \rightarrow q)$

1.2. A principle of classical logic is *modus ponens*, which asserts that the proposition $(p \wedge (p \rightarrow q)) \rightarrow q$ is a tautology, or that $p \wedge (p \wedge q)$ logically implies q. Is there any way to define the conditional statement $p \rightarrow q$, other than the way we defined it, that makes it false when p is true and q is false and makes the modus ponens proposition a tautology? Explain.

1.3. Suppose m_1 and m_2 are integers representing months ($1 \leq m_i \leq 12$), and d_1 and d_2 are integers representing days (d_i is at least 1 and no

larger than the number of days in month m_i). For each i, the pair (m_i, d_i) can be thought of as representing a date; for example, (9, 18) represents September 18. We wish to write a logical proposition involving the four integers that says (m_1, d_1) comes before (m_2, d_2) in the calendar.

a. Find such a proposition that is a disjunction of two propositions (i.e., combines them using $\vee$).

b. Find such a proposition that is a conjunction of two propositions (combines them using $\wedge$).

1.4. In each case below, say whether the statement is a tautology, a contradiction, or neither.

a. $p \vee \neg(p \rightarrow p)$

b. $p \wedge \neg(p \rightarrow p)$

c. $p \rightarrow \neg p$

d. $(p \rightarrow \neg p) \vee (\neg p \rightarrow p)$

e. $(p \rightarrow \neg p) \wedge (\neg p \rightarrow p)$

f. $(p \wedge q) \vee (\neg p) \vee (\neg q)$

1.5. In the nine propositions $p \wedge q \vee r$, $p \vee q \wedge r$, $\neg p \wedge q$, $\neg p \vee q$, $\neg p \rightarrow q$, $p \vee q \rightarrow r$, $p \wedge q \rightarrow r$, $p \rightarrow q \vee r$, and $p \rightarrow q \wedge r$, the standard convention if no parentheses are used is to give $\neg$ the highest precedence, $\wedge$ the next-highest, $\vee$ the next-highest after that, and $\rightarrow$ the lowest. For example, $\neg p \vee r$ would normally be interpreted $(\neg p) \vee r$ and $p \rightarrow q \vee r$ would normally be interpreted $p \rightarrow (q \vee r)$. Are there any of the nine whose truth value would be unchanged if the precedence of the two operators involved were reversed? If so, which ones?

1.6. Prove that every string of length 4 over the alphabet $\{a, b\}$ contains the substring xx, for some nonnull string x. One way is to consider all sixteen cases, but try to reduce the number of cases as much as possible.

1.7. Describe each of the following infinite sets using the format $\{\_\_\_\_\_ \mid n \in \mathcal{N}\}$, without using "..." in the expression on the left side of the vertical bar.

a. $\{0, -1, 2, -3, 4, -5, \ldots\}$

b. $\{\{0\}, \{1\}, \{2\}, \ldots\}$

c. $\{\{0\}, \{0, 1\}, \{0, 1, 2\}, \{0, 1, 2, 3\}, \ldots\}$

d. $\{\{0\}, \{0, 1\}, \{0, 1, 2, 3\}, \{0, 1, 2, 3, 4, 5, 6, 7\}, \{0, 1, \ldots, 15\}, \{0, 1, 2, \ldots, 31\}, \ldots\}$

1.8. In each case below, find an expression for the indicated set, involving A, B, C, and any of the operations $\cup$, $\cap$, $-$, and $'$.

a. $\{x | x \in A$ or $x \in B$ but not both$\}$

b. $\{x | x$ is an element of exactly one of the three sets A, B, and $C\}$

c. $\{x | x$ is an element of at most one of the three sets A, B, and $C\}$

d. $\{x | x$ is an element of exactly two of the three sets A, B, and $C\}$

1.9. For each integer n, denote by C_n the set of all real numbers less than n, and for each positive number n let D_n be the set of all real numbers less than $1/n$. Express each of the following unions or intersections in a simpler way. For example, the answer to (a) is C_{10}. The answer is not always one of the sets C_i or D_i, but there is an equally simple answer in each case. Since ∞ is not a number, the expressions C_∞ and D_∞ do not make sense and should not appear in your answers.

a. $\bigcup\{C_n \mid 1 \leq n \leq 10\}$
b. $\bigcup\{D_n \mid 1 \leq n \leq 10\}$
c. $\bigcap\{C_n \mid 1 \leq n \leq 10\}$
d. $\bigcap\{D_n \mid 1 \leq n \leq 10\}$
e. $\bigcup\{C_n \mid 1 \leq n\}$
f. $\bigcup\{D_n \mid 1 \leq n\}$
g. $\bigcap\{C_n \mid 1 \leq n\}$
h. $\bigcap\{D_n \mid 1 \leq n\}$
i. $\bigcup\{C_n \mid n \in \mathcal{Z}\}$
j. $\bigcap\{C_n \mid n \in \mathcal{Z}\}$

1.10. List the elements of $2^{2^{\{0,1\}}}$, and number the items in your list.

1.11. In each case below, say whether the given statement is true for the universe $(0, 1) = \{x \in \mathcal{R} \mid 0 < x < 1\}$, and say whether it is true for the universe $[0, 1] = \{x \in \mathcal{R} \mid 0 \leq x \leq 1\}$. For each of the four cases, you should therefore give two true-or-false answers.

a. $\forall x(\exists y(x > y))$
b. $\forall x(\exists y(x \geq y))$
c. $\exists y(\forall x(x > y))$
d. $\exists y(\forall x(x \geq y))$

1.12. a. How many elements are there in the set

$$\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}\}\}?$$

b. Describe precisely the algorithm you used to answer part (a).

1.13. Simplify the given set as much as possible in each case below. Assume that all the numbers involved are real numbers.

a. $\bigcap\{\{x \mid |x - a| < r\} \mid r > 0\}$
b. $\bigcup\{\{x \mid |x - a| \leq r\} \mid r > 0$

1.14. Suppose that A and B are nonempty sets and $A \times B \subseteq B \times A$. Show that $A = B$. Suggestion: show that $A \subseteq B$ and $B \subseteq A$, using proof by contradiction in each case.

1.15. Suppose that A and B are subsets of a universal set U.

a. What is the relationship between $2^{A \cup B}$ and $2^A \cup 2^B$? (Under what circumstances are they equal? If they are not equal, is one necessarily a subset of the other, and if so, which one?) Give reasons for your answers.

b. Same question for $2^{A \cap B}$ and $2^A \cap 2^B$.

c. Same question for $2^{(A')}$ and $(2^A)'$ (both subsets of 2^U).

1.16. Suppose A and B are finite sets, A has n elements, and $f : A \to B$.

a. If f is one-to-one, what can you say about the number of elements of B?

b. If f is onto, what can you say about the number of elements of B?

1.17. In each case below, say whether the indicated function is one-to-one and what its range is.

a. $m : \mathcal{N} \to \mathcal{N}$ defined by $m(x) = \min(x, 2)$

b. $M : \mathcal{N} \to \mathcal{N}$ defined by $M(x) = \max(x, 2)$

c. $s : \mathcal{N} \to \mathcal{N}$ defined by $s(x) = m(x) + M(x)$

d. $f : \mathcal{N} - \{0\} \to 2^{\mathcal{N}}$, where $f(n)$ is the set of prime factors of n

e. (Here A is the set of all finite sets of primes and B is the set $\mathcal{N} - \{0\}$.) $g : A \to B$, where $g(S)$ is the product of the elements of S. (The product of the elements of the empty set is 1.)

1.18. Find a formula for a function from $\mathcal{Z}$ to $\mathcal{N}$ that is a bijection.

1.19. In each case, say whether the function is one-to-one and whether it is onto.

a. $f : \mathcal{Z} \times \mathcal{Z} \to \mathcal{Z} \times \mathcal{Z}$, defined by $f(a, b) = (a + b, a - b)$

b. $f : \mathcal{R} \times \mathcal{R} \to \mathcal{R} \times \mathcal{R}$, defined by $f(a, b) = (a + b, a - b)$

1.20. Suppose A and B are sets and $f : A \to B$. For a subset S of A, we use the notation $f(S)$ to denote the set $\{f(x) \mid x \in S\}$. Let S and T be subsets of A.

a. Is the set $f(S \cup T)$ a subset of $f(S) \cup f(T)$? If so, give a proof; if not, give a counterexample (i.e., say what the sets A, B, S, and T are and what the function f is).

b. Is the set $f(S) \cup f(T)$ a subset of $f(S \cup T)$? Give either a proof or a counterexample.

c. Repeat part (a) with intersection instead of union.

d. Repeat part (b) with intersection instead of union.

e. In each of the first four parts where your answer is no, what extra assumption on the function f would make the answer yes? Give reasons for your answer.

1.21. Let E be the set of even natural numbers, S the set of nonempty subsets of E, T the set of nonempty subsets of $\mathcal{N}$, and $\mathcal{P}$ the set of partitions of $\mathcal{N}$ into two nonempty subsets.

a. Suppose $f : T \to \mathcal{P}$ is defined by the formula $f(A) = \{A, \mathcal{N} - A\}$ (in other words, for a nonempty subset A of $\mathcal{N}$, $f(A)$ is the partition of $\mathcal{N}$ consisting of the two subsets A and $\mathcal{N} - A$). Is f a bijection from T to $\mathcal{P}$? Why or why not?

b. Suppose that $g : S \to \mathcal{P}$ is defined by $g(A) = \{A, \mathcal{N} - A\}$. Is g a bijection from S to $\mathcal{P}$? Why or why not?

1.22. Suppose U is a set, $\circ$ is a binary operation on U, and S_0 is a subset of U. Define the subset A of U recursively as follows:

$$S_0 \subseteq A; \text{ for every } x \text{ and } y \text{ in } A, x \circ y \in A$$

(In other words, A is the smallest subset of A that contains the elements of S_0 and is closed under $\circ$.) Show that

$$A = \bigcap \{S \mid S_0 \subseteq S \subseteq U \text{ and } S \text{ is closed under } \circ\}$$

1.23. In each case below, a relation on the set $\{1, 2, 3\}$ is given. Of the three properties, reflexivity, symmetry, and transitivity, determine which ones the relation has. Give reasons.

a. $R = \{(1, 3), (3, 1), (2, 2)\}$

b. $R = \{(1, 1), (2, 2), (3, 3), (1, 2)\}$

c. $R = \emptyset$

1.24. For each of the eight lines of the table below, construct a relation on $\{1, 2, 3\}$ that fits the description.

reflexive	symmetric	transitive
true	true	true
true	true	false
true	false	true
true	false	false
false	true	true
false	true	false
false	false	true
false	false	false

1.25. Each case below gives a relation on the set of all nonempty subsets of $\mathcal{N}$. In each case, say whether the relation is reflexive, whether it is symmetric, and whether it is transitive.

a. R is defined by: ARB if and only if $A \subseteq B$.

b. R is defined by: ARB if and only if $A \cap B \neq \emptyset$.

c. R is defined by: ARB if and only if $1 \in A \cap B$.

1.26. Let R be a relation on a set S. Write three quantified statements (the domain being S in each case), which say, respectively, that R is not reflexive, R is not symmetric, and R is not transitive.

1.27. Suppose S is a nonempty set, $A = 2^S$, and the relation R on A is defined as follows: For every X and every Y in A, XRY if and only if there is a bijection from X to Y.

a. Show that R is an equivalence relation.

b. If S is a finite set with n elements, how many equivalence classes does the equivalence relation R have?

c. Again assuming that S is finite, describe a function $f : A \to \mathcal{N}$ so that for every X and Y in A, XRY if and only if $f(X) = f(Y)$.

1.28. Suppose A and B are sets, $f : A \to B$ is a function, and R is the relation on A so that for $x, y \in A$, xRy if and only if $f(x) = f(y)$.

a. Show that R is an equivalence relation on A.

b. If $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, $B = \mathcal{N}$, and $f(x) = (x - 3)^2$ for every $x \in A$, how many equivalence classes are there, and what are the elements of each one?

c. Suppose A has p elements and B has q elements. If the function f is one-to-one (not necessarily onto), how many equivalence classes does the equivalence relation R have? If the function f is onto (not necessarily one-to-one), how many equivalence classes does R have?

1.29. Show that for every set A and every equivalence relation R on A, there is a set B and a function $f : A \to B$ such that R is the relation described in Exercise 1.28.

1.30. For a positive integer n, find a function $f : \mathcal{N} \to \mathcal{N}$ so that the equivalence relation $\equiv_n$ on $\mathcal{N}$ can be described as in Exercise 1.28.

1.31. Show that for every language L, $LL^* = L^*$ if and only if $\Lambda \in L$.

1.32. For a finite language L, let $|L|$ denote the number of elements of L. For example, $|\{\Lambda, a, ababb\}| = 3$. This notation has nothing to do with the length $|x|$ of a string x. The statement $|L_1L_2| = |L_1||L_2|$ says that the number of strings in the concatenation L_1L_2 is the same as the product of the two numbers $|L_1|$ and $|L_2|$. Is this always true? If so, give reasons, and if not, find two finite languages $L_1, L_2 \subseteq \{a, b\}^*$ such that $|L_1L_2| \neq |L_1||L_2|$.

1.33. Let L_1 and L_2 be subsets of $\{a, b\}^*$.

a. Show that if $L_1 \subseteq L_2$, then $L_1^* \subseteq L_2^*$.

b. Show that $L_1^* \cup L_2^* \subseteq (L_1 \cup L_2)^*$.

c. Give an example of two languages L_1 and L_2 such that $L_1^* \cup L_2^* \neq (L_1 \cup L_2)^*$.

d. †One way for the two languages $L_1^* \cup L_2^*$ and $(L_1 \cup L_2)^*$ to be equal is for one of the two languages L_1 and L_2 to be a subset of the other, or more generally, for one of the two languages L_1^* and L_2^* to be a subset of the other. Find an example of languages L_1 and L_2 for which neither of L_1^*, L_2^* is a subset of the other, but $L_1^* \cup L_2^* = (L_1 \cup L_2)^*$.

1.34. †Suppose that $x, y \in \{a, b\}^*$ and neither is Λ. Show that if $xy = yx$, then for some string z and two integers i and j, $x = z^i$ and $y = z^j$.

1.35. †Consider the language $L = \{yy \mid y \in \{a, b\}^*\}$. We know that $L = L\{\Lambda\} = \{\Lambda\}L$, because every language L has this property. Is there any other way to express L as the concatenation of two languages? Prove your answer.

1.36. a. Consider the language L of all strings of a's and b's that do not end with b and do not contain the substring bb. Find a finite language S such that $L = S^*$.

b. Show that there is no language S such that S^* is the language of all strings of a's and b's that do not contain the substring bb.

1.37. Let L_1, L_2, and L_3 be languages over some alphabet Σ. In each case below, two languages are given. Say what the relationship is between them. (Are they always equal? If not, is one always a subset of the other?) Give reasons for your answers, including counterexamples if appropriate.

a. $L_1(L_2 \cap L_3)$, $L_1L_2 \cap L_1L_3$

b. $L_1^* \cap L_2^*$, $(L_1 \cap L_2)^*$

c. $L_1^*L_2^*$, $(L_1L_2)^*$

1.38. In each case below, write a quantified statement, using the formal notation discussed in the chapter, that expresses the given statement. In both cases the set A is assumed to be a subset of the domain, not necessarily the entire domain.

a. There are at least two distinct elements in the set A satisfying the condition P (i.e., for which the proposition $P(x)$ holds).

b. There is exactly one element x in the set A satisfying the condition P.

1.39. Consider the following 'proof' that every symmetric, transitive relation R on a set A must also be reflexive:

> Let a be any element of A. Let b be any element of A for which aRb. Then since R is symmetric, bRa. Now since R is transitive, and since aRb and bRa, it follows that aRa. Therefore R is reflexive.

Your answer to Exercise 1.24 shows that this proof cannot be correct. What is the first incorrect statement in the proof, and why is it incorrect?

1.40. †Suppose A is a set having n elements.

a. How many relations are there on A?

b. How many reflexive relations are there on A?

c. How many symmetric relations are there on A?

d. How many relations are there on A that are both reflexive and symmetric?

1.41. Suppose R is a relation on a nonempty set A.

a. Define $R^s = R \cup \{(x, y) \mid yRx\}$. Show that R^s is symmetric and is the smallest symmetric relation on A containing R (i.e., for any symmetric relation R_1 with $R \subseteq R_1$, $R^s \subseteq R_1$).

b. Define R^t to be the intersection of all transitive relations on A containing R. Show that R^t is transitive and is the smallest transitive relation on A containing R.

c. Let $R^u = R \cup \{(x, y) \mid \exists z(xRz \text{ and } zRy)\}$. Is R^u equal to the set R^t in part (b)? Either prove that it is, or give an example in which it is not.

The relations R^s and R^t are called the symmetric closure and transitive closure of R, respectively.

1.42. Suppose R is an equivalence relation on a set A. A subset $S \subseteq A$ is *pairwise inequivalent* if no two distinct elements of S are equivalent. S is a *maximal* pairwise inequivalent set if S is pairwise inequivalent and for every element of A, there is an element of S equivalent to it. Show that a set S is a maximal pairwise inequivalent set if and only if it contains exactly one element of each equivalence class.

1.43. Suppose R_1 and R_2 are equivalence relations on a set A. As discussed in Section 1.3, the equivalence classes of R_1 and R_2 form partitions P_1 and P_2, respectively, of A. Show that $R_1 \subseteq R_2$ if and only if the partition P_1 is *finer* than P_2 (i.e., every subset in the partition P_2 is the union of one or more subsets in the partition P_1).

1.44. Each case below gives, a recursive definition of a subset L of $\{a, b\}^*$. Give a simple nonrecursive definition of L in each case.

a. $a \in L$; for any $x \in L$, xa and xb are in L.

b. $a \in L$; for any $x \in L$, bx and xb are in L.

c. $a \in L$; for any $x \in L$, ax and xb are in L.

d. $a \in L$; for any $x \in L$, xb, xa, and bx are in L.

e. $a \in L$; for any $x \in L$, xb, ax, and bx are in L.

f. $a \in L$; for any $x \in L$, xb and xba are in L.

1.45. Prove using mathematical induction that for every nonnegative integer n,

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$$

(If $n = 0$, the sum on the left is 0 by definition.)

1.46. Suppose r is a real number other than 1. Prove using mathematical induction that for every nonnegative integer n,

$$\sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}$$

1.47. Prove using mathematical induction that for every nonnegative integer n,

$$1 + \sum_{i=1}^{n} i * i! = (n+1)!$$

1.48. Prove using mathematical induction that for every integer $n \geq 4$, $n! > 2^n$.

1.49. Suppose x is any real number greater than -1. Prove using mathematical induction that for every nonnegative integer n, $(1 + x)^n \geq 1 + nx$. (Be sure you say in your proof exactly how you use the assumption that $x > -1$.)

1.50. Prove using mathematical induction that for every positive integer n,

$$\sum_{i=1}^{n} i * 2^i = (n-1) * 2^{n+1} + 2$$

1.51. Prove using mathematical induction that for every nonnegative integer n, n is either even or odd but not both. (By definition, an integer n is even if there is an integer i so that $n = 2 * i$, and n is odd if there is an integer i so that $n = 2 * i + 1$.)

1.52. Prove that for every language $L \subseteq \{a, b\}^*$, if $L^2 \subseteq L$, then $LL^* \subseteq L$.

1.53. Suppose that Σ is an alphabet, and that $f : \Sigma^* \to \Sigma^*$ has the property that $f(\sigma) = \sigma$ for every $\sigma \in \Sigma$ and $f(xy) = f(x)f(y)$ for every $x, y \in \Sigma^*$. Prove that for every $x \in \Sigma^*$, $f(x) = x$.

1.54. Prove that for every positive integer n, there is a nonnegative integer i and an odd integer j so that $n = 2^i * j$.

1.55. Show using mathematical induction that for every $x \in \{a, b\}^*$ such that x begins with a and ends with b, x contains the substring ab.

1.56. Show using mathematical induction that every nonempty subset A of $\mathcal{N}$ has a smallest element. (Perhaps the hardest thing about this problem is finding a way of formulating the statement so that it involves an integer n and can therefore be proved by induction. Why is it *not* feasible to prove that for every integer $n \geq 1$, every subset A of $\mathcal{N}$ containing at least n elements has a smallest element?)

1.57. Some recursive definitions of functions on $\mathcal{N}$ don't seem to be based directly on the recursive definition of $\mathcal{N}$ in Example 1.15. The *Fibonacci* function f is usually defined as follows.

$$f(0) = 0;\ \ f(1) = 1;\ \text{ for every } n > 1,\ f(n) = f(n-1) + f(n-2).$$

Here we need to give both the values $f(0)$ and $f(1)$ in the first part of the definition, and for each larger n, $f(n)$ is defined using both $f(n-1)$ and $f(n-2)$. If there is any doubt in your mind, you can use strong induction to verify that for every $n \in \mathcal{N}$, $f(n)$ is actually defined. Use strong induction to show that for every $n \in \mathcal{N}$, $f(n) \leq (5/3)^n$. (Note that in the induction step, you can use the recursive formula only if $n > 1$; checking the case $n = 1$ separately is comparable to performing a second basis step.)

1.58. The numbers a_n, for $n \geq 0$, are defined recursively as follows.

$$a_0 = -2;\ \ a_1 = -2;\quad \text{for } n \geq 2,\ a_n = 5a_{n-1} - 6a_{n-2}$$

Use strong induction to show that for every $n \geq 0$, $a_n = 2 * 3^n - 4 * 2^n$. (Refer to Example 1.24.)

1.59. Show that the set B in Example 1.11 is precisely the set $S = \{2^i * 5^j \mid i, j \in \mathcal{N}\}$.

1.60. Suppose the language $L \subseteq \{a, b\}^*$ is defined recursively as follows:

$$\Lambda \in L;\qquad \text{for every } x \in L, \text{ both } ax \text{ and } axb \text{ are elements of } L.$$

Show that $L = L_0$, where $L_0 = \{a^i b^j \mid i \geq j\}$. To show that $L \subseteq L_0$ you can use structural induction, based on the recursive definition of L. In the other direction, use strong induction on the length of a string in L_0.

1.61. Find a recursive definition for the language $L = \{a^i b^j \mid i \leq j \leq 2i\}$, and show that it is correct (i.e., show that the language described by the recursive definition is precisely L). In order to come up with a recursive definition, it may be helpful to start with recursive definitions for each of the languages $\{a^i b^i \mid i \geq 0\}$ and $\{a^i b^{2i} \mid i \geq 0\}$.

1.62. In each case below, find a recursive definition for the language L and show that it is correct.

a. $L = \{a^i b^j \mid j \geq 2i\}$

b. $L = \{a^i b^j \mid j \leq 2i\}$

1.63. For a string x in the language *Expr* defined in Example 1.19, $n_a(x)$ denotes the number of a's in the string, and we will use $n_{\text{op}}(x)$ to stand for the number of operators in x (the number of occurrences of $+$ or $*$). Show that for every $x \in Expr$, $n_a(x) = 1 + n_{\text{op}}(x)$.

1.64. For a string x in *Expr*, show that x does not start or end with $+$ and does not contain the substring $++$. (In this case it would be feasible to prove, first, that strings in *Expr* do not start or end with $+$, and then that they don't contain the substring $++$; as Example 1.26 suggests, however, it is possible to prove both statements in the same induction proof.)

1.65. Suppose $L \subseteq \{a, b\}^*$ is defined as follows:

$$\Lambda \in L; \quad \text{for every } x \in L, \text{ both } xa \text{ and } xba \text{ are in } L.$$

Show that for every $x \in L$, both of the following statements are true.

a. $n_a(x) \geq n_b(x)$.

b. x does not contain the substring bb.

1.66. Suppose $L \subseteq \{a, b\}^*$ is defined as follows:

$\Lambda \in L$; for every x and y in L, the strings axb, bxa, and xy are in L.

Show that $L = AEqB$, the language of all strings x in $\{a, b\}^*$ satisfying $n_a(x) = n_b(x)$.

1.67. †Suppose $L \subseteq \{a, b\}^*$ is defined as follows:

$\Lambda \in L$; for every x and y in L, the strings $axby$ and $bxay$ are in L.

Show that $L = AEqB$, the language of all strings x in $\{a, b\}^*$ satisfying $n_a(x) = n_b(x)$.

1.68. †Suppose $L \subseteq \{a, b\}^*$ is defined as follows:

$$a \in L; \quad \text{for every } x \text{ and } y \text{ in } L, \text{ the strings } ax, bxy, xby, \text{ and } xyb \text{ are in } L.$$

Show that $L = L_0$, the language of all strings x in $\{a, b\}^*$ satisfying $n_a(x) > n_b(x)$.

1.69. For a relation R on a set S, the *transitive closure* of R is the relation R^t defined as follows: $R \subseteq R^t$; for every x, every y, and every z in S, if $(x, y) \in R^t$ and $(y, z) \in R^t$, then $(x, z) \in R^t$. (We can summarize the definition by saying that R^t is the smallest transitive relation containing R.) Show that if R_1 and R_2 are relations on S satisfying $R_1 \subseteq R_2$, then $R_1^t \subseteq R_2^t$.

CHAPTER

2

Finite Automata and the Languages They Accept

In this chapter, we introduce the first of the models of computation we will study. A *finite automaton* is a model of a particularly simple computing device, which acts as a language acceptor. We will describe how one works and look at examples of languages that can be accepted this way. Although the examples are simple, they illustrate how finite automata can be useful, both in computer science and more generally. We will also see how their limitations prevent them from being general models of computation, and exactly what might keep a language from being accepted by a finite automaton. The simplicity of the finite automaton model makes it possible, not only to characterize in an elegant way the languages that can be accepted, but also to formulate an algorithm for simplifying one of these devices as much as possible.

2.1 FINITE AUTOMATA: EXAMPLES AND DEFINITIONS

In this section we introduce a type of computer that is simple, partly because the output it produces in response to an input string is limited to "yes" or "no", but mostly because of its primitive memory capabilities during a computation.

Any computer whose outputs are either "yes" or "no" acts as a *language acceptor*; the language the computer accepts is the set of input strings that cause it to produce the answer yes. In this chapter, instead of thinking of the computer as receiving an input string and then producing an answer, it's a little easier to think of it as receiving individual input symbols, one at a time, and producing after every one the answer for the *current string* of symbols that have been read so far. Before the computer has received any input symbols, the current string is Λ, and

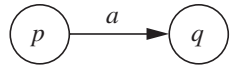
there is an answer for that string too. If the current string is *abbab*, for example, the computer might have produced the answers "yes, no, yes, yes, no, no" so far, one for each of the six prefixes of *abbab*.

The *very* simplest device for accepting a language is one whose response doesn't even depend on the input symbols it receives. There are only two possibilities: to announce at each step that the current string is accepted, and to announce at each step that it is not accepted. These two language acceptors are easy to construct, because they don't have to remember anything about the input symbols they have received, but of course they are not very useful. The only languages they can accept are the entire set Σ^* and the empty language $\emptyset$.

Slightly more complicated is the case in which the answer depends on the last input symbol received and not on any symbols before that. For example, if $\Sigma = \{a, b\}$, a device might announce every time it receives the symbol a, and only in that case, that the current string is accepted. In this case, the language it accepts is the set of all strings that end with a.

These are examples of a type of language acceptor called a *finite automaton* (FA), or *finite-state machine*. At each step, a finite automaton is in one of a finite number of *states* (it is a *finite* automaton because its set of states is finite). Its response depends only on the current state and the current symbol. A "response" to being in a certain state and receiving a certain input symbol is simply to enter a certain state, possibly the same one it was already in. The FA "announces" acceptance or rejection in the sense that its current state is either an *accepting* state or a *nonaccepting* state. In the two trivial examples where the response is always the same, only one state is needed. For the language of strings ending with a, two states are sufficient, an accepting state for the strings ending with a and a nonaccepting state for all the others.

Before a finite automaton has received any input, it is in its initial state, which is an accepting state precisely if the null string is accepted. Once we know how many states there are, which one is the initial state, and which ones are the accepting states, the only other information we need in order to describe the operation of the machine is the *transition function*, which specifies for each combination of state and input symbol the state the FA enters. The transition function can be described by either a table of values or (the way we will use most often) a transition diagram. In the diagram, states are represented by circles, transitions are represented by arrows with input symbols as labels,



and accepting states are designated by double instead of single circles.

The initial state will have an arrow pointing to it that doesn't come from another state.

An FA can proceed through the input, automaton-like, by just remembering at each step what state it's in and changing states in response to input symbols in accordance with the transition function. With the diagram, we can trace the computation for a particular input string by simply following the arrows that correspond to the symbols of the string.

A Finite Automaton Accepting the Language of Strings Ending in *aa* — EXAMPLE 2.1

In order to accept the language

$$L_1 = \{x \in \{a, b\}^* \mid x \text{ ends with } aa\}$$

an FA can operate with three states, corresponding to the number of consecutive a's that must still be received next in order to produce a string in L_1: two, because the current string does not end with a; one, because the current string ends in a but not in aa; or none, because the current string is already in L_1. It is easy to see that a transition diagram can be drawn as in Figure 2.2.

In state q_0, the input symbol b doesn't represent any progress toward obtaining a string in L_1, and it causes the finite automaton to stay in q_0; input a allows it to go to q_1. In q_1, the input b undoes whatever progress we had made and takes us back to q_0, while an a gives us a string in L_1. In q_2, the accepting state, input a allows us to stay in q_2, because the last two symbols of the current string are still both a, and b sends us back to the initial state q_0.

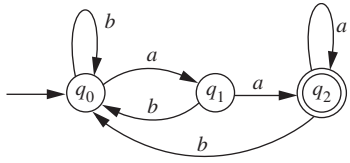


Figure 2.2 |
An FA accepting the strings ending with aa.

An FA Accepting the Language of Strings Ending in *b* and Not Containing the Substring *aa* — EXAMPLE 2.3

Let L_2 be the language

$$L_2 = \{x \in \{a, b\}^* \mid x \text{ ends with } b \text{ and does not contain the substring } aa\}$$

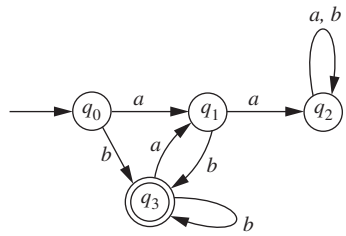


Figure 2.4 |
An FA accepting the strings ending with b and not containing aa.

In Example 2.1, no matter what the current string is, if the next two input symbols are both a, the FA ends up in an accepting state. In accepting L_2, if the next two input symbols are a's, not only do we want to end up in a nonaccepting state, but we want to make sure that from that nonaccepting state we can never reach an accepting state. We can copy the previous example by having three states q_0, q_1, and q_2, in which the current strings don't end in a, end in exactly one a, and end in two a's, respectively. This time all three are nonaccepting states, and from q_2 both transitions return to q_2, so that once our FA reaches this state it will never return to an accepting state.

The only other state we need is an accepting state q_3. Once the FA reaches this state, by receiving the symbol b in either q_0 or q_1, it stays there as long as it continues to receive b's and moves to q_1 on input a. The transition diagram for this FA is shown in Figure 2.4.

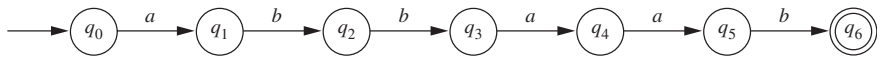
EXAMPLE 2.5 An FA Illustrating a String Search Algorithm

Suppose we have a set of strings over $\{a, b\}$ and we want to identify all the ones containing a particular substring, say $abbaab$. A reasonable way to go about it is to build an FA accepting the language

$$L_3 = \{x \in \{a, b\}^* \mid x \text{ contains the substring } abbaab\}$$

and use it to test each string in the set. Once we have the FA, the number of steps required to test each string is no more than the number of symbols in the string, so that we can be confident this is an efficient approach.

The idea behind the FA is the same as in Example 2.1, but using a string with both a's and b's will make it easier to identify the underlying principle. We can start by drawing this diagram:



For each i, the FA will be in state q_i whenever the current string ends with the prefix of $abbaab$ having length i and not with any longer prefix. Now we just have to try to add transitions to complete the diagram. The transitions leaving q_6 simply return to q_6, because this FA should accept the strings containing, not ending with, $abbaab$. For each $i < 6$, we already have one transition from q_i, and we have to decide where to send the other one.

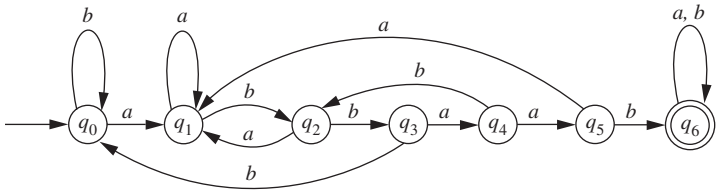


Figure 2.6 |
An FA accepting the strings containing the substring *abbaab*.

Consider $i = 4$. One string that causes the FA to be in state q_4 is *abba*, and we must decide what state corresponds to the string *abbab*. Because *abbab* ends with *ab*, and not with any longer prefix of *abbaab*, the transition should go to q_2. The other cases are similar, and the resulting FA is shown in Figure 2.6.

If we want an FA accepting all the strings ending in *abbaab*, instead of all the strings containing this substring, we can use the transition diagram in Figure 2.6 with different transitions from the accepting state. The transition on input *a* should go from state q_6 to some earlier state corresponding to a prefix of *abbaab*. Which prefix? The answer is *a*, the longest one that is a suffix of *abbaaba*. In other words, we can proceed as if we were drawing the FA accepting the set of strings containing *abbaabb* and after six symbols we received input *a* instead of *b*. Similarly, the transition from q_6 on input *b* should go to state q_3.

An FA Accepting Binary Representations of Integers Divisible by 3

EXAMPLE 2.7

We consider the language L of strings over the alphabet $\{0, 1\}$ that are the binary representations of natural numbers divisible by 3. Another way of saying that n is divisible by 3 is to say that n mod 3, the remainder when n is divided by 3, is 0. This seems to suggest that the only information concerning the current string x that we really need to remember is the remainder when the number represented by x is divided by 3.

The question is, if we know the remainder when the number represented by x is divided by 3, is that enough to find the remainders when the numbers represented by $x0$ and $x1$ are divided by 3? And that raises the question: What are the numbers represented by $x0$ and $x1$?

Just as adding 0 to the end of a decimal representation corresponds to multiplying by ten, adding 0 to the end of a binary representation corresponds to multiplying by 2. Just as adding 1 to the end of a decimal representation corresponds to multiplying by ten and then adding 1 (example: $39011 = 10 * 3901 + 1$), adding 1 to the end of a binary representation corresponds to multiplying by 2 and then adding 1 (example: 1110 represents 14, and 11101 represents 29).

Now we are ready to answer the first question: If x represents n, and n mod 3 is r, then what are $2n$ mod 3 and $(2n + 1)$ mod 3? It is almost correct that the answers are $2r$ and $2r + 1$; the only problem is that these numbers may be 3 or bigger, and in that case we must do another mod 3 operation.

These facts are enough for us to construct our FA. We begin with states corresponding to remainders 0, 1, and 2. The only one of these that is an accepting state is 0, because remainder 0 means that the integer is divisible by 3, and the transitions from these states

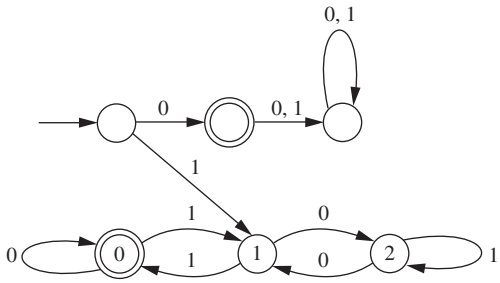


Figure 2.8 |
An FA accepting binary representations of integers divisible by 3.

follow the rules outlined above. These states do not include the initial state, because the null string doesn't qualify as a binary representation of a natural number, or the accepting state corresponding to the string 0. We will disallow leading 0's in binary representations, except for the number 0 itself, and so we need one more state for the strings that start with 0 and have more than one digit. The resulting transition diagram is shown in Figure 2.8.

EXAMPLE 2.9 Lexical Analysis

Another real-world problem for which finite automata are ideally suited is lexical analysis, the first step in compiling a program written in a high-level language.

Before a C compiler can begin to determine whether a string such as

```
main(){ double b=41.3; b *= 4; ...
```

satisfies the many rules for the syntax of C, it must be able to break up the string into *tokens*, which are the indecomposable units of the program. Tokens include reserved words (in this example, "`main`" and "`double`"), punctuation symbols, identifiers, operators, various types of parentheses and brackets, numeric literals such as "`41.3`" and "`4`", and a few others.

Programming languages differ in their sets of reserved words, as well as in their rules for other kinds of tokens. For example, "`41.`" is a legal token in C but not in Pascal, which requires a numeric literal to have at least one digit on both sides of a decimal point.

In any particular language, the rules for constructing tokens are reasonably simple. Testing a substring to see whether it represents a valid token can be done by a finite automaton in software form; once this is done, the string of alphabet symbols can be replaced by a sequence of tokens, each one represented in a form that is easier for the compiler to use in its later processing.

We will illustrate a lexical-analysis FA for a C-like language in a greatly simplified case, in which the only tokens are identifiers, semicolons, the assignment operator =, the reserved word `aa`, and numeric literals consisting of one or more digits and possibly a decimal point. An identifier will start with a lowercase letter and will contain only lowercase letters and/or digits. Saying that `aa` is reserved means that it cannot be an identifier, although longer identifiers might begin with `aa` or contain it as a substring. The job of the FA will be to accept strings that consist of one or more consecutive tokens. (The FA will not be required

to determine whether a particular sequence of tokens makes sense; that job will have to be performed at a later stage in the compilation.) The FA will be in an accepting state each time it finishes reading another legal token, and the state will be one that is reserved for a particular type of token; in this sense, the lexical analyzer will be able to classify the tokens.

Another way to simplify the transition diagram considerably is to make another assumption: that two consecutive tokens are always separated by a blank space.

The crucial transitions of the FA are shown in Figure 2.10. The input alphabet is the set containing the 26 lowercase letters, the 10 digits, a semicolon, an equals sign, a decimal point, and the blank space Δ. We have used a few abbreviations: D for any numerical digit, L for any lowercase letter other than `a`, M for any numerical digit or letter other than `a`, and N for any letter or digit. You can check that all possible transitions from the initial state are shown. From every other state, transitions not shown go to a "reject" state, from which all transitions return to that state; no attempt is made to continue the lexical analysis once an error is detected.

The two portions of the diagram that require a little care are the ones involving tokens with more than one symbol. State 3 corresponds to the identifier `a`, state 4 to the reserved word `aa`, and state 5 to any other identifier. Transitions to state 5 are possible from state 3 with any letter or digit except `a`, from states 4 or 5 with any letter or digit, and from the initial state with any letter other than `a`. State 6 corresponds to numeric literals without decimal points and state 7 to those with decimal points. State 8 is not an accepting state, because a numeric literal must have at least one digit.

This FA could be incorporated into lexical-analysis software as follows. Each time a symbol causes the FA to make a transition out of the initial state, we mark that symbol in the string; each time we are in one of the accepting states and receive a blank space, we mark

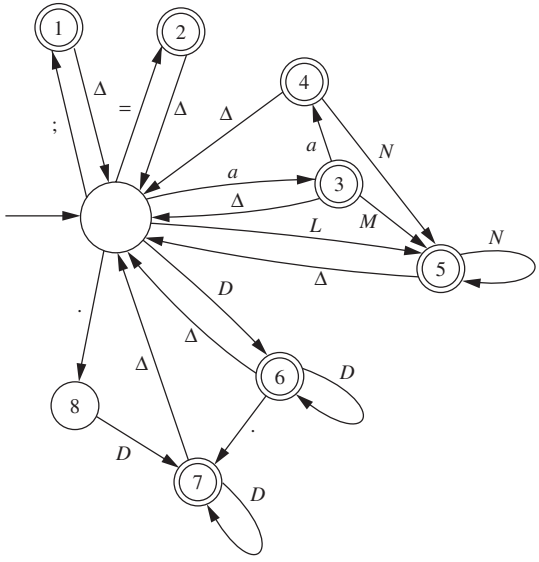


Figure 2.10 |
An FA illustrating a simplified version of lexical analysis.

the symbol just before the blank; and the token, whose type is identified by the accepting state, is represented by the substring that starts with the first of these two symbols and ends with the second.

The restriction that tokens be separated by blanks makes the job of recognizing the beginnings and ends of tokens very simple, but in practice there is no such rule, and we could construct an FA without it. The transition diagram would be considerably more cluttered; the FA would not be in the initial state at the beginning of each token, and many more transitions between the other states would be needed.

Without a blank space to tell us where a token ends, we would normally adopt the convention that each token extends as far as possible. A substring like "`=2b3aa1`" would then be interpreted as containing two tokens, "=" and "2", and at least the first five symbols of a third. There are substrings, such as "`3...2`", that cannot be part of any legal string. There are others, like "`1..2..3`", that can but only if the extending-as-far-as-possible policy is abandoned. Rejecting this particular string is probably acceptable anyway, because no way of breaking it into tokens is compatible with the syntax of C.

See Example 3.5 for more discussion of tokens and lexical analysis.

Giving the following official definition of a finite automaton and developing some related notation will make it easier to talk about these devices precisely.

Definition 2.11 A Finite Automaton

A *finite automaton* (FA) is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$, where

Q is a finite set of *states*;
Σ is a finite *input alphabet*;
$q_0 \in Q$ is the *initial* state;
$A \subseteq Q$ is the set of *accepting* states;
$\delta : Q \times \Sigma \to Q$ is the *transition* function.

For any element q of Q and any symbol $\sigma \in \Sigma$, we interpret $\delta(q, \sigma)$ as the state to which the FA moves, if it is in state q and receives the input σ.

The first line of the definition deserves a comment. What does it mean to say that a simple computer is a 5-tuple? This is simply a formalism that allows us to define an FA in a concise way. Describing a finite automaton precisely requires us to specify five things, and it is easier to say

Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA

than it is to say

Let M be an FA with state set Q, input alphabet Σ, initial state q_0, set of accepting states A, and transition function δ.

We write $\delta(q, \sigma)$ to mean the state an FA M goes to from q after receiving the input symbol σ. The next step is to extend the notation to allow a corresponding expression $\delta^*(q, x)$ that will represent the state the FA ends up in if it starts out in state q and receives the string x of input symbols. In other words, we want to define an "extended transition function" δ^* from $Q \times \Sigma^*$ to Q. The easiest way to define it is recursively, using the recursive definition of Σ^* in Example 1.17. We begin by defining $\delta^*(q, \Lambda)$, and since we don't expect the state of M to change as a result of getting the input string Λ, we give the expression the value q.

Definition 2.12 The Extended Transition Function δ^*

Let $M = (Q, \Sigma, q_0, A, \delta)$ be a finite automaton. We define the extended transition function

$$\delta^* : Q \times \Sigma^* \to Q$$

as follows:

1. For every $q \in Q$, $\delta^*(q, \Lambda) = q$
2. For every $q \in Q$, every $y \in \Sigma^*$, and every $\sigma \in \Sigma$,

$$\delta^*(q, y\sigma) = \delta(\delta^*(q, y), \sigma)$$

The recursive part of the definition says that we can evaluate $\delta^*(q, x)$ if we know that $x = y\sigma$, for some string y and some symbol σ, and if we know what state the FA is in after starting in q and processing the symbols of y. We do it by just starting in that state and applying one last transition, the one corresponding to the symbol σ. For example, if M contains the transitions

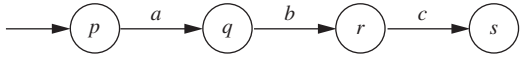


Figure 2.13

then

$$\begin{aligned}
\delta^*(p, abc) &= \delta(\delta^*(p, ab), c) \\
&= \delta(\delta(\delta^*(p, a), b), c) \\
&= \delta(\delta(\delta^*(p, \Lambda a), b), c) \\
&= \delta(\delta(\delta(\delta^*(p, \Lambda), a), b), c) \\
&= \delta(\delta(\delta(p, a), b), c) \\
&= \delta(\delta(q, b), c) \\
&= \delta(r, c) \\
&= s
\end{aligned}$$

Looking at the diagram, of course, we can get the same answer by just following the arrows. The point of this derivation is not that it's always the simplest

way to find the answer by hand, but that the recursive definition is a reasonable way of defining the extended transition function, and that the definition provides a systematic algorithm.

Other properties you would expect δ^* to satisfy can be derived from our definition. For example, a natural generalization of the recursive statement in the definition is the formula

$$\delta^*(q, xy) = \delta^*(\delta^*(q, x), y)$$

which is true for every state q and every two strings x and y in Σ^*. The proof is by structural induction on y and is similar to the proof of the formula $r(xy) = r(y)r(x)$ in Example 1.27.

The extended transition function makes it possible to say concisely what it means for an FA to accept a string or a language.

Definition 2.14 Acceptance by a Finite Automaton

Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA, and let $x \in \Sigma^*$. The string x is *accepted* by M if

$$\delta^*(q_0, x) \in A$$

and is *rejected* by M otherwise. The *language* accepted by M is the set

$$L(M) = \{x \in \Sigma^* \mid x \text{ is accepted by } M\}$$

If L is a language over Σ, L is accepted by M if and only if $L = L(M)$.

Notice what the last statement in Definition 2.14 does not say. It doesn't say that L is accepted by M if every string in L is accepted by M. To take this as the definition would not be useful (it's easy to describe a one-state FA that accepts every string in Σ^*). A finite automaton accepting a language L does its job by distinguishing between strings in L and strings not in L: accepting the strings in L and rejecting all the others.

2.2 ACCEPTING THE UNION, INTERSECTION, OR DIFFERENCE OF TWO LANGUAGES

Suppose L_1 and L_2 are both languages over Σ. If $x \in \Sigma^*$, then knowing whether $x \in L_1$ and whether $x \in L_2$ is enough to determine whether $x \in L_1 \cup L_2$. This means that if we have one algorithm to accept L_1 and another to accept L_2, we can easily formulate an algorithm to accept $L_1 \cup L_2$. In this section we will show that if we actually have finite automata accepting L_1 and L_2, then there is a finite automaton accepting $L_1 \cup L_2$, and that the same approach also gives us FAs accepting $L_1 \cap L_2$ and $L_1 - L_2$.

The idea is to construct an FA that effectively executes the two original ones simultaneously, one whose current state records the current states of both. This isn't difficult; we can simply use "states" that are ordered pairs (p, q), where p and q are states in the two original FAs.

Theorem 2.15

Suppose $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ are finite automata accepting L_1 and L_2, respectively. Let M be the FA $(Q, \Sigma, q_0, A, \delta)$, where

$$Q = Q_1 \times Q_2$$
$$q_0 = (q_1, q_2)$$

and the transition function δ is defined by the formula

$$\delta((p, q), \sigma) = (\delta_1(p, \sigma), \delta_2(q, \sigma))$$

for every $p \in Q_1$, every $q \in Q_2$, and every $\sigma \in \Sigma$. Then

1. If $A = \{(p, q) \mid p \in A_1 \text{ or } q \in A_2\}$, M accepts the language $L_1 \cup L_2$.
2. If $A = \{(p, q) \mid p \in A_1 \text{ and } q \in A_2\}$, M accepts the language $L_1 \cap L_2$.
3. If $A = \{(p, q) \mid p \in A_1 \text{ and } q \notin A_2\}$, M accepts the language $L_1 - L_2$.

Proof

We consider statement 1, and the other two are similar. The way the transition function δ is defined allows us to say that at any point during the operation of M, if (p, q) is the current state, then p and q are the current states of M_1 and M_2, respectively. This will follow immediately from the formula

$$\delta^*(q_0, x) = (\delta_1^*(q_1, x), \delta_2^*(q_2, x))$$

which is true for every $x \in \Sigma^*$. The proof is by structural induction on x and is left to Exercise 2.12. For every string x, x is accepted by M precisely if $\delta^*(q_0, x) \in A$, and according to the definition of A in statement 1 and the formula for δ^*, this is true precisely if $\delta_1^*(q_1, x) \in A_1$ or $\delta_2^*(q_2, x) \in A_2$—i.e., precisely if $x \in L_1 \cup L_2$.

As we will see in Example 2.16, we don't always need to include every ordered pair in the state set of the composite FA.

Constructing an FA Accepting $L_1 \cap L_2$ — EXAMPLE 2.16

Let L_1 and L_2 be the languages

$$L_1 = \{x \in \{a, b\}^* \mid aa \text{ is not a substring of } x\}$$
$$L_2 = \{x \in \{a, b\}^* \mid x \text{ ends with } ab\}$$

Finite automata M_1 and M_2 accepting these languages are easy to obtain and are shown in Figure 2.17a. The construction in Theorem 2.15 produces an FA with the nine states shown in Figure 2.17b. Rather than drawing all eighteen transitions, we start at the initial state (A, P), draw the two transitions to (B, Q) and (A, P) using the definition of δ in the theorem, and continue in this way, at each step drawing transitions from a state that has already been reached by some other transition. At some point, we have six states such that every transition from one of these six goes to one of these six. Since none of the remaining three states is reachable from any of the first six, we can leave them out.

If we want our finite automaton to accept $L_1 \cup L_2$, then we designate as accepting states the ordered pairs among the remaining six that involve at least one of the accepting states A, B, and R. The result is shown in Figure 2.17c.

If instead we want to accept $L_1 \cap L_2$, then the only accepting state is (A, R), since (B, R) was one of the three omitted. This allows us to simplify the FA even further. None of the three states (C, P), (C, Q), and (C, R) is accepting, and every transition from one of these three goes to one of these three; therefore, we can combine them all into a single nonaccepting state. An FA accepting $L_1 \cap L_2$ is shown in Figure 2.17d. The FA we would get for $L_1 - L_2$ is similar and also has just four states, but in that case two are accepting states.

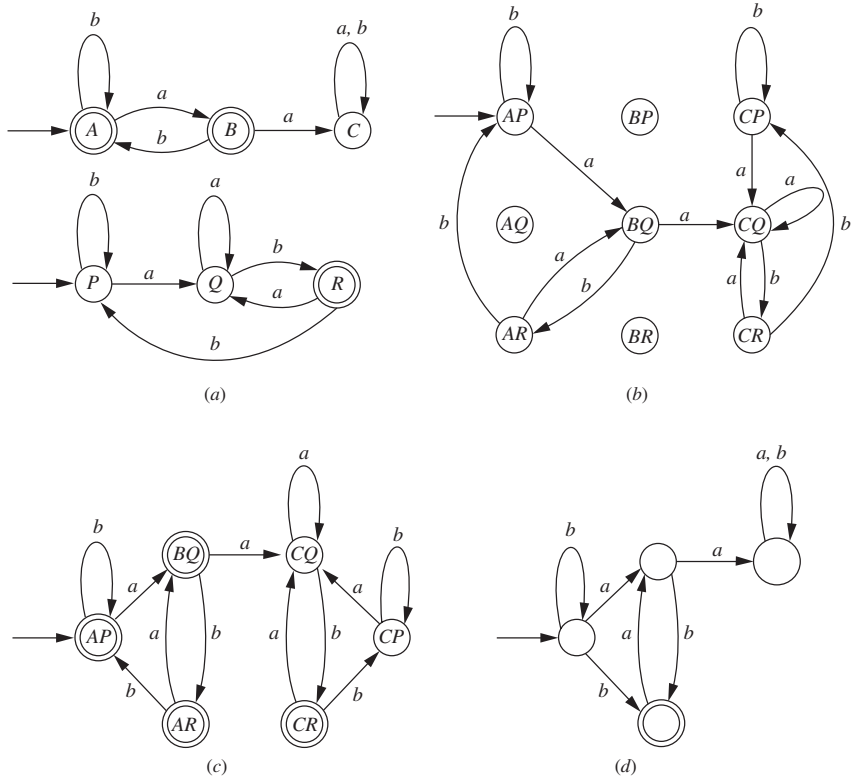


Figure 2.17 |
Constructing an FA to accept the intersection of two languages.

EXAMPLE 2.18 An FA Accepting Strings That Contain Either *ab* or *bba*

Figure 2.19a shows FAs M_1 and M_2 accepting $L_1 = \{x \in \{a, b\}^* \mid x$ contains the substring $ab\}$ and $L_2 = \{x \in \{a, b\}^* \mid x$ contains the substring $bba\}$, respectively. They are both obtained by the technique described in Example 2.5. Using Theorem 2.15 to construct an FA accepting $L_1 \cup L_2$ could result in one with twelve states, but Figure 2.19b illustrates an approach that seems likely to require considerably fewer. If it works, the FA will need only the states we've drawn, and the two paths to the accepting state will correspond to strings in L_1 and strings in L_2, respectively.

This approach does work; the five states shown are sufficient, and it is not difficult to complete the transitions from the three intermediate states. Instead, let's see whether the construction in the theorem gives us the same answer or one more complicated. Figure 2.19c shows a partially completed diagram; to complete it, we must draw the transitions from $(3, q)$ and $(2, s)$ and any additional states that may be required. So far we have six states, and you can check that $(3, p)$, $(3, r)$, and $(3, s)$ will also appear if we continue this way. Notice, however, that because states 3 and s are accepting, and transitions from either state return to that state, every state we add will be an ordered pair involving 3 or s or both, and every transition from one of these accepting states will return to one. The conclusion is that we can combine $(3, q)$ and $(2, s)$ and we don't need any more states; the final diagram is in Figure 2.19d.

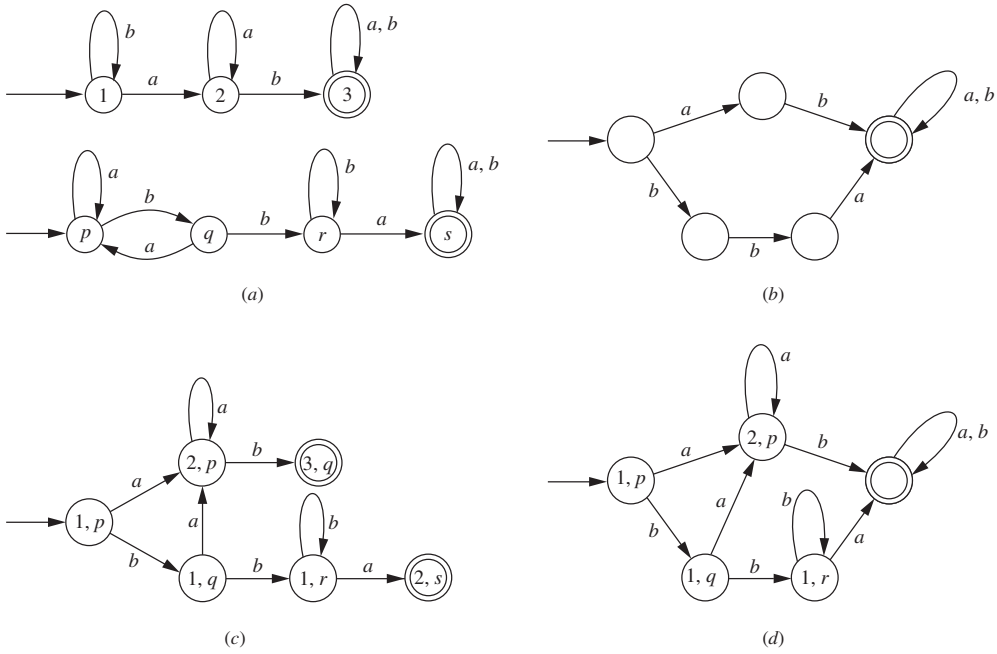


Figure 2.19 |
Constructing an FA to accept strings containing either *ab* or *bba*.

The construction in Theorem 2.15 will always work, but the FA it produces may not be the simplest possible. Fortunately, if we need the simplest possible one, we don't need to rely on the somewhat unsystematic methods in these two examples; we will see in Section 2.6 how to start with an arbitary FA and find one with the fewest possible states accepting the same language.

2.3 DISTINGUISHING ONE STRING FROM ANOTHER

The finite automaton M in Example 2.1, accepting the language L of strings in $\{a, b\}^*$ ending with aa, had three states, corresponding to the three possible numbers of a's still needed to have a string in L. As simple as this sounds, it's worth taking a closer look. Could the FA be constructed with fewer than three states? And can we be sure that three are enough? These are different questions; we will answer the first in this section and return to the second in Section 2.5.

Any FA with three states ignores, or forgets, almost all the information pertaining to the current string. In the case of M, it makes no difference whether the current string is aba or $aabbabbabaaaba$; the only relevant feature of these two strings is that both end with a and neither ends with aa. However, it does make a difference whether the current string is aba or ab, even though neither string is in L. It makes a difference because of what input symbols might come next. If the next input symbol is a, the current string at that point would be $abaa$ in the first case and aba in the second; one string is in L and the other isn't. The FA has to be able to distinguish aba and ab now, so that in case the next input symbol is a it will be able to distinguish the two corresponding longer strings. We will describe the difference between aba and ab by saying that they are *distinguishable* with respect to L: there is at least one string z such that of the two strings $abaz$ and abz, one is in L and the other isn't.

Definition 2.20 Strings Distinguishable with Respect to *L*

If L is a language over the alphabet Σ, and x and y are strings in Σ^*, then x and y are *distinguishable with respect to* L, or L*-distinguishable*, if there is a string $z \in \Sigma^*$ such that either $xz \in L$ and $yz \notin L$, or $xz \notin L$ and $yz \in L$. A string z having this property is said to distinguish x and y with respect to L. An equivalent formulation is to say that x and y are L-distinguishable if $L/x \neq L/y$, where

$$L/x = \{z \in \Sigma^* \mid xz \in L\}$$

The two strings x and y are L-indistinguishable if $L/x = L/y$, which means that for every $z \in \Sigma^*$, $xz \in L$ if and only if $yz \in L$.

The strings in a set $S \subseteq \Sigma^*$ are *pairwise* L*-distinguishable* if for every pair x, y of distinct strings in S, x and y are L-distinguishable.

The crucial fact about two L-distinguishable strings, or more generally about a set of pairwise L-distinguishable strings, is given in Theorem 2.21, and it will provide the answer to the first question we asked in the first paragraph.

Theorem 2.21

Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA accepting the language $L \subseteq \Sigma^*$. If x and y are two strings in Σ^* that are L-distinguishable, then $\delta^*(q_0, x) \neq \delta^*(q_0, y)$. For every $n \geq 2$, if there is a set of n pairwise L-distinguishable strings in Σ^*, then Q must contain at least n states.

Proof

If x and y are L-distinguishable, then for some string z, one of the strings xz, yz is in L and the other isn't. Because M accepts L, this means that one of the states $\delta^*(q_0, xz)$, $\delta^*(q_0, yz)$ is an accepting state and the other isn't. In particular,

$$\delta^*(q_0, xz) \neq \delta^*(q_0, yz)$$

According to Exercise 2.5, however,

$$\delta^*(q_0, xz) = \delta^*(\delta^*(q_0, x), z)$$
$$\delta^*(q_0, yz) = \delta^*(\delta^*(q_0, y), z)$$

Because the left sides are different, the right sides must be also, and so $\delta^*(q_0, x) \neq \delta^*(q_0, y)$.

The second statement in the theorem follows from the first: If M had fewer than n states, then at least two of the n strings would cause M to end up in the same state, but this is impossible if the two strings are L-distinguishable.

Returning to our example, we can now say why there must be three states in an FA accepting L, the language of strings ending with aa. We already have an FA with three states accepting L. Three states are actually necessary if there are three pairwise L-distinguishable strings, and we can find three such strings by choosing one corresponding to each state. We choose Λ, a, and aa. The string a distinguishes Λ and a, because $\Lambda a \notin L$ and $aa \in L$; the string Λ distinguishes Λ and aa; and it also distinguishes a and aa.

As the next example illustrates, the first statement in Theorem 2.21 can be useful in constructing a finite automaton to accept a language, because it can help us decide at each step whether a transition should go to a state we already have or whether we need to add another state.

Constructing an FA to Accept $\{aa, aab\}^*\{b\}$ **EXAMPLE 2.22**

Let L be the language $\{aa, aab\}^*\{b\}$. In Chapter 3 we will study a systematic way to construct finite automata for languages like this one. It may not be obvious at this stage that

it will even be possible, but we will proceed by adding states as needed and hope that we will eventually have enough.

The null string is not in L, and so the initial state should not be an accepting state. The string b is in L, the string a is not, and the two strings Λ and a are L-distinguishable because $\Lambda ab \notin L$ and $aab \in L$. We have therefore determined that we need at least the states in Figure 2.23a.

The language L contains b but no other strings beginning with b. It also contains no strings beginning with ab. These two observations suggest that we introduce a state s to take care of all strings that fail for either reason to be a prefix of an element of L (Fig.2.23b). Notice that if two strings are L-distinguishable, at least one of them must be a prefix of an element of L; therefore, two strings ending up in state s cannot be L-distinguishable. All transitions from s will return to s.

Suppose the FA is in state p and receives the input a. It can't stay in p, because the strings a and aa are distinguished relative to L by the string ab. It can't return to the initial state, because Λ and aa are L-distinguishable. Therefore, we need a new state t. From t, the input b must lead to an accepting state, because $aab \in L$; this accepting state cannot be r, because aab and a are L-distinguishable; call the new accepting state u. One of the strings that gets the FA to state u is aab. If we receive another b in state u, the situation is the same as for an initial b; $aabb$ and b are both in L, but neither is a prefix of any longer string in L. We can therefore let $\delta(u, b) = r$.

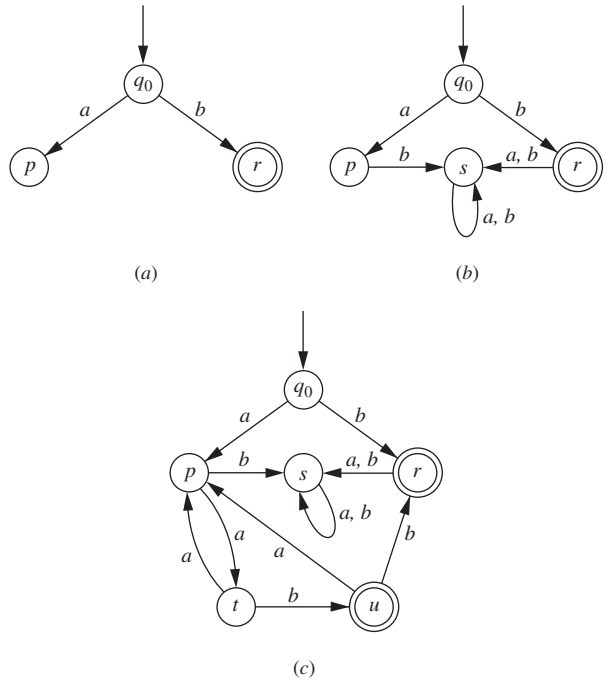


Figure 2.23 |
Constructing an FA to accept $\{aa, aab\}^*\{b\}$.

We have yet to define $\delta(t, a)$ and $\delta(u, a)$. States t and u can be thought of as representing the end of one of the strings aa and aab. (The reason u is an accepting state is that one of these two strings, aab, also happens to be the other one followed by b.) In either case, if the next symbol is a, we should view it as the first symbol in *another* occurrence of one of these two strings. For this reason, we can define $\delta(t, a) = \delta(u, a) = p$, and we arrive at the FA shown in Figure 2.23c.

It may be clear already that because we added each state only if necessary, the FA we have constructed is the one with the fewest possible states. If we had not constructed it ourselves, we could use Theorem 2.21 to show this. The FA has six states, and applying the second statement in the theorem seems to require that we produce six pairwise L-distinguishable strings. Coming up with six strings is easy—we can choose one corresponding to each state—but verifying directly that they are pairwise L-distinguishable requires looking at all 21 choices of two of them. A slightly easier approach, since there are four nonaccepting states and two accepting states, is to show that the four strings that are not in L are pairwise L-distinguishable and the two strings in L are L-distinguishable. The argument in the proof of the theorem can easily be adapted to show that every FA accepting L must then have at least four nonaccepting states and two accepting states. This way we have to consider only seven sets of two strings and for each set find a string that distinguishes the two relative to L.

An FA Accepting Strings with *a* in the *n*th Symbol from the End **EXAMPLE 2.24**

Suppose n is a positive integer, and L_n is the language of strings in $\{a, b\}^*$ with at least n symbols and an a in the nth position from the end.

The first observation about accepting this language is that if a finite automaton "remembers" the most recent n input symbols it has received, or remembers the entire current string as long as its length is less than n, then it has enough information to continue making correct decisions. Another way to say this is that no symbol that was received more than n symbols ago should play any part in deciding whether the current string is accepted.

We can also see that if $i < n$ and x is any string of length i, then the string $b^{n-i}x$ can be treated the same as x by an FA accepting L_n. For example, suppose $n = 5$ and $x = baa$. Neither of the strings $bbbaa$ and baa is in L_5, three more symbols are required in both cases before an element of L_5 will be obtained, and from that point on the two current strings will always agree in the last five symbols. As a result, an FA accepting L_n will require no more than 2^n states, the number of strings of length n.

Finally, if we can show that the strings of length n are pairwise L_n-distinguishable, Theorem 2.21 will tell us that we need this many states. Let x and y be distinct strings of length n. They must differ in the ith symbol (from the left), for some i with $1 \le i \le n$. Every string z of length $i - 1$ distinguishes these two relative to L_n, because xz and yz differ in the ith symbol, which is the nth symbol from the right.

Figure 2.25 shows an FA with four states accepting L_2. The labeling technique we have used here also works for $n > 2$; if each state is identified with a string x of length n, and $x = \sigma_1 y$ where $|y| = n - 1$, the transition function can be described by the formula

$$\delta(\sigma_1 y, \sigma) = y\sigma$$

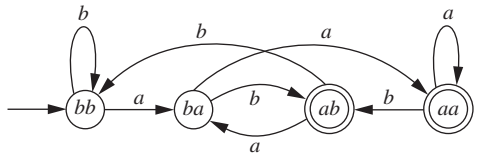


Figure 2.25 |
An FA accepting L_n in the case $n = 2$.

We can carry the second statement of Theorem 2.21 one step further, by considering a language L for which there are infinitely many pairwise L-distinguishable strings.

Theorem 2.26
For every language $L \subseteq \Sigma^*$, if there is an infinite set S of pairwise L-distinguishable strings, then L cannot be accepted by a finite automaton.

Proof
If S is infinite, then for every n, S has a subset with n elements. If M were a finite automaton accepting L, then Theorem 2.21 would say that for every n, M would have at least n states. No *finite* automaton can have this property!

It is not hard to find languages with the property in Theorem 2.26. In Example 2.27 we take L to be the language *Pal* from Example 1.18, the set of palindromes over $\{a,b\}$. Not only is there an infinite set of pairwise L-distinguishable strings, but *all* the strings in $\{a,b\}^*$ are pairwise L-distinguishable.

EXAMPLE 2.27 For Every Pair *x*, *y* of Distinct Strings in $\{a,b\}^*$, *x* and *y* Are Distinguishable with Respect to *Pal*

First suppose that $x \neq y$ and $|x| = |y|$. Then x^r, the reverse of x, distinguishes the two with respect to *Pal*, because $xx^r \in Pal$ and $yx^r \notin Pal$. If $|x| \neq |y|$, we assume x is shorter. If x is not a prefix of y, then $xx^r \in Pal$ and $yx^r \notin Pal$. If x is a prefix of y, then $y = xz$ for some nonnull string z. If we choose the symbol σ (either a or b) so that $z\sigma$ is not a palindrome, then $x\sigma x^r \in Pal$ and $y\sigma x^r = xz\sigma x^r \notin Pal$.

An explanation for this property of *Pal* is easy to find. If a computer is trying to accept *Pal*, has read the string x, and starts to receive the symbols of another string z, it can't be expected to decide whether z is the reverse of x unless it can actually remember every symbol of x. The only thing a finite automaton M can remember is what state it's in, and there are only a finite number of states. If x is a sufficiently long string, remembering every symbol of x is too much to expect of M.

2.4 | THE PUMPING LEMMA

A finite automaton accepting a language operates in a very simple way. Not surprisingly, the languages that can be accepted in this way are "simple" languages, but it is not yet clear exactly what this means. In this section, we will see one property that every language accepted by an FA must have.

Suppose that $M = (Q, \Sigma, q_0, A, \delta)$ is an FA accepting $L \subseteq \Sigma^*$ and that Q has n elements. If x is a string in L with $|x| = n - 1$, so that x has n distinct prefixes, it is still conceivable that M is in a different state after processing every one. If $|x| \geq n$, however, then by the time M has read the symbols of x, it must have entered some state twice; there must be two different prefixes u and uv (saying they are different is the same as saying that $v \neq \Lambda$) such that

$$\delta^*(q_0, u) = \delta^*(q_0, uv)$$

This means that if $x \in L$ and w is the string satisfying $x = uvw$, then we have the situation illustrated by Figure 2.28. In the course of reading the symbols of $x = uvw$, M moves from the initial state to an accepting state by following a path that contains a loop, corresponding to the symbols of v. There may be more than one such loop, and more than one such way of breaking x into three pieces u, v, and w; but at least one of the loops must have been completed by the time M has read the first n symbols of x. In other words, for at least one of the choices of u, v, and w such that $x = uvw$ and v corresponds to a loop, $|uv| \leq n$.

The reason this is worth noticing is that it tells us there must be many more strings besides x that are also accepted by M and are therefore in L: strings that cause M to follow the same path but traverse the loop a different number of times. The string obtained from x by omitting the substring v is in L, because M doesn't have to traverse the loop at all. For each $i \geq 2$, the string uv^iw is in L, because M can take the loop i times before proceeding to the accepting state.

The statement we have now proved is known as the Pumping Lemma for Regular Languages. "Pumping" refers to the idea of pumping up the string x by inserting additional copies of the string v (but remember that we also get one of the new strings by leaving out v). "Regular" won't be defined until Chapter 3, but we will see after we define regular languages that they turn out to be precisely the ones that can be accepted by finite automata.

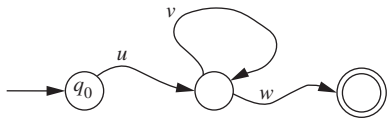


Figure 2.28 |
What the three strings u, v, and w in the pumping lemma might look like.

Theorem 2.29 The Pumping Lemma for Regular Languages
Suppose L is a language over the alphabet Σ. If L is accepted by a finite automaton $M = (Q, \Sigma, q_0, A, \delta)$, and if n is the number of states of M, then for every $x \in L$ satisfying $|x| \geq n$, there are three strings u, v, and w such that $x = uvw$ and the following three conditions are true:

1. $|uv| \leq n$.
2. $|v| > 0$ (i.e., $v \neq \Lambda$).
3. For every $i \geq 0$, the string $uv^i w$ also belongs to L.

Later in this section we will find ways of applying this result for a language L that is accepted by an FA. But the most common application is to show that a language *cannot* be accepted by an FA, by showing that it doesn't have the property described in the pumping lemma.

A proof using the pumping lemma that L cannot be accepted by a finite automaton is a proof by contradiction. We assume, for the sake of contradiction, that L can be accepted by M, an FA with n states, and we try to select a string in L with length at least n so that statements 1–3 lead to a contradiction. There are a few places in the proof where it's easy to go wrong, so before looking at an example, we consider points at which we have to be particularly careful.

Before we can think about applying statements 1–3, we must have a string $x \in L$ with $|x| \geq n$. What is n? It's the number of states in M, but we don't know what M is—the whole point of the proof is to show that it can't exist! In other words, our choice of x must involve n. We can't say "let $x = aababaabbab$", because there's no reason to expect that $11 \geq n$. Instead, we might say "let $x = a^n ba^{2n}$", or "let $x = b^{n+1}a^n b$", or something comparable, depending on L.

The pumping lemma tells us some properties that *every* string in L satisfies, as long as its length is at least n. It is very possible that for some choices of x, the fact that x has these properties does not produce any contradiction. If we don't get a contradiction, we haven't proved anything, and so we look for a string x that *will* produce one. For example, if we are trying to show that the language of palindromes over $\{a, b\}$ cannot be accepted by an FA, there is no point in considering a string x containing only a's, because all the new strings that we will get by using the pumping lemma will also contain only a's, and they're all palindromes too. No contradiction!

Once we find a string x that looks promising, the pumping lemma says that there is *some* way of breaking x into shorter strings u, v, and w satisfying the three conditions. It doesn't tell us what these shorter strings are, only that they satisfy conditions 1–3. If $x = a^n b^n a^n$, we can't say "let $u = a^{10}$, $v = a^{n-10}$, and $w = b^n a^n$". It's not enough to show that *some* choices for u, v, and w produce a contradiction—we have to show that we *must* get a contradiction, no matter what u, v, and w are, as long as they satisfy conditions 1–3.

Let's try an example.

EXAMPLE 2.30 The Language *AnBn*

Let L be the language *AnBn* introduced in Example 1.18:

$$L = \{a^i b^i \mid i \geq 0\}$$

It would be surprising if *AnBn* could be accepted by an FA; if the beginning input symbols are a's, a computer accepting L surely needs to remember how many of them there are, because otherwise, once the input switches to b's, it won't be able to compare the two numbers.

Suppose for the sake of contradiction that there is an FA M having n states and accepting L. We choose $x = a^n b^n$. Then $x \in L$ and $|x| \geq n$. Therefore, by the pumping lemma, there are strings u, v, and w such that $x = uvw$ and the conditions 1–3 in the theorem are true.

Because $|uv| \leq n$ (by condition 1) and the first n symbols of x are a's (because of the way we chose x), all the symbols in u and v must be a's. Therefore, $v = a^k$ for some $k > 0$ (by condition 2). We can get a contradiction from statement 3 by using any number i other than 1, because $uv^i w$ will still have exactly n b's but will no longer have exactly n a's. The string uv^2w, for example, is $a^{n+k}b^n$, obtained by inserting k additional a's into the first part of x. This is a contradiction, because the pumping lemma says $uv^2w \in L$, but $n + k \neq n$.

Not only does the string uv^2w fail to be in L, but it also fails to be in the bigger language *AEqB* containing all strings in $\{a, b\}^*$ with the same number of a's as b's. Our proof, therefore, is also a proof that *AEqB* cannot be accepted by an FA.

EXAMPLE 2.31 The Language $\{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$

Let L be the language

$$L = \{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$$

The first sentence of a proof using the pumping lemma is always the same: Suppose for the sake of contradiction that there is an FA M that accepts L and has n states. There are more possibilities for x than in the previous example; we will suggest several choices, all of which satisfy $|x| \geq n$ but some of which work better than others in the proof.

First we try $x = b^n a^{2n}$. Then certainly $x \in L$ and $|x| \geq n$. By the pumping lemma, $x = uvw$ for some strings u, v, and w satisfying conditions 1–3. Just as in Example 2.30, it follows from conditions 1 and 2 that $v = b^k$ for some $k > 0$. We can get a contradiction from condition 3 by considering $uv^i w$, where i is large enough that $n_b(uv^i w) \geq n_a(uv^i w)$. Since $|v| \geq 1$, $i = n + 1$ is guaranteed to be large enough. The string $uv^{n+1}w$ has at least n more b's than x does, and therefore at least $2n$ b's, but it still has exactly $2n$ a's.

Suppose that instead of $b^n a^{2n}$ we choose $x = a^{2n} b^n$. This time $x = uvw$, where v is a string of one or more a's and $uv^i w \in L$ for every $i \geq 0$. The way to get a contradiction now is to consider $uv^0 w$, which has fewer a's than x does. Unfortunately, this produces a contradiction only if $|v| = n$. Since we don't know what $|v|$ is, the proof will not work for this choice of x.

The problem is not that x contains a's before b's; rather, it is that the original numbers of a's and b's are too far apart to guarantee a contradiction. Getting a contradiction in this case means making an inequality fail; if we start with a string in which the inequality is

just barely satisfied, then ideally any change in the right direction will cause it to fail. A better choice, for example, is $x = a^{n+1}b^n$. (If we had used $x = b^n a^{n+1}$ instead of $b^n a^{2n}$ for our first choice, we could have used $i = 2$ instead of $i = n$ to get a contradiction.)

Letting $x = (ab)^n a$ is also a bad choice, but for a different reason. We know that $x = uvw$ for some strings u, v, and w satisfying conditions 1–3, but now we don't have enough information about the string v. It might be $(ab)^k a$ for some k, so that uv^0w produces a contradiction; it might be $(ba)^k b$, so that uv^2w produces a contradiction; or it might be either $(ab)^k$ or $(ba)^k$, so that changing the number of copies of v doesn't change the relationship between n_a and n_b and doesn't give us a contradiction.

EXAMPLE 2.32 The Language $L = \{a^{i^2} \mid i \geq 0\}$

Whether a string of a's is an element of L depends only on its length; in this sense, our proof will be more about numbers than about strings.

Suppose L can be accepted by an FA M with n states. Let us choose x to be the string a^{n^2}. Then according to the pumping lemma, $x = uvw$ for some strings u, v, and w satisfying conditions 1–3. Conditions 1 and 2 tell us that $0 < |v| \leq n$. Therefore,

$$n^2 = |uvw| < |uv^2w| = n^2 + |v| \leq n^2 + n < n^2 + 2n + 1 = (n + 1)^2$$

This is a contradiction, because condition 3 says that $|uv^2w|$ must be i^2 for some integer i, but there is no integer i whose square is strictly between n^2 and $(n + 1)^2$.

EXAMPLE 2.33 Languages Related to Programming Languages

Almost exactly the same pumping-lemma proof that we used in Example 2.30 to show *AnBn* cannot be accepted by a finite automaton also works for several other languages, including several that a compiler for a high-level programming language must be able to accept. These include both the languages *Balanced* and *Expr* introduced in Example 1.19, because $(^m)^n$ is a balanced string, and $(^m a)^n$ is a legal expression, if and only if $m = n$.

Another example is the set L of legal C programs. We don't need to know much about the syntax of C to show that this language can't be accepted by an FA—only that the string

```
main(){{{ ... }}}
```

with m occurrences of "{" and n occurrences of "}", is a legal C program precisely if $m = n$. As usual, we start our proof by assuming that L is accepted by some FA with n states and letting x be the string `main() {`^{n}`}`^{n}. If $x = uvw$, and these three strings satisfy conditions 1–3, then the easiest way to obtain a contradiction is to use $i = 0$ in condition 3. The string v cannot contain any right brackets because of condition 1; if the shorter string uw is missing any of the symbols in "`main()`", then it doesn't have the legal header necessary for a C program, and if it is missing any of the left brackets, then the two numbers don't match.

As the pumping lemma demonstrates, one way to answer questions about a language is to examine a finite automaton that accepts it. In particular, for languages that can be accepted by FAs, there are several *decision problems* (questions with

yes-or-no answers) we can answer this way, and some some of them have decision algorithms that take advantage of the pumping lemma.

Decision Problems Involving Languages Accepted by Finite Automata

EXAMPLE 2.34

The most fundamental question about a language L is which strings belong to it. The *membership problem* for a language L accepted by an FA M asks, for an arbitrary string x over the input alphabet of M, whether $x \in L(M)$. We can think of the problem as specific to M, so that an *instance* of the problem is a particular string x; we might also ask the question for an arbitrary M and an arbitrary x, and consider an instance to be an ordered pair (M, x). In either case, the way a finite automaton works makes it easy to find a decision algorithm for the problem. Knowing the string x and the specifications for $M = (Q, \Sigma, q_0, A, \delta)$ allows us to compute the state $\delta^*(q_0, x)$ and check to see whether it is an element of A.

The two problems below are examples of other questions we might ask about $L(M)$:

1. Given an FA $M = (Q, \Sigma, q_0, A, \delta)$, is $L(M)$ nonempty?
2. Given an FA $M = (Q, \Sigma, q_0, A, \delta)$, is $L(M)$ infinite?

One way for the language $L(M)$ to be empty is for A to be empty, but this is not the only way. The real question is not whether M has any accepting states, but whether it has any that are *reachable* from q_0. The same algorithm that we used in Example 1.21 to find the set of cities reachable from city S can be used here; another algorithm that is less efficient but easier to describe is the following.

■ Decision Algorithm for Problem 1

Let n be the number of states of M. Try strings in order of length, to see whether any are accepted by M; $L(M) \neq \emptyset$ if and only if there is a string of length less than n that is accepted, where n is the number of states of M.

The reason this algorithm works is that according to the pumping lemma, for every string $x \in L(M)$ with $|x| \geq n$, there is a shorter string in $L(M)$, the one obtained by omitting the middle portion v. Therefore, it is impossible for the shortest string accepted by M to have length n or more.

It may be less obvious that an approach like this will work for problem 2, but again we can use the pumping lemma. If n is the number of states of M, and x is a string in $L(M)$ with $|x| \geq n$, then there are infinitely many longer strings in $L(M)$. On the other hand, if $x \in L(M)$ and $|x| \geq 2n$, then $x = uvw$ for some strings u, v, and w satisfying the conditions in the pumping lemma, and $uv^0w = uw$ is a shorter string in $L(M)$ whose length is still n or more, because $|v| \leq n$. In other words, if there is a string in L with length at least n, the shortest such string must have length less than $2n$. It follows that the algorithm below, which is undoubtedly inefficient, is a decision algorithm for problem 2.

■ Decision Algorithm for Problem 2

Try strings in order of length, starting with strings of length n, to see whether any are accepted by M. $L(M)$ is infinite if and only if a string x is found with $n \leq |x| < 2n$.

2.5 HOW TO BUILD A SIMPLE COMPUTER USING EQUIVALENCE CLASSES

In Section 2.3 we considered a three-state finite automaton M accepting L, the set of strings over $\{a, b\}$ that end with aa. We showed that three states were necessary by finding three pairwise L-distinguishable strings, one for each state of M. Now we are interested in why three states are enough. Of course, if M really does accept L, then three are enough; we can check that this is the case by showing that if x and y are two strings that cause M to end up in the same state, then M doesn't need to distinguish them, because they are not L-distinguishable.

Each state of M corresponds to a set of strings, and we described the three sets in Example 2.1: the strings that do not end with a, the strings that end with a but not aa, and the strings in L. If x is a string in the second set, for example, then for every string z, $xz \in L$ precisely if $z = a$ or z itself ends in aa. We don't need to know what x is in order to say this, only that x ends with a but not aa. Therefore, no two strings in this set are L-distinguishable. A similar argument works for each of the other two sets.

We will use "L-indistinguishable" to mean not L-distinguishable. We can now say that if S is any one of these three sets, then

1. Any two strings in S are L-indistinguishable.
2. No string of S is L-indistinguishable from a string not in S.

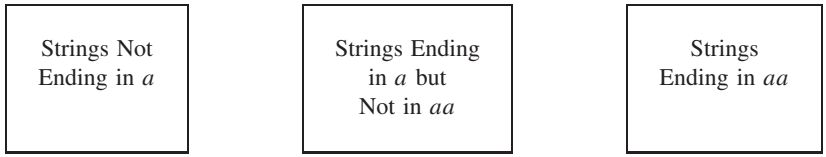
If the "L-indistinguishability" relation is an equivalence relation, then as we pointed out at the end of Section 1.3, these two statements about a set S say precisely that S is one of the equivalence classes. The relation is indeed an equivalence relation. Remember that x and y are L-indistinguishable if and only if $L/x = L/y$ (see Definition 2.20); this characterization makes it easy to see that the relation is reflexive, symmetric, and transitive, because the equality relation has these properties.

Definition 2.35 The *L*-Indistinguishability Relation

For a language $L \subseteq \{a, b\}^*$, we define the relation I_L (an equivalence relation) on Σ^* as follows: For $x, y \in \Sigma^*$,

$$x \, I_L \, y \text{ if and only if } x \text{ and } y \text{ are } L\text{-indistinguishable}$$

In the case of the language L of strings ending with aa, we started with the three-state FA and concluded that for each state, the corresponding set was one of the equivalence classes of I_L. What if we didn't have an FA but had figured out what the equivalence classes were, and that there were only three?



Then we would have at least the first important ingredient of a finite automaton accepting L: a finite set, whose elements we could call *states*. Calling a set of strings a state is reasonable, because we already had in mind an association between a state and the set of strings that led the FA to that state. But in case it seems questionable, states don't have to *be* anything special—they only have to be representable by circles (or, as above, rectangles), with some coherent way of defining an initial state, a set of accepting states, and a transition function.

Once we start to describe this FA, the details fall into place. The initial state should be the equivalence class containing the string Λ, because Λ is one of the strings corresponding to the initial state of any FA. Because we want the FA to accept L, the accepting states (in this case, only one) should be the equivalence classes containing elements of L.

Let us take one case to illustrate how the transitions can be defined. The third equivalence class is the set containing strings ending with aa. If we choose an arbitrary element, say $abaa$, we have one string that we want to correspond to that state. If the next input symbol is b, then the current string that results is $abaab$. Now we simply have to determine which of the three sets this string belongs to, and the answer is the first. You can see in the other cases as well that what we end up with is the diagram shown in Figure 2.2.

That's almost all there is to the construction of the FA, although there are one or two subtleties in the proof of Theorem 2.36. The other half of the theorem says that there is an FA accepting L *only* if the set of equivalence classes of I_L is finite. The two parts therefore give us an if-and-only-if characterization of the languages that can be accepted by finite automata. The pumping lemma in Section 2.4 does not; we will see in Example 2.39 that there are languages L for which, although L is not accepted by any FA, the pumping lemma does not allow us to prove this and Theorem 2.36 does.

Theorem 2.36

If $L \subseteq \Sigma^*$ can be accepted by a finite automaton, then the set Q_L of equivalence classes of the relation I_L on Σ^* is finite. Conversely, if the set Q_L is finite, then the finite automaton $M_L = (Q_L, \Sigma, q_0, A, \delta)$ accepts L, where $q_0 = [\Lambda]$, $A = \{q \in Q_L \mid q \subseteq L\}$, and for every $x \in \Sigma^*$ and every $a \in \Sigma$,

$$\delta([x], a) = [xa]$$

Finally, M_L has the fewest states of any FA accepting L.

Proof

If Q_L is infinite, then a set S containing exactly one string from each equivalence class is an infinite set of pairwise L-distinguishable strings. If there were an FA accepting L, it would have k states, for some k; but S has $k + 1$ pairwise L-distinguishable strings, and it follows from Theorem 2.21 that every FA accepting L must have at least $k + 1$ states. Therefore, there is no FA accepting L.

If Q_L is finite, on the other hand, we want to show that the FA M_L accepts L. First, however, we must consider the question of whether the definition of M_L even makes sense. The formula $\delta([x], a) = [xa]$ is supposed to assign an equivalence class (exactly one) to the ordered pair $([x], a)$. The equivalence class $[x]$ containing the string x might also contain another string y, so that $[x] = [y]$. In order for the formula to be a sensible definition of δ, it should be true that in this case

$$[xa] = \delta([x], a) = \delta([y], a) = [ya]$$

The question, then, is whether the statement $[x] = [y]$ implies the statement $[xa] = [ya]$. Fortunately, it does. If $[x] = [y]$, then xI_Ly, which means that for every string z', xz' and yz' are either both in L or both not in L; therefore, for every z, xaz and yaz are either both in L or both not in L (because of the previous statement with $z' = az$), and so xaI_Lya.

The next step in the proof is to verify that with this definition of δ, the formula

$$\delta^*([x], y) = [xy]$$

holds for every two strings $x, y \in \Sigma^*$. This is a straightforward proof by structural induction on y, which uses the definition of δ for this FA and the definition of δ^* for any FA.

From this formula, it follows that

$$\delta^*(q_0, x) = \delta^*([\Lambda], x) = [x]$$

It follows from our definition of A that x is accepted by M_L if and only if $[x] \subseteq L$. What we want is that x is accepted if and only if $x \in L$, and so we must show that $[x] \subseteq L$ if and only if $x \in L$. If the first statement is true, then the second is, because $x \in [x]$. On the other hand, if $x \in L$, and y is any element of $[x]$, then yI_Lx. Since $x\Lambda \in L$, and x and y are L-indistinguishable, $y\Lambda = y \in L$. Therefore, $[x] \subseteq L$.

A set containing one element from each equivalence class of I_L is a set of pairwise L-distinguishable strings. Therefore, by the second statement of Theorem 2.21, every FA accepting L must have at least as many states as there are equivalence classes. M_L, with exactly this number of states, has the fewest possible.

The statement that L can be accepted by an FA if and only if the set of equivalence classes of I_L is finite is known as the Myhill-Nerode Theorem.

As a practical matter, if we are trying to construct a finite automaton to accept a language L, it is often easier to attack the problem directly, as in Example 2.22, than to determine the equivalence classes of I_L. Theorem 2.36 is interesting because it can be interpreted as an answer to the question of how much a computer accepting a language L needs to remember about the current string x. It can forget everything about x except the equivalence class it belongs to. The theorem provides an elegant description of an "abstract" finite automaton accepting a language, and we will see in the next section that it will help us if we already have an FA and are trying to simplify it by eliminating all the unnecessary states.

If identifying equivalence classes is not always the easiest way to construct an FA accepting a language L, identifying the equivalence classes from an existing FA $M = (Q, \Sigma, q_0, A, \delta)$ is relatively straightforward. For each state q, we define

$$L_q = \{x \in \Sigma^* \mid \delta^*(q_0, x) = q\}$$

Every one of the sets L_q is a subset of some equivalence class of I_L; otherwise (if for some q, L_q contained strings in two different equivalence classes), there would be two L-distinguishable strings that caused M to end up in the same state, which would contradict Theorem 2.21. It follows that the number of equivalence classes is no larger than the number of states. If M has the property that strings corresponding to different states are L-distinguishable, then the two numbers are the same, and the equivalence classes are precisely the sets L_q, just as for the language of strings ending in aa discussed at the beginning of this section.

It is possible that an FA M accepting L has more states than I_L has equivalence classes. This means that for at least one equivalence class S, there are two different states q_1 and q_2 such that L_{q_1} and L_{q_2} are both subsets of S. We will see in the next section that in this case M has more states than it needs, and we will obtain an algorithm to simplify M by combining states wherever possible. The sets L_q for the simplified FA are the equivalence classes of I_L, as in the preceding paragraph.

In the next example in this section, we return to the language *AnBn* and calculate the equivalence classes. We have already used the pumping lemma to show there is no FA accepting this language (Example 2.30), so we will not be surprised to find that there are infinitely many equivalence classes.

The Equivalence Classes of I_L, Where L = *AnBn* **EXAMPLE 2.37**

As we observed in Example 2.30, accepting $AnBn = \{a^n b^n \mid n \geq 0\}$ requires that we remember how many a's we have read, so that we can compare that number to the number of b's. A precise way to say this is that two different strings of a's are L-distinguishable: if $i \neq j$, then $a^i b^i \in L$ and $a^j b^i \notin L$. Therefore, the equivalence classes $[a^j]$ are all distinct. If we were interested only in showing that the set of equivalence classes is infinite, we could stop here.

Exactly what are the elements of $[a^j]$? Not only is the string a^j L-distinguishable from a^i, but it is L-distinguishable from *every* other string x: A string that distinguishes the two is ab^{j+1}, because $a^j ab^{j+1} \in L$ and $xab^{j+1} \notin L$. Therefore, there are no other strings in the set $[a^j]$, and

$$[a^j] = \{a^j\}$$

Each of the strings a^i is a prefix of an element of L. Other prefixes of elements of L include elements of L themselves and strings of the form $a^i b^j$ where $i > j$. All other strings in $\{a, b\}^*$ are nonprefixes of elements of L.

Two nonnull elements of L are L-indistinguishable, because if a string other than Λ is appended to the end of either one, the result is not in L; and every nonnull string in L can be distinguished from every string not in L by the string Λ. Therefore, the set $L - \{\Lambda\}$ is an equivalence class of I_L.

The set of nonprefixes of elements of L is another equivalence class: No two nonprefixes can be distinguished relative to L, and if $xy \in L$, then the string y distinguishes x from every nonprefix.

We are left with the strings $a^i b^j$ with $i > j > 0$. Let's consider an example, say $x = a^7 b^3$. The only string z with $xz \in L$ is b^4. However, there are many other strings y that share this property with x; every string $a^{i+4} b^i$ with $i > 0$ does. The equivalence class $[a^7 b^3]$ is the set $\{a^{i+4} b^i \mid i > 0\} = \{a^5 b, a^6 b^2, a^7 b^3, \ldots\}$. Similarly, for every $k > 0$, the set $\{a^{i+k} b^i \mid i > 0\}$ is an equivalence class.

To summarize, L and the set of nonprefixes of elements of L are two equivalence classes that are infinite sets. For each $j \geq 0$, the set with the single element a^j is an equivalence class; and for every $k > 0$, the infinite set $\{a^{k+i} b^i \mid i > 0\}$ is an equivalence class. We have now accounted for all the strings in $\{a, b\}^*$.

EXAMPLE 2.38 The Equivalence Classes of I_L, Where $L = \{a^{n^2} \mid n \in \mathcal{N}\}$

We show that if L is the language in Example 2.36 of strings in $\{a\}^*$ with length a perfect square, then the elements of $\{a\}^*$ are pairwise L-distinguishable. Suppose i and j are two integers with $0 \leq i < j$; we look for a positive integer k so that $j + k$ is a perfect square and $i + k$ is not, so that $a^j a^k \in L$ and $a^i a^k \notin L$. Let $k = (j + 1)^2 - j = j^2 + j + 1$. Then $j + k = (j + 1)^2$ but $i + k = j^2 + i + j + 1 > j^2$, so that $i + k$ falls between j^2 and $(j + 1)^2$.

The language $Pal \subseteq \{a, b\}^*$ (Examples 1.18 and 2.31) was the first one we found for which no equivalence class of I_L has more than one element. That example was perhaps a little more dramatic than this one, in which the argument depends only on the length of the string. Of course, palindromes over a one-symbol alphabet are not very interesting. If we took L to be the set of all strings in $\{a, b\}^*$ with length a perfect square, each equivalence class would correspond to a natural number n and would contain all strings of that length.

According to the pumping lemma, if L is accepted by an FA M, then there is a number n, depending on M, such that we can draw some conclusions about

every string in L with length at least n. In the applications in Section 2.4, we didn't need to know where the number n came from, only that it existed. If there is no such number (that is, if the assumption that there is leads to a contradiction), then L cannot be accepted by an FA. If there is such a number—never mind where it comes from—does it follow that there is an FA accepting L? The answer is no, as the following slightly complicated example illustrates.

A Language Can Satisfy the Conclusions of the Pumping Lemma and Still Not Be Accepted by a Finite Automaton

EXAMPLE 2.39

Consider the language

$$L = \{a^i b^j c^j \mid i \geq 1 \text{ and } j \geq 0\} \cup \{b^j c^k \mid j \geq 0 \text{ and } k \geq 0\}$$

Strings in L contain a's, then b's, then c's; if there is at least one a in the string, then the number of b's and the number of c's have to be the same, and otherwise they don't.

We will show that for the number $n = 1$, the statement in the pumping lemma is true for L. Suppose $x \in L$ and $|x| \geq 1$. If there is at least one a in the string x, and $x = a^i b^j c^j$, we can define

$$u = \Lambda \qquad v = a \qquad w = a^{i-1} b^j c^j$$

Every string of the form $uv^i w$ is still of the form $a^k b^j c^j$ and is therefore still an element of L, whether k is 0 or not. If x is $b^i c^j$, on the other hand, then again we define u to be Λ and v to be the first symbol in x, which is either b or c. It is also true in this case that $uv^i w \in L$ for every $i \geq 0$.

We can use Theorem 2.26 to show that there is no finite automaton accepting our language L, because there is an infinite set of pairwise L-distinguishable strings. If S is the set $\{ab^n \mid n \geq 0\}$, any two distinct elements ab^m and ab^n are distinguished by the string c^m.

2.6 MINIMIZING THE NUMBER OF STATES IN A FINITE AUTOMATON

Suppose we have a finite automaton $M = (Q, \Sigma, q_0, A, \delta)$ accepting $L \subseteq \Sigma^*$. For a state q of M, we have introduced the notation L_q to denote the set of strings that cause M to be in state q:

$$L_q = \{x \in \Sigma^* \mid \delta^*(q_0, x) = q\}$$

The first step in reducing the number of states of M as much as possible is to eliminate every state q for which $L_q = \emptyset$, along with transitions from these states. None of these states is reachable from the initial state, and eliminating them does not change the language accepted by M. For the remainder of this section, we assume that all the states of M are reachable from q_0.

We have defined an equivalence relation on Σ^*, the L-indistinguishability relation I_L, and we have seen that for each state q in M, all the strings in L_q are L-indistinguishable. In other words, the set L_q is a subset of one of the equivalence classes of L_q.

The finite automaton we described in Theorem 2.36, with the fewest states of any FA accepting L, is the one in which each state corresponds precisely to (according to our definition, *is*) one of the equivalence classes of I_L. For each state q in this FA, L_q is as large as possible—it contains every string in some equivalence class of I_L. Every FA in which this statement doesn't hold has more states than it needs. There are states p and q such that the strings in L_p are L-indistinguishable from those in L_q; if M doesn't need to distinguish between these two types of strings, then q can be eliminated, and the set L_p can be enlarged by adding the strings in L_q.

The equivalence relation on Σ^* gives us an equivalence relation $\equiv$ on the set Q of states of M. For $p, q \in Q$,

$p \equiv q$ if and only if strings in L_p are L-indistinguishable from strings in L_q

This is the same as saying

$p \equiv q$ if and only if L_p and L_q are subsets of the same equivalence class of I_L

Two strings x and y are L-distinguishable if for some string z, exactly one of the two strings xz, yz is in L. Two states p and q fail to be equivalent if strings x and y corresponding to p and q, respectively, are L-distinguishable, and this means:

$p \not\equiv q$ if and only if, for some string z, exactly one of the states $\delta^*(p, z), \delta^*(q, z)$ is in A

In order to simplify M by eliminating unnecessary states, we just need to identify the unordered pairs (p, q) for which the two states can be combined into one. The definition of $p \not\equiv q$ makes it easier to identify the pairs (p, q) for which p and q cannot be combined, the ones for which $p \not\equiv q$. We look systematically for a string z that might distinguish the states p and q (or distinguish a string in L_p from one in L_q). With this in mind, we let S_M be the set of such unordered pairs.

S_M is the set of unordered pairs (p, q) of distinct states satisfying $p \not\equiv q$

A Recursive Definition of S_M

The set S_M can be defined as follows:

1. For every pair (p, q) with $p \neq q$, if exactly one of the two states is in A, $(p, q) \in S_M$.
2. For every pair (r, s) of distinct states, if there is a symbol $\sigma \in \Sigma$ such that the pair $(\delta(r, \sigma), \delta(s, \sigma))$ is in S_M, then $(r, s) \in S_M$.

In the basis statement we get the pairs of states that can be distinguished by Λ. If the states $\delta(r, \sigma)$ and $\delta(s, \sigma)$ are distinguished by the string z, then the states r and s are distinguished by the string σz; as a result, if the states r and s can be

distinguished by a string of length n, then the pair (r, s) can be added to the set by using the recursive part of the definition n or fewer times.

Because the set S_M is finite, this recursive definition provides an algorithm for identifying the elements of S_M.

Algorithm 2.40 Identifying the Pairs *(p, q)* with $p \not\equiv q$ List all unordered pairs of distinct states (p, q). Make a sequence of passes through these pairs as follows. On the first pass, mark each pair (p, q) for which exactly one of the two states p and q is in A. On each subsequent pass, examine each unmarked pair (r, s); if there is a symbol $\sigma \in \Sigma$ such that $\delta(r, \sigma) = p$, $\delta(s, \sigma) = q$, and the pair (p, q) has already been marked, then mark (r, s).

After a pass in which no new pairs are marked, stop. At that point, the marked pairs (p, q) are precisely those for which $p \not\equiv q$. ■

Algorithm 2.40 is the crucial ingredient in the algorithm to simplify the FA by minimizing the number of states. When Algorithm 2.40 terminates, every pair (p, q) that remains unmarked represents two states that can be combined into one, because the corresponding sets L_p and L_q are subsets of the same equivalence class. It may happen that every pair ends up marked; this means that for distinct states p and q, strings in p are already L-distinguishable from strings in q, and M already has the fewest states possible.

We finish up by making one final pass through the states. The first state to be considered represents an equivalence class, or a state in our new minimal FA. After that, a state q represents a new equivalence class, or a new state, only if the pair (p, q) is marked for every state p considered previously. Each time we consider a state q that does not produce a new state in the minimal FA, because it is equivalent to a previous state p, we will add it to the set of original states that are being combined with p.

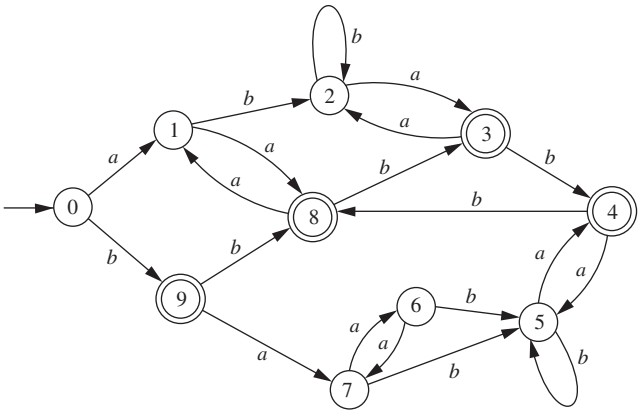
Once we have the states in the resulting minimum-state FA, determining the transitions is straightforward. Example 2.41 illustrates the algorithm.

Applying the Minimization Algorithm **EXAMPLE 2.41**

Figure 2.42a shows a finite automaton with ten states, numbered 0–9, and Figure 2.42b shows the unordered pairs (p, q) with $p \neq q$. The pairs marked 1 are the ones marked on pass 1, in which exactly one state is an accepting state, and those marked 2 or 3 are the ones marked on the second or third pass. In this example, the third pass is the last one on which new pairs were marked.

How many passes are required and which pairs are marked on each one may depend on the order in which the pairs are considered during each pass. The results in Figure 2.42b are obtained by proceeding one vertical column at a time, and considering the pairs in each column from top to bottom.

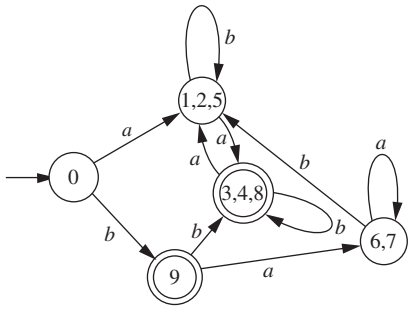
The pair (6, 3) is one of the pairs marked on the first pass, since 3 is an accepting state and 6 is not. When the pair (7, 2) is considered on the second pass, it is marked because $\delta(7, a) = 6$ and $\delta(2, a) = 3$. When the pair (9, 3) is considered later on the second pass, it is



(*a*)

1	2							
2	2							
3	1	1	1					
4	1	1	1					
5	2			1	1			
6	2	2	2	1	1	2		
7	2	2	2	1	1	2		
8	1	1	1			1	1	1
9	1	1	1	2	3	1	1	1	2
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

(*b*)



(*c*)

Figure 2.42 |
Minimizing the number of states in an FA.

also marked, because $\delta(9, a) = 7$ and $\delta(3, a) = 2$. The pair $(7, 5)$ is marked on the second pass. We have $\delta(9, a) = 7$ and $\delta(4, a) = 5$, but $(9, 4)$ was considered earlier on the second pass, and so it is not marked until the third pass.

With the information from Figure 2.42b, we can determine the states in the minimal FA as follows. State 0 will be a new state. State 1 will be a new state, because the pair $(1, 0)$ is marked. State 2 will not, because $(2, 1)$ is unmarked, which means we combine states 2 and 1. State 3 will be a new state. State 4 will be combined with 3. State 5 will be combined with states 1 and 2, because both $(5, 1)$ and $(5, 2)$ are unmarked. State 6 will be a new state; state 7 is combined with state 6; state 8 is combined with 3 and 4; and state 9 is a new state. At this point, we have the five states shown in Figure 2.42c.

If we designate each state in the FA by the set of states in the original FA that were combined to produce it, we can compute the transitions from the new state by considering

any of the elements of that set. For example, one of the new states is $\{1, 2, 5\}$; in the original FA, $\delta(1, a) = 8$, which tells us that the a-transition from $\{1, 2, 5\}$ goes to $\{3, 4, 8\}$. (If there are any inconsistencies, such as $\delta(5, a)$ not being an element of $\{3, 4, 8\}$, then we've made a mistake somewhere!)

EXERCISES

2.1. In each part below, draw an FA accepting the indicated language over $\{a, b\}$.

a. The language of all strings containing exactly two a's.

b. The language of all strings containing at least two a's.

c. The language of all strings that do not end with ab.

d. The language of all strings that begin or end with aa or bb.

e. The language of all strings not containing the substring aa.

f. The language of all strings in which the number of a's is even.

g. The language of all strings in which both the number of a's and the number of b's are even.

h. The language of all strings containing no more than one occurrence of the string aa. (The string aaa contains two occurrences of aa.)

i. The language of all strings in which every a (if there are any) is followed immediately by bb.

j. The language of all strings containing both bb and aba as substrings.

k. The language of all strings containing both aba and bab as substrings.

2.2. For each of the FAs pictured in Fig. 2.43, give a simple verbal description of the language it accepts.

2.3. a. Draw a transition diagram for an FA that accepts the string $abaa$ and no other strings.

b. For a string $x \in \{a, b\}^*$ with $|x| = n$, how many states are required for an FA accepting x and no other strings? For each of these states, describe the strings that cause the FA to be in that state.

c. For a string $x \in \{a, b\}^*$ with $|x| = n$, how many states are required for an FA accepting the language of all strings in $\{a, b\}^*$ that begin with x? For each of these states, describe the strings that cause the FA to be in that state.

2.4. Example 2.7 describes an FA accepting L_3, the set of strings in $\{0, 1\}^*$ that are binary representations of integers divisible by 3. Draw a transition diagram for an FA accepting L_5.

2.5. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA, q is an element of Q, and x and y are strings in Σ^*. Using structural induction on y, prove the formula

$$\delta^*(q, xy) = \delta^*(\delta^*(q, x), y)$$

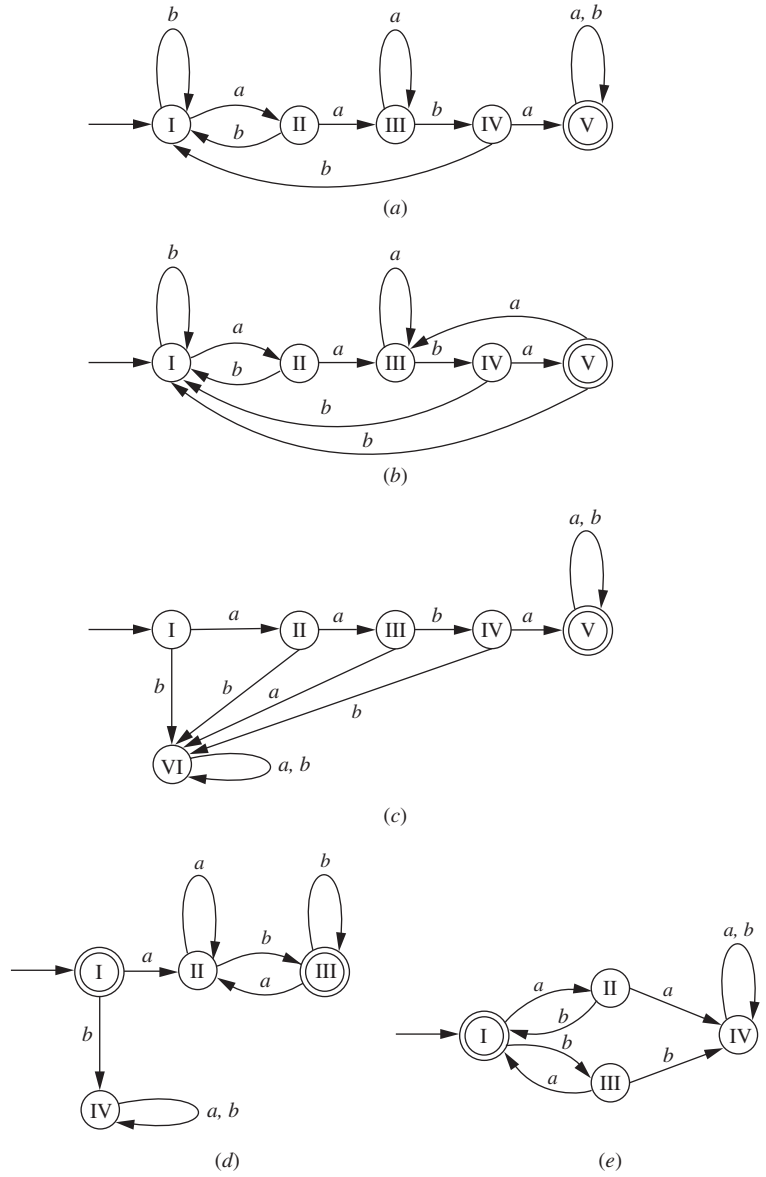


Figure 2.43

2.6. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA, q is an element of Q, and $\delta(q, \sigma) = q$ for every $\sigma \in \Sigma$. Show using structural induction that for every $x \in \Sigma^*$, $\delta^*(q, x) = q$.

2.7. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA. Let $M_1 = (Q, \Sigma, q_0, R, \delta)$, where R is the set of states p in Q for which $\delta^*(p, z) \in A$ for some string z. What

is the relationship between the language accepted by M_1 and the language accepted by M? Prove your answer.

2.8. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA. Below are other conceivable methods of defining the extended transition function δ^* (see Definition 2.12). In each case, determine whether it is in fact a valid definition of a function on the set $Q \times \Sigma^*$, and why. *If* it is, show using mathematical induction that it defines the same function that Definition 2.12 does.

a. For every $q \in Q$, $\delta^*(q, \Lambda) = q$; for every $y \in \Sigma^*$, $\sigma \in \Sigma$, and $q \in Q$, $\delta^*(q, y\sigma) = \delta^*(\delta^*(q, y), \sigma)$.

b. For every $q \in Q$, $\delta^*(q, \Lambda) = q$; for every $y \in \Sigma^*$, $\sigma \in \Sigma$, and $q \in Q$, $\delta^*(q, \sigma y) = \delta^*(\delta(q, \sigma), y)$.

c. For every $q \in Q$, $\delta^*(q, \Lambda) = q$; for every $q \in Q$ and every $\sigma \in \Sigma$, $\delta^*(q, \sigma) = \delta(q, \sigma)$; for every $q \in Q$, and every x and y in Σ^*, $\delta^*(q, xy) = \delta^*(\delta^*(q, x), y)$.

2.9. In order to test a string for membership in a language like the one in Example 2.1, we need to examine only the last few symbols. More precisely, there is an integer n and a set S of strings of length n such that for every string x of length n or greater, x is in the language if and only if $x = yz$ for some $z \in S$.

a. Show that every language L having this property can be accepted by an FA.

b. Show that every finite language has this property.

c. Give an example of an infinite language that can be accepted by an FA but does not have this property.

2.10. Let M_1 and M_2 be the FAs pictured in Figure 2.44, accepting languages L_1 and L_2, respectively.
Draw FAs accepting the following languages.

a. $L_1 \cup L_2$

b. $L_1 \cap L_2$

c. $L_1 - L_2$

2.11. (For this problem, refer to the proof of Theorem 2.15.) Show that for every $x \in \Sigma^*$ and every $(p, q) \in Q$, $\delta^*((p, q), x) = (\delta_1^*(p, x), \delta_2^*(q, x))$.

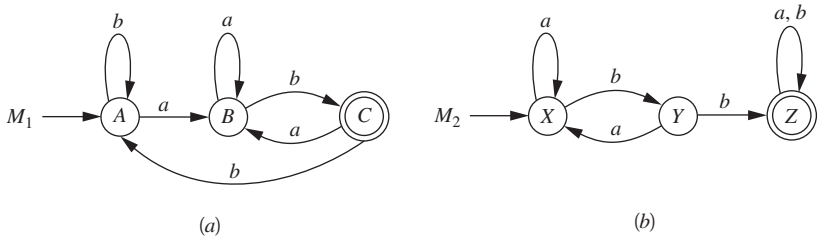


Figure 2.44

2.12. For each of the following languages, draw an FA accepting it.

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

b. $\{bb, ba\}^*$

c. $\{a, b\}^*\{b, aa\}\{a, b\}^*$

d. $\{bbb, baa\}^*\{a\}$

e. $\{a\} \cup \{b\}\{a\}^* \cup \{a\}\{b\}^*\{a\}$

f. $\{a, b\}^*\{ab, bba\}$

g. $\{b, bba\}^*\{a\}$

h. $\{aba, aa\}^*\{ba\}^*$

2.13. For the FA pictured in Fig. 2.17d, show that there cannot be any other FA with fewer states accepting the same language. (See Example 2.24, in which the same result is established for the FA accepting the language L_n.)

2.14. Let z be a fixed string of length n over the alphabet $\{a, b\}$. Using the argument in Example 2.5, we can find an FA with $n + 1$ states accepting the language of all strings in $\{a, b\}^*$ that end in z. The states correspond to the $n + 1$ distinct prefixes of z. Show that there can be no FA with fewer than $n + 1$ states accepting this language.

2.15. Suppose L is a subset of $\{a, b\}^*$. If $x_0, x_1, \ldots$ is a sequence of distinct strings in $\{a, b\}^*$ such that for every $n \geq 0$, x_n and x_{n+1} are L-distinguishable, does it follow that the strings $x_0, x_1, \ldots$ are pairwise L-distinguishable? Either give a proof that it does follow, or find an example of a language L and strings $x_0, x_1, \ldots$ that represent a counterexample.

2.16. Let $L \subseteq \{a, b\}^*$ be an infinite language, and for each $n \geq 0$, let $L_n = \{x \in L \mid |x| = n\}$. Denote by $s(n)$ the number of states an FA must have in order to accept L_n. What is the smallest that $s(n)$ can be if $L_n \neq \emptyset$? Give an example of an infinite language $L \subseteq \{a, b\}^*$ such that for every n satisfying $L_n \neq \emptyset$, $s(n)$ is this minimum number.

2.17. Let L be the language $AnBn = \{a^n b^n \mid n \geq 0\}$.

a. Find two distinct strings x and y in $\{a, b\}^*$ that are not L-distinguishable.

b. Find an infinite set of pairwise L-distinguishable strings.

2.18. Let n be a positive integer and $L = \{x \in \{a, b\}^* \mid |x| = n$ and $n_a(x) = n_b(x)\}$. What is the minimum number of states in any FA that accepts L? Give reasons for your answer.

2.19. Let n be a positive integer, and let L be the set of all strings in Pal of length $2n$. In other words,

$$L = \{xx^r \mid x \in \{a, b\}^n\}$$

What is the minimum number of states in any FA that accepts L? Give reasons for your answer.

2.20. Suppose L and L_1 are both languages over Σ, and M is an FA with alphabet Σ. Let us say that M accepts L *relative to* L_1 if M accepts every string in the set $L \cap L_1$ and rejects every string in the set $L_1 - L$. Note that this is not in general the same as saying that M accepts the language $L \cap L_1$.

Now suppose each of the languages $L_1, L_2, \ldots$ (subsets of Σ^*) can be accepted by an FA, $L_i \subseteq L_{i+1}$ for each i, and $\cup_{i=1}^{\infty} L_i = \Sigma^*$. For each i, let n_i be the minimum number of states required to accept L relative to L_i. If there is no FA accepting L relative to L_i, we say n_i is ∞.

a. Show that for each i, $n_i \leq n_{i+1}$.

b. Show that if the sequence n_i is bounded (i.e., there is a constant C such that $n_i \leq C$ for every i), then L can be accepted by an FA. Show in particular that if there is some fixed FA M that accepts L relative to L_i for every i, then M accepts L.

2.21. For each of the following languages $L \subseteq \{a, b\}^*$, show that the elements of the infinite set $\{a^n \mid n \geq 0\}$ are pairwise L-distinguishable.

a. $L = \{a^n b a^{2n} \mid n \geq 0\}$

b. $L = \{a^i b^j a^k \mid k > i + j\}$

c. $L = \{a^i b^j \mid j = i \text{ or } j = 2i\}$

d. $L = \{a^i b^j \mid j \text{ is a multiple of } i\}$

e. $L = \{x \in \{a, b\}^* \mid n_a(x) < 2n_b(x)\}$

f. $L = \{x \in \{a, b\}^* \mid \text{no prefix of } x \text{ has more b's than a's}\}$

g. $L = \{a^{n^3} \mid n \geq 1\}$

h. $L = \{ww \mid w \in \{a, b\}^*\}$

2.22. For each of the languages in Exercise 2.21, use the pumping lemma to show that it cannot be accepted by an FA.

2.23. By ignoring some of the details in the statement of the pumping lemma, we can easily get these two weaker statements.

I. If $L \subseteq \Sigma^*$ is an infinite language that can be accepted by an FA, then there are strings u, v, and w such that $|v| > 0$ and $uv^i w \in L$ for every $i \geq 0$.

II. If $L \subseteq \Sigma^*$ is an infinite language that can be accepted by an FA, then there are integers p and q such that $q > 0$ and for every $i \geq 0$, L contains a string of length $p + iq$.

For each language L in Exercise 2.21, decide whether statement II is enough to show that L cannot be accepted by an FA, and explain your

answer. If statement II is not sufficient, decide whether statement I is, and explain your answer.

2.24. Prove the following generalization of the pumping lemma, which can sometimes make it unnecessary to break the proof into cases. If L can be accepted by an FA, then there is an integer n such that for any $x \in L$, and any way of writing x as $x = x_1x_2x_3$ with $|x_2| = n$, there are strings u, v, and w such that

a. $x_2 = uvw$

b. $|v| > 0$

c. For every $m \geq 0$, $x_1uv^mwx_3 \in L$

2.25. Find a language $L \subseteq \{a, b\}^*$ such that, in order to prove that L cannot be accepted by an FA, the pumping lemma is not sufficient but the statement in Exercise 2.24 is.

2.26. The pumping lemma says that if M accepts a language L, and if n is the number of states of M, then for every $x \in L$ satisfying $|x| \geq n$, ... Show that the statement provides no information if L is finite: If M accepts a finite language L, and n is the number of states of M, then L can contain no strings of length n or greater.

2.27. Describe decision algorithms to answer each of the following questions.

a. Given two FAs M_1 and M_2, are there any strings that are accepted by neither?

b. Given an FA $M = (Q, \Sigma, q_0, A, \delta)$ and a state $q \in Q$, is there an x with $|x| > 0$ such that $\delta^*(q, x) = q$?

c. Given an FA M accepting a language L, and given two strings x and y, are x and y distinguishable with respect to L?

d. Given an FA M accepting a language L, and a string x, is x a prefix of an element of L?

e. Given an FA M accepting a language L, and a string x, is x a suffix of an element of L?

f. Given an FA M accepting a language L, and a string x, is x a substring of an element of L?

g. Given two FAs M_1 and M_2, is $L(M_1)$ a subset of $L(M_2)$?

h. Given two FAs M_1 and M_2, is every element of $L(M_1)$ a prefix of an element of $L(M_2)$?

2.28. Suppose L is a language over $\{a, b\}$, and there is a fixed integer k such that for every $x \in \Sigma^*$, $xz \in L$ for some string z with $|z| \leq k$. Does it follow that there is an FA accepting L? Why or why not?

2.29. For each statement below, decide whether it is true or false. If it is true, prove it. If it is not true, give a counterexample. All parts refer to languages over the alphabet $\{a, b\}$.

a. If $L_1 \subseteq L_2$, and L_1 cannot be accepted by an FA, then L_2 cannot.

b. If $L_1 \subseteq L_2$, and L_2 cannot be accepted by an FA, then L_1 cannot.

c. If neither L_1 nor L_2 can be accepted by an FA, then $L_1 \cup L_2$ cannot.

d. If neither L_1 nor L_2 can be accepted by an FA, then $L_1 \cap L_2$ cannot.

e. If L cannot be accepted by an FA, then L' cannot.

f. If L_1 can be accepted by an FA and L_2 cannot, then $L_1 \cup L_2$ cannot.

g. If L_1 can be accepted by an FA, L_2 cannot, and $L_1 \cap L_2$ can, then $L_1 \cup L_2$ cannot.

h. If L_1 can be accepted by an FA and neither L_2 nor $L_1 \cap L_2$ can, then $L_1 \cup L_2$ cannot.

i. If each of the languages $L_1, L_2, \ldots$ can be accepted by an FA, then $\cup_{n=1}^{\infty} L_n$ can.

j. If none of the languages $L_1, L_2, \ldots$ can be accepted by an FA, and $L_i \subseteq L_{i+1}$ for each i, then $\cup_{n=1}^{\infty} L_n$ cannot be accepted by an FA.

2.30. †A set S of nonnegative integers is an *arithmetic progression* if for some integers n and p,

$$S = \{n + ip \mid i \geq 0\}$$

Let A be a subset of $\{a\}^*$, and let $S = \{|x| \mid x \in A\}$.

a. Show that if S is an arithmetic progression, then A can be accepted by an FA.

b. Show that if A can be accepted by an FA, then S is the union of a finite number of arithmetic progressions.

2.31. †This exercise involves languages of the form

$$L = \{x \in \{a, b\}^* \mid n_a(x) = f(n_b(x))\}$$

for some function f from the set of natural numbers to itself. Example 2.30 shows that if f is the function defined by $f(n) = n$, then L cannot be accepted by an FA. If f is any constant function (e.g., $f(n) = 4$), there is an FA accepting L. One might ask whether this is still true when f is not restricted quite so severely.

a. Show that if L can be accepted by an FA, the function f must be bounded (for some integer B, $f(n) \leq B$ for every n). (Suggestion: suppose not, and apply the pumping lemma to strings of the form $a^{f(n)}b^n$.)

b. Show that if $f(n) = n \bmod 2$, then L can be accepted by an FA.

c. The function f in part (b) is an *eventually periodic* function; that is, there are integers n_0 and p, with $p > 0$, such that for every $n \geq n_0$, $f(n) = f(n + p)$. Show that if f is any eventually periodic function, L can be accepted by an FA.

d. Show that if L can be accepted by an FA, then f must be eventually periodic. (Suggestion: as in part (a), find a class of strings to which you can apply the pumping lemma.)

2.32. For which languages $L \subseteq \{a, b\}^*$ does the equivalence relation I_L have exactly one equivalence class?

2.33. Let x be a string of length n in $\{a, b\}^*$, and let $L = \{x\}$. How many equivalence classes does I_L have? Describe them.

2.34. Show that if $L \subseteq \Sigma^*$, and there is a string $x \in \Sigma^*$ that is not a prefix of an element of L, then the set of all strings that are not prefixes of elements of L is an infinite set that is one of the equivalence classes of I_L.

2.35. Let $L \subseteq \Sigma^*$ be any language. Show that if $[\Lambda]$ (the equivalence class of I_L containing Λ) is not $\{\Lambda\}$, then it is infinite.

2.36. For a certain language $L \subseteq \{a, b\}^*$, I_L has exactly four equivalence classes. They are $[\Lambda]$, $[a]$, $[ab]$, and $[b]$. It is also true that the three strings a, aa, and abb are all equivalent, and that the two strings b and aba are equivalent. Finally, $ab \in L$, but Λ and a are not in L, and b is not even a prefix of any element of L. Draw an FA accepting L.

2.37. Suppose $L \subseteq \{a, b\}^*$ and I_L has three equivalence classes. Suppose they can be described as the three sets $[a]$, $[aa]$, and $[aaa]$, and also as the three sets $[b]$, $[bb]$, and $[bbb]$. How many possibilities are there for the language L? For each one, draw a transition diagram for an FA accepting it.

2.38. In each part, find every possible language $L \subseteq \{a, b\}^*$ for which the equivalence classes of I_L are the three given sets.

a. $\{a, b\}^*\{b\}$, $\{a, b\}^*\{ba\}$, $\{\Lambda, a\} \cup \{a, b\}^*\{aa\}$

b. $(\{a, b\}\{a\}^*\{b\})^*$, $(\{a, b\}\{a\}^*\{b\})^*\{a\}\{a\}^*$, $(\{a, b\}^*\{a\}^*\{b\})^*\{b\}\{a\}^*$

c. $\{\Lambda\}$, $\{a\}(\{b\} \cup \{a\}\{a\}^*\{b\})^*$, $\{b\}(\{a\} \cup \{b\}\{b\}^*\{a\})^*$

2.39. In Example 2.37, if the language is changed to $\{a^n b^n \mid n > 0\}$, so that it does not contain Λ, are there any changes in the partition of $\{a, b\}^*$ corresponding to I_L? Explain.

2.40. Consider the language $L = AEqB = \{x \in \{a, b\}^* \mid n_a(x) = n_b(x)\}$.

a. Show that if $n_a(x) - n_b(x) = n_a(y) - n_b(y)$, then $x \; I_L \; y$.

b. Show that if $n_a(x) - n_b(x) \neq n_a(y) - n_b(y)$, then x and y are L-distinguishable.

c. Describe all the equivalence classes of I_L.

2.41. Let $L \subseteq \Sigma^*$ be a language, and let L_1 be the set of prefixes of elements of L. What is the relationship, if any, between the two partitions of Σ^* corresponding to the equivalence relations I_L and I_{L_1}, respectively? Explain.

2.42. a. List all the subsets A of $\{a, b\}^*$ having the property that for some language $L \subseteq \{a, b\}^*$ for which I_L has exactly two equivalence classes, $A = [\Lambda]$.

b. For each set A that is one of your answers to (a), how many distinct languages L are there such that I_L has two equivalence classes and $[\Lambda]$ is A?

2.43. Let $L = \{ww \mid w \in \{a, b\}^*\}$. Describe all the equivalence classes of I_L.

2.44. Let L be the language *Balanced* of balanced strings of parentheses. Describe all the equivalence classes of I_L.

2.45. †Let L be the language of all fully parenthesized algebraic expressions involving the operator $+$ and the identifier a. (L can be defined recursively by saying that $a \in L$ and $(x + y) \in L$ for every x and y in L.) Describe all the equivalence classes of I_L.

2.46. †For a language L over Σ, and two strings x and y in Σ^* that are L-distinguishable, let

$$d_{L,x,y} = \min\{|z| \mid z \text{ distinguishes } x \text{ and } y \text{ with respect to } L\}$$

a. For the language $L = \{x \in \{a, b\}^* \mid x \text{ ends in } aba\}$, find the maximum of the numbers $d_{L,x,y}$ over all possible pairs of L-distinguishable strings x and y.

b. If L is the language of balanced strings of parentheses, and if x and y are L-distinguishable strings with $|x| = m$ and $|y| = n$, find an upper bound involving m and n on the numbers $d_{L,x,y}$.

2.47. For an arbitrary string $x \in \{a, b\}^*$, denote by $x^\sim$ the string obtained by replacing all a's by b's and vice versa. For example, $\Lambda^\sim = \Lambda$ and $(abb)^\sim = baa$.

a. Define

$$L = \{xx^\sim \mid x \in \{a, b\}^*\}$$

Determine the equivalence classes of I_L.

b. Define

$$L_1 = \{xy \mid x \in \{a, b\}^* \text{ and } y \text{ is either } x \text{ or } x^\sim\}$$

Determine the equivalence classes of I_{L_1}.

2.48. †Let $L = \{x \in \{a, b\}^* \mid n_b(x) \text{ is an integer multiple of } n_a(x)\}$. Determine the equivalence classes of I_L.

2.49. Let L be a language over Σ. We know that I_L is a *right-invariant* equivalence relation; i.e., for any x and y in Σ^* and any $a \in \Sigma$, if $x \; I_L \; y$, then $xa \; I_L \; ya$. It follows from Theorem 2.36 that if the set of equivalence classes of I_L is finite, L can be accepted by an FA, and in this case L is the union of some (zero or more) of these equivalence classes. Show that if R is *any* right-invariant equivalence relation such that the set of equivalence classes of R is finite and L is the union of some of the equivalence classes of R, then L can be accepted by an FA.

2.50. †If P is a partition of $\{a, b\}^*$ (a collection of pairwise disjoint subsets whose union is $\{a, b\}^*$), then there is an equivalence relation R on $\{a, b\}^*$ whose equivalence classes are precisely the subsets in P. Let us say that P is right-invariant if the resulting equivalence relation is.

a. Show that for a subset S of $\{a, b\}^*$, S is one of the subsets of some right-invariant partition (not necessarily a finite partition) of $\{a, b\}^*$ if

and only if the following condition is satisfied: for every $x, y \in S$, and every $z \in \{a, b\}^*$, xz and yz are either both in S or both not in S.

b. To what simpler condition does this one reduce in the case where S is a finite set?

c. Show that if a finite set S satisfies this condition, then there is a finite right-invariant partition having S as one of its subsets.

d. For an arbitrary set S satisfying the condition in part (a), there might be no finite right-invariant partition having S as one of its subsets. Characterize those sets S for which there is.

2.51. For two languages L_1 and L_2 over Σ, we define the *quotient* of L_1 and L_2 to be the language

$$L_1/L_2 = \{x \mid \text{ for some } y \in L_2, xy \in L_1\}$$

Show that if L_1 can be accepted by an FA and L_2 is any language, then L_1/L_2 can be accepted by an FA.

2.52. Suppose L is a language over Σ, and $x_1, x_2, \ldots, x_n$ are strings that are pairwise L-distinguishable. How many distinct strings are necessary in order to distinguish between the x_i's? In other words, what is the smallest number k such that for some set $\{z_1, z_2, \ldots, z_k\}$, any two distinct x_i's are distinguished, relative to L, by some z_l? Prove your answer. (Here is a way of thinking about the question that may make it easier. Think of the x_i's as points on a piece of paper, and think of the z_l's as cans of paint, each z_l representing a different primary color. Saying that z_l distinguishes x_i and x_j means that one of those two points is colored with that primary color and the other isn't. We allow a single point to have more than one primary color applied to it, and we assume that two distinct combinations of primary colors produce different resulting colors. Then the question is, how many different primary colors are needed in order to color the points so that no two points end up the same color?)

2.53. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA accepting L. We know that if $p, q \in Q$ and $p \not\equiv q$, then there is a string z such that exactly one of the two states $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A. Show that there is an integer n such that for every p and q with $p \not\equiv q$, such a z can be found whose length is no greater than n, and say what n is.

2.54. Show that L can be accepted by an FA if and only if there is an integer n such that, for every pair of L-distinguishable strings, the two strings can be distinguished by a string of length $\leq n$. (Use the two previous exercises.)

2.55. For each of the FAs pictured in Fig. 2.45, use the minimization algorithm described in Section 2.6 to find a minimum-state FA recognizing the same language. (It's possible that the given FA may already be minimal.)

2.56. Suppose that in applying the minimization algorithm in Section 2.6, we establish some fixed order in which to process the pairs, and we follow the same order on each pass.

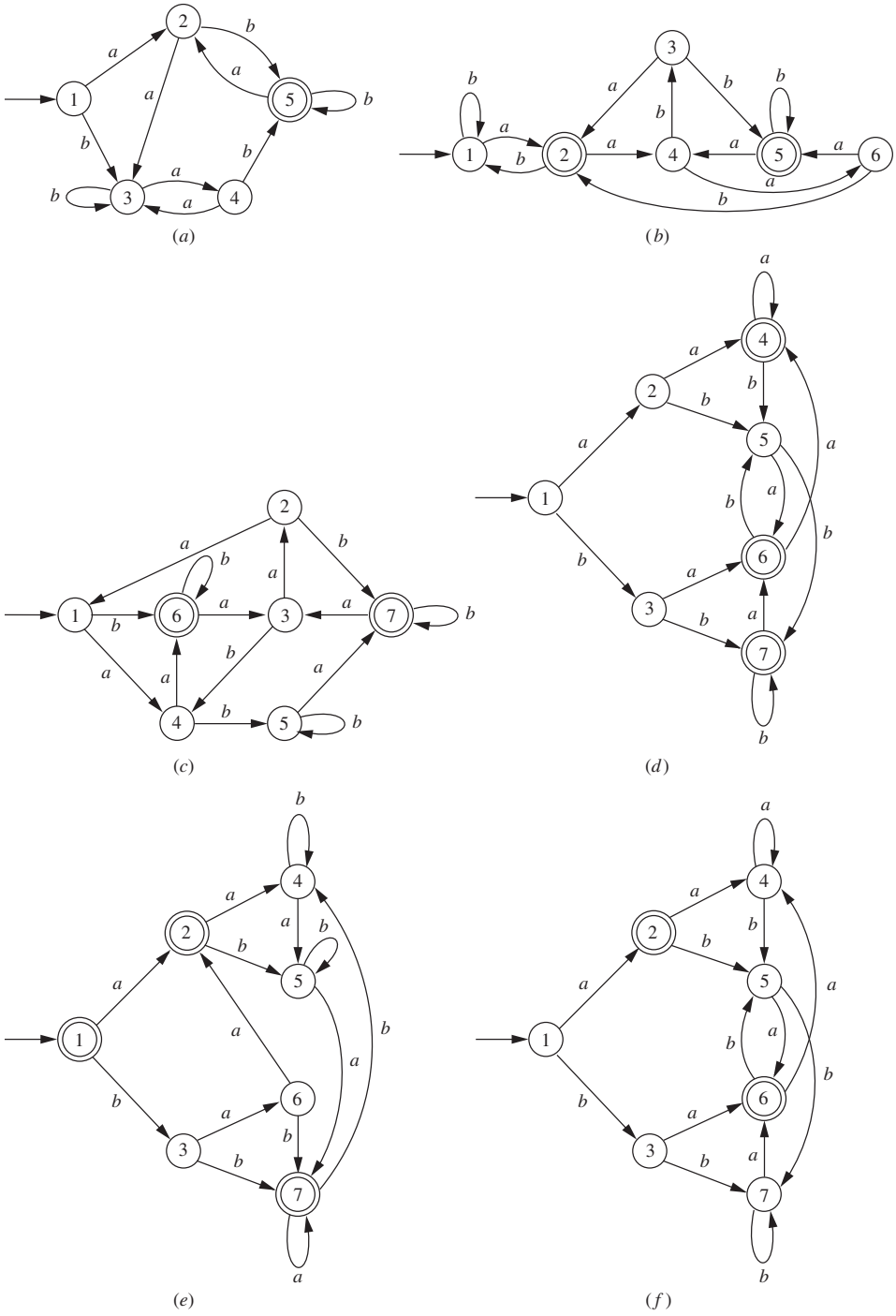
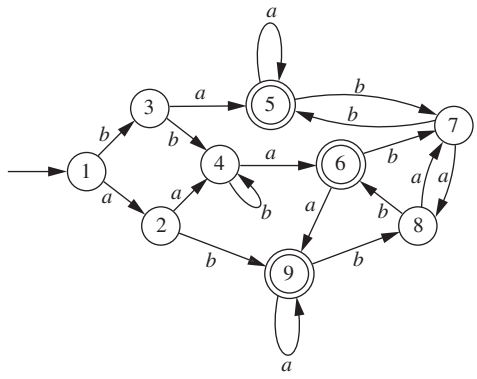


Figure 2.45 |



(*g*)

Figure 2.45 |
Continued

a. What is the maximum number of passes that might be required? Describe an FA, and an ordering of the pairs, that would require this number.

b. Is there always a fixed order (depending on M) that would guarantee that no pairs are marked after the first pass, so that the algorithm terminates after two passes?

2.57. Each case below defines a language over $\{a, b\}$. In each case, decide whether the language can be accepted by an FA, and prove that your answer is correct.

a. The set of all strings x beginning with a nonnull string of the form ww.

b. The set of all strings x containing some nonnull substring of the form ww.

c. The set of all strings x having some nonnull substring of the form www. (You may assume the following fact: there are arbitrarily long strings in $\{a, b\}^*$ that do not contain any nonnull substring of the form www.)

d. The set of odd-length strings with middle symbol a.

e. The set of even-length strings of length at least 2 with the two middle symbols equal.

f. The set of strings of the form xyx for some x with $|x| \geq 1$.

g. The set of non-palindromes.

h. The set of strings in which the number of a's is a perfect square.

i. The set of strings having the property that in every prefix, the number of a's and the number of b's differ by no more than 2.

j. The set of strings having the property that in some prefix, the number of a's is 3 more than the number of b's.

k. The set of strings in which the number of a's and the number of b's are both divisible by 5.

l. The set of strings x for which there is an integer $k > 1$ (possibly depending on x) such that the number of a's in x and the number of b's in x are both divisible by k.

m. (Assuming that L can be accepted by an FA), $Max(L) = \{x \in L \mid$ there is no nonnull string y so that $xy \in L\}$.

n. (Assuming that L can be accepted by an FA), $Min(L) = \{x \in L \mid$ no prefix of x other than x itself is in $L\}$.

2.58. Find an example of a language $L \subseteq \{a, b\}^*$ such that L^* cannot be accepted by an FA.

2.59. Find an example of a language L over $\{a, b\}$ such that L cannot be accepted by an FA but L^* can.

2.60. Find an example of a language L over $\{a, b\}$ such that L cannot be accepted by an FA but LL can.

2.61. †Show that if L is any language over a one-symbol alphabet, then L^* can be accepted by an FA.

2.62. †Consider the two FAs in Fig. 2.46.
If you examine them closely you can see that they are really identical, except that the states have different names: state p corresponds to state A, q corresponds to B, and r corresponds to C. Let us describe this correspondence by the "relabeling function" i; that is, $i(p) = A$, $i(q) = B$, $i(r) = C$. What does it mean to say that under this correspondence, the two FAs are "really identical"? It means several things: First, the initial states correspond to each other; second, a state is an accepting state if and only if the corresponding state is; and finally, the transitions among the states of the first FA are the same as those among the corresponding states of the other. For example, if δ_1 and δ_2 are the transition functions, then

$$\delta_1(p, a) = p \text{ and } \delta_2(i(p), a) = i(p)$$
$$\delta_1(p, b) = q \text{ and } \delta_2(i(p), b) = i(q)$$

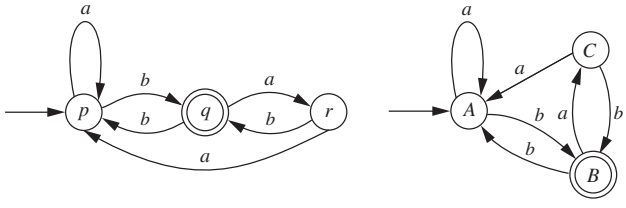


Figure 2.46 |

These formulas can be rewritten

$$\delta_2(i(p), a) = i(\delta_1(p, a)) \text{ and } \delta_2(i(p), b) = i(\delta_1(p, b))$$

and these and all the other relevant formulas can be summarized by the general formula

$$\delta_2(i(s), \sigma) = i(\delta_1(s, \sigma)) \text{ for every state } s \text{ and alphabet symbol } \sigma$$

In general, if $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ are FAs, and $i : Q_1 \to Q_2$ is a bijection (i.e., one-to-one and onto), we say that i is an *isomorphism from M_1 to M_2* if these conditions are satisfied:

i. $i(q_1) = q_2$

ii. for every $q \in Q_1$, $i(q) \in A_2$ if and only if $q \in A_1$

iii. for every $q \in Q_1$ and every $\sigma \in \Sigma$, $i(\delta_1(q, \sigma)) = \delta_2(i(q), \sigma)$

and we say M_1 is *isomorphic to M_2* if there is an isomorphism from M_1 to M_2. This is simply a precise way of saying that M_1 and M_2 are "essentially the same".

a. Show that the relation $\sim$ on the set of FAs over Σ, defined by $M_1 \sim M_2$ if M_1 is isomorphic to M_2, is an equivalence relation.

b. Show that if i is an isomorphism from M_1 to M_2 (notation as above), then for every $q \in Q_1$ and $x \in \Sigma^*$,

$$i(\delta_1^*(q, x)) = \delta_2^*(i(q), x)$$

c. Show that two isomorphic FAs accept the same language.

d. How many one-state FAs over the alphabet $\{a, b\}$ are there, no two of which are isomorphic?

e. How many pairwise nonisomorphic two-state FAs over $\{a, b\}$ are there, in which both states are reachable from the initial state and at least one state is accepting?

f. How many distinct languages are accepted by the FAs in the previous part?

g. Show that the FAs described by these two transition tables are isomorphic. The states are 1–6 in the first, A–F in the second; the initial states are 1 and A, respectively; the accepting states in the first FA are 5 and 6, and D and E in the second.

q	$\delta_1(q, a)$	$\delta_1(q, b)$
1	3	5
2	4	2
3	1	6
4	4	3
5	2	4
6	3	4

q	$\delta_2(q, a)$	$\delta_2(q, b)$
A	B	E
B	A	D
C	C	B
D	B	C
E	F	C
F	C	F

2.63. Suppose that $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ are both FAs accepting the language L, and that both have as few states as possible. Show that M_1 and M_2 are isomorphic (see Exercise 2.62). Note that in both cases, the sets L_q forming the partition of Σ^* are precisely the equivalence classes of I_L. This tells you how to come up with a bijection from Q_1 to Q_2. What you must do next is to show that the other conditions of an isomorphism are satisfied.

2.64. Use Exercise 2.63 to describe another decision algorithm to answer the question "Given two FAs, do they accept the same language?"

CHAPTER

3

Regular Expressions, Nondeterminism, and Kleene's Theorem

A simple way of describing a language is to describe a finite automaton that accepts it. As with the models of computation we will study later, an alternative approach is to use some appropriate notation to describe how the strings of the language can be generated. Languages that can be accepted by finite automata are the same as *regular* languages, which can be represented by formulas called regular expressions involving the operations of union, concatenation, and Kleene star. In the case of finite automata, demonstrating this equivalence (by proving the two parts of Kleene's theorem) is simplified considerably by introducing *nondeterminism*, which will also play a part in the computational models we will study later. Here, although allowing nondeterminism seems at first to enhance the accepting power of these devices, we will see that it can be eliminated.

3.1 REGULAR LANGUAGES AND REGULAR EXPRESSIONS

Three of the languages over $\{a, b\}$ that we considered in Chapter 2 are L_1, the language of strings ending in aa; L_2, the language of strings containing either the substring ab or the substring bba; and L_3, the language $\{aa, aab\}^*\{b\}$. Like L_3, both L_1 and L_2 can be expressed by a formula involving the operations of union, concatenation, and Kleene *: L_1 is $\{a, b\}^*\{aa\}$ and L_2 is $\{a, b\}^*(\{ab\} \cup \{bba\})\{a, b\}^*$. Languages that have formulas like these are called *regular* languages. In this section we give a recursive definition of the set of regular languages over an alphabet Σ, and later in this chapter we show that these are precisely the languages that can be accepted by a finite automaton.

Definition 3.1 Regular Languages over an Alphabet Σ

If Σ is an alphabet, the set $\mathcal{R}$ of regular languages over Σ is defined as follows.

1. The language $\emptyset$ is an element of $\mathcal{R}$, and for every $a \in \Sigma$, the language $\{a\}$ is in $\mathcal{R}$.
2. For any two languages L_1 and L_2 in $\mathcal{R}$, the three languages

$$L_1 \cup L_2, \quad L_1 L_2, \quad \text{and} \quad L_1^*$$

are elements of $\mathcal{R}$.

The language $\{\Lambda\}$ is a regular language over Σ, because $\emptyset^* = \{\Lambda\}$. If $\Sigma = \{a, b\}$, then $L_1 = \{a, b\}^*\{aa\}$ can be obtained from the definition by starting with the two languages $\{a\}$ and $\{b\}$ and then using the recursive statement in the definition four times: The language $\{a, b\}$ is the union $\{a\} \cup \{b\}$; $\{aa\}$ is the concatenation $\{a\}\{a\}$; $\{a, b\}^*$ is obtained by applying the Kleene star operation to $\{a, b\}$; and the final language is the concatenation of $\{a, b\}^*$ and $\{aa\}$.

A regular language over Σ has an explicit formula. A *regular expression* for the language is a slightly more user-friendly formula. The only differences are that in a regular expression, parentheses replace {} and are omitted whenever the rules of precedence allow it, and the union symbol $\cup$ is replaced by $+$. Here are a few examples (see Example 3.5 for a discussion of the last one):

Regular Language	***Corresponding Regular Expression***
$\emptyset$	$\emptyset$
$\{\Lambda\}$	Λ
$\{a, b\}^*$	$(a + b)^*$
$\{aab\}^*\{a, ab\}$	$(aab)^*(a + ab)$
$(\{aa, bb\} \cup \{ab, ba\}\{aa, bb\}^*\{ab, ba\})^*$	$(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$

When we write a regular expression like Λ or aab, which contains neither $+$ nor * and corresponds to a one-element language, the regular expression looks just like the string it represents. A more general regular expression involving one or both of these operations can't be mistaken for a string; we can think of it as representing the general form of strings in the language. A regular expression describes a regular language, and a regular language can be described by a regular expression.

We say that two regular expressions are equal if the languages they describe are equal. Some regular-expression identities are more obvious than others. The formula

$$(a^*b^*)^* = (a + b)^*$$

is true because the language corresponding to a^*b^* contains both a and b. The formula

$$(a + b)^*ab(a + b)^* + b^*a^* = (a + b)^*$$

is true because the first term on the left side corresponds to the strings in $\{a, b\}^*$ that contain the substring ab and the second term, b^*a^*, corresponds to the strings that don't.

EXAMPLE 3.2 The Language of Strings in $\{a, b\}^*$ with an Odd Number of a's

A string with an odd number of a's has at least one a, and the additional a's can be grouped into pairs. There can be arbitrarily many b's before the first a, between any two consecutive a's, and after the last a. The expression

$$b^*ab^*(ab^*a)^*b^*$$

is not correct, because it doesn't allow b's between the second a in one of the repeating pairs ab^*a and the first a in the next pair. One correct regular expression describing the language is

$$b^*ab^*(ab^*ab^*)^*$$

The expression

$$b^*a(b^*ab^*ab^*)^*$$

is also not correct, because it doesn't allow strings with just one a to end with b, and the expression

$$b^*a(b^*ab^*a)^*b^*$$

corrects the mistake. Another correct expression is

$$b^*a(b + ab^*a)^*$$

All of these could also be written with the single a on the right, as in

$$(b + ab^*a)^*ab^*$$

EXAMPLE 3.3 The Language of Strings in $\{a, b\}^*$ Ending with b and Not Containing aa

If a string does not contain the substring aa, then every a in the string either is followed immediately by b or is the last symbol in the string. If the string ends with b, then every a is followed immediately by b. Therefore, every string in the language L of strings that end with b and do not contain aa matches the regular expression $(b + ab)^*$. This regular expression does not describe L, however, because it allows the null string, which does not end with b. At least one of the two strings b and ab must occur, and so a regular expression for L is

$$(b + ab)^*(b + ab)$$

Strings in {*a*,*b*}* in Which Both the Number of *a*'s and the Number of *b*'s Are Even

EXAMPLE 3.4

One of the regular expressions given in Example 3.2, $b^*a(b + ab^*a)^*$, describes the language of strings with an odd number of a's, and the final portion of it, $(b + ab^*a)^*$, describes the language of strings with an even number of a's. We can interpret the two terms inside the parentheses as representing the two possible ways of adding to the string without changing the parity (the evenness or oddness) of the number of a's: adding a string that has no a's, and adding a string that has two a's. Every string x with an even number of a's has a prefix matching one of these two terms, and x can be decomposed into nonoverlapping substrings that match one of these terms.

Let L be the subset of $\{a, b\}^*$ containing the strings x for which both $n_a(x)$ and $n_b(x)$ are even. Every element of L has even length. We can use the same approach to find a regular expression for L, but this time it's sufficient to consider substrings of even length. The easiest way to add a string of even length without changing the parity of the number of a's or the number of b's is to add aa or bb. If a nonnull string $x \in L$ does not begin with one of these, then it starts with either ab or ba, and the shortest substring following this that restores the evenness of n_a and n_b must also end with ab or ba, because its length is even and strings of the form aa and bb don't change the parity of n_a or n_b.

The conclusion is that every nonnull string in L has a prefix that matches the regular expression

$$aa + bb + (ab + ba)(aa + bb)^*(ab + ba)$$

and that a regular expression for L is

$$(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$$

Regular Expressions and Programming Languages

EXAMPLE 3.5

In Example 2.9 we built a finite automaton to carry out a very simple version of lexical analysis: breaking up a part of a computer program into tokens, which are the basic building blocks from which the expressions or statements are constructed. The last two sections of this chapter are devoted to proving that finite automata can accept exactly the same languages that regular expressions can describe, and in this example we construct regular expressions for two classes of tokens.

An identifier in the C programming language is a string of length 1 or more that contains only letters, digits, and underscores ("_") and does not begin with a digit. If we use the abbreviations l for "letter," either uppercase or lowercase, and d for "digit," then l stands for the regular expression

$$\mathrm{a} + \mathrm{b} + \mathrm{c} + \ldots + \mathrm{z} + \mathrm{A} + \mathrm{B} + \ldots + \mathrm{Z}$$

and d for the regular expression

$$0 + 1 + 2 + \cdots + 9$$

(which has nothing to do with the integer 45), and a regular expression for the language of C identifiers is

$$(l + \_)(l + d + \_)^*$$

Next we look for a regular expression to describe the language of numeric "literals," which typically includes strings such as 14, +1, −12, 14.3, −.99, 16., 3E14, −1.00E2, 4.1E−1, and .3E+2. Let us assume that such an expression may or may not begin with a plus sign or a minus sign; it will contain one or more decimal digits, and possibly a decimal point, and it may or may not end with a subexpression starting with E. If there is such a subexpression, the portion after E may or may not begin with a sign and will contain one or more decimal digits.

Our regular expression will involve the abbreviations d and l introduced above, and we will use s to stand for "sign" (either Λ or a plus sign or a minus sign) and p for a decimal point. It is not hard to convince yourself that a regular expression covering all the possibilities is

$$s(dd^*(\Lambda + pd^*) + pdd^*)(\Lambda + \mathrm{E}sdd^*)$$

In some programming languages, numeric literals are not allowed to contain a decimal point unless there is at least one digit on both sides. A regular expression incorporating this requirement is

$$sdd^*(\Lambda + pdd^*)(\Lambda + \mathrm{E}sdd^*)$$

Other tokens in a high-level language can also be described by regular expressions, in most cases even simpler than the ones in this example. Lexical analysis is the first phase in compiling a high-level-language program. There are programs called lexical-analyzer generators; the input provided to such a program is a set of regular expressions describing the structure of tokens, and the output produced by the program is a software version of an FA that can be incorporated as a token-recognizing module in a compiler. One of the most widely used lexical-analyzer generators is `lex`, a tool provided in the Unix operating system. It can be used in many situations that require the processing of structured input, but it is often used together with `yacc`, another Unix tool. The lexical analyzer produced by `lex` creates a string of tokens; and the *parser* produced by `yacc`, on the basis of grammar rules provided as input, is able to determine the syntactic structure of the token string. (`yacc` stands for *yet another compiler compiler*.)

Regular expressions come up in Unix in other ways as well. The Unix text editor allows the user to specify a regular expression and searches for patterns in the text that match it. Other commands such as `grep` (global regular expression print) and `egrep` (extended global regular expression print) allow a user to search a file for strings that match a specified regular expression.

3.2 NONDETERMINISTIC FINITE AUTOMATA

The goal in the rest this chapter is to prove that regular languages, defined in Section 3.1, are precisely the languages accepted by finite automata. In order to do this, we will introduce a more general "device," a *nondeterministic* finite automaton. The advantage of this approach is that it's much easier to start with an arbitrary regular expression and draw a transition diagram for something

that accepts the corresponding language and has an obvious connection to the regular expression. The only problem is that the *something* might not be a finite automaton, although it has a superficial resemblance to one, and we have to figure out how to interpret it in order to think of it as a physical device at all.

Accepting the Language $\{aa,aab\}^*\{b\}$) **EXAMPLE 3.6**

There is a close resemblance between the diagram in Figure 3.7 and the regular expression $(aa + aab)^*b$. The top loop corresponds to aa, the bottom one corresponds to aab, and the remaining b-transition corresponds to the last b in the regular expression. To the extent that we think of it as a transition diagram like that of an FA, its resemblance to the regular expression suggests that the string *aaaabaab*, for example, should be accepted, because it allows us to start at q_0, take the top loop once, the bottom loop once, the top loop again, and finish up with the transition to the accepting state.

This diagram, however, is not the transition diagram for an FA, because there are three transitions from q_0 and fewer than two from several other states. The input string *aaaabaab allows* us to reach the accepting state, but it also allows us to follow, or at least start to follow, other paths that don't result in acceptance. We can imagine an idealized "device" that would work by using the input symbols to follow paths shown in the diagram, making arbitrary choices at certain points. It has to be *nondeterministic*, in the sense that the path it follows is not determined by the input string. (If the first input symbol is a, it chooses whether to start up the top path or down the bottom one.) Even if there were a way to build a physical device that acted like this, it wouldn't accept this language in the same way that an FA could. Suppose we watched it process a string x. If it ended up in the accepting state at the end, we could say that x was in the language; if it didn't, all we could say for sure is that the moves it chose to make did not lead to the accepting state.

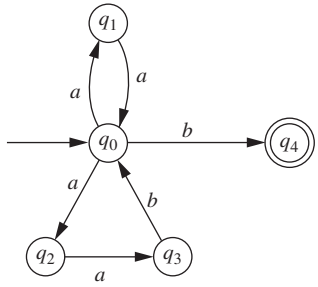


Figure 3.7 |
Using nondeterminism to accept $\{aa,aab\}^*\{b\}$.

Allowing nondeterminism, by relaxing the rules for an FA, makes it easy to draw diagrams corresponding to regular expressions. However, we should no longer think of the diagram as representing an explicit algorithm for accepting the language, because an algorithm refers to a sequence of steps that are determined by the input and would be the same no matter how many times the algorithm was executed on the same input.

If the diagram doesn't represent an explicit accepting algorithm, what good is it? One way to answer this is to think of the diagram as describing a number of different sequences of steps that might be followed. We can visualize these sequences for the input string *aaaabaab* by drawing a *computation tree*, pictured in Figure 3.8.

A level of the tree corresponds to the input (the prefix of the entire input string) read so far, and the states appearing on this level are those in which the device *could* be, depending on the choices it has made so far. Two paths in the tree, such as the one that

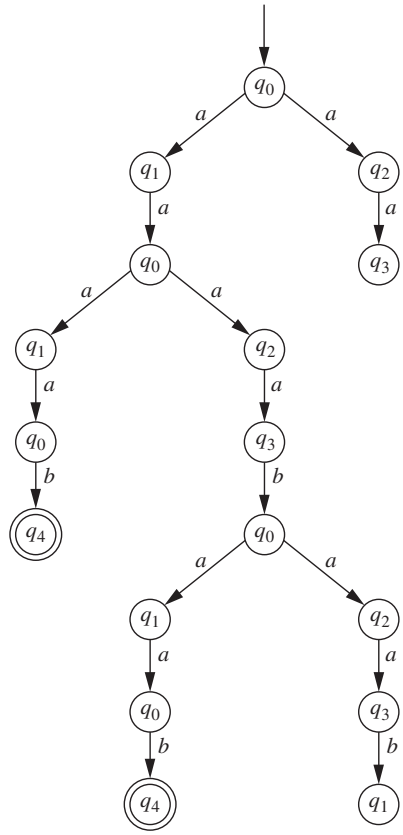


Figure 3.8 |
The computation tree for Figure 3.7 and the input string *aaaabaab*.

starts by treating the initial a as the first symbol of aab, terminate prematurely, because the next input symbol does not allow a move from the current state. One path, which corresponds to interpreting the input string as $(aa)(aab)(aab)$, allows all the input to be read and ends up in a nonaccepting state. The path in which the device makes the "correct" choice at each step ends up at the accepting state when all the input symbols have been read.

If we had the transition diagram in Figure 3.7 and were trying to use it to accept the language, we could systematically keep track of the current sequence of steps, and use a backtracking strategy whenever we couldn't proceed any further or finished in a nonaccepting state. The result would, in effect, be to search the computation tree using a depth-first search. In the next section we will see how to develop an ordinary finite automaton that effectively executes a breadth-first search of the tree, by keeping track after each input symbol of all the possible states the various sequences of steps could have led us to.

Accepting the Language $\{aab\}^*\{a,aba\}^*$ **EXAMPLE 3.9**

In this example we consider the regular expression $(aab)^*(a + aba)^*$. The techniques of Example 3.6 don't provide a simple way to draw a transition diagram related to the regular expression, but Figure 3.10 illustrates another type of nondeterminism that does.

The new feature is a "Λ-transition," which allows the device to change state with no input. If the input a is received in state 0, there are three options: take the transition from state 0 corresponding to the a in aab; move to state 3 and take the transition corresponding to a; and move to state 3 and take the transition corresponding to the a in aba. The diagram shows two a-transitions from state 3, but because of the Λ-transition, we would have a choice of moves even if there were only one.

Figure 3.11 shows a computation tree illustrating the possible sequences of moves for the input string $aababa$. The Λ-transition is drawn as a horizontal arrow, so that as in the previous example, a new level of the tree corresponds to a new input symbol.

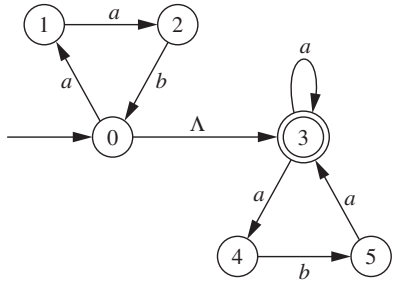


Figure 3.10 |
Using nondeterminism to accept $\{aab\}^*\{a, aba\}^*$.

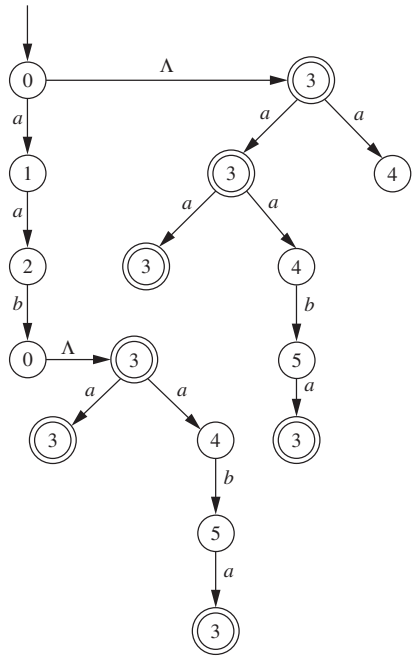


Figure 3.11 |
The computation tree for Figure 3.10 and the input string *aababa*.

The string is accepted, because the device can choose to take the first loop, execute the Λ-transition, and take the longer loop from state 3.

The transition diagrams in our first two examples show four of the five ingredients of an ordinary finite automaton. The one that must be handled differently is the transition function δ. For a state q and an alphabet symbol σ, it is no longer correct to say that $\delta(q, \sigma)$ is a state: There may be no transitions from state q on input σ, or one, or more than one. There may also be Λ-transitions. We can incorporate both of these features by making two changes: first, enlarging the domain of δ to include ordered pairs (q, Λ) as well as the pairs in $Q \times \Sigma$; and second, making the values of δ *sets* of states instead of individual states.

Definition 3.12 A Nondeterministic Finite Automaton

A *nondeterministic finite automaton* (NFA) is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$, where

Q is a finite set of states;
Σ is a finite input alphabet;

$q_0 \in Q$ is the initial state;
$A \subseteq Q$ is the set of accepting states;
$\delta : Q \times (\Sigma \cup \{\Lambda\}) \to 2^Q$ is the transition function.

For every element q of Q and every element σ of $\Sigma \cup \{\Lambda\}$, we interpret $\delta(q, \sigma)$ as the set of states to which the FA can move, if it is in state q and receives the input σ, or, if $\sigma = \Lambda$, the set of states other than q to which the NFA can move from state q without receiving any input symbol.

In Example 3.9, for example, $\delta(0, a) = \{1\}$, $\delta(0, \Lambda) = \{3\}$, $\delta(0, b) = \emptyset$, and $\delta(0, a) = \{3, 4\}$.

In the case of an NFA $M = (Q, \Sigma, q_0, A, \delta)$, we want $\delta^*(q, x)$ to tell us all the states M can get to by starting at q and using the symbols in the string x. We can still define the function δ^* recursively, but the mathematical notation required to express this precisely is a little more involved, particularly if M has Λ-transitions.

In order to define $\delta^*(q, x\sigma)$, where $x \in \Sigma^*$ and $\sigma \in \Sigma$, we start by considering $\delta^*(q, x)$, just as in the simple case of an ordinary FA. This is now a set of states, and for each state p in this set, $\delta(p, \sigma)$ is itself a set. In order to include all the possibilities, we need to consider

$$\bigcup \{\delta(p, a) \mid p \in \delta^*(q, x)\}$$

Finally, we must keep in mind that in the case of Λ-transitions, "using all the symbols in the string x" really means using all the symbols in x and perhaps Λ-transitions where they are possible. In the recursive step of the definition of δ^*, once we have the union we have just described, we must consider all the additional states we might be able to reach from elements of this union, using nothing but Λ-transitions.

You can probably see at this point how the following definition will be helpful in our discussion.

Definition 3.13 The Λ-Closure of a Set of States

Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an NFA, and $S \subseteq Q$ is a set of states. The Λ-closure of S is the set $\Lambda(S)$ that can be defined recursively as follows.

1. $S \subseteq \Lambda(S)$.
2. For every $q \in \Lambda(S)$, $\delta(q, \Lambda) \subseteq \Lambda(S)$.

In exactly the same way as in Example 1.21, we can convert the recursive definition of $\Lambda(S)$ into an algorithm for evaluating it, as follows.

Algorithm to Calculate $\Lambda(S)$ Initialize T to be S. Make a sequence of passes, in each pass considering every $q \in T$ and adding to T every state in $\delta(q, \Lambda)$ that is not already an element. Stop after the first pass in which T is not changed. The final value of T is $\Lambda(S)$. ■

A state is in $\Lambda(S)$ if it is an element of S or can be reached from an element of S using one or more Λ-transitions.

With the help of Definition 3.13 we can now define the extended transition function δ^* for a nondeterministic finite automaton.

Definition 3.14 The Extended Transition Function δ^* for an NFA, and the Definition of Acceptance

Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA. We define the extended transition function

$$\delta^* : Q \times \Sigma^* \to 2^Q$$

as follows:

1. For every $q \in Q$, $\delta^*(q, \Lambda) = \Lambda(\{q\})$.
2. For every $q \in Q$, every $y \in \Sigma^*$, and every $\sigma \in \Sigma$,

$$\delta^*(q, y\sigma) = \Lambda\left(\bigcup\{\delta(p, \sigma) \mid p \in \delta^*(q, y)\}\right)$$

A string $x \in \Sigma^*$ is accepted by M if $\delta^*(q_0, x) \cap A \neq \emptyset$. The language $L(M)$ accepted by M is the set of all strings accepted by M.

For the NFA in Example 3.9, which has only one Λ-transition, it is easy to evaluate $\delta^*(aababa)$ by looking at the computation tree in Figure 3.11. The two states on the first level of the diagram are 0 and 3, the elements of the Λ-closure of $\{0\}$. The states on the third level, for example, are 2, 3, and 4, because $\delta^*(0, aa) = \{2, 3, 4\}$. When we apply the recursive part of the definition to evaluate $\delta^*(0, aab)$, we first evaluate

$$\bigcup\{\delta(p, b) \mid p \in \{2, 3, 4\}\} = \delta(2, b) \cup \delta(3, b) \cup \delta(4, b) = \{0\} \cup \emptyset \cup \{5\}$$
$$= \{0, 5\}$$

and then we compute the Λ-closure of this set, which contains the additional element 3.

For an NFA M with no Λ-transitions, both statements in the definition can be simplified, because for every subset S of Q, $\Lambda(S) = S$.

We illustrate the definition once more in a slightly more extended example.

EXAMPLE 3.15 Applying the Definitions of $\Lambda(S)$ and δ^*

We start by evaluating the Λ-closure of the set $\{v\}$ in the NFA whose transition diagram is shown in Figure 3.16. When we apply the algorithm derived from Definition 3.13, after one

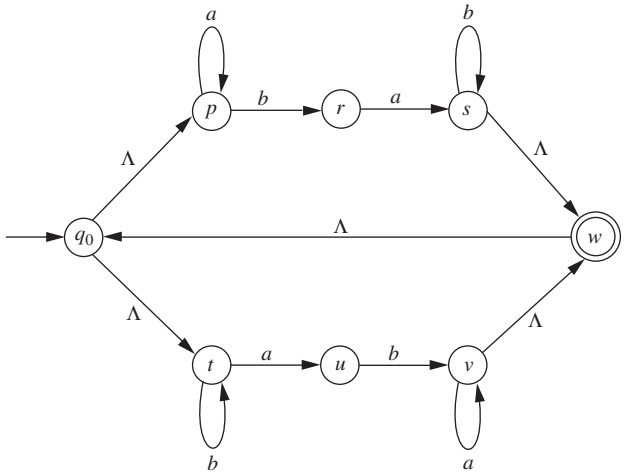


Figure 3.16 |
Evaluating the extended transition function when there are Λ-transitions.

pass T is $\{v, w\}$, after two passes it is $\{v, w, q_0\}$, after three passes it is $\{v, w, q_0, p, t\}$, and during the next pass it remains unchanged. The set $\Lambda(\{s\})$ is therefore $\{v, w, q_0, p, t\}$.

If we want to apply the definition of δ^* to evaluate $\delta^*(q_0, aba)$, the easiest way is to begin with Λ, the shortest prefix of aba, and work our way up one symbol at a time.

$$
\begin{aligned}
\delta^*(q_0, \Lambda) &= \Lambda(\{q_0\}) \\
&= \{q_0, p, t\} \\
\delta^*(q_0, a) &= \Lambda\left(\bigcup\{\delta(k, a) \mid k \in \delta^*(q_0, \Lambda)\}\right) \\
&= \Lambda\left(\delta(q_0, a) \cup \delta(p, a) \cup \delta(t, a)\right) \\
&= \Lambda\left(\emptyset \cup \{p\} \cup \{u\}\right) \\
&= \Lambda(\{p, u\}) \\
&= \{p, u\} \\
\delta^*(q_0, ab) &= \Lambda\left(\bigcup\{\delta(k, b) \mid k \in \{p, u\}\}\right) \\
&= \Lambda(\delta(p, b) \cup \delta(u, b)) \\
&= \Lambda(\{r, v\}) \\
&= \{r, v, w, q_0, p, t\} \\
\delta^*(q_0, aba) &= \Lambda\left(\bigcup\{\delta(k, a) \mid k \in \{r, v, w, q_0, p, t\}\}\right) \\
&= \Lambda(\delta(r, a) \cup \delta(v, a) \cup \delta(w, a) \cup \delta(q_0, a) \cup \delta(p, a) \cup \delta(t, a)) \\
&= \Lambda(\{s\} \cup \{v\} \cup \emptyset \cup \emptyset \cup \{p\} \cup \{u\}) \\
&= \Lambda(\{s, v, p, u\}) \\
&= \{s, v, p, u, w, q_0, t\}
\end{aligned}
$$

The evaluation of $\Lambda(\{r, v\})$ is very similar to that of $\Lambda(\{v\})$, since there are no Λ-transitions from r, and the evaluation of $\Lambda(\{s, v, p, u\})$ is also similar. Because $\delta^*(q_0, aba)$ contains the accepting state w, the string aba is accepted.

A state r is an element of $\delta^*(q, x)$ if in the transition diagram there is a path from q to r, in which there are transitions for every symbol in x and the next transition at each step corresponds either to the next symbol in x or to Λ. In simple examples, including this one, you may feel that it's easier to evaluate δ^* by looking at the diagram and determining by inspection what states you can get to. One reason for having a precise recursive definition of δ^* and a systematic algorithm for evaluating it is that otherwise it's easy to overlook things.

3.3 THE NONDETERMINISM IN AN NFA CAN BE ELIMINATED

We have observed nondeterminism in two slightly different forms in our discussion of NFAs. It is most apparent if there is a state q and an alphabet symbol σ such that several different transitions are possible in state q on input σ. A choice of moves can also occur as a result of Λ-transitions, because there may be states from which the NFA can make either a transition on an input symbol or one on no input.

We will see in this section that both types of nondeterminism can be eliminated. The idea in the second case is to introduce new transitions so that we no longer need Λ-transitions: In every case where there is no σ-transition from p to q but the NFA can go from p to q by using one or more Λ's as well as σ, we will introduce the σ-transition. The resulting NFA may have even more nondeterminism of the first type than before, but it will be able to accept the same strings without using Λ-transitions.

The way we eliminate nondeterminism from an NFA having no Λ-transitions is simply to define it away, by finding an appropriate definition of *state*. We have used this technique twice before, in Section 2.2 when we considered states that were ordered pairs, and in Section 2.5 when we defined a state to be a set of strings. Here a similar approach is already suggested by the way we define the transition function of an NFA, whose value is a set of states. If we say that for an element p of a set $S \subseteq Q$, the transition on input σ can possibly go to several states, it sounds like nondeterminism; if we say that starting with an element of the set S, the set of states to which we can go on input σ is

$$\bigcup \{\delta(p, \sigma) \mid p \in S\}$$

and if both S and this set qualify as states in our new definition, then it sounds as though we have eliminated the nondeterminism. The only question then is whether the FA we obtain accepts the same strings as the NFA we started with.

Theorem 3.17
For every language $L \subseteq \Sigma^*$ accepted by an NFA $M = (Q, \Sigma, q_0, A, \delta)$, there is an NFA M_1 with no Λ-transitions that also accepts L.

Proof

As we have already mentioned, we may need to add transitions in order to guarantee that the same strings will be accepted even when the Λ-transitions are eliminated. In addition, if $q_0 \notin A$ but $\Lambda \in L$, we will also make q_0 an accepting state of M_1 in order to guarantee that M_1 accepts Λ.

We define

$$M_1 = (Q, \Sigma, q_0, A_1, \delta_1)$$

where for every $q \in Q$, $\delta_1(q, \Lambda) = \emptyset$, and for every $q \in Q$ and every $\sigma \in \Sigma$,

$$\delta_1(q, \sigma) = \delta^*(q, \sigma)$$

Finally, we define

$$A_1 = \begin{cases} A \cup \{q_0\} & \text{if } \Lambda \in L \\ A & \text{if not} \end{cases}$$

For every state q and every $x \in \Sigma^*$, the way we have defined the extended transition function δ^* for the NFA M tells us that $\delta^*(q, x)$ is the set of states M can reach by using the symbols of x together with Λ-transitions. The point of our definition of δ_1 is that we want $\delta_1^*(q, x)$ to be the same set, even though M_1 has no Λ-transitions. This may not be true for $x = \Lambda$, because $\delta^*(q, \Lambda) = \Lambda(\{q\})$ and $\delta_1(q, \Lambda) = \{q\}$; this is the reason for the definition of A_1 above. We sketch the proof that for every q and every x with $|x| \geq 1$,

$$\delta_1^*(q, x) = \delta^*(q, x)$$

The proof is by structural induction on x. If $x = a \in \Sigma$, then by definition of δ_1, $\delta_1(q, x) = \delta^*(q, x)$, and because M_1 has no Λ-transitions, $\delta_1(q, x) = \delta_1^*(q, x)$ (see Exercise 3.24).

Suppose that for some y with $|y| \geq 1$, $\delta_1^*(q, y) = \delta^*(q, y)$ for every state q, and let σ be an arbitrary element of Σ.

$$\begin{aligned} \delta_1^*(q, y\sigma) &= \bigcup \{\delta_1(p, \sigma) \mid p \in \delta_1^*(q, y)\} \\ &= \bigcup \{\delta_1(p, \sigma) \mid p \in \delta^*(q, y)\} \text{ (by the induction hypothesis)} \\ &= \bigcup \{\delta^*(p, \sigma) \mid p \in \delta^*(q, y)\} \text{ (by definition of } \delta_1) \end{aligned}$$

The last step in the induction proof is to check that this last expression is indeed $\delta^*(q, y\sigma)$. This is a special case of the general formula

$$\delta^*(q, yz) = \bigcup \{\delta^*(p, z) \mid p \in \delta^*(q, y)\}$$

See Exercise 3.30 for the details.

Now we can verify that $L(M_1) = L(M) = L$. If the string Λ is accepted by M, then it is accepted by M_1, because in this case $q_0 \in A_1$ by definition. If $\Lambda \notin L(M)$, then $A = A_1$; therefore, $q_0 \notin A_1$, and $\Lambda \notin L(M_1)$.

Suppose that $|x| \geq 1$. If $x \in L(M)$, then $\delta^*(q_0, x)$ contains an element of A; therefore, since $\delta^*(q_0, x) = \delta_1^*(q_0, x)$ and $A \subseteq A_1$, $x \in L(M_1)$.

Now suppose $|x| \geq 1$ and $x \in L(M_1)$. Then $\delta_1^*(q_0, x)$ contains an element of A_1. The state q_0 is in A_1 only if $\Lambda \in L$; therefore, if $\delta_1^*(q_0, x)$ (which is the same as $\delta^*(q_0, x)$) contains q_0, it also contains every element of A in $\Lambda(\{q_0\})$. In any case, if $x \in L(M_1)$, then $\delta_1^*(q_0, x)$ must contain an element of A, which implies that $x \in L(M)$.

Theorem 3.18

For every language $L \subseteq \Sigma^*$ accepted by an NFA $M = (Q, \Sigma, q_0, A, \delta)$, there is an FA $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ that also accepts L.

Proof

Because of Theorem 3.17, it is sufficient to prove the theorem in the case when M has no Λ-transitions. The formulas defining δ^* are simplified accordingly: $\delta^*(q, \Lambda) = \{q\}$ and $\delta^*(q, x\sigma) = \cup\{\delta(p, \sigma) \mid p \in \delta^*(q, x)\}$.

The finite automaton M_1 can be defined as follows, using the *subset construction*: The states of M_1 are sets of states of M, or

$$Q_1 = 2^Q$$

The initial state q_1 of Q_1 is $\{q_0\}$. For every $q \in Q_1$ and every $\sigma \in \Sigma$,

$$\delta_1(q, \sigma) = \bigcup\{\delta(p, \sigma) \mid p \in q\}$$

and the accepting states of M_1 are defined by the formula

$$A_1 = \{q \in Q_1 \mid q \cap A \neq \emptyset\}$$

The last definition is the correct one, because a string x should be accepted by M_1 if, when the NFA M processes x, there is at least one state it might end up in that is an element of A.

There is no doubt that M_1 is an ordinary finite automaton. The expression $\delta_1^*(q_1, x)$, however, is a set of states of M—not because M_1 is nondeterministic, but because we have defined states of M_1 to be sets of states of M. The fact that the two devices accept the same language follows from the fact that for every $x \in \Sigma^*$,

$$\delta_1^*(q_1, x) = \delta^*(q_0, x)$$

and we now prove this formula using structural induction on x. We must keep in mind during the proof that δ_1^* and δ^* are defined in different ways, because M_1 is an FA and M is an NFA.

If $x = \Lambda$, then

$$\begin{aligned}\delta_1^*(q_1, x) &= \delta_1^*(q_1, \Lambda) \\ &= q_1 \text{ (by the definition of } \delta_1^*)\end{aligned}$$

$$= \{q_0\} \text{ (by the definition of } q_1)$$
$$= \delta^*(q_0, \Lambda) \text{ (by the definition of } \delta^*)$$
$$= \delta^*(q_0, x)$$

The induction hypothesis is that x is a string for which $\delta_1^*(q_1, x) = \delta^*(q_0, x)$, and we must show that for every $\sigma \in \Sigma$, $\delta_1^*(q_1, x\sigma) = \delta^*(q_0, x\sigma)$.

$$\begin{aligned}\delta_1^*(q_1, x\sigma) &= \delta_1(\delta_1^*(q_1, x), \sigma) \text{ (by the definition of } \delta_1^*)\\ &= \delta_1(\delta^*(q_0, x), \sigma) \text{ (by the induction hypothesis)}\\ &= \bigcup\{\delta(p, \sigma) \mid p \in \delta^*(q_0, x)\} \text{ (by the definition of } \delta_1)\\ &= \delta^*(q_0, x\sigma) \text{ (by the definition of } \delta^*)\end{aligned}$$

A string x is accepted by M_1 precisely if $\delta_1^*(q_1, x) \in A_1$. We know now that this is true if and only if $\delta^*(q_0, x) \in A_1$; and according to the definition of A_1, this is true if and only if $\delta^*(q_0, x) \cap A \neq \emptyset$. Therefore, x is accepted by M_1 if and only if x is accepted by M.

We present three examples: one that illustrates the construction in Theorem 3.17, one that illustrates the subset construction in Theorem 3.18, and one in which we use both to convert an NFA with Λ-transitions to an ordinary FA.

Eliminating Λ-Transitions from an NFA

EXAMPLE 3.19

Figure 3.20a shows the transition diagram for an NFA M with Λ-transitions; it is not hard to see that it accepts the language corresponding to the regular expression $(a^*ab(ba)^*)^*$. We show in tabular form the values of the transition function δ, as well as the values $\delta^*(q, a)$ and $\delta^*(q, b)$ that will give us the transition function δ_1 in the resulting NFA M_1.

q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, \Lambda)$	$\delta^*(q, a)$	$\delta^*(q, b)$
1	$\emptyset$	$\emptyset$	$\{2\}$	$\{2, 3\}$	$\emptyset$
2	$\{2, 3\}$	$\emptyset$	$\emptyset$	$\{2, 3\}$	$\emptyset$
3	$\emptyset$	$\{4\}$	$\emptyset$	$\emptyset$	$\{1, 2, 4\}$
4	$\emptyset$	$\{5\}$	$\{1\}$	$\{2, 3\}$	$\{5\}$
5	$\{4\}$	$\emptyset$	$\emptyset$	$\{1, 2, 4\}$	$\emptyset$

For example, the value $\delta^*(5, a)$ is the set $\{1, 2, 4\}$, because $\delta(5, a) = \{4\}$ and there are Λ-transitions from 4 to 1 and from 1 to 2.

Figure 3.20b shows the NFA M_1, whose transition function has the values in the last two columns of the table. In this example, the initial state of M is already an accepting state, and so drawing the new transitions and eliminating the Λ-transitions are the only steps required to obtain M_1.

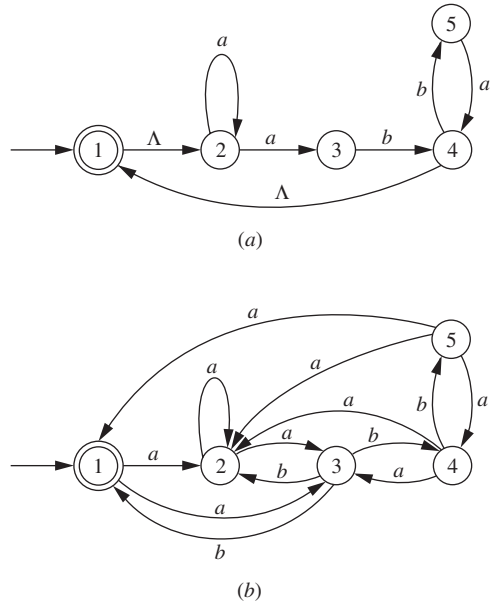


Figure 3.20 |
Eliminating Λ-transitions from an NFA.

EXAMPLE 3.21 Using the Subset Construction to Eliminate Nondeterminism

We consider the NFA $M = (Q, \{a, b\}, 0, A, \delta)$ in Example 3.6, shown in Figure 3.7. Instead of labeling states as q_i, here we will use only the subscript i. We will describe the FA $M_1 = (2^Q, \{a, b\}, \{0\}, A_1, \delta_1)$ obtained from the construction in the proof of Theorem 3.18. Because a set with n elements has 2^n subsets, using this construction might require an exponential increase in the number of states. As this example will illustrate, we can often get by with fewer by considering only the states of M_1 (subsets of Q) that are reachable from $\{0\}$, the initial state of M_1.

It is helpful, and in fact recommended, to use a transition table for δ in order to obtain the values of δ_1. The table is shown below.

q	$\delta(q, a)$	$\delta(q, b)$
0	{1, 2}	{4}
1	{0}	Ø
2	{3}	Ø
3	Ø	{0}
4	Ø	Ø

The transition diagram for M_1 is shown in Figure 3.22. For example, $\delta_1(\{1, 2\}, a) = \delta(1, a) \cup \delta(2, a) = \{0, 3\}$. If you compare Figure 3.22 to Figure 2.23c, you will see that they are the same except for the way the states are labeled. The subset construction doesn't always produce the FA with the fewest possible states, but in this example it does.

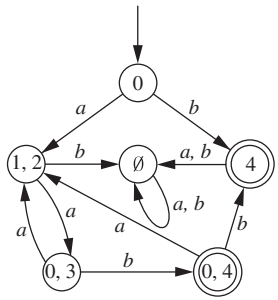


Figure 3.22 |
Applying the subset construction to the NFA in Example 3.21.

Converting an NFA with Λ-Transitions to an FA **EXAMPLE 3.23**

For the NFA pictured in Figure 3.24a, we show the transition function in tabular form below, as well as the transition function for the resulting NFA without Λ-transitions. It is pictured in Figure 3.24b.

q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, \Lambda)$	$\delta^*(q, a)$	$\delta^*(q, b)$
1	{1}	Ø	{2, 4}	{1, 2, 3, 4, 5}	{4, 5}
2	{3}	{5}	Ø	{3}	{5}
3	Ø	{2}	Ø	Ø	{2}
4	{5}	{4}	Ø	{5}	{4}
5	Ø	Ø	Ø	Ø	Ø

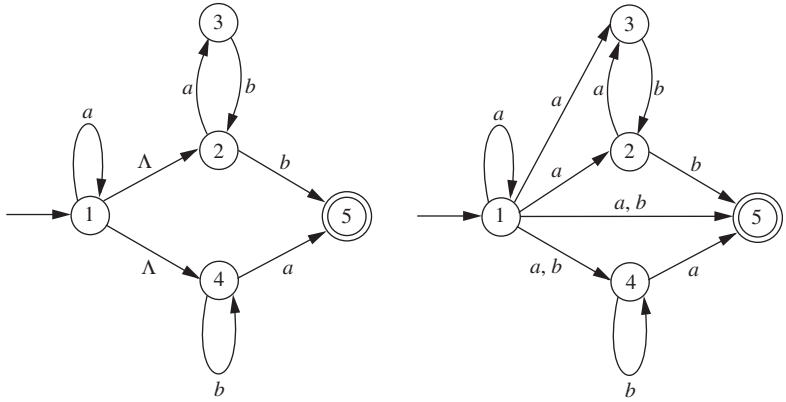


Figure 3.24 |
Converting an NFA to an FA.

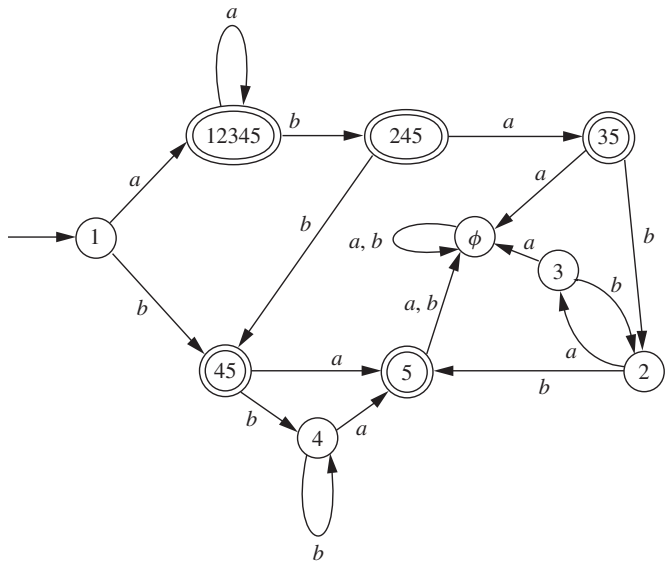


Figure 3.24 |
Continued

The subset construction gives us a slightly greater variety of subsets this time, but still considerably fewer than the total number of subsets of Q. The final FA is shown in Figure 3.24c.

3.4 | KLEENE'S THEOREM, PART 1

If we are trying to construct a device that accepts a regular language L, we can proceed one state at a time, as in Example 2.22, deciding at each step which strings it is necessary to distinguish. Adding each additional state may get harder as the number of states grows, but if we know somehow that there is an FA accepting L, we can be sure that the procedure will eventually terminate and produce one.

We have examples to show that for certain regular expressions, nondeterminism simplifies the problem of drawing an accepting device. In this section we will use nondeterminism to show that we can do this for every regular expression. Furthermore, we now have algorithms to convert the resulting NFA to an FA. The conclusion will be that on the one hand, the state-by-state approach will always work; and on the other hand, there is a systematic procedure that is also guaranteed to work and may be more straightforward.

The general result is one half of Kleene's theorem, which says that regular languages are the languages that can be accepted by finite automata. We will discuss the first half in this section and the second in Section 3.5.

Theorem 3.25 Kleene's Theorem, Part 1
For every alphabet Σ, every regular language over Σ can be accepted by a finite automaton.

Proof
Because of Theorems 3.17 and 3.18, it's enough to show that every regular language over Σ can be accepted by an NFA. The set of regular languages over Σ is defined recursively in Definition 3.1, and we will prove the theorem by structural induction.

The languages $\emptyset$ and $\{\sigma\}$ (where $\sigma \in \Sigma$) can be accepted by the two NFAs in Figure 3.26, respectively. The induction hypothesis is that L_1 and L_2 are both regular languages over Σ and that for both $i = 1$ and $i = 2$, L_i can be accepted by an NFA $M_i = (Q_i, \Sigma, q_i, A_i, \delta_i)$. We can assume, by renaming states if necessary, that Q_1 and Q_2 are disjoint. In the induction step we must show that there are NFAs accepting the three languages $L(M_1) \cup L(M_2)$, $L(M_1)L(M_2)$, and $L(M_1)^*$.

In each case we will give an informal definition and a diagram showing the idea of the construction. For simplicity, each diagram shows the two NFAs M_1 and M_2 as having two accepting states, both distinct from the initial state.

An NFA M_u accepting $L(M_1) \cup L(M_2)$ is shown in Figure 3.27a. Its states are those of M_1 and M_2 and one additional state q_u that is the initial state. The transitions include all the ones in M_1 and M_2 as well as Λ-transitions from q_u to q_1 and q_2, the initial states of M_1 and M_2. Finally, the accepting states are simply the states in $A_1 \cup A_2$.

If $x \in L(M_1)$, for example, M_u can accept x by taking the Λ-transition from q_u to q_1 and then executing the moves that would allow M_1 to accept x. On the other hand, if x is any string accepted by M_u, there is a path from q_u to an element of A_1 or A_2. The first transition in the path must be a Λ-transition, which takes M_u to q_1 or q_2. Because $Q_1 \cap Q_2 = \emptyset$, the remainder of the path causes x to be accepted either by M_1 or by M_2.

An NFA M_c accepting $L(M_1)L(M_2)$ is shown in Figure 3.27b. No new states need to be added to those of M_1 and M_2. The initial state is q_1, and the accepting states are the elements of A_2. The transitions include all those of M_1 and M_2 and a new Λ-transition from every element of A_1 to q_2. If x is the string x_1x_2, where x_i is accepted by M_i for each i, then M_c can process x by moving from q_1 to a state in A_1 using Λ's and the symbols of x_1, taking the Λ-transition to q_2, and moving to a state in A_2 using Λ's and the symbols of x_2. Conversely, if x is a string accepted

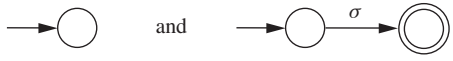


Figure 3.26

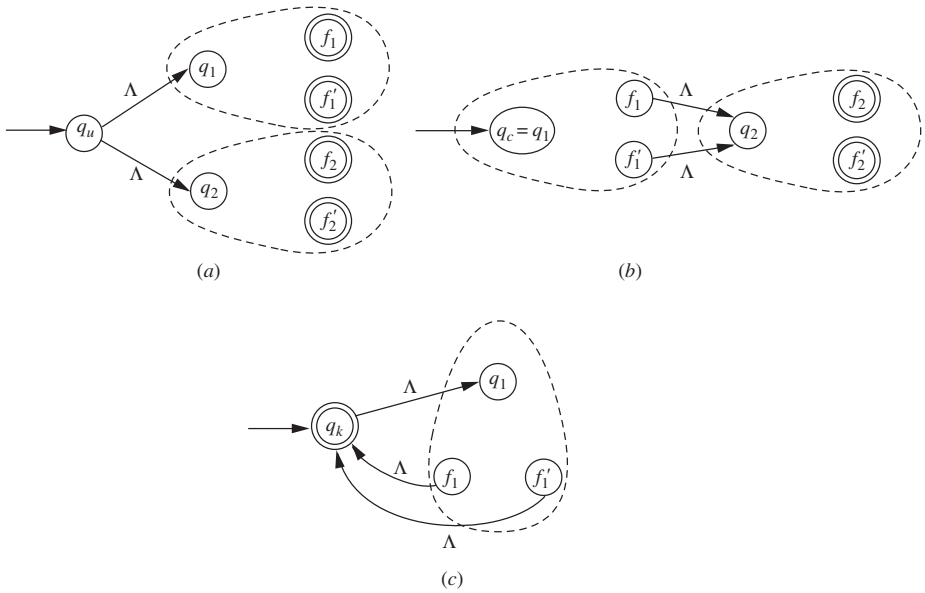


Figure 3.27 |
Schematic diagram for Kleene's theorem, Part 1.

by M_c, then at some point during the computation, M_c must execute the Λ-transition from an element of A_1 to q_2. If x_1 is the prefix of x whose symbols have been processed at that point, then x_1 must be accepted by M_1; the remaining suffix of x is accepted by M_2, because it corresponds to a path from q_2 to an element of A_2 that cannot involve any transitions other than those of M_2.

Finally, an NFA M_k accepting $L(M_1)^*$ is shown in Figure 3.27c. Its states are the elements of Q_1 and a new initial state q_k that is also the only accepting state. The transitions are those of M_1, a Λ-transition from q_k to q_1, and a Λ-transition from every element of A_1 to q_k. We can see by structural induction that every element of $L(M_1)^*$ is accepted. The null string is, because q_k is an accepting state. Now suppose that $x \in L(M_1)^*$ is accepted and that $y \in L(M_1)$. When M^* is in a state in A_1 after processing x, it can take a Λ-transition to q_k and another to q_1, process y so as to end up in an element of A_1, and finish up by returning to q_k with a Λ-transition. Therefore, xy is accepted by M^*.

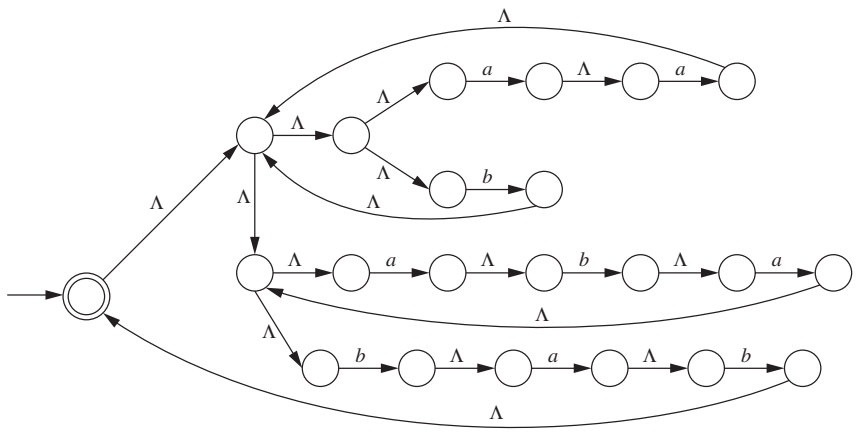
We can argue in the opposite direction by using mathematical induction on the number of times M^* enters the state q_k in the process of accepting a string. If M^* visits q_k only once in accepting x, then $x = \Lambda$, which is an element of $L(M_1)^*$. If we assume that $n \geq 1$ and that every string accepted by M^* that causes M^* to enter q_k n or fewer times is in $L(M_1)^*$, then consider a string x that causes M^* to enter q_k $n + 1$ times

and is accepted. Let x_1 be the prefix of x that is accepted when M^* enters q_k the nth time, and let x_2 be the remaining part of x. By the induction hypothesis, $x_1 \in L(M_1)^*$. In processing x_2, M^* moves to q_1 on a Λ-transition and then from q_1 to an element of A_1 using Λ-transitions in addition to the symbols of x_2. Therefore, $x_2 \in L(M_1)$, and it follows that $x \in L(M_1)^*$.

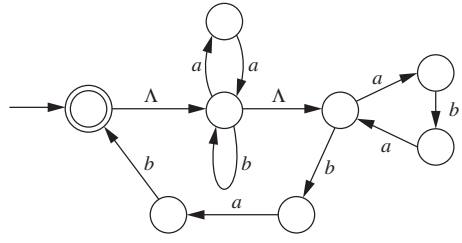
An NFA Corresponding to $((aa + b)^*(aba)^*bab)^*$ **EXAMPLE 3.28**

The three portions of the induction step in the proof of Theorem 3.25 provide algorithms for constructing an NFA corresponding to an arbitrary regular expression. These can be combined into a general algorithm that could be used to automate the process.

The transition diagram in Figure 3.29a shows a literal application of the three algorithms in the case of the regular expression $((aa + b)^*(aba)^*bab)^*$. In this case there is no need for all the Λ-transitions that are called for by the algorithms, and a simplified NFA is



(*a*)



(*b*)

Figure 3.29 |
Constructing an NFA for the regular expression $((aa + b)^*(aba)^*bab)^*$.

shown in Figure 3.29b. At least two Λ-transitions are still helpful in order to preserve the resemblance between the transition diagram and the regular expression. The algorithms can often be shortened in examples, but for each step where one of them calls for an extra state and/or a Λ-transition, there are examples to show that dispensing with the extra state or the transition doesn't always work (see Exercises 3.45–3.48).

3.5 | KLEENE'S THEOREM, PART 2

In this section we prove that if L is accepted by a finite automaton, then L is regular. The proof will provide an algorithm for starting with an FA that accepts L and finding a regular expression that describes L.

Theorem 3.30 Kleene's Theorem, Part 2
For every finite automaton $M = (Q, \Sigma, q_0, A, \delta)$, the language $L(M)$ is regular.

Proof
For states p and q, we introduce the notation $L(p, q)$ for the language

$$L(p, q) = \{x \in \Sigma^* \mid \delta^*(p, x) = q\}$$

If we can show that for every p and q in Q, $L(p, q)$ is regular, then it will follow that $L(M)$ is, because

$$L(M) = \bigcup \{L(q_0, q) \mid q \in A\}$$

and the union of a finite collection of regular languages is regular.

We will show that each language $L(p, q)$ is regular by expressing it in terms of simpler languages that are regular. Strings in $L(p, q)$ cause M to move from p to q in any manner whatsoever. One way to think about simpler ways of moving from p to q is to think about the number of transitions involved; the problem with this approach is that there is no upper limit to this number, and so no obvious way to obtain a final regular expression. A similar approach that is more promising is to consider the distinct states through which M passes as it moves from p to q. We can start by considering how M can go from p to q without going through any states, and at each step add one more state to the set through which M is allowed to go. This procedure will terminate when we have enlarged the set to include all possible states.

If $x \in L(p, q)$, we say x causes M to go from p to q *through* a state r if there are nonnull strings x_1 and x_2 such that $x = x_1x_2$, $\delta^*(p, x_1) = r$, and $\delta^*(r, x_2) = q$. In using a string of length 1 to go from p to q, M does not go through any state. (If M loops from p back to p on the symbol a, it does not go through p even though the string a causes it to leave p and enter p.) In using a string of length $n \geq 2$, it goes through a state $n - 1$ times, but if $n > 2$ these states may not be distinct.

Now we assume that Q has n elements and that they are numbered from 1 to n. For $p, q \in Q$ and $j \geq 0$, we let $L(p, q, j)$ be the set of strings in $L(p, q)$ that cause M to go from p to q without going through any state numbered higher than j.

The set $L(p, q, 0)$ is the set of strings that allow M to go from p to q without going through any state at all. This includes the set of alphabet symbols σ for which $\delta(p, \sigma) = q$, and in the case when $p = q$ it also includes the string Λ. In any case, $L(p, q, 0)$ is a finite set of strings and therefore regular.

Suppose that for some number $k \geq 0$, $L(p, q, k)$ is regular for every p and every q in Q, and consider how a string can be in $L(p, q, k + 1)$. The easiest way is for it to be in $L(p, q, k)$, because if M goes through no state numbered higher than k, it certainly goes through no state numbered higher than $k + 1$. The other strings in $L(p, q, k + 1)$ are those that cause M to go from p to q by going through state $k + 1$ and no higher-numbered states. A path of this type goes from p to $k + 1$; it may return to $k + 1$ one or more times; and it finishes by going from $k + 1$ to q (see Figure 3.31). On each of these individual portions, the path starts or stops at state $k + 1$ but doesn't go through any state numbered higher than k.

Every string in $L(p, q, k + 1)$ can be described in one of these two ways, and every string that has one of these two forms is in $L(p, q, k + 1)$. The resulting formula is

$$L(p, q, k + 1) = L(p, q, k) \cup L(p, k + 1, k)L(k + 1, k + 1, k)^* \\ L(k + 1, q, k)$$

We have the ingredients, both for a proof by mathematical induction that $L(p, q)$ is regular and for an algorithm to obtain a regular expression for this language. $L(p, q, 0)$ can be described by a regular expression; for each $k < n$, $L(p, q, k + 1)$ is described by the formula above; and $L(p, q, n) = L(p, q)$, because the condition that the path go through no state numbered higher than n is no restriction at all if there are no states numbered higher than n. As we observed at the beginning of the proof, the last step in obtaining a regular expression for $L(M)$ is to use the $+$ operation to combine the expressions for the languages $L(q_0, q)$, where $q \in A$.

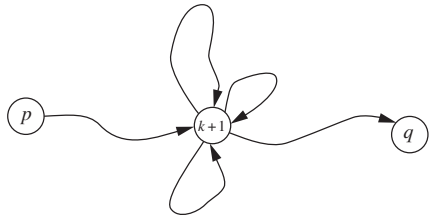


Figure 3.31
Going from p to q by going through $k + 1$.

EXAMPLE 3.32 Finding a Regular Expression Corresponding to an FA

Let M be the finite automaton pictured in Figure 3.33.

If we let $r(i, j, k)$ denote a regular expression corresponding to the language $L(i, j, k)$ described in the proof of Theorem 3.30, then $L(M)$ is described by the regular expression $r(M)$, where

$$r(M) = r(1, 1, 3) + r(1, 2, 3)$$

We might try calculating this expression from the top down, at least until we can see how many of the terms $r(i, j, k)$ we will need that involve smaller values of k. The recursive formula in the proof of the theorem tells us that

$$r(1, 1, 3) = r(1, 1, 2) + r(1, 3, 2)r(3, 3, 2)^*r(3, 1, 2)$$
$$r(1, 2, 3) = r(1, 2, 2) + r(1, 3, 2)r(3, 3, 2)^*r(3, 2, 2)$$

Applying the formula to the expressions $r(i, j, 2)$ that we apparently need, we obtain

$$r(1, 1, 2) = r(1, 1, 1) + r(1, 2, 1)r(2, 2, 1)^*r(2, 1, 1)$$
$$r(1, 3, 2) = r(1, 3, 1) + r(1, 2, 1)r(2, 2, 1)^*r(2, 3, 1)$$
$$r(3, 3, 2) = r(3, 3, 1) + r(3, 2, 1)r(2, 2, 1)^*r(2, 3, 1)$$
$$r(3, 1, 2) = r(3, 1, 1) + r(3, 2, 1)r(2, 2, 1)^*r(2, 1, 1)$$
$$r(1, 2, 2) = r(1, 2, 1) + r(1, 2, 1)r(2, 2, 1)^*r(2, 2, 1)$$
$$r(3, 2, 2) = r(3, 2, 1) + r(3, 2, 1)r(2, 2, 1)^*r(2, 2, 1)$$

At this point it is clear that we need every one of the expressions $r(i, j, 1)$, and we now start at the bottom and work our way up. The three tables below show the expressions $r(i, j, 0)$, $r(i, j, 1)$, and $r(i, j, 2)$ for all combinations of i and j. (Only six of the nine entries in the last table are required.)

p	$r(p, 1, 0)$	$r(p, 2, 0)$	$r(p, 3, 0)$
1	$a + \Lambda$	b	$\emptyset$
2	a	Λ	b
3	a	b	Λ

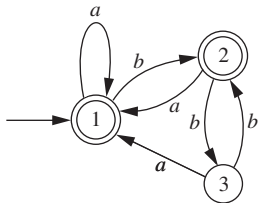


Figure 3.33 An FA for which we want an equivalent regular expression.

p	$r(p,1,1)$	$r(p,2,1)$	$r(p,3,1)$
1	a^*	a^*b	$\emptyset$
2	aa^*	$\Lambda + aa^*b$	b
3	aa^*	a^*b	Λ

p	$r(p,1,2)$	$r(p,2,2)$	$r(p,3,2)$
1	$a^*(baa^*)^*$	$a^*(baa^*)^*b$	$a^*(baa^*)^*bb$
2	$aa^*(baa^*)^*$	$(aa^*b)^*$	$(aa^*b)^*b$
3	$aa^* + a^*baa^*(baa^*)^*$	$a^*b(aa^*b)^*$	$\Lambda + a^*b(aa^*b)^*b$

For example,

$$\begin{aligned} r(2,2,1) &= r(2,2,0) + r(2,1,0)r(1,1,0)^*r(1,2,0) \\ &= \Lambda + (a)(a+\Lambda)^*(b) \\ &= \Lambda + aa^*b \\ r(3,1,2) &= r(3,1,1) + r(3,2,1)r(2,2,1)^*r(2,1,1) \\ &= aa^* + (a^*b)(\Lambda + aa^*b)^*(aa^*) \\ &= aa^* + a^*b(aa^*b)^*aa^* \\ &= aa^* + a^*baa^*(baa^*)^* \end{aligned}$$

The terms required for the final regular expression can now be obtained from the last table. As you can see, these expressions get very involved, even though we have already made some attempts to simplify them. There is no guarantee that the final regular expression is the simplest possible (it seems clear in this case that it is not), but at least we have a systematic way of generating a regular expression corresponding to $L(M)$.

EXERCISES

3.1. In each case below, find a string of minimum length in $\{a, b\}^*$ *not* in the language corresponding to the given regular expression.

a. $b^*(ab)^*a^*$

b. $(a^* + b^*)(a^* + b^*)(a^* + b^*)$

c. $a^*(baa^*)^*b^*$

d. $b^*(a + ba)^*b^*$

3.2. Consider the two regular expressions

$$r = a^* + b^* \qquad s = ab^* + ba^* + b^*a + (a^*b)^*$$

a. Find a string corresponding to r but not to s.

b. Find a string corresponding to s but not to r.

c. Find a string corresponding to both r and s.

d. Find a string in $\{a, b\}^*$ corresponding to neither r nor s.

3.3. Let r and s be arbitrary regular expressions over the alphabet Σ. In each case below, find a simpler equivalent regular expression.

a. $r(r^*r + r^*) + r^*$

b. $(r + \Lambda)^*$

c. $(r + s)^*rs(r + s)^* + s^*r^*$

3.4. It is not difficult to show using mathematical induction that for every integer $n \geq 2$, there are nonnegative integers i and j such that $n = 2i + 3j$. With this in mind, simplify the regular expression $(aa + aaa)(aa + aaa)^*$.

3.5. In each case below, give a simple description of the smallest set of languages that contains all the "basic" languages $\emptyset$, $\{\Lambda\}$, and $\{\sigma\}$ (for every $\sigma \in \Sigma$) and is closed under the specified operations.

a. union

b. concatenation

c. union and concatenation

3.6. Suppose w and z are strings in $\{a, b\}^*$. Find regular expressions corresponding to each of the languages defined recursively below.

a. $\Lambda \in L$; for every $x \in L$, then wx and xz are elements of L.

b. $a \in L$; for every $x \in L$, wx, xw, and xz are elements of L.

c. $\Lambda \in L$; $a \in L$; for every $x \in L$, wx and zx are in L.

3.7. Find a regular expression corresponding to each of the following subsets of $\{a, b\}^*$.

a. The language of all strings containing exactly two a's.

b. The language of all strings containing at least two a's.

c. The language of all strings that do not end with ab.

d. The language of all strings that begin or end with aa or bb.

e. The language of all strings not containing the substring aa.

f. The language of all strings in which the number of a's is even.

g. The language of all strings containing no more than one occurrence of the string aa. (The string aaa should be viewed as containing two occurrences of aa.)

h. The language of all strings in which every a is followed immediately by bb.

i. The language of all strings containing both bb and aba as substrings.

j. The language of all strings not containing the substring aaa.

k. The language of all strings not containing the substring bba.

l. The language of all strings containing both bab and aba as substrings.

m. The language of all strings in which the number of a's is even and the number of b's is odd.

n. The language of all strings in which both the number of a's and the number of b's are odd.

3.8. a. The regular expression $(b + ab)^*(a + ab)^*$ describes the set of all strings in $\{a, b\}^*$ not containing the substring _____x_____ for any x. (Fill in the blanks appropriately.)

b. The regular expression $(a + b)^*(aa^*bb^*aa^* + bb^*aa^*bb^*)(a + b)^*$ describes the set of all strings in $\{a, b\}^*$ containing both the substrings _____ and _____. (Fill in the blanks appropriately.)

3.9. Show that every finite language is regular.

3.10. a. If L is the language corresponding to the regular expression $(aab + bbaba)^*baba$, find a regular expression corresponding to $L^r = \{x^r \mid x \in L\}$.

b. Using the example in part (a) as a model, give a recursive definition (based on Definition 3.1) of the reverse e^r of a regular expression e.

c. Show that for every regular expression e, if the language L corresponds to e, then L^r corresponds to e^r.

3.11. The *star height* of a regular expression r over Σ, denoted by $sh(r)$, is defined as follows:

i. $sh(\emptyset) = 0$.

ii. $sh(\Lambda) = 0$.

iii. $sh(\sigma) = 0$ for every $\sigma \in \Sigma$.

iv. $sh((rs)) = sh((r + s)) = \max(sh(r), sh(s))$.

v. $sh((r^*)) = sh(r) + 1$.

Find the star heights of the following regular expressions.

a. $(a(a + a^*aa) + aaa)^*$

b. $(((a + a^*aa)aa)^* + aaaaaa^*)^*$

3.12. For both the regular expressions in the previous exercise, find an equivalent regular expression of star height 1.

3.13. Let c and d be regular expressions over Σ.

a. Show that the formula $r = c + rd$, involving the variable r, is true if the regular expression cd^* is substituted for r.

b. Show that if Λ is not in the language corresponding to d, then any regular expression r satisfying $r = c + rd$ corresponds to the same language as cd^*.

3.14. Describe precisely an algorithm that could be used to eliminate the symbol $\emptyset$ from any regular expression that does not correspond to the empty language.

3.15. Describe an algorithm that could be used to eliminate the symbol Λ from any regular expression whose corresponding language does not contain the null string.

3.16. The *order* of a regular language L is the smallest integer k for which $L^k = L^{k+1}$, if there is one, and ∞ otherwise.

a. Show that the order of L is finite if and only if there is an integer k such that $L^k = L^*$, and that in this case the order of L is the smallest k such that $L^k = L^*$.

b. What is the order of the regular language $\{\Lambda\} \cup \{aa\}\{aaa\}^*$?

c. What is the order of the regular language $\{a\} \cup \{aa\}\{aaa\}^*$?

d. What is the order of the language corresponding to the regular expression $(\Lambda + b^*a)(b + ab^*ab^*a)^*$?

3.17. †A *generalized regular expression* is defined the same way as an ordinary regular expression, except that two additional operations, intersection and complement, are allowed. So, for example, the generalized regular expression $abb\emptyset' \cap (\emptyset' aaa\emptyset')'$ represents the set of all strings in $\{a, b\}^*$ that start with *abb* and don't contain the substring *aaa*.

a. Show that the subset $\{aba\}^*$ of $\{a, b\}^*$ can be described by a generalized regular expression with no occurrences of *.

b. Can the subset $\{aaa\}^*$ be described this way? Give reasons for your answer.

3.18. Figure 3.34, at the bottom of this page, shows a transition diagram for an NFA. For each string below, say whether the NFA accepts it.

a. *aba*

b. *abab*

c. *aaabbb*

3.19. Find a regular expression corresponding to the language accepted by the NFA pictured in Figure 3.34. You should be able to do it without applying Kleene's theorem: First find a regular expression describing the most general way of reaching state 4 the first time, and then find a regular expression describing the most general way, starting in state 4, of moving to state 4 the next time.

3.20. For each of the NFAs shown in Figure 3.35 on the next page, find a regular expression corresponding to the language it accepts.

3.21. On the next page, after Figure 3.35, is the transition table for an NFA with states 1–5 and input alphabet $\{a, b\}$. There are no Λ-transitions.

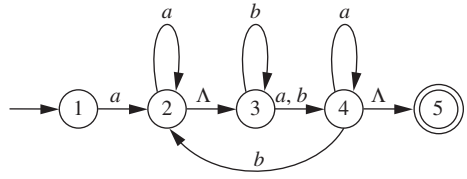


Figure 3.34 |

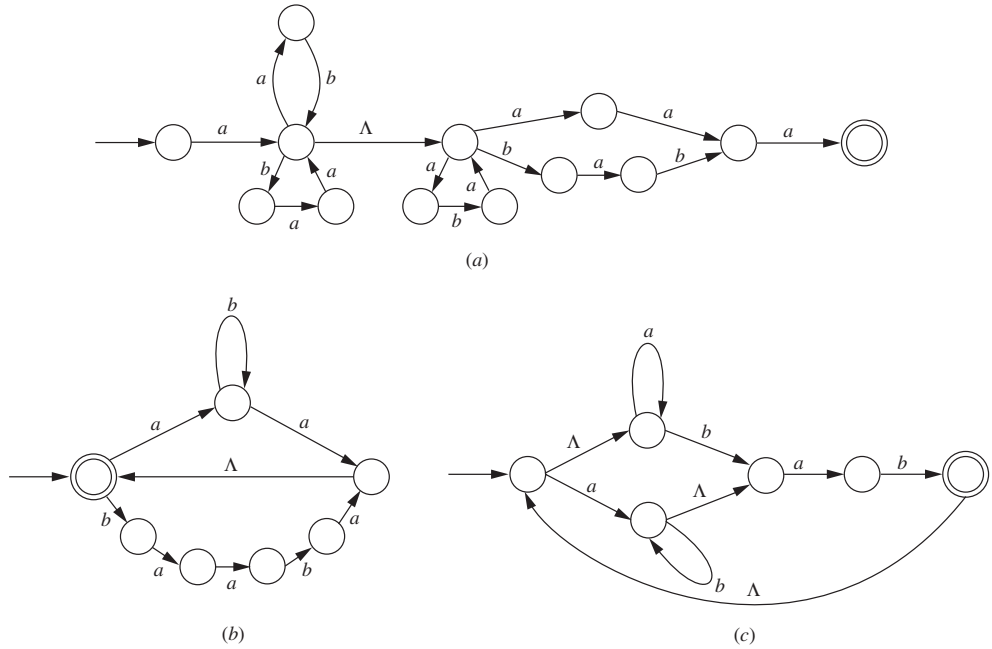


Figure 3.35 |

q	$\delta(q, a)$	$\delta(q, b)$
1	{1, 2}	{1}
2	{3}	{3}
3	{4}	{4}
4	{5}	Ø
5	Ø	{5}

a. Draw a transition diagram.
b. Calculate $\delta^*(1, ab)$.
c. Calculate $\delta^*(1, abaab)$.

3.22. A transition table is given for an NFA with seven states.

q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, \Lambda)$
1	Ø	Ø	{2}
2	{3}	Ø	{5}
3	Ø	{4}	Ø
4	{4}	Ø	{1}
5	Ø	{6, 7}	Ø
6	{5}	Ø	Ø
7	Ø	Ø	{1}

Find:

a. $\Lambda(\{2, 3\})$

b. $\Lambda(\{1\})$

c. $\Lambda(\{3, 4\})$

d. $\delta^*(1, ba)$

e. $\delta^*(1, ab)$

f. $\delta^*(1, ababa)$

3.23. A transition table is given for another NFA with seven states.

q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, \Lambda)$
1	{5}	∅	{4}
2	{1}	∅	∅
3	∅	{2}	∅
4	∅	{7}	{3}
5	∅	∅	{1}
6	∅	{5}	{4}
7	{6}	∅	∅

Calculate $\delta^*(1, ba)$.

3.24. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA with no Λ-transitions. Show that for every $q \in Q$ and every $\sigma \in \Sigma$, $\delta^*(q, \sigma) = \delta(q, \sigma)$.

3.25. It is easy to see that if $M = (Q, \Sigma, q_0, A, \delta)$ is an FA accepting L, then the FA $M' = (Q, \Sigma, q_0, Q - A, \delta)$ accepts L' (the FA obtained from Theorem 2.15 by writing $L' = \Sigma^* - L$ is essentially M'). Does this still work if M is an NFA? If so, prove it. If not, find a counterexample.

3.26. In Definition 3.14, δ^* is defined recursively in an NFA by first defining $\delta^*(q, \Lambda)$ and then defining $\delta^*(q, y\sigma)$, where $y \in \Sigma^*$ and $\sigma \in \Sigma$. Give an acceptable recursive definition in which the recursive part of the definition defines $\delta^*(q, \sigma y)$ instead.

3.27. Which of the following, if any, would be a correct substitute for the second part of Definition 3.14? Give reasons for your answer.

a. $\delta^*(q, \sigma y) = \Lambda(\bigcup\{\delta^*(r, y) \mid r \in \delta(q, \sigma)\})$

b. $\delta^*(q, \sigma y) = \bigcup\{\Lambda(\delta^*(r, y)) \mid r \in \delta(q, \sigma)\}$

c. $\delta^*(q, \sigma y) = \bigcup\{\delta^*(r, y) \mid r \in \Lambda(\delta(q, \sigma))\}$

d. $\delta^*(q, \sigma y) = \bigcup\{\Lambda(\delta^*(r, y)) \mid r \in \Lambda(\delta(q, \sigma))\}$

3.28. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA. This exercise involves properties of the Λ-closure of a set S. Since $\Lambda(S)$ is defined recursively, structural induction can be used to show that $\Lambda(S)$ is a subset of some other set.

a. Show that if S and T are subsets of Q for which $S \subseteq T$, then $\Lambda(S) \subseteq \Lambda(T)$.

b. Show that for any $S \subseteq Q$, $\Lambda(\Lambda(S)) = \Lambda(S)$.

c. Show that if $S, T \subseteq Q$, then $\Lambda(S \cup T) = \Lambda(S) \cup \Lambda(T)$.

d. Show that if $S \subseteq Q$, then $\Lambda(S) = \bigcup\{\Lambda(\{p\}) \mid p \in S\}$.

e. Draw a transition diagram to illustrate the fact that $\Lambda(S \cap T)$ and $\Lambda(S) \cap \Lambda(T)$ are not always the same. Which is always a subset of the other?

f. Draw a transition diagram illustrating the fact that $\Lambda(S')$ and $\Lambda(S)'$ are not always the same. Which is always a subset of the other? Under what circumstances are they equal?

3.29. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA. A set $S \subseteq Q$ is called Λ-*closed* if $\Lambda(S) = S$.

a. Show that the union of two Λ-closed sets is Λ-closed.

b. Show that the intersection of two Λ-closed sets is Λ-closed.

c. Show that for any subset S of Q, $\Lambda(S)$ is the smallest Λ-closed set of which S is a subset.

3.30. †Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA. Show that for every $q \in Q$ and every $x, y \in \Sigma^*$,

$$\delta^*(q, xy) = \bigcup\{\delta^*(r, y) \mid r \in \delta^*(q, x)\}$$

3.31. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA, and let $M_1 = (Q, \Sigma, q_0, A, \delta_1)$ be the NFA with no Λ-transitions for which $\delta_1(q, \sigma) = \{\delta(q, \sigma)\}$ for every $q \in Q$ and $\sigma \in \Sigma$. Show that for every $q \in Q$ and $x \in \Sigma^*$, $\delta_1^*(q, x) = \{\delta(q, x)\}$. Recall that the two functions δ^* and δ_1^* are defined differently.

3.32. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA accepting a language L. Assume that there are no transitions to q_0, that A has only one element, q_f, and that there are no transitions from q_f.

a. Let M_1 be obtained from M by adding Λ-transitions from q_0 to every state that is reachable from q_0 in M. (If p and q are states, q is reachable from p if there is a string $x \in \Sigma^*$ such that $q \in \delta^*(p, x)$.) Describe (in terms of L) the language accepted by M_1.

b. Let M_2 be obtained from M by adding Λ-transitions to q_f from every state from which q_f is reachable in M. Describe in terms of L the language accepted by M_2.

c. Let M_3 be obtained from M by adding both the Λ-transitions in (a) and those in (b). Describe the language accepted by M_3.

3.33. Give an example of a regular language L containing Λ that cannot be accepted by any NFA having only one accepting state and no Λ-transitions, and show that your answer is correct.

3.34. Can every regular language not containing Λ be accepted by an NFA having only one accepting state and no Λ-transitions? Prove your answer.

3.35. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA, let m be the maximum size of any of the sets $\delta^*(q, \sigma)$ for $q \in Q$ and $\sigma \in \Sigma$, and let x be a string of length n over the input alphabet.

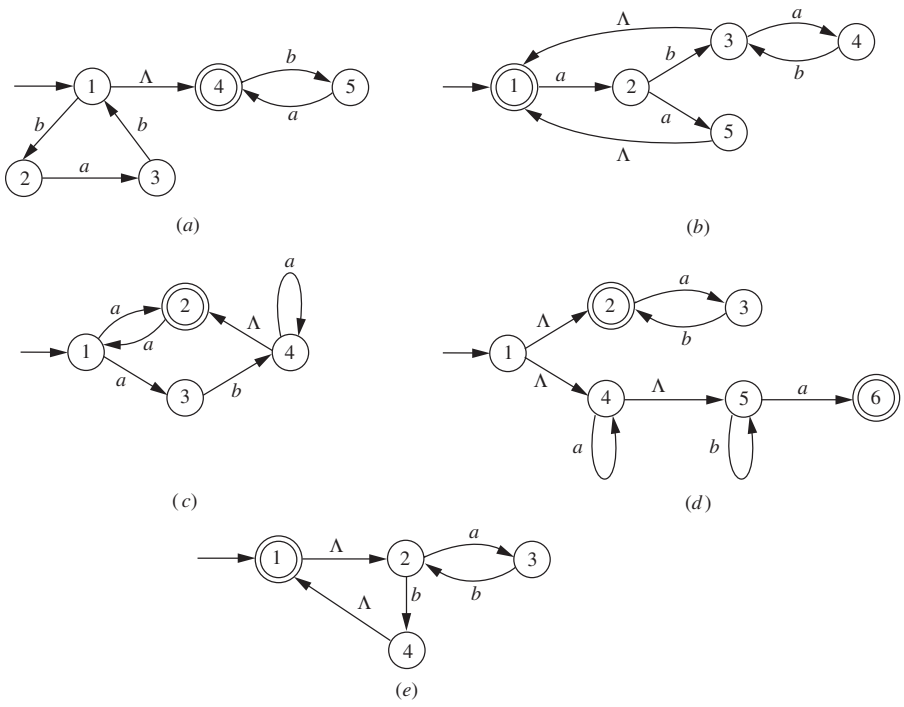


Figure 3.36 |

a. What is the maximum number of distinct paths that there might be in the computation tree corresponding to x?

b. In order to determine whether x is accepted by M, it is sufficient to replace the complete computation tree by one that is perhaps smaller, obtained by "pruning" the original one so that no level of the tree contains more nodes than the number of states in M (and no level contains more nodes than there are at that level of the original tree). Explain why this is possible, and how it might be done.

3.36. Let $M = (Q, \Sigma, q_0, A, \delta)$ be an NFA. The NFA M_1 obtained by eliminating Λ-transitions from M might have more accepting states than M, because the initial state q_0 is made an accepting state if $\Lambda(\{q_0\}) \cap A \neq \emptyset$. Explain why it is not necessary to make *all* the states q for which $\Lambda(\{q\}) \cap A \neq \emptyset$ accepting states in M_1.

3.37. In each part of Figure 3.36 is pictured an NFA. Use the algorithm described in the proof of Theorem 3.17 to draw an NFA with no Λ-transitions accepting the same language.

3.38. Each part of Figure 3.37 pictures an NFA. Using the subset construction, draw an FA accepting the same language. Label the final picture so as to make it clear how it was obtained from the subset construction.

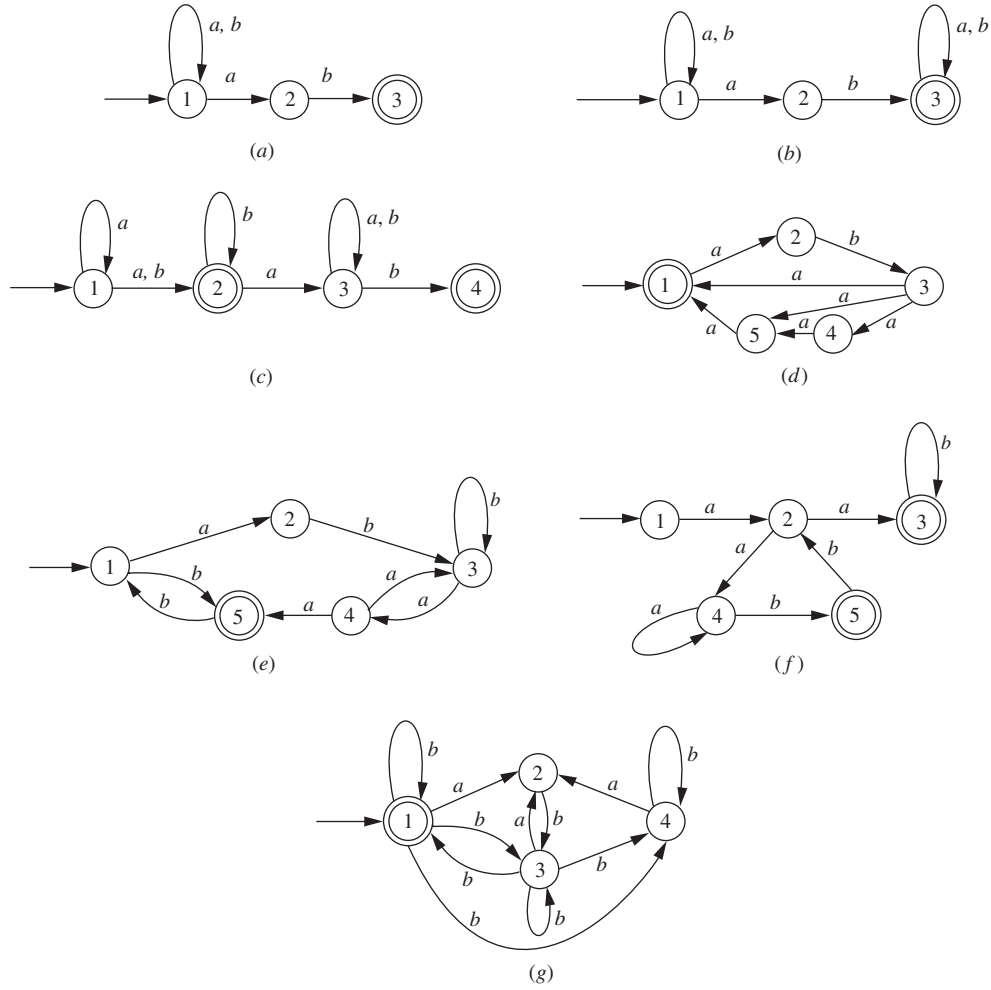


Figure 3.37 |

3.39. Suppose $L \subseteq \Sigma^*$ is a regular language. If every FA accepting L has at least n states, then every NFA accepting L has at least ____ states. (Fill in the blank, and explain your answer.)

3.40. Each part of Figure 3.38 shows an NFA. Draw an FA accepting the same language.

3.41. For each of the following regular expressions, draw an NFA accepting the corresponding language, so that there is a recognizable correspondence between the regular expression and the transition diagram.

a. $(b + bba)^*a$

b. $(a + b)^*(abb + ababa)(a + b)^*$

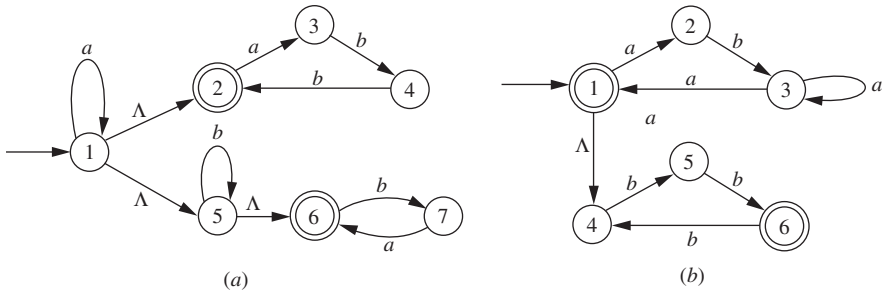


Figure 3.38

c. $(a + b)(ab)^*(abb)^*$

d. $(a + b)^*(abba^* + (ab)^*ba)$

e. $(a^*bb)^* + bb^*a^*$

3.42. For part (e) of Exercise 3.41, draw the NFA that is obtained by a literal application of Kleene's theorem, without any simplifications.

3.43. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an NFA accepting a language L. Let M_1 be the NFA obtained from M by adding Λ-transitions from each element of A to q_0. Describe (in terms of L) the language $L(M_1)$.

3.44. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an NFA accepting a language L.

a. Describe how to construct an NFA M_1 with no transitions to its initial state so that M_1 also accepts L.

b. Describe how to construct an NFA M_2 with exactly one accepting state and no transitions from that state, so that M_2 also accepts L.

3.45. Suppose M is an NFA with exactly one accepting state q_f that accepts the language $L \subseteq \{a, b\}^*$. In order to find NFAs accepting the languages $\{a\}^*L$ and $L\{a\}^*$, we might try adding a-transitions from q_0 to itself and from q_f to itself, respectively. Draw transition diagrams to show that neither technique always works.

3.46. In the construction of M_u in the proof of Theorem 3.25, consider this alternative to the construction described: Instead of a new state q_u and Λ-transitions from it to q_1 and q_2, make q_1 the initial state of the new NFA, and create a Λ-transition from it to q_2. Either prove that this works in general, or give an example in which it fails.

3.47. In the construction of M_c in the proof of Theorem 3.25, consider the simplified case in which M_1 has only one accepting state. Suppose that we eliminate the Λ-transition from the accepting state of M_1 to q_2, and merge these two states into one. Either show that this would always work in this case, or give an example in which it fails.

3.48. In the construction of M^* in the proof of Theorem 3.25, suppose that instead of adding a new state q_0, with Λ-transitions from it to q_1 and to it

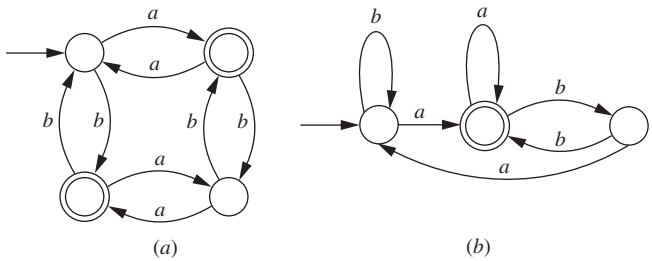


Figure 3.39

from each accepting state of Q_1, we make q_1 both the initial state and the accepting state, and create Λ-transitions from each accepting state of M_1 to q_0. Either show that this works in general, or give an example in which it fails.

3.49. Figure 3.39 shows FAs M_1 and M_2 accepting languages L_1 and L_2, respectively. Draw NFAs accepting each of the following languages, using the constructions in the proof of Theorem 3.25.

a. $L_2^* \cup L_1$

b. $L_2 L_1^*$

c. $L_1 L_2 \cup (L_2 L_1)^*$

3.50. Draw NFAs with no Λ-transitions accepting $L_1 L_2$ and $L_2 L_1$, where L_1 and L_2 are as in Exercise 3.49. Do this by connecting the two given diagrams directly, by arrows with appropriate labels.

3.51. Use the algorithm of Theorem 3.30 to find a regular expression corresponding to each of the FAs shown in Figure 3.40. In each case, if the FA has n states, construct tables showing $L(p, q, j)$ for each j with $0 \leq j \leq n - 1$.

3.52. Suppose M is an FA with the three states 1, 2, and 3, and 1 is both the initial state and the only accepting state. The expressions $r(p, q, 2)$ corresponding to the languages $L(p, q, 2)$ are shown in the table below. Write a regular expression describing $L(M)$.

p	$r(p, 1, 2)$	$r(p, 2, 2)$	$r(p, 3, 2)$
1	Λ	aa^*	$b + aa^*b$
2	$\emptyset$	a^*	a^*b
3	a	aaa^*	$\Lambda + b + ab + aaa^*b$

3.53. Suppose Σ_1 and Σ_2 are alphabets, and the function $f : \Sigma_1^* \to \Sigma_2^*$ is a *homomorphism*; i.e., $f(xy) = f(x)f(y)$ for every $x, y \in \Sigma_1^*$.

a. Show that $f(\Lambda) = \Lambda$.

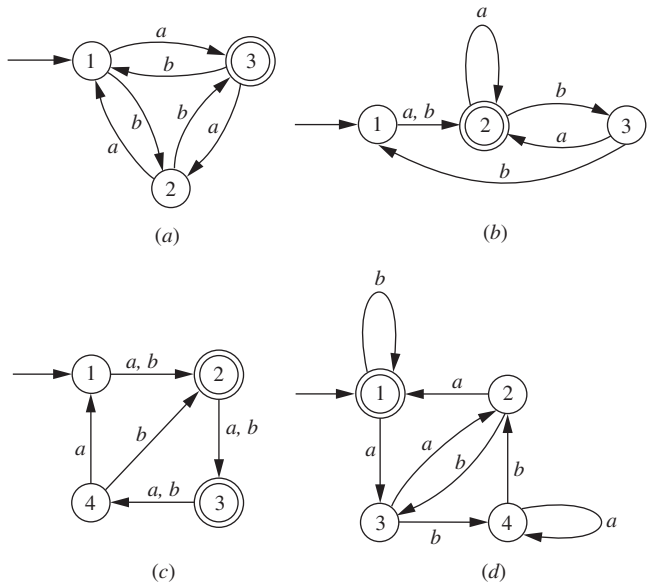


Figure 3.40 |

b. Show that if $L \subseteq \Sigma_1^*$ is regular, then $f(L)$ is regular. ($f(L)$ is the set $\{y \in \Sigma_2^* \mid y = f(x) \text{ for some } x \in L\}$.)

c. Show that if $L \subseteq \Sigma_2^*$ is regular, then $f^{-1}(L)$ is regular. ($f^{-1}(L)$ is the set $\{x \in \Sigma_1^* \mid f(x) \in L\}$.)

3.54. Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an NFA. For two (not necessarily distinct) states p and q, we define the regular expression $e(p, q)$ as follows: $e(p, q) = l + r_1 + r_2 + \cdots + r_k$, where l is either Λ (if $\delta(p, \Lambda)$ contains q) or $\emptyset$, and the r_i's are all the elements σ of Σ for which $\delta(p, \sigma)$ contains q. It's possible for $e(p, q)$ to be $\emptyset$, if there are no transitions from p to q; otherwise, $e(p, q)$ represents the "most general" transition from p to q.

If we generalize this by allowing $e(p, q)$ to be an arbitrary regular expression over Σ, we get what is called an *expression graph*. If p and q are two states in an expression graph G, and $x \in \Sigma^*$, we say that x allows G to move from p to q if there are states $p_0, p_1, \ldots, p_m$, with $p_0 = p$ and $p_m = q$, such that x corresponds to the regular expression $e(p_0, p_1)e(p_1, p_2) \ldots e(p_{n-1}, p_n)$. This allows us to say how G accepts a string x (x allows G to move from the initial state to an accepting state), and therefore to talk about the language accepted by G. It is easy to see that in the special case where G is simply an NFA, the two definitions for the language accepted by G coincide. It is also not hard to convince

yourself, using Theorem 3.25, that for any expression graph G, the language accepted by G can be accepted by an NFA.

We can use the idea of an expression graph to obtain an alternate proof of Theorem 3.30, as follows. Starting with an FA M accepting L, we may easily convert it to an NFA M_1 accepting L, so that M_1 has no transitions to its initial state q_0, exactly one accepting state q_f (which is different from q_0), and no transitions from q_f. The remainder of the proof is to specify a reduction technique to reduce by one the number of states other than q_0 and q_f, obtaining an equivalent expression graph at each step, until q_0 and q_f are the only states remaining. The regular expression $e(q_0, q_f)$ then describes the language accepted. If p is the state to be eliminated, the reduction step involves redefining $e(q, r)$ for every pair of states q and r other than p.

Describe in more detail how this reduction can be done. Then apply this technique to the FAs in Figure 3.40 to obtain regular expressions corresponding to their languages.

CHAPTER

4

Context-Free Languages

Regular languages and finite automata are too simple and too restrictive to be able to handle languages that are at all complex. Using *context-free grammars* allows us to generate more interesting languages; much of the syntax of a high-level programming language, for example, can be described this way. In this chapter we start with the definition of a context-free grammar and look at a number of examples. A particularly simple type of context-free grammar provides another way to describe the regular languages we discussed in Chapters 2 and 3. Later in the chapter we study derivations in a context-free grammar, how a derivation might be related to the structure of the string being derived, and the presence of ambiguity in a grammar, which can complicate this relationship. We also consider a few ways that a grammar might be simplified to make it easier to answer questions about the strings in the corresponding language.

4.1 USING GRAMMAR RULES TO DEFINE A LANGUAGE

The term *grammar* applied to a language like English refers to the rules for constructing phrases and sentences. For us a grammar is also a set of rules, simpler than the rules of English, by which strings in a language can be generated. In the first few examples in this section, a grammar is another way to write the recursive definitions that we used in Chapter 1 to define languages.

EXAMPLE 4.1 The language *AnBn*

In Example 1.18, we defined the language $AnBn = \{a^n b^n \mid n \geq 0\}$ using this recursive definition:

1. $\Lambda \in AnBn$.
2. for every $S \in AnBn$, $aSb \in AnBn$.

Let us think of S as a *variable*, representing an arbitrary string in *AnBn*. Rule 1, which we rewrite as $S \to \Lambda$, says that the arbitrary string could simply be Λ, obtained by substituting Λ for the variable S. To obtain any other string, we must begin with rule 2, which we write $S \to aSb$. This rule says that a string in *AnBn* can have the form aSb (or that S can be replaced by aSb), where the new occurrence of S represents some other element of *AnBn*. Replacing S by aSb is the first step in a *derivation* of the string, and the remaining steps will be further applications of rules 1 and 2 that will give a value to the new S. The derivation will continue as long as the string contains the variable S, so that in this example the last step will always be to replace S by Λ.

If α and β are strings, and α contains at least one occurrence of S, the notation

$$\alpha \Rightarrow \beta$$

will mean that β is obtained from α by using one of the two rules to replace a single occurrence of S by either Λ or aSb. Using this notation, we would write

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

to describe the sequence of steps (three applications of rule 2, then one application of rule 1) we used to derive the string $aaabbb \in AnBn$.

The notation is simplified further by writing rules 1 and 2 as

$$S \to \Lambda \mid aSb$$

and interpreting | as "or". When we write the rules of a grammar this way, we give concatenation higher precedence than |, which means in our example that the two alternatives in the formula $S \to \Lambda \mid aSb$ are Λ and aSb, not Λ and a.

The Language *Expr* **EXAMPLE 4.2**

In Example 1.19 we considered the language *Expr* of algebraic expressions involving the binary operators $+$ and $*$, left and right parentheses, and a single identifier a. We used the following recursive definition to describe *Expr*:

1. $a \in Expr$.
2. For every x and y in *Expr*, $x + y$ and $x * y$ are in *Expr*.
3. For every $x \in Expr$, $(x) \in Expr$.

Rule 2 involves two different letters x and y, because the two expressions being combined with either $+$ or $*$ are not necessarily the same. However, they both represent elements of *Expr*, and we still need only one variable in the grammar rules corresponding to the recursive definition:

$$S \to a \mid S + S \mid S * S \mid (S)$$

To derive the string $a + (a * a)$, for example, we could use the sequence of steps

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + (S) \Rightarrow a + (S * S) \Rightarrow a + (a * S) \Rightarrow a + (a * a)$$

The string $S + S$ that we obtain from the first step of this derivation suggests that the final expression should be interpreted as the sum of two subexpressions. The subexpressions we

end up with are obviously not the same, but they are both elements of *Expr* and can therefore both be derived from S.

A derivation of an expression in *Expr* is related to the way we choose to interpret the expression, and there may be several possible choices. The expression $a + a * a$, for example, has at least these two derivations:

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * a$$

and

$$S \Rightarrow S * S \Rightarrow S + S * S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * a$$

The first steps in these two derivations yield the strings $S + S$ and $S * S$, respectively. The first derivation suggests that the expression is the sum of two subexpressions, the second that it is the product of two subexpressions.

The rules of precedence normally adopted for algebraic expressions say, among other things, that multiplication has higher precedence than addition. If we adopt this precedence rule, then when we evaluate the expression $a + a * a$, we first evaluate the product $a * a$, so that we interpret the expression as a sum, not a product. Because there is nothing in our grammar rules to suggest that the first derivation is preferable to the second, one possible conclusion is that it might be better to use another grammar, in which every string has essentially only one derivation. We will return to this question in Section 4.4, when we discuss *ambiguity* in a grammar.

We have made the language *Expr* simple by restricting the expressions in several ways. We could easily add grammar rules to allow other operations besides $+$ and $*$; if we wanted to allow other "atomic" expressions besides the identifier a (more general identifiers, or numeric literals such as 16 and 1.3E−2, or both), we could add another variable, A, to get

$$S \rightarrow A \mid S + S \mid S * S \mid (S)$$

and then look for grammar rules beginning with A that would generate all the subexpressions we wanted. The next example involves another language L for which grammar rules generating L require more than one variable.

EXAMPLE 4.3 Palindromes and Nonpalindromes

We see from Example 1.18 that the language *Pal* of palindromes over the alphabet $\{a, b\}$ can be generated by the grammar rules

$$S \rightarrow \Lambda \mid a \mid b \mid aSa \mid bSb$$

What about its complement *NonPal*? The last two grammar rules in the definition of *Pal* still seem to work: For every nonpalindrome x, both axa and bxb are also nonpalindromes. But a recursive definition of *NonPal* cannot be as simple as the one for *Pal*, because there is no finite set of strings comparable to $\{\Lambda, a, b\}$ that can serve as basis elements in the definition (see Exercise 4.6).

To find the crucial feature of a nonpalindrome, let's look at one, say

$$x = abbbbaaba$$

The string x is $abyba$, where $y = bbbaa$. Working our way in from the ends, comparing the symbol on the left with the one on the right, we wouldn't know until we got to y whether x was a palindrome or not, but once we saw that y looked like bza for some string z, it would be clear, even without looking at any symbols of z. Here is a definition of *NonPal*:

1. For every $A \in \{a, b\}^*$, aAb and bAa are elements of *NonPal*;
2. For every S in *NonPal*, aSa and bSb are in *NonPal*.

In order to obtain grammar rules generating *NonPal*, we can introduce A as a second variable, representing an arbitrary element of $\{a, b\}^*$, and use grammar rules starting with A that correspond to the recursive definition in Example 1.17. The complete set of rules for *NonPal* is

$$S \to aSa \mid bSb \mid aAb \mid bAa$$
$$A \to Aa \mid Ab \mid \Lambda$$

A derivation of $abbbbaaba$ in this grammar is

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abbAaba$$
$$\Rightarrow abbAaaba \Rightarrow abbAbaaba \Rightarrow abbAbbaaba \Rightarrow abbbbaaba$$

In order to generate a language L using the kinds of grammars we are discussing, it is often necessary to include several variables. The *start* variable is distinguished from the others, and we will usually denote it by S. Each remaining variable can be thought of as representing an arbitrary string in some auxiliary language involved in the definition of L (the language of all strings that can be derived from that variable). We can still interpret the grammar as a recursive definition of L, except that we must extend our notion of recursion to include *mutual* recursion: rather than one object defined recursively in terms of itself, several objects defined recursively in terms of each other.

English and Programming-Language Syntax **EXAMPLE 4.4**

You can easily see how grammar rules can be used to describe simple English syntax. Many useful sentences can be generated by the rule

<declarative sentence> → <subject phrase> <verb phrase> <object>

provided that reasonable rules are found for each of the three variables on the right. Three examples are "haste makes waste", "the ends justify the means", and "we must extend our notion" (from the last sentence in Example 4.3). You can also see how difficult it would be to find grammars of a reasonable size that would allow more sophisticated sentences without also allowing gibberish. (Try to formulate some rules to generate the preceding sentence, "You can also see how ... gibberish". Unless your approach is to provide almost as many rules as sentences, the chances are that the rules will also generate strings that aren't really English sentences.)

The syntax of programming languages is much simpler. Two types of statements in C are *if* statements and *for* statements.

<statement> → ... | <*if* − statement> | <*for* − statement>

Assuming we are using "if statements" to include both those with *else* and those without, we might write

$$\begin{aligned} <\textit{if}-\text{statement}> &\rightarrow \text{if (<expr>) <statement> } | \\ &\quad\ \text{if (<expr>) <statement> else <statement>} \\ <\textit{for}-\text{statement}> &\rightarrow \text{for (<expr>; <expr>; <expr>) <statement>} \end{aligned}$$

where <expr> is another variable, for which the productions would need to be described.

The logic of a program often requires that a "statement" include several statements. We can define a compound statement as follows:

$$\begin{aligned} &\text{<compound statement>} \rightarrow \text{\{ <statement-sequence> \}} \\ &\text{<statement-sequence>} \rightarrow \Lambda \mid \text{<statement> <statement-sequence>} \end{aligned}$$

A *syntax diagram* such as the one in Figure 4.5 accomplishes the same thing.

Figure 4.5 |

A path through the diagram begins with {, ends with }, and traverses the loop zero or more times.

4.2 | CONTEXT-FREE GRAMMARS: DEFINITIONS AND MORE EXAMPLES

Definition 4.6 Context-Free Grammars

A *context-free grammar* (CFG) is a 4-tuple $G = (V, \Sigma, S, P)$, where V and Σ are disjoint finite sets, $S \in V$, and P is a finite set of formulas of the form $A \rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.

Elements of Σ are called *terminal symbols*, or *terminals*, and elements of V are *variables*, or *nonterminals*. S is the *start* variable, and elements of P are *grammar rules*, or *productions*.

As in Section 4.1, we will reserve the symbol $\rightarrow$ for productions in a grammar, and we will use $\Rightarrow$ for a step in a derivation. The notations

$$\alpha \Rightarrow^n \beta \qquad \text{and} \qquad \alpha \Rightarrow^* \beta$$

refer to a sequence of n steps and a sequence of zero or more steps, respectively, and we sometimes write

$$\alpha \Rightarrow_G \beta \qquad \text{or} \qquad \alpha \Rightarrow_G^n \beta \qquad \text{or} \qquad \alpha \Rightarrow_G^* \beta$$

to indicate explicitly that the steps involve productions in the grammar G. If $G = (V, \Sigma, S, P)$, the first statement means that there are strings α_1, α_2, and γ in $(V \cup \Sigma)^*$ and a production $A \to \gamma$ in P such that

$$\alpha = \alpha_1 A \alpha_2$$
$$\beta = \alpha_1 \gamma \alpha_2$$

In other words, β can be obtained from α in one step by applying the production $A \to \gamma$. Whenever there is no chance of confusion, we will drop the subscript G.

In the situation we have just described, in which $\alpha = \alpha_1 A \alpha_2$ and $\beta = \alpha_1 \gamma \alpha_2$, the formula $\alpha \Rightarrow \beta$ represents a step in a derivation; if our definition of productions allowed $\alpha \to \beta$ to be a production, we might say that the variable A could be replaced by γ, depending on its *context*—i.e., depending on the values of α_1 and α_2. What makes a context-free grammar context-free is that the left side of a production is a single variable and that we may apply the production to any string containing that variable, independent of the context. In Chapter 8 we will consider grammars, and productions, that are not context-free.

Definition 4.7 The Language Generated by a CFG

If $G = (V, \Sigma, S, P)$ is a CFG, the language generated by G is

$$L(G) = \{x \in \Sigma^* \mid S \Rightarrow_G^* x\}$$

A language L is a *context-free language* (CFL) if there is a CFG G with $L = L(G)$.

EXAMPLE 4.8 The Language *AEqB*

Exercises 1.65 and 4.16 both allow us to find context-free grammars for the language $AEqB = \{x \in \{a, b\}^* \mid n_a(x) = n_b(x)\}$ that use only the variable S. In this example we consider a grammar with three variables that is based on a definition involving mutual recursion.

If x is a nonnull string in $AEqB$, then either $x = ay$, where $y \in L_b = \{z \mid n_b(z) = n_a(z) + 1\}$, or $x = by$, where $y \in L_a = \{z \mid n_a(z) = n_b(z) + 1\}$. Let us use the variables A and B to represent L_a and L_b, respectively, and try to find a CFG for $AEqB$ that involves the three variables S, A, and B. So far, the appropriate productions are

$$S \to \Lambda \mid aB \mid bA$$

All we need are productions starting with A and B.

If a string x in L_a starts with a, then the remaining substring is an element of $AEqB$. What if it starts with b? Then $x = by$, where y has two more a's than b's. The crucial observation here is that every string y having two more a's than b's must be the concatenation of two strings y_1 and y_2, each with *one* more a. To see this, think about the relative numbers of a's and b's in each prefix of y; specifically, for each prefix z, let $d(z) = n_a(z) - n_b(z)$.

The shortest prefix of y is Λ, and $d(\Lambda)$ is obviously 0; the longest prefix is y itself, and $d(y) = 2$ by assumption. Each time we add a single symbol to a prefix, the d-value changes (either increases or decreases) by 1. If a quantity starts at 0, changes by 1 at each step, and ends up at 2, it must be 1 at some point! Therefore, for some string y_1 with $d(y_1) = 1$, and some other string y_2, $y = y_1 y_2$, and since $d(y) = d(y_1) + d(y_2) = 2$, $d(y_2)$ must also be 1.

The argument is exactly the same for a string in L_b that starts with a. We conclude that *AEqB* is generated by the CFG with productions

$$S \to \Lambda \mid aB \mid bA$$
$$A \to aS \mid bAA$$
$$B \to bS \mid aBB$$

One feature of this CFG is that if we call A the start variable instead of S, it also works as a grammar generating the language L_a, and similarly for B and L_b.

We can obtain many more examples of context-free languages from the following theorem, which describes three ways of starting with CFLs and constructing new ones.

Theorem 4.9
If L_1 and L_2 are context-free languages over an alphabet Σ, then $L_1 \cup L_2$, $L_1 L_2$, and L_1^* are also CFLs.

Proof
Suppose $G_1 = (V_1, \Sigma, S_1, P_1)$ generates L_1 and $G_2 = (V_2, \Sigma, S_2, P_2)$ generates L_2. We consider the three new languages one at a time, and in the first two cases we assume, by renaming variables if necessary, that G_1 and G_2 have no variables in common.

1. We construct a CFG $G_u = (V_u, \Sigma, S_u, P_u)$ generating $L_1 \cup L_2$, as follows. S_u is a new variable not in either V_1 or V_2,

   $$V_u = V_1 \cup V_2 \cup \{S_u\}$$

   and

   $$P_u = P_1 \cup P_2 \cup \{S_u \to S_1 \mid S_2\}$$

   For every $x \in L_1 \cup L_2$, we can derive x in the grammar G_u by starting with either $S_u \to S_1$ or $S_u \to S_2$ and continuing with the derivation in either G_1 or G_2. On the other hand, if $S_u \Rightarrow^*_{G_U} x$, the first step in any derivation must be either $S_u \Rightarrow S_1$ or $S_u \Rightarrow S_2$, because those are the only productions with left side S_u. In the first case, the remaining steps must involve productions in G_1, because no variables in V_2 can appear, and so $x \in L_1$; similarly, in the second case $x \in L_2$. Therefore, $L(G_u) = L_1 \cup L_2$.

2. To obtain $G_c = (V_c, \Sigma, S_c, P_c)$ generating L_1L_2, we add the single variable S_c to the set $V_1 \cup V_2$, just as in the union, and we let

$$P_c = P_1 \cup P_2 \cup \{S_c \rightarrow S_1S_2\}$$

For a string $x = x_1x_2 \in L_1L_2$, where $x_1 \in L_1$ and $x_2 \in L_2$, a derivation of x in G_c is

$$Sc \Rightarrow S_1S_2 \Rightarrow^* x_1S_2 \Rightarrow^* x_1x_2$$

where the second step (actually a sequence of steps) is a derivation of x_1 in G_1 and the third step is a derivation of x_2 in G_2. Conversely, since the first step in any derivation in G_c must be $S_c \Rightarrow S_1S_2$, every string x derivable from S_c must have the form $x = x_1x_2$, where for each i, x_i is derivable from S_i in G_c. But because $V_1 \cap V_2 = \emptyset$, being derivable from S_i in G_c means being derivable from S_i in G_i, so that $x \in L_1L_2$.

3. We can define a CFG $G^* = (V, \Sigma, S, P)$ generating L_1^* by letting $V = V_1 \cup \{S\}$, where S is a variable not in V_1, and adding productions that generate all possible strings S_1^k, where $k \geq 0$. Let

$$P = P_1 \cup \{S \rightarrow SS_1 \mid \Lambda\}$$

Every string in L_1^* is an element of $L(G_1)^k$ for some $k \geq 0$ and can therefore be obtained from S_1^k; therefore, $L_1^* \subseteq L(G^*)$. On the other hand, every string x in $L(G^*)$ must be derivable from S_1^k for some $k \geq 0$, and we may conclude that $x \in L(G_1)^k \subseteq L(G_1)^*$, because the only productions in P starting with S_1 are the ones in G_1.

The Language $\{a^ib^jc^k \mid j \neq i + k\}$ EXAMPLE 4.10

Let L be the language $\{a^ib^jc^k \mid j \neq i + k\} \subseteq \{a, b, c\}^*$. The form of each element of L might suggest that we try expressing L as the concatenation of three languages L_1, L_2, and L_3, which contain strings of a's, strings of b's, and strings of c's, respectively. This approach doesn't work. Both $a^2b^4c^3$ and $a^3b^4c^2$ are in L; if a^2 were in L_1, b^4 were in L_2, and c^2 were in L_3, then $a^2b^4c^2$, which isn't an element of L, would be in $L_1L_2L_3$.

Instead, we start by noticing that $j \neq i + k$ means that either $j > i + k$ or $j < i + k$, so that L is the union

$$L = L_1 \cup L_2 = \{a^ib^jc^k \mid j > i + k\} \cup \{a^ib^jc^k \mid j < i + k\}$$

There's still no way to express L_1 or L_2 as a threefold concatenation using the approach we tried originally (see Exercise 4.11), but L_1 can be expressed as a concatenation of a slightly different form. Observe that for any natural numbers i and k,

$$a^ib^{i+k}c^k = (a^ib^i)(b^kc^k)$$

and a string in L_1 differs from a string like this only in having at least one extra b in the middle. In other words,

$$L_1 = MNP = \{a^ib^i \mid i \geq 0\}\ \{b^m \mid m > 0\}\ \{b^kc^k \mid k \geq 0\}$$

and it will be easy to find CFGs for the three languages M, N, and P.

L_2 is slightly more complicated. For a string $a^i b^j c^k$ in L_1, knowing that $j > i + k$ tells us in particular that $j > i$; for a string $a^i b^j c^k$ in L_2, such that $j < i + k$, it is helpful to know either how j is related to i or how it is related to k. Let us also split L_2 into a union of two languages:

$$L_2 = L_3 \cup L_4 = \{a^i b^j c^k \mid j < i\} \cup \{a^i b^j c^k \mid i \leq j < i + k\}$$

Now we can use the same approach for L_3 and L_4 as we used for L_1. We can write a string in L_3 as

$$a^i b^j c^k = (a^{i-j})\ (a^j b^j)\ (c^k)$$

where $i - j > 0$, $j \geq 0$, and $k \geq 0$ (and these inequalities are the only constraints on $i - j$, j, and k), and a string in L_4 as

$$a^i b^j c^k = (a^i b^i)\ (b^{j-i} c^{j-i})\ (c^{k-j+i})$$

where i, $j - i$, and $k - j + i$ are natural numbers that are arbitrary except that $i > 0$, $j - i \geq 0$, and $k - j + i > 0$ (i.e., $k + i > j$). It follows that

$$L_3 = QRT = \{a^i \mid i > 0\}\ \{b^i c^i \mid i \geq 0\}\ \{c^i \mid i \geq 0\}$$
$$L_4 = UVW = \{a^i b^i \mid i > 0\}\ \{b^i c^i \mid i \geq 0\}\ \{c^i \mid i > 0\}$$

We have now expressed L as the union of the three languages L_1, L_3, and L_4, and each of these three can be written as the concatenation of three languages. The context-free grammar we are looking for will have productions

$$S \to S_1 \mid S_3 \mid S_4 \qquad S_1 \to S_M S_N S_P \qquad S_3 \to S_Q S_R S_T \qquad S_4 \to S_U S_V S_W$$

as well as productions that start with the nine variables S_M, S_N, ..., S_W. We present productions to take care of the first three of these and leave the last six to you.

$$S_M \to aS_M b \mid \Lambda \qquad S_N \to bS_N \mid b \qquad S_P \to bS_N c \mid \Lambda$$

4.3 REGULAR LANGUAGES AND REGULAR GRAMMARS

The three operations in Theorem 4.9 are the ones involved in the recursive definition of regular languages (Definition 3.1). The "basic" regular languages over an alphabet Σ (the languages $\emptyset$ and $\{\sigma\}$, for every $\sigma \in \Sigma$) are context-free languages. These two statements provide the ingredients that would be necessary for a proof using structural induction that every regular language is a CFL.

EXAMPLE 4.11 A CFG Corresponding to a Regular Expression

Let $L \subseteq \{a, b\}^*$ be the language corresponding to the regular expression

$$bba(ab)^* + (ab + ba^*b)^*ba$$

A literal application of the constructions in Theorem 4.9 would be tedious: the expression $ab + ba^*b$ alone involves all three operations, including three uses of concatenation. We don't really need to introduce separate variables for the languages $\{a\}$ and $\{b\}$, and there are other similar shortcuts to reduce the number of variables in the grammar. (Keep in mind that just as when we constructed an NFA for a given regular expression in Chapter 3, sometimes a literal application of the constructions helps to avoid errors.)

Here we might start by introducing the productions $S \to S_1 \mid S_2$, since the language is a union of two languages L_1 and L_2. The productions

$$S_1 \to S_1ab \mid bba$$

are sufficient to generate L_1. L_2 is complicated enough that we introduce another variable T for the language corresponding to $ab + ba^*b$, and the productions

$$S_2 \to TS_2 \mid ba$$

will then take care of L_2. Finally, the productions

$$T \to ab \mid bUb \qquad U \to aU \mid \Lambda$$

are sufficient to generate the language represented by T. The complete CFG for L has five variables and the productions

$$S \to S_1 \mid S_2 \qquad S_1 \to S_1ab \mid bba \qquad S_2 \to TS_2 \mid ba$$
$$T \to ab \mid bUb \qquad U \to aU \mid \Lambda$$

Not only can every regular language be generated by a context-free grammar, but it can be generated by a CFG of a particularly simple form. To introduce this type of grammar, we consider the finite automaton in Figure 4.12, which accepts the language $\{a, b\}^*\{ba\}$.

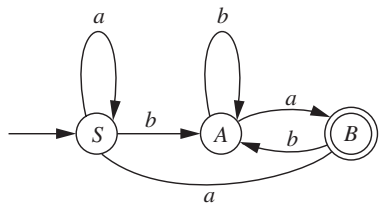
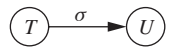


Figure 4.12
An FA accepting $\{a, b\}^*\{ba\}$.

The idea behind the corresponding CFG is that states in the FA correspond to variables in the grammar, and for each transition



the grammar will have a production $T \to \sigma U$. The six transitions in the figure result in the six productions

$$S \to aS \mid bA \qquad A \to bA \mid aB \qquad B \to bA \mid aS$$

The only other productions in the grammar will be Λ-productions (i.e., the right side will be Λ), and so the current string in an uncompleted derivation will consist of a string of terminal symbols and a single variable at the end. The sequence of transitions

$$S \overset{b}{\to} A \overset{b}{\to} A \overset{a}{\to} B \overset{a}{\to} S \overset{b}{\to} A \overset{a}{\to} B$$

corresponds to the sequence of steps

$$S \Rightarrow bA \Rightarrow bbA \Rightarrow bbaB \Rightarrow bbaaS \Rightarrow bbaabA \Rightarrow bbaabaB$$

We can see better now how the correspondence between states and variables makes sense: a state is the way an FA "remembers" how to react to the next input symbol, and the variable at the end of the current string can be thought of as the state of the derivation—the way the derivation remembers, for each possible terminal symbol, how to generate the appropriate little piece of the string that should come next.

The way the string *bbaaba* is finally accepted by the FA is that the current state B is an accepting state, and the way the corresponding derivation terminates is that the variable B at the end of the current string will be replaced by Λ.

Definition 4.13 Regular Grammars

A context-free grammar $G = (V, \Sigma, S, P)$ is *regular* if every production is of the form $A \to \sigma B$ or $A \to \Lambda$, where $A, B \in V$ and $\sigma \in \Sigma$.

Theorem 4.14

For every language $L \subset \Sigma^*$, L is regular if and only if $L = L(G)$ for some regular grammar G.

Proof

If L is a regular language, then for some finite automaton $M = (Q, \Sigma, q_0, A, \delta)$, $L = L(M)$. As in the grammar above for the FA in Figure 4.12, we define $G = (V, \Sigma, S, P)$ by letting V be Q, letting S be the initial state q_0, and letting P be the set containing a production $T \to aU$ for

every transition $\delta(T, a) = U$ in M, and a production $T \to \Lambda$ for every accepting state T of M. G is a regular grammar. It is easy to see, just as in the example, that for every $x = a_1a_2 \ldots a_n$, the transitions on these symbols that start at q_0 end at an accepting state if and only if there is a derivation of x in G; in other words, $L = L(G)$.

In the other direction, if G is a regular grammar with $L = L(G)$, we can reverse the construction to produce $M = (Q, \Sigma, q_0, A, \delta)$. Q is the set of variables of G, q_0 is the start variable, A is the set of states (variables) for which there are Λ-productions in G, and for every production $T \to \sigma U$ there is a transition $T \xrightarrow{\sigma} U$ in M. We cannot expect that M is an ordinary finite automaton, because for some combinations of T and a, there may be either more than one or fewer than one U for which $T \to \sigma U$ is a production. But M is an NFA, and the argument in the first part of the proof is still sufficient; for every string x, there is a sequence of transitions involving the symbols of x that starts at q_0 and ends in an accepting state if and only if there is a derivation of x in G. Therefore, L is regular, because it is the language accepted by the NFA M.

The word *regular* is sometimes used to describe grammars that are slightly different from the ones in Definition 4.13. Grammars in which every production has one of the forms $A \to \sigma B$, $A \to \sigma$, or $A \to \Lambda$ are equivalent to finite automata in the same way that our regular grammars are. Grammars with only the first two of these types of productions generate regular languages, and for every language L, $L - \{\Lambda\}$ can be generated by such a grammar. Similarly, a language L is regular if and only if the set of nonnull strings in L can be generated by a grammar in which all productions have the form $A \to xB$ or $A \to x$, where A and B are variables and x is a nonnull string of terminals. Grammars of this last type are also called *linear*. Some of these variations are discussed in the exercises.

4.4 | DERIVATION TREES AND AMBIGUITY

In most of our examples so far, we have been interested in what strings a context-free grammar generates. As Example 4.2 suggests, it is also useful to consider *how* a string is generated by a CFG. A derivation may provide information about the structure of the string, and if a string has several possible derivations, one may be more appropriate than another.

Just as diagramming a sentence might help to exhibit its grammatical structure, drawing a *tree* to represent a derivation of a string helps to visualize the steps of the derivation and the corresponding structure of the string. In a derivation tree, the root node represents the start variable S. For each interior node N, the portion of the tree consisting of N and its children represents a production $A \to \alpha$ used in the derivation. N represents the variable A, and the children, from left to right, represent the symbols of α. (If the production is $A \to \Lambda$, the node N has a single child representing Λ.) Each leaf node in the tree represents either a terminal symbol

or Λ, and the string being derived is the one obtained by reading the leaf nodes from left to right.

A derivation of a string x in a CFG is a sequence of steps, and a single step is described by specifying the current string, a variable appearing in the string, a particular occurrence of that variable, and the production starting with that variable. We will adopt the phrase *variable-occurrence* to mean a particular occurrence of a particular variable. (For example, if S and A are variables, the string $S + A * S$ contains three variable-occurrences.) Using this terminology, we may say that a step in a derivation is determined by the current string, a particular variable-occurrence in the string, and the production to be applied to that variable-occurrence.

It is easy to see that for each derivation in a CFG, there is exactly one derivation tree. The derivation begins with the start symbol S, which corresponds to the root node of the tree, and the children of that node are determined by the first production in the derivation. At each subsequent step, a production is applied, involving a variable-occurrence corresponding to a node N in the tree; that production determines the portion of the tree consisting of N and its children. For example, the derivation

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + (S) \Rightarrow a + (S * S) \Rightarrow a + (a * S) \Rightarrow a + (a * a)$$

in the CFG for *Expr* in Example 4.2 corresponds to the derivation tree in Figure 4.15.

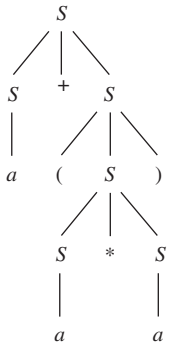


Figure 4.15 A derivation tree for $a + (a * a)$ in Example 4.2.

There are other derivations that also correspond to this tree. Every derivation of the string $a + (a * a)$ must begin

$$S \Rightarrow S + S$$

but now there are two possible ways to proceed, since the next step could involve either of the two occurrences of S. If we chose the rightmost one, so as to obtain $S + (S)$, we would still have two choices for the step after that.

When we said above, "if a string has several possible derivations, one may be more appropriate than another", we were referring, not to derivations that differed in this way, but to derivations corresponding to different derivation trees. Among all the derivations corresponding to the derivation tree in Figure 4.15 (see Exercise 4.30), there are no *essential* differences, but only differences having to do with which occurrence of S we choose for the next step. We could eliminate these choices by agreeing, at every step where the current string has more than one variable-occurrence, to use the *leftmost* one.

Definition 4.16 Leftmost and Rightmost Derivations

A derivation in a context-free grammar is a *leftmost* derivation (LMD) if, at each step, a production is applied to the leftmost variable-occurrence in the current string. A rightmost derivation (RMD) is defined similarly.

Theorem 4.17

If G is a context-free grammar, then for every $x \in L(G)$, these three statements are equivalent:

1. x has more than one derivation tree.
2. x has more than one leftmost derivation.
3. x has more than one rightmost derivation.

Proof

We will show that x has more than one derivation tree if and only if x has more than one LMD, and the equivalence involving rightmost derivations will follow similarly.

If there are two different derivation trees for the string x, each of them has a corresponding leftmost derivation. The two LMDs must be different—otherwise that derivation would correspond to two different derivation trees, and this is impossible for any derivation, leftmost or otherwise.

If there are two different leftmost derivations of x, let the corresponding derivation trees be T_1 and T_2. Suppose that in the first step where the two derivations differ, this step is

$$xA\beta \Rightarrow x\alpha_1\beta$$

in one derivation and

$$xA\beta \Rightarrow x\alpha_2\beta$$

in the other. Here x is a string of terminals, because the derivations are leftmost; A is a variable; and $\alpha_1 \neq \alpha_2$. In both T_1 and T_2 there is a node corresponding to the variable-occurrence A, and the respective portions of the two trees to the left of this node must be identical, because the leftmost derivations have been the same up to this point. These two nodes have different sets of children, and so $T_1 \neq T_2$.

Definition 4.18 Ambiguity in a CFG

A context-free grammar G is *ambiguous* if for at least one $x \in L(G)$, x has more than one derivation tree (or, equivalently, more than one leftmost derivation).

We will return to the algebraic-expression grammar of Example 4.2 shortly, but first we consider an example of ambiguity arising from the definition of if-statements in Example 4.4.

EXAMPLE 4.19 The Dangling *else*

In the C programming language, an if-statement can be defined by these grammar rules:

$$\begin{aligned} S \rightarrow\ & \text{if } (\ E\ )\ S\ \mid \\ & \text{if } (\ E\ )\ S\ \text{ else } S\ \mid \\ & OS \end{aligned}$$

(In our notation, E is short for <expression>, S for <statement>, and OS for <otherstatement>.) A statement in C that illustrates the ambiguity of these rules is

```
if (e1) if (e2) f(); else g();
```

The problem is that although in C the *else* should be associated with the second *if*, as in

```
if (e1) { if (e2) f(); else g(); }
```

there is nothing in these grammar rules to rule out the other interpretation:

```
if (e1) { if (e2) f(); } else g();
```

The two derivation trees in Figure 4.21 show the two possible interpretations of the statement; the correct one is in Figure 4.21a.

There are equivalent grammar rules that allow only the correct interpretation. One possibility is

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow \text{if } (\ E\ )\ S_1\ \text{ else } S_1 \mid OS \\ S_2 &\rightarrow \text{if } (\ E\ )\ S\ \mid \\ &\qquad \text{if } (\ E\ )\ S_1\ \text{ else } S_2 \end{aligned}$$

These rules generate the same strings as the original ones and are unambiguous. We will not prove either fact, but you can see how the second might be true. The variable S_1 represents a statement in which every *if* is matched by a corresponding *else*, and every statement derived from S_2 contains at least one unmatched *if*. The only variable appearing before *else* in these rules is S_1; because the *else* cannot match any of the *if* s in the statement derived from S_1, it must match the *if* that appeared at the same time it did.

EXAMPLE 4.20 Ambiguity in the CFG for *Expr* in Example 4.2

In the grammar G in Example 4.2, with productions

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

the string $a + a * a$ has the two leftmost derivations

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * a$$
$$S \Rightarrow S * S \Rightarrow S + S * S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * a$$

which correspond to the derivation trees in Figure 4.22.

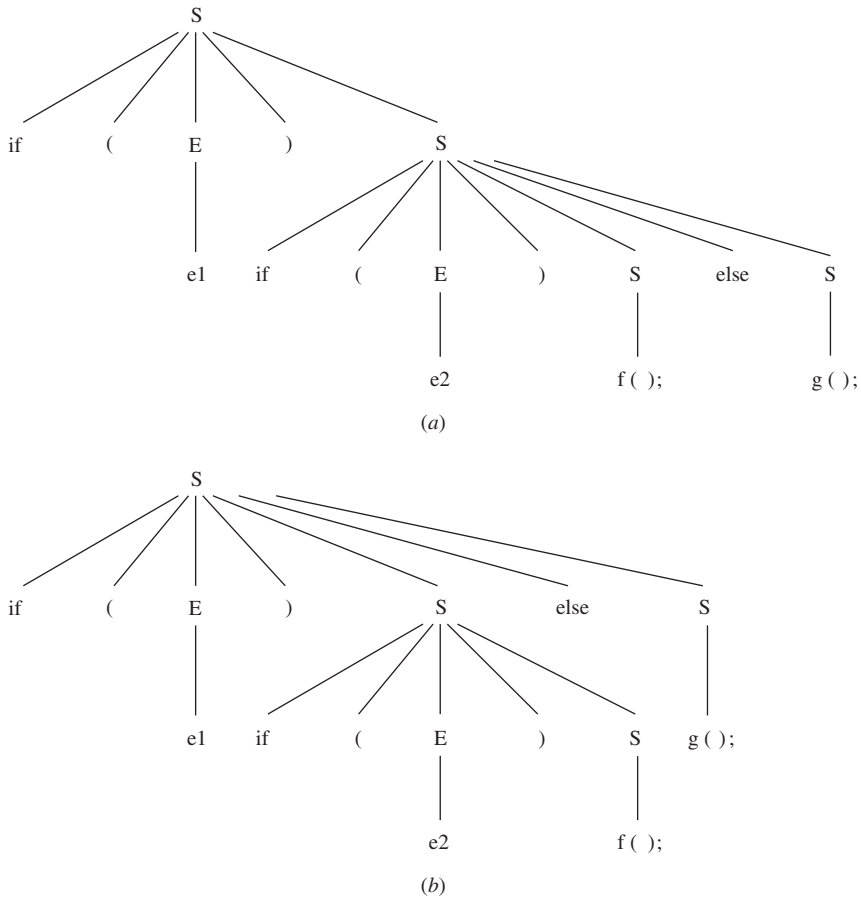


Figure 4.21 |
Two possible interpretations of a dangling *else*.

We observed in Example 4.2 that the first of these two LMDs (or the first of the two derivation trees) matches the interpretation of $a + a * a$ as a sum, rather than as a product; one reason for the ambiguity is that the grammar allows either of the two operations $+$ and $*$ to be given higher precedence. Just as adding braces to the C statements in Example 4.19 allowed only the correct interpretation, the addition of parentheses in this expression, to obtain $a + (a * a)$, has the same effect; the only leftmost derivation of this string is

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + (S) \Rightarrow a + (S * S) \Rightarrow a + (a * S) \Rightarrow a + (a * a)$$

Another rule in algebra that is not enforced by this grammar is the rule that says operations of the same precedence are performed left-to-right. For this reason, both the expressions $a + a + a$ and $a * a * a$ also illustrate the ambiguity of the grammar. The first expression has the (correct) LMD

$$S \Rightarrow S + S \Rightarrow S + S + S \Rightarrow a + S + S \Rightarrow a + a + S \Rightarrow a + a + a$$

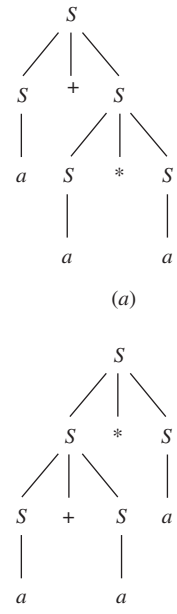


Figure 4.22 |
Two derivation trees for $a + a * a$ in Example 4.2.

corresponding to the interpretation $(a + a) + a$, as well as the LMD

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S + S \Rightarrow a + a + S \Rightarrow a + a + a$$

corresponding to the other way of grouping the terms.

In order to find an unambiguous CFG G_1 generating this language, we look for productions that do not allow any choice regarding either the precedence of the two operators or the order in which operators of the same precedence are performed. To some extent, we can ignore the parentheses at least temporarily. Let S_1 be the start symbol of G_1.

Saying that multiplication should have higher precedence than addition means that when there is any doubt, an expression should be treated as a sum rather than a product. For this reason, we concentrate first on productions that will give us sums of terms. The problem with $S_1 \to S_1 + S_1$ is that if we are attempting to derive a sum of three or more terms, there are too many choices as to how many terms will come from the first S_1 and how many from the second. Saying that additions are performed left-to-right, or that $+$ "associates to the left", suggests that we think of a sum of n terms as another sum plus *one* additional term. We try

$$S_1 \to S_1 + T \mid T$$

where T represents a term—that is, an expression that may be part of a sum but is not itself a sum.

It is probably clear already that we would no longer want $S_1 \to S_1 * S_1$, even if it didn't cause the same problems as $S_1 \to S_1 + S_1$; we don't want to say that an expression can be either a sum or a product, because that was one of the sources of ambiguity. Instead, we say that *terms* can be products. Products of what? *Factors*. This suggests

$$T \to T * F \mid F$$

where, in the same way that $S_1 + T$ is preferable to $T + S_1$, $T * F$ is preferable to $F * T$. We now have a hierarchy with three levels: expressions are sums of terms, and terms are products of factors. Furthermore, we have incorporated the rule that each operator associates to the left.

What remains is to deal with a and expressions with parentheses. The productions $S_1 \to T$ and $T \to F$ allow us to have an expression with a single term and a term with a single factor; thus, although a by itself is a valid expression, it is best to call it a factor, because it is neither a product nor a sum. Similarly, an expression in parentheses should also be considered a factor. What is in the parentheses should be an expression, because once we introduce a pair of parentheses we can start all over with what's inside.

The CFG that we have now obtained is $G_1 = (V, \Sigma, S_1, P)$, where $V = \{S_1, T, F\}$, $\Sigma = \{a, +, *, (,)\}$, and P contains the productions

$$\begin{aligned} S_1 &\to S_1 + T \mid T \\ T &\to T * F \mid F \\ F &\to a \mid (S_1) \end{aligned}$$

Both halves of the statement $L(G) = L(G_1)$ can be proved by mathematical induction on the length of a string. The details, particularly for the statement $L(G) \subseteq L(G_1)$, are somewhat involved and are left to the exercises.

The rest of this section is devoted to proving that G_1 is unambiguous. In the proof, for a string $x \in L(G_1)$, it is helpful to talk about a symbol in x that is *within parentheses*, and we want this to mean that it lies between the left and right parentheses that are introduced in a single step of a derivation of x. One might ask, however, whether there can be two derivations of x, and a symbol that has this property with respect to one of the two but not the other. The purpose of Definition 4.23 and Theorem 4.24 is to establish that this formulation is a satisfactory one and that "within parentheses" doesn't depend on which derivation we are using. In the proof of Theorem 4.24, we will use the result obtained in Example 1.25, that balanced strings of parentheses are precisely the strings with equal numbers of left and right parentheses and no prefix having more right than left parentheses.

Definition 4.23 The Mate of a Left Parenthesis in a Balanced String

The *mate* of a left parenthesis in a balanced string is the first right parenthesis following it for which the string starting with the left and ending with the right has equal numbers of left and right parentheses. (Every left parenthesis in a balanced string has a mate—see Exercise 4.41.)

Theorem 4.24

For every $x \in L(G_1)$, every derivation of x in G_1, and every step of this derivation in which two parentheses are introduced, the right parenthesis is the mate of the left.

Proof

Suppose $x = x_1({}_0z)_0x_2$, where ${}_0$ and $)_0$ are two occurrences of parentheses that are introduced in the same step in some derivation of x. Then $F \Rightarrow ({}_0S_1)_0 \Rightarrow^* ({}_0z)_0$. It follows that z is a balanced string, and by the definition of "mate", the mate of ${}_0$ cannot appear after $)_0$.

If it appears before $)_0$, however, then $z = z_1)_1z_2$, for some strings z_1 and z_2, where $)_1$ is the mate of ${}_0$ and z_1 has equal numbers of left and right parentheses. This implies that the prefix $z_1)_1$ of z has more right parentheses than left, which is impossible if z is balanced. Therefore, $)_0$ is the mate of ${}_0$.

Definition 4.23 and Theorem 4.24 allow us to say that "between the two parentheses produced in a single step of a derivation" is equivalent to "between some left parenthesis and its mate", so that either of these can be taken as the definition of "within parentheses".

Theorem 4.25

The context-free grammar G_1 with productions

$$S_1 \rightarrow S_1 + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow a \mid (S_1)$$

is unambiguous.

Proof

We prove the following statement, which implies the result: For every x derivable from one of the variables S_1, T, or F in G_1, x has only one leftmost derivation from that variable. The proof is by mathematical induction on $|x|$. The basis step, for $x = a$, is easy, because for each of the three variables, a has exactly one derivation from that variable.

The induction hypothesis is that $k \geq 1$ and that for every y derivable from S_1, T, or F for which $|y| \leq k$, y has only one leftmost derivation from that variable. We wish to show that the same is true for a string $x \in L(G_1)$ with $|x| = k + 1$.

We start with the case in which x contains at least one occurrence of $+$ that is not within parentheses. Because the only occurrences of $+$ in strings derivable from T or F *are* within parentheses, every derivation of x must begin $S_1 \Rightarrow S_1 + T$, and the occurrence of $+$ in this step is the last one in x that is not within parentheses. Every leftmost derivation of x from S_1 must then have the form

$$S_1 \Rightarrow S_1 + T \Rightarrow^* y + T \Rightarrow^* y + z$$

where the last two steps represent leftmost derivations of y from S_1 and z from T, respectively, and the $+$ is still the last one not within parentheses. According to the induction hypothesis, y has only one leftmost derivation from S_1, and z has only one from T; therefore, x has only one LMD from S_1.

Next we consider the case in which every occurrence of $+$ in x is within parentheses but there is at least one occurrence of $*$ not within parentheses. Because x cannot be derived from F, every derivation of x from S_1 must begin $S_1 \Rightarrow T \Rightarrow T * F$, and every derivation from T must begin $T \Rightarrow T * F$. In either case, this occurrence of $*$ must be the last one in x not within parentheses. As before, the subsequent steps of every LMD must be

$$T * F \Rightarrow^* y * F \Rightarrow^* y * z$$

in which the derivations of y from T and z from F are both leftmost. The induction hypothesis tells us that there is only one way for each of these derivations to proceed, and that there is only one leftmost derivation of x from S_1 or T.

Finally, suppose that every occurrence of $+$ or $*$ in x is within parentheses. Then x can be derived from any of the three variables, but every derivation from S_1 begins

$$S_1 \Rightarrow T \Rightarrow F \Rightarrow (S_1)$$

and every derivation from T or F begins the same way but with the first one or two steps omitted. Therefore, $x = (y)$, where $S_1 \Rightarrow^* y$. By the induction hypothesis, y has only one LMD from S_1, and it follows that x has only one from each of the three variables.

4.5 SIMPLIFIED FORMS AND NORMAL FORMS

Questions about the strings generated by a context-free grammar G are sometimes easier to answer if we know something about the form of the productions. Sometimes this means knowing that certain types of productions never occur, and sometimes it means knowing that every production has a certain simple form. For example, suppose we want to know whether a string x is generated by G, and we look for an answer by trying all derivations with one step, then all derivations with two steps, and so on. If we don't find a derivation that produces x, how long do we have to keep trying?

The number t of terminals in the current string of a derivation cannot decrease at any step. If G has no Λ-productions (of the form $A \rightarrow \Lambda$), then the length l of the current string can't decrease either, which means that the sum $t + l$ can't decrease. If we also know that G has no "unit productions" (of the form $A \rightarrow B$, where A and B are variables), then the sum must increase, because every step either adds a terminal or increases the number of variables. A derivation of x starts with S, for which $l + t = 1$, and ends with x, for which $l + t = 2|x|$; therefore, no derivation has more than $2|x| - 1$ steps. If we try all derivations with this many steps or fewer and don't find one that generates x, we may conclude that $x \notin L(G)$.

In this section we show how to modify an arbitrary CFG G so that the modified grammar has no productions of either of these types but still generates $L(G)$, except possibly for the string Λ. We conclude by showing how to modify G further (eliminating both these types of productions is the first step) so as to obtain a CFG that is still essentially equivalent to G but is in *Chomsky normal form*, so that every production has one of two very simple forms.

A simple example will suggest the idea of the algorithm to eliminate Λ-productions. Suppose one of the productions in G is

$$A \rightarrow BCDCB$$

and that from both the variables B and C, Λ can be derived (in one or more steps), as well as other nonnull strings of terminals. Once the algorithm has been applied, the steps that replace B and C by Λ will no longer be possible, but we must still be able to get all the nonnull strings from A that we could have gotten using these

steps. This means that we should retain the production $A \to BCDCB$, but we should add $A \to CDCB$ (because we could have replaced the first B by Λ and the other occurrences of B and C by nonnull strings), $A \to DCB$, $A \to CDB$, and so on—every one of the fifteen productions that result from leaving out one or more of the four occurrences of B and C in the right-hand string.

The necessary first step, of course, is to identify the variables like B and C from which Λ can be derived. We will refer to these as *nullable* variables. It is easy to see that the set of nullable variables can be obtained using the following recursive definition.

Definition 4.26 A Recursive Definition of the Set of Nullable Variables of *G*

1. Every variable A for which there is a production $A \to \Lambda$ is nullable.
2. If $A_1, A_2, \ldots, A_k$ are nullable variables (not necessarily distinct), and
$$B \to A_1 A_2 \ldots A_k$$
is a production, then B is nullable.

This definition leads immediately to an algorithm for identifying the nullable variables (see Example 1.21).

Theorem 4.27

For every context-free grammar $G = (V, \Sigma, S, P)$, the following algorithm produces a CFG $G_1 = (V, \Sigma, S, P_1)$ having no Λ-productions and satisfying $L(G_1) = L(G) - \{\Lambda\}$.

1. Identify the nullable variables in V, and initialize P_1 to P.
2. For every production $A \to \alpha$ in P, add to P_1 every production obtained from this one by deleting from α one or more variable-occurrences involving a nullable variable.
3. Delete every Λ-production from P_1, as well as every production of the form $A \to A$.

Proof

It is obvious that G_1 has no Λ-productions. We show that for every $A \in V$ and every nonnull $x \in \Sigma^*$, $A \Rightarrow_G^* x$ if and only if $A \Rightarrow_{G_1}^* x$.

First we show, using mathematical induction on n, that for every $n \geq 1$, every $A \in V$, and every $x \in \Sigma^*$ with $x \neq \Lambda$, if $A \Rightarrow_G^n x$, then $A \Rightarrow_{G_1}^* x$. For the basis step, suppose $A \Rightarrow_G^1 x$. Then $A \to x$ is a production in P, and since $x \neq \Lambda$, this production is also in P_1.

Suppose that $k \geq 1$ and that for every $n \leq k$, every variable A, and every nonnull $x \in \Sigma^*$ for which $A \Rightarrow_G^n x$, $A \Rightarrow_{G_1}^* x$. Now suppose that

$A \Rightarrow_G^{k+1} x$, for some $x \in \Sigma^* - \{\Lambda\}$. Let the first step in some $(k+1)$-step derivation of x from A in G be

$$A \Rightarrow X_1 X_2 \ldots X_m$$

where each X_i is either a variable or a terminal. Then $x = x_1 x_2 \ldots x_m$, where for each i, either $x_i = X_i$ (a terminal), or x_i is a string derivable from the variable X_i in k or fewer steps. When all the (nullable) variables X_i for which the corresponding x_i is Λ are deleted from the string $X_1 X_2 \ldots X_m$, there are still some X_i's left, because $x \neq \Lambda$, and so the resulting production is an element of P_1. Furthermore, the induction hypothesis tells us that for each variable X_i remaining, $X_i \Rightarrow_{G_1}^* x_i$. Therefore, $A \Rightarrow_{G_1}^* x$.

For the converse, we show, using mathematical induction on n, that for every $n \geq 1$, every variable A, and every $x \in \Sigma^*$, if $A \Rightarrow_{G_1}^n x$, then $A \Rightarrow_G^* x$. If $A \Rightarrow_{G_1}^1 x$, then $A \rightarrow x$ is a production in P_1. It follows that $A \rightarrow \alpha$ is a production in P, where α is a string from which x can be obtained by deleting zero or more occurrences of nullable variables. Therefore, $A \Rightarrow_G^* x$, because we can begin a derivation with the production $A \rightarrow \alpha$ and continue by deriving Λ from each of the nullable variables that was deleted from α to obtain x.

Suppose that $k \geq 1$, that for every $n \leq k$, every A, and every string x with $A \Rightarrow_{G_1}^n x$, $A \Rightarrow_G^* x$, and that $A \Rightarrow_{G_1}^{k+1} x$. Again, let the first step of some $(k+1)$-step derivation of x from A in G_1 be

$$A \Rightarrow X_1 X_2 \ldots X_m$$

Then $x = x_1 x_2 \ldots x_m$, where for each i, either $x_i = X_i$ or x_i is a string derivable from the variable X_i in the grammar G_1 in k or fewer steps. By the induction hypothesis, $X_i \Rightarrow_G^* x_i$ for each i. By definition of G_1, there is a production $A \rightarrow \alpha$ in P so that $X_1 X_2 \ldots X_m$ can be obtained from α by deleting some variable-occurrences involving nullable variables. This implies that $A \Rightarrow_G^* X_1 X_2 \ldots X_m$, and therefore that we can derive x from A in G, by first deriving $X_1 \ldots X_m$ and then deriving each x_i from the corresponding X_i.

The procedure we use to eliminate unit productions from a CFG is rather similar. We first identify pairs of variables (A, B) for which $A \Rightarrow^* B$ (not only those for which $A \rightarrow B$ is a production); then, for each such pair (A, B), and each nonunit production $B \rightarrow \alpha$, we add the production $A \rightarrow \alpha$.

If we make the simplifying assumption that we have already eliminated Λ-productions from the grammar, then a sequence of steps by which $A \Rightarrow^* B$ involves only unit productions. This allows us, for a variable A, to formulate the following recursive definition of an "A-derivable" variable (a variable B different from A for which $A \Rightarrow^* B$):

1. If $A \to B$ is a production and $B \neq A$, then B is A-derivable.
2. If C is A-derivable, $C \to B$ is a production, and $B \neq A$, then B is A-derivable.
3. No other variables are A-derivable.

Theorem 4.28

For every context-free grammar $G = (V, \Sigma, S, P)$ without Λ-productions, the CFG $G_1 = (V, \Sigma, S, P_1)$ produced by the following algorithm generates the same language as G and has no unit productions.

1. Initialize P_1 to be P, and for each $A \in V$, identify the A-derivable variables.
2. For every pair (A, B) of variables for which B is A-derivable, and every nonunit production $B \to \alpha$, add the production $A \to \alpha$ to P_1.
3. Delete all unit productions from P_1.

Proof

The proof that $L(G_1) = L(G)$ is a straightforward induction proof and is omitted.

Definition 4.29 Chomsky Normal Form

A context-free grammar is said to be in *Chomsky normal form* if every production is of one of these two types:

$$A \to BC \quad \text{(where } B \text{ and } C \text{ are variables)}$$
$$A \to \sigma \quad \text{(where } \sigma \text{ is a terminal symbol)}$$

Theorem 4.30

For every context-free grammar G, there is another CFG G_1 in Chomsky normal form such that $L(G_1) = L(G) - \{\Lambda\}$.

Proof

We describe briefly the algorithm that can be used to construct the grammar G_1, and it is not hard to see that it generates the same language as G, except possibly for the string Λ.

The first step is to apply the algorithms presented in Theorems 4.6 and 4.7 to eliminate Λ-productions and unit productions. The second step is to introduce for every terminal symbol σ a variable X_σ and a production $X_\sigma \to \sigma$, and in every production whose right side has at least two symbols, to replace every occurrence of a terminal by the corresponding

variable. At this point, every production looks like either $A \to \sigma$ or

$$A \to B_1 B_2 \ldots B_k$$

where the B_i's are variables and $k \geq 2$.

The last step is to replace each production having more than two variable-occurrences on the right by an equivalent set of productions, each of which has exactly two variable-occurrences. This step is described best by an example. The production

$$A \to BACBDCBA$$

would be replaced by

$$A \to BY_1 \quad Y_1 \to AY_2 \quad Y_2 \to CY_3 \quad Y_3 \to BY_4 \quad Y_4 \to DY_5$$
$$Y_5 \to CY_6 \quad Y_6 \to BA$$

where the new variables $Y_1, \ldots, Y_6$ are specific to this production and are not used anywhere else.

Converting a CFG to Chomsky Normal Form

EXAMPLE 4.31

This example will illustrate all the algorithms described in this section: to identify nullable variables, to eliminate Λ-productions, to identify A-derivable variables for each A, to eliminate unit productions, and to convert to Chomsky normal form. Let G be the context-free grammar with productions

$$S \to TU \mid V$$
$$T \to aTb \mid \Lambda$$
$$U \to cU \mid \Lambda$$
$$V \to aVc \mid W$$
$$W \to bW \mid \Lambda$$

which can be seen to generate the language $\{a^i b^j c^k \mid i = j \text{ or } i = k\}$.

1. (Identifying nullable variables) The variables T, U, and W are nullable because they are involved in Λ-productions; V is nullable because of the production $V \to W$; and S is also, either because of the production $S \to TU$ or because of $S \to V$. So all the variables are!
2. (Eliminating Λ-productions) Before the Λ-productions are eliminated, the following productions are added:

   $$S \to T \qquad S \to U \qquad T \to ab \qquad U \to c \qquad V \to ac \qquad W \to b$$

   After eliminating Λ-productions, we are left with

   $$S \to TU \mid T \mid U \mid V \qquad T \to aTb \mid ab \qquad U \to cU \mid c$$
   $$V \to aVc \mid ac \mid W \qquad W \to bW \mid b$$

3. (Identifying A-derivable variables, for each A) The S-derivable variables obviously include T, U, and V, and they also include W because of the production $V \to W$. The V-derivable variable is W.

4. (Eliminating unit productions) We add the productions

$$S \to aTb \mid ab \mid cU \mid c \mid aVc \mid ac \mid bW \mid b \qquad V \to bW \mid b$$

before eliminating unit productions. At this stage, we have

$$\begin{aligned} S &\to TU \mid aTb \mid ab \mid cU \mid c \mid aVc \mid ac \mid bW \mid b \\ T &\to aTb \mid ab \\ U &\to cU \mid c \\ V &\to aVc \mid ac \mid bW \mid b \\ W &\to bW \mid b \end{aligned}$$

5. (Converting to Chomsky normal form) We replace a, b, and c by X_a, X_b, and X_c, respectively, in productions whose right sides are not single terminals, obtaining

$$\begin{aligned} S &\to TU \mid X_aTX_b \mid X_aX_b \mid X_cU \mid c \mid X_aVX_c \mid X_aX_c \mid X_bW \mid b \\ T &\to X_aTX_b \mid X_aX_b \\ U &\to X_cU \mid c \\ V &\to X_aVX_c \mid X_aX_c \mid X_bW \mid b \\ W &\to X_bW \mid b \end{aligned}$$

This grammar fails to be in Chomsky normal form only because of the productions $S \to X_aTX_b$, $S \to X_aVX_c$, $T \to X_aTX_b$, and $V \to X_aVX_c$. When we take care of these as described above, we obtain the final CFG G_1 with productions

$$\begin{aligned} S &\to TU \mid X_aY_1 \mid X_aX_b \mid X_cU \mid c \mid X_aY_2 \mid X_aX_c \mid X_bW \mid b \\ Y_1 &\to TX_b \\ Y_2 &\to VX_c \\ T &\to X_aY_3 \mid X_aX_b \\ Y_3 &\to TX_b \\ U &\to X_cU \mid c \\ V &\to X_aY_4 \mid X_aX_c \mid X_bW \mid b \\ Y_4 &\to VX_c \\ W &\to X_bW \mid b \end{aligned}$$

(We obviously don't need both Y_1 and Y_3, and we don't need both Y_2 and Y_4, so we could simplify G_1 slightly.)

EXERCISES

4.1. In each case below, say what language (a subset of $\{a, b\}^*$) is generated by the context-free grammar with the indicated productions.

a. $S \to aS \mid bS \mid \Lambda$

b. $S \to SS \mid bS \mid a$

c. $S \to SaS \mid b$

d. $S \to SaS \mid b \mid \Lambda$

e. $S \to TT \qquad T \to aT \mid Ta \mid b$

f. $S \to aSa \mid bSb \mid aAb \mid bAa \qquad A \to aAa \mid bAb \mid a \mid b \mid \Lambda$

g. $S \to aT \mid bT \mid \Lambda \qquad T \to aS \mid bS$

h. $S \to aT \mid bT \qquad T \to aS \mid bS \mid \Lambda$

4.2. Find a context-free grammar corresponding to the "syntax diagram" in Figure 4.32.

4.3. In each case below, find a CFG generating the given language.

a. The set of odd-length strings in $\{a, b\}^*$ with middle symbol a.

b. The set of even-length strings in $\{a, b\}^*$ with the two middle symbols equal.

c. The set of odd-length strings in $\{a, b\}^*$ whose first, middle, and last symbols are all the same.

4.4. In both parts below, the productions in a CFG G are given. In each part, show first that for every string $x \in L(G)$, $n_a(x) = n_b(x)$; then find a string $x \in \{a, b\}^*$ with $n_a(x) = n_b(x)$ that is not in $L(G)$.

a. $S \to SabS \mid SbaS \mid \Lambda$

b. $S \to aSb \mid bSa \mid abS \mid baS \mid Sab \mid Sba \mid \Lambda$

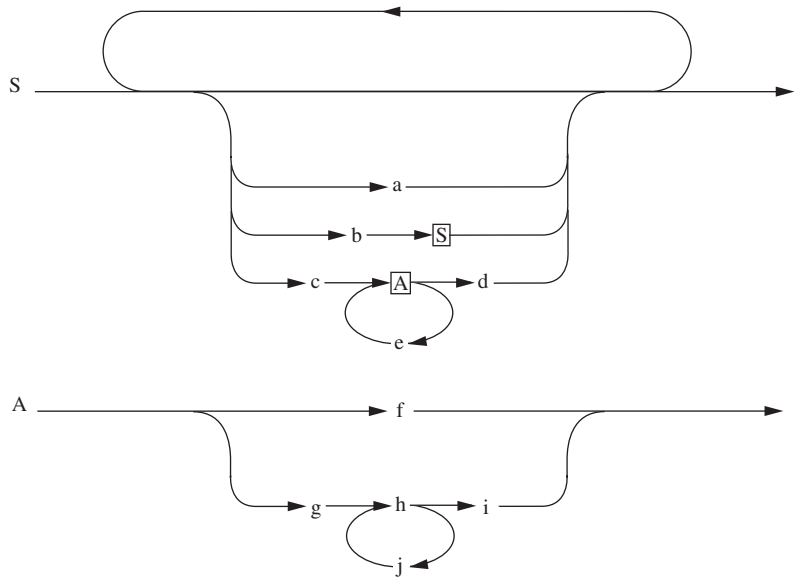


Figure 4.32

4.5. Consider the CFG with productions

$$S \to aSbScS \mid aScSbS \mid bSaScS \mid bScSaS \mid cSaSbS \mid cSbSaS \mid \Lambda$$

Does this generate the language $\{x \in \{a, b, c\}^* \mid n_a(x) = n_b(x) = n_c(x)\}$? Prove your answer.

4.6. Show that the language of all nonpalindromes over $\{a, b\}$ (see Example 4.3) cannot be generated by any CFG in which $S \to aSa \mid bSb$ are the only productions with variables on the right side.

4.7. Describe the language generated by the CFG with productions

$$S \to ST \mid \Lambda \qquad T \to aS \mid bT \mid b$$

Give an induction proof that your answer is correct.

4.8. What language over $\{a, b\}$ does the CFG with productions

$$S \to aaS \mid bbS \mid Saa \mid Sbb \mid abSab \mid abSba \mid baSba \mid baSab \mid \Lambda$$

generate? Prove your answer.

4.9. Suppose that $G_1 = (V_1, \{a, b\}, S_1, P_1)$ and $G_2 = (V_2, \{a, b\}, S_2, P_2)$ are CFGs and that $V_1 \cap V_2 = \emptyset$.

a. It is easy to see that no matter what G_1 and G_2 are, the CFG $G_u = (V_u, \{a, b\}, S_u, P_u)$ defined by $V_u = V_1 \cup V_2$, $S_u = S_1$, and $P_u = P_1 \cup P_2 \cup \{S_1 \to S_2\}$ generates every string in $L(G_1) \cup L(G_2)$. Find grammars G_1 and G_2 (you can use $V_1 = \{S_1\}$ and $V_2 = \{S_2\}$) and a string $x \in L(G_u)$ such that $x \notin L(G_1) \cup L(G_2)$.

b. As in part (a), the CFG $G_c = (V_c, \{a, b\}, S_c, P_c)$ defined by $V_c = V_1 \cup V_2$, $S_c = S_1$, and $P_c = P_1 \cup P_2 \cup \{S_1 \to S_1S_2\}$ generates every string in $L(G_1)L(G_2)$. Find grammars G_1 and G_2 (again with $V_1 = \{S_1\}$ and $V_2 = \{S_2\}$) and a string $x \in L(G_c)$ such that $x \notin L(G_1)L(G_2)$.

c. The CFG $G^* = (V, \{a, b\}, S, P)$ defined by $V = V_1$, $S = S_1$, and $P = P_1 \cup \{S_1 \to S_1S_1 \mid \Lambda\}$ generates every string in $L(G_1)^*$. Find a grammar G_1 with $V_1 = \{S_1\}$ and a string $x \in L(G^*)$ such that $x \notin L(G)^*$.

4.10. Find context-free grammars generating each of the languages below.

a. $\{a^i b^j \mid i \leq j\}$

b. $\{a^i b^j \mid i < j\}$

c. $\{a^i b^j \mid j = 2i\}$

d. $\{a^i b^j \mid i \leq j \leq 2i\}$

e. $\{a^i b^j \mid j \leq 2i\}$

f. $\{a^i b^j \mid j < 2i\}$

4.11. a. Show that the language $L = \{a^i b^j c^k \mid j > i + k\}$ cannot be written in the form $L = L_1L_2L_3$, where L_1, L_2, and L_3 are subsets of $\{a\}^*$, $\{b\}^*$, and $\{c\}^*$, respectively.

b. Show the same thing for the language $L = \{a^i b^j c^k \mid j < i + k\}$.

4.12. Find a context-free grammar generating the language $\{a^i b^j c^k \mid i \neq j + k\}$.

4.13. †Find context-free grammars generating each of these languages, and prove that your answers are correct.

a. $\{a^i b^j \mid i \leq j \leq 3i/2\}$

b. $\{a^i b^j \mid i/2 \leq j \leq 3i/2\}$

4.14. Let L be the language generated by the CFG with productions

$$S \to aSb \mid ab \mid SS$$

Show that no string in L begins with abb.

4.15. Show using mathematical induction that every string produced by the context-free grammar with productions

$$S \to a \mid aS \mid bSS \mid SSb \mid SbS$$

has more a's than b's.

4.16. Prove that the CFG with productions $S \to aSbS \mid aSbS \mid \Lambda$ generates the language $L = \{x \in \{a, b\}^* \mid n_a(x) = n_b(x)\}$.

4.17. †Show that the CFG with productions

$$S \to aSaSbS \mid aSbSaS \mid bSaSaS \mid \Lambda$$

generates the language $\{x \in \{a, b\}^* \mid n_a(x) = 2n_b(x)\}$.

4.18. †Show that the following CFG generates the language $\{x \in \{a, b\}^* \mid n_a(x) = 2n_b(x)\}$.

$$S \to SS \mid bTT \mid TbT \mid TTb \mid \Lambda \qquad T \to aS \mid SaS \mid Sa \mid a$$

4.19. Let G be the CFG with productions $S \to a \mid aS \mid bSS \mid SSb \mid SbS$. Show that every x in $\{a, b\}^*$ with $n_a(x) > n_b(x)$ is an element of $L(G)$.

4.20. Let G be the context-free grammar with productions $S \to SaT \mid \Lambda$ $T \to TbS \mid \Lambda$. Show using mathematical induction that $L(G)$ is the language of all strings in $\{a, b\}^*$ that don't start with b. One direction is easy. For the other direction, it might be easiest to prove two statements simultaneously: (i) every string that doesn't start with b can be derived from S; (ii) every string that doesn't start with a can be derived from T.

4.21. Definition 3.1 and Theorem 4.9 provide the ingredients for a structural-induction proof that every regular language is a CFL. Give the proof.

4.22. Show that if G is a context-free grammar in which every production has one of the forms $A \to aB$, $A \to a$, and $A \to \Lambda$ (where A and B are variables and a is a terminal), then $L(G)$ is regular. Suggestion: construct an NFA accepting $L(G)$, in which there is a state for each variable in G and one additional state F, the only accepting state.

4.23. Suppose $L \subseteq \Sigma^*$. Show that L is regular if and only if $L = L(G)$ for some context-free grammar G in which every production is either of the form $A \to Ba$ or of the form $A \to \Lambda$, where A and B are variables and a

is a terminal. (The grammars in Definition 4.13 are sometimes called *right-regular* grammars, and the ones in this exercise are sometimes called left-regular.)

4.24. Let us call a context-free grammar G a grammar of type R if every production has one of the three forms $A \to aB$, $A \to Ba$, or $A \to \Lambda$ (where A and B are variables and a is a terminal). Every regular language can be generated by a grammar of type R. Is every language that is generated by a grammar of type R regular? If so, give a proof; if not, find a counterexample.

4.25. Show that for a language $L \subseteq \Sigma^*$, the following statements are equivalent.

a. L is regular.

b. L can be generated by a grammar in which all productions are either of the form $A \to xB$ or of the form $A \to \Lambda$ (where A and B are variables and $x \in \Sigma^*$).

c. L can be generated by a grammar in which all productions are either of the form $A \to Bx$ or of the form $A \to \Lambda$ (where A and B are variables and $x \in \Sigma^*$).

4.26. In each part, draw an NFA (which might be an FA) accepting the language generated by the CFG having the given productions.

a. $S \to aA \mid bC \qquad A \to aS \mid bB \qquad B \to aC \mid bA$
$C \to aB \mid bS \mid \Lambda$

b. $S \to bS \mid aA \mid \Lambda \qquad A \to aA \mid bB \mid b \qquad B \to bS$

4.27. Find a regular grammar generating the language $L(M)$, where M is the FA shown in Figure 4.33.

4.28. Draw an NFA accepting the language generated by the grammar with productions

$$S \to abA \mid bB \mid aba$$
$$A \to b \mid aB \mid bA$$
$$B \to aB \mid aA$$

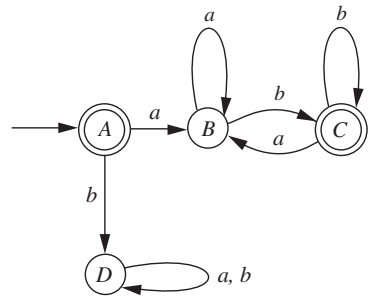


Figure 4.33 |

4.29. Each of the following grammars, though not regular, generates a regular language. In each case, find a regular grammar generating the language.

a. $S \to SSS \mid a \mid ab$

b. $S \to AabB \qquad A \to aA \mid bA \mid \Lambda \qquad B \to Bab \mid Bb \mid ab \mid b$

c. $S \to AAS \mid ab \mid aab \qquad A \to ab \mid ba \mid \Lambda$

d. $S \to AB \qquad A \to aAa \mid bAb \mid a \mid b \qquad B \to aB \mid bB \mid \Lambda$

e. $S \to AA \mid B \qquad A \to AAA \mid Ab \mid bA \mid a \qquad B \to bB \mid \Lambda$

4.30. a. Write the rightmost derivation of the string $a + (a * a)$ corresponding to the derivation tree in Figure 4.15.

b. How many distinct derivations (not necessarily leftmost or rightmost) does the string $a + (a * a)$ have in the CFG with productions $S \to a \mid S + S \mid S * S \mid (S)$?

4.31. Again we consider the CFG in Example 4.2, with productions $S \to a \mid S + S \mid S * S \mid (S)$.

a. How many distinct derivation trees does the string $a + (a * a)/a - a$ have in this grammar?

b. How many derivation trees are there for the string $(a + (a + a)) + (a + a)$?

4.32. In the CFG in the previous exercise, suppose we define n_i to be the number of distinct derivation trees for the string $a + a + \ldots + a$ in which there are i a's. Then $n_1 = n_2 = 1$.

a. Find a recursive formula for n_i, by first observing that if $i > 1$ the root of a derivation tree has two children labeled S, and then considering all possibilities for the two subtrees.

b. How many derivation trees are there for the string $a + a + a + a + a$?

c. How many derivation trees are there for the string $a + a + a + a + a + a + a + a + a + a + a$?

4.33. Consider the C statements

```
x = 1; if (a > 2) if (a > 4) x = 2; else x = 3;
```

a. What is the resulting value of `x` if these statements are interpreted according to the derivation tree in Figure 4.21a and $a = 3$?

b. Same question as in (a), but when $a = 1$.

c. What is the resulting value of `x` if these statements are interpreted according to the derivation tree in Figure 4.21b and $a = 3$?

d. Same question as in (c), but when $a = 1$.

4.34. Show that the CFG with productions

$$S \to a \mid Sa \mid bSS \mid SSb \mid SbS$$

is ambiguous.

4.35. Consider the context-free grammar with productions

$$S \to AB$$
$$A \to aA \mid \Lambda$$
$$B \to ab \mid bB \mid \Lambda$$

Every derivation of a string in this grammar must begin with the production $S \to AB$. Clearly, any string derivable from A has only one derivation from A, and likewise for B. Therefore, the grammar is unambiguous. True or false? Why?

4.36. For each part of Exercise 4.1, decide whether the grammar is ambiguous or not, and prove your answer.

4.37. Show that the CFG in Example 4.8 is ambiguous.

4.38. In each case below, show that the grammar is ambiguous, and find an equivalent unambiguous grammar.

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

b. $S \to ABA \qquad A \to aA \mid \Lambda \qquad B \to bB \mid \Lambda$

c. $S \to aSb \mid aaSb \mid \Lambda$

d. $S \to aSb \mid abS \mid \Lambda$

4.39. Describe an algorithm for starting with a regular grammar and finding an equivalent unambiguous grammar.

In the exercises that follow, take as the definition of "balanced string of parentheses" the criterion in Example 1.25: The string has an equal number of left and right parentheses, and no prefix has more right than left.

4.40. a. Show that for every string x of left and right parentheses, x is a prefix of a balanced string if and only if no prefix of x has more right parentheses than left.

b. Show that the language of prefixes of balanced strings of parentheses is generated by the CFG with productions $S \to (S)S \mid (S \mid \Lambda$.

4.41. Show that every left parenthesis in a balanced string has a mate.

4.42. Show that if x is a balanced string of parentheses, then either $x = (y)$ for some balanced string y or $x = x_1x_2$ for two balanced strings x_1 and x_2 that are both shorter than x.

4.43. Show that if $(_0$ is an occurrence of a left parenthesis in a balanced string, and $)_0$ is its mate, then $(_0$ is the rightmost left parentheses for which the string consisting of it and $)_0$ and everything in between is balanced.

4.44. †Show that both of the CFGs below generate the language of balanced strings of parentheses.

a. The CFG with productions $S \to S(S) \mid \Lambda$

b. The CFG with productions $S \to (S)S \mid \Lambda$

4.45. Show that both of the CFGs in the previous exercise are unambiguous.

4.46. Let x be a string of left and right parentheses. A *complete pairing* of x is a partition of the parentheses of x in to pairs such that (i) each pair consists of one left parenthesis and one right parenthesis appearing somewhere after it; and (ii) the parentheses *between* those in a pair are themselves the union of pairs. Two parentheses in a pair are said to be *mates* with respect to that pairing.

a. Show that there is at most one complete pairing of a string of parentheses.

b. Show that a string of parentheses has a complete pairing if and only if it is a balanced string, and in this case the two definitions of mates coincide.

4.47. †Let G be the CFG with productions

$$S \to S + S \mid S * S \mid (S) \mid a$$

and G_1 the CFG with productions

$$S_1 \to S_1 + T \mid T \qquad T \to T * F \mid F \qquad F \to (S_1) \mid a$$

(see Section 4.4).

a. Show that $L(G_1) \subseteq L(G)$. One approach is to use induction and to consider three cases in the induction step, corresponding to the three possible ways a derivation of the string in $L(G_1)$ might begin:

$$S_1 \Rightarrow S_1 + T$$
$$S_1 \Rightarrow T \;\Rightarrow\; T * F$$
$$S_1 \Rightarrow T \;\Rightarrow\; F \;\Rightarrow\; (S_1)$$

b. Show that $L(G) \subseteq L(G_1)$. Again three cases are appropriate in the induction step:

i. x has a derivation in G beginning $S \Rightarrow (S)$

ii. x has a derivation beginning $S \Rightarrow S + S$

iii. Every derivation of x in G begins $S \Rightarrow S * S$

In the second case it may be helpful to let $x = x_1 + x_2 + \ldots + x_n$, where each $x_i \in L(G_1)$ and n is as large as possible. In the third case it may be helpful to let $x = x_1 * x_2 * \ldots x_n$, where each $x_i \in L(G)$ (not $L(G_1)$) and n is as large as possible.

4.48. Show that the nullable variables defined by Definition 4.7 are precisely those variables A for which $A \Rightarrow^* \Lambda$.

4.49. In each case below, find a context-free grammar with no Λ-productions that generates the same language, except possibly for Λ, as the given CFG.

a. $S \to AB \mid \Lambda \qquad A \to aASb \mid a \qquad B \to bS$

b. $S \to AB \mid ABC$
$A \to BA \mid BC \mid \Lambda \mid a$
$B \to AC \mid CB \mid \Lambda \mid b$
$C \to BC \mid AB \mid A \mid c$

4.50. In each case, given the context-free grammar G, find a CFG G' with no Λ-productions and no unit productions that generates the language $L(G) - \{\Lambda\}$.

a. G has productions

$$S \rightarrow ABA \qquad A \rightarrow aA \mid \Lambda \qquad B \rightarrow bB \mid \Lambda$$

b. G has productions

$$S \rightarrow aSa \mid bSb \mid \Lambda \qquad A \rightarrow aBb \mid bBa \qquad B \rightarrow aB \mid bB \mid \Lambda$$

c. G has productions

$$S \rightarrow A \mid B \mid C \qquad A \rightarrow aAa \mid B \qquad B \rightarrow bB \mid bb$$
$$C \rightarrow aCaa \mid D \qquad D \rightarrow baD \mid abD \mid aa$$

4.51. A variable A in a context-free grammar $G = (V, \Sigma, S, P)$ is *live* if $A \Rightarrow^* x$ for some $x \in \Sigma^*$. Give a recursive definition, and a corresponding algorithm, for finding all live variables in G.

4.52. A variable A in a context-free grammar $G = (V, \Sigma, S, P)$ is *reachable* if $S \Rightarrow^* \alpha A \beta$ for some $\alpha, \beta \in (\Sigma \cup V)^*$. Give a recursive definition, and a corresponding algorithm, for finding all reachable variables in G.

4.53. †A variable A is a context-free grammar $G = (V, \Sigma, S, P)$ is *useful* if for some string $x \in \Sigma^*$, there is a derivation of x that takes the form

$$S \Rightarrow^* \alpha A \beta \Rightarrow^* x$$

A variable that is not useful is *useless*. Clearly if a variable is either not live or not reachable (see the two preceding exercises), then it is useless. The converse is not true, as the grammar with productions $S \rightarrow AB$ and $A \rightarrow a$ illustrates. (The variable A is both live and reachable but still useless.)

a. Let G be a CFG. Suppose $G1$ is obtained by eliminating all dead variables from G and eliminating all productions in which dead variables appear. Suppose $G2$ is then obtained from $G1$ by eliminating all variables unreachable in $G1$, as well as productions in which such variables appear. Show that $G2$ contains no useless variables, and $L(G2) = L(G)$.

b. Give an example to show that if the two steps are done in the opposite order, the resulting grammar may still have useless variables.

c. In each case, given the context-free grammar G, find an equivalent CFG with no useless variables.

i. G has productions

$$S \rightarrow ABC \mid BaB \qquad A \rightarrow aA \mid BaC \mid aaa$$
$$B \rightarrow bBb \mid a \qquad C \rightarrow CA \mid AC$$

ii. G has productions

$$S \rightarrow AB \mid AC \quad A \rightarrow aAb \mid bAa \mid a \quad B \rightarrow bbA \mid aaB \mid AB$$
$$C \rightarrow abCa \mid aDb \qquad D \rightarrow bD \mid aC$$

4.54. In each case below, given the context-free grammar G, find a CFG G_1 in Chomsky normal form generating $L(G) - \{\Lambda\}$.

a. G has productions $S \rightarrow SS \mid (S) \mid \Lambda$

b. G has productions $S \rightarrow S(S) \mid \Lambda$

c. G has productions

$$S \rightarrow AaA \mid CA \mid BaB \quad A \rightarrow aaBa \mid CDA \mid aa \mid DC$$
$$B \rightarrow bB \mid bAB \mid bb \mid aS \quad C \rightarrow Ca \mid bC \mid D \quad D \rightarrow bD \mid \Lambda$$

4.55. For alphabets Σ_1 and Σ_2, a *homomorphism* from Σ_1^* to Σ_2^* is defined in Exercise 3.53. Show that if $f : \Sigma_1^* \rightarrow \Sigma_2^*$ is a homomorphism and $L \subseteq \Sigma_1^*$ is a context-free language, then $f(L) \subseteq \Sigma_2^*$ is also a CFG.

4.56. †Let G be the context-free grammar with productions

$$S \rightarrow aS \mid aSbS \mid c$$

and let G_1 be the one with productions

$$S_1 \rightarrow T \mid U \qquad T \rightarrow aTbT \mid c \qquad U \rightarrow aS_1 \mid aTbU$$

(G_1 is a simplified version of the second grammar in Example 4.19.)

a. Show that G is ambiguous.

b. Show that G and G_1 generate the same language.

c. Show that G_1 is unambiguous.

4.57. †Show that if a context-free grammar is unambiguous, then the grammar obtained from it by the algorithm in Theorem 4.27 is also unambiguous.

4.58. †Show that if a context-free grammar with no Λ-productions is unambiguous, then the one obtained from it by the algorithm in Theorem 4.28 is also unambiguous.

CHAPTER

5

Pushdown Automata

A language can be generated by a context-free grammar precisely if it can be accepted by a *pushdown automaton*, which is similar in some respects to a finite automaton but has an auxiliary memory that operates according to the rules of a *stack*. The default mode in a pushdown automaton (PDA) is to allow nondeterminism, and unlike the case of finite automata, the nondeterminism cannot always be eliminated. We give the definition of a pushdown automaton and consider a few simple examples, both with and without nondeterminism. We then describe two ways of obtaining a PDA from a given context-free grammar, so that in each case the moves made by the device as it accepts a string are closely related to a derivation of the string in the grammar. We also discuss the reverse construction, which produces a context-free grammar corresponding to a given PDA. Both methods of obtaining a PDA from a context-free grammar produce nondeterministic devices in general, but we present two examples in which well-behaved grammars allow us to eliminate the nondeterminism, so as to produce a *parser* for the language.

5.1 | DEFINITIONS AND EXAMPLES

In this chapter we will describe how an abstract computing device called a *pushdown automaton* (PDA) accepts a language, and the languages that can be accepted this way are precisely the context-free languages. To introduce these devices, we will start with two simple CFLs that are not regular and extend the definition of a finite automaton in a way that will allow the more general device, when restricted like an FA to a single left-to-right scan of the input string, to remember enough information to accept the language.

Consider the languages

$$AnBn = \{a^n b^n \mid n \geq 0\}$$
$$SimplePal = \{xcx^r \mid x \in \{a, b\}^*\}$$

AnBn was the first example of a context-free language presented in Chapter 4; *SimplePal*, whose elements are palindromes of a particularly simple type, is just as easy to describe by a context-free grammar. For this discussion, though, we will not refer directly to grammars, just as we constructed examples of FAs at first without knowing how to obtain one from an arbitrary regular expression.

When a pushdown automaton reads an input symbol, it will be able to save it (or perhaps save one or more other symbols) in its memory. In processing an input string that might be in *AnBn*, all we need to remember is the *number* of a's, but saving the a's themselves is about as easy a way as any to do that. For *SimplePal*, it seems clear that we really do need to remember the individual symbols themselves.

Suppose a pushdown automaton is attempting to decide whether an input string is in *AnBn* and that it has read and saved a number of a's. If it now reads the input symbol b, two things should happen. First, it should change states to register the fact that from now on, the only legal input symbols are b's. This is the way it can remember that it has read at least one b. Second, it should delete one of the a's from its memory, because one fewer b is now required in order for the input string to be accepted. Obviously, it doesn't matter which a it deletes.

Now suppose a PDA is processing an input string that might be an element of *SimplePal* and that it has read and saved a string of a's and b's. When it reads a c, a change of state is appropriate, because now each symbol it reads will be handled differently: Another c will be illegal, and an a or b will not be saved but used to match a symbol in the memory. Which one? This time it does matter; the symbol used should be the one that was saved most recently.

In both these examples, it will be sufficient for the memory to operate like a *stack*, which uses a last-in-first-out rule. There is no limit to the number of symbols that can be on the stack, but the PDA has immediate access only to the *top* symbol, the one added most recently. We will adopt the terminology that is commonly used with stacks, which explains where the term "pushdown automaton" comes from: when a symbol is added to the memory, it is *pushed* onto the stack, so that it becomes the new top symbol, and when a symbol is deleted it is *popped* off the stack. By allowing a slightly more general type of operation, we will be able to describe a move using a more uniform notation that will handle both these cases. A PDA will be able in a single move to replace the symbol X currently on top of the stack by a *string* α of stack symbols. The cases $\alpha = \Lambda$, $\alpha = X$, and $\alpha = YX$ (where Y is a single symbol) correspond to popping X off the stack, leaving the stack unchanged, and pushing Y onto the stack, respectively (assuming in the last case that the left end of the string YX corresponds to the top of the stack).

A single move of a pushdown automaton will depend on the current state, the next input, and the symbol currently on top of the stack. In the move, the PDA will be allowed to change states, as well as to modify the stack in the way we have described. Before we give the precise definition of a pushdown automaton, there are a few other things to mention. First, Λ-transitions (moves in which no input symbol is read) are allowed, so that "the next input" means either a symbol in the input alphabet or Λ. Second, a PDA is defined in general to be nondeterministic and at some stages of its operation may have a choice of more than one move.

Finally, a PDA will be assumed to begin operation with an initial start symbol Z_0 on its stack and will not be permitted to move unless the stack contains at least one symbol; provided that Z_0 is never removed and no additional copies of it are pushed onto the stack, saying that this symbol is on top means that the stack is effectively empty.

Definition 5.1 A Pushdown Automaton

A *pushdown automaton* (PDA) is a 7-tuple $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$, where

Q is a finite set of states.

Σ and Γ are finite sets, the *input* and *stack* alphabets.

q_0, the initial state, is an element of Q.

Z_0, the initial stack symbol, is an element of Γ.

A, the set of accepting states, is a subset of Q.

δ, the transition function, is a function from $Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma$ to the set of finite subsets of $Q \times \Gamma^*$.

Tracing the moves of a PDA on an input string is more complicated than tracing a finite automaton, because of the stack. When we do this, we will keep track of the current state, the portion of the input string that has not yet been read, and the complete stack contents. The stack contents will be represented by a string of stack symbols, and the leftmost symbol is assumed to be the one on top. A *configuration* of the PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ is a triple

$$(q, x, \alpha)$$

where $q \in Q$, $x \in \Sigma^*$, and $\alpha \in \Gamma^*$. Although it is not required, the last symbol of the string α will often be Z_0, because in most cases it is not removed and remains at the bottom of the stack. We write

$$(p, x, \alpha) \vdash_M (q, y, \beta)$$

to mean that one of the possible moves in the first configuration takes M to the second. This can happen in two ways, depending on whether the move reads an input symbol or is a Λ-transition. In the first case, $x = \sigma y$ for some $\sigma \in \Sigma$, and in the second case $x = y$. We can summarize both cases by saying that $x = \sigma y$ for some $\sigma \in \Sigma \cup \{\Lambda\}$. If $\alpha = X\gamma$ for some $X \in \Gamma$ and some $\gamma \in \Gamma^*$, then $\beta = \xi\gamma$ for some string ξ for which $(q, \xi) \in \delta(p, \sigma, X)$.

More generally, we write

$$(p, x, \alpha) \vdash_M^n (q, y, \beta)$$

if there is a sequence of n moves taking from M from the first configuration to the second, and

$$(p, x, \alpha) \vdash_M^* (q, y, \beta)$$

if there is a sequence of zero or more moves taking M from the first configuration to the second. In the three notations $\vdash_M$, $\vdash_M^n$, and $\vdash_M^*$, if there is no confusion we usually omit the subscript M.

Definition 5.2 Acceptance by a PDA

If $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ and $x \in \Sigma^*$, the string x is *accepted* by M if

$$(q_0, x, Z_0) \vdash_M^* (q, \Lambda, \alpha)$$

for some $\alpha \in \Gamma^*$ and some $q \in A$. A language $L \subseteq \Sigma^*$ is said to be accepted by M if L is precisely the set of strings accepted by M; in this case, we write $L = L(M)$. Sometimes a string accepted by M, or a language accepted by M, is said to be accepted *by final state*.

According to this definition, whether or not a string x is accepted depends only on the current state when x has been processed, not on the stack contents. An *accepting configuration* will be any configuration in which the state is an accepting state. Specifying an input string as part of a configuration can be slightly confusing; keep in mind that if $x, y \in \Sigma^*$, and

$$(q_0, xy, Z_0) \vdash_M^* (q, y, \alpha)$$

and $q \in A$, then although the original input is xy and the PDA reaches an accepting configuration, it is not xy that has been accepted, but only x. In order to be accepted, a string must have been read in its entirety by the PDA.

PDAs Accepting the Languages *AnBn* and *SimplePal* EXAMPLE 5.3

We have already described in general terms how pushdown automata can be constructed to accept the languages *AnBn* and *SimplePal* defined above. Now we present the PDAs in more detail. In this chapter, for the most part, we will rely on transition tables rather than transition diagrams, although for the PDA accepting *SimplePal* we will present both.

The PDA for *AnBn* is $M_1 = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$, where $Q = \{q_0, q_1, q_2, q_3\}$, $A = \{q_0, q_3\}$, and the transitions are those in Table 5.4.

M_1 is in an accepting state initially, which allows Λ to be accepted. The only legal input symbol in this state is a, and the first a takes M_1 to q_1, the state in which additional a's are read and pushed onto the stack. Reading the first b is the signal to move to q_2, the state in which b's are read (these are the only legal input symbols when M_1 is in the state q_2) and used to cancel a's on the stack. The input string is accepted when a b cancels the last a on the stack, so that the number of b's and the number of a's are equal. At that point, M_1 makes a Λ-transition to the other accepting state, from which no moves are possible. The reason a second accepting state is necessary is that without it the PDA could start all over once it has processed a string in *AnBn*, and end up accepting strings such as *aabbab*.

Table 5.4 | Transition Table for a PDA Accepting *AnBn*

Move Number	State	Input	Stack Symbol	Move(s)
1	q_0	a	Z_0	(q_1, aZ_0)
2	q_1	a	a	(q_1, aa)
3	q_1	b	a	(q_2, Λ)
4	q_2	b	a	(q_2, Λ)
5	q_2	Λ	Z_0	(q_3, Z_0)
(all other combinations)				none

The sequence of moves causing *aabb* to be accepted is

$$(q_0, aabb, Z_0) \vdash (q_1, abb, aZ_0) \vdash (q_1, bb, aaZ_0)$$
$$\vdash (q_2, b, aZ_0) \vdash (q_2, \Lambda, Z_0) \vdash (q_3, \Lambda, Z_0)$$

The language *SimplePal* does not contain Λ, and as a result we can get by with one fewer state in a pushdown automaton accepting it. As in the first PDA, there is one state q_0 for processing the first half of the string, another, q_1, for the second half, and an accepting state q_2 from which there are no moves. The PDA for this language is also similar to the first one in that in state q_0, symbols in the first half of the string are pushed onto the stack; in state q_1 there is only one legal input symbol at each step and it is used to cancel the top stack symbol; and the PDA moves to the accepting state on a Λ-transition when the stack is empty except for Z_0.

The moves by which the string *abcba* are accepted are shown below:

$$(q_0, abcba, Z_0) \vdash (q_0, bcba, aZ_0) \vdash (q_0, cba, baZ_0) \vdash (q_1, ba, baZ_0)$$
$$\vdash (q_1, a, aZ_0) \vdash (q_1, \Lambda, Z_0) \vdash (q_2, \Lambda, Z_0)$$

A string can fail to be accepted either because it is never processed completely or because the current state at the end of the processing is not an accepting state. These two scenarios are illustrated by the strings *acab* and *abc*, respectively:

$$(q_0, acab, Z_0) \vdash (q_0, cab, aZ_0) \vdash (q_1, ab, aZ_0) \vdash (q_1, b, Z_0) \vdash (q_2, b, Z_0)$$
$$(q_0, abc, Z_0) \vdash (q_0, bc, aZ_0) \vdash (q_0, c, baZ_0) \vdash (q_1, \Lambda, baZ_0)$$

In the first case, although the sequence of moves ends in the accepting state, it is not *acab* that has been accepted, but only the prefix *aca*.

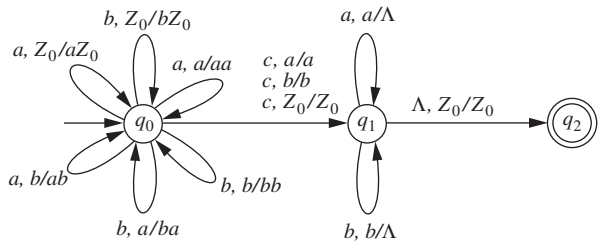


Figure 5.5 |
A transition diagram for a PDA accepting *SimplePal*.

Table 5.6 | Transition Table for a PDA Accepting *SimplePal*

Move Number	State	Input	Stack Symbol	Move(s)
1	q_0	a	Z_0	(q_0, aZ_0)
2	q_0	b	Z_0	(q_0, bZ_0)
3	q_0	a	a	(q_0, aa)
4	q_0	b	a	(q_0, ba)
5	q_0	a	b	(q_0, ab)
6	q_0	b	b	(q_0, bb)
7	q_0	c	Z_0	(q_1, Z_0)
8	q_0	c	a	(q_1, a)
9	q_0	c	b	(q_1, b)
10	q_1	a	a	(q_1, Λ)
11	q_1	b	b	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)				none

Figure 5.5 shows a transition diagram for the PDA in Table 5.6. The labels on the arrows are more complicated than in the case of a finite automaton, because each one includes the input as well as the change made to the stack. In the second part of the label, the top stack symbol comes before the slash and the string that replaces it comes after. Even with the extra information, however, a diagram of this type does not capture the machine's behavior in the way an FA's transition diagram does. Tracing the moves means being able to remember information, perhaps the entire contents of the stack, at each step.

Although the definition of a pushdown automaton allows nondeterminism, neither of these two examples illustrates it—at least not if we take nondeterminism to mean the possibility of a choice of moves. In the PDA in the next example, there are some configurations in which several moves are possible, and we will see later that the corresponding language $L(M)$ can be accepted by a PDA only if this feature is present. The existence of such languages is another significant difference between pushdown automata and finite automata.

A Pushdown Automaton Accepting *Pal* EXAMPLE 5.7

Pal is the language of palindromes over $\{a, b\}$ discussed in Examples 1.18 and 4.3. An odd-length string in *Pal* is just like an element of *SimplePal* in Example 5.3 except that its middle symbol is a or b, not c. Like the PDA above accepting *SimplePal*, this one has only the three states q_0, q_1, and q_2. q_2 is the accepting state, and q_0 and q_1 are the states the device is in when it is processing the first half and the second half of the string, respectively.

Our strategy for accepting *SimplePal* depended strongly on the c in the middle, which told us that we should stop pushing input symbols onto the stack and start using them to cancel symbols currently on the stack. The question is, without this symbol marking the middle of the string, how can a pushdown automaton know when it is time to change

from the pushing-onto-the-stack state q_0 to the popping-off-the-stack state q_1? The answer is that it can't, and this is where the nondeterminism is necessary. Let's begin, however, by emphasizing that once the PDA makes this change of state, there is no more nondeterminism; we proceed the way we did with *SimplePal*, so that we reach the accepting state q_2 only if the symbols we read from this point on match the ones currently on the stack. This is what guarantees that even though we have many choices of moves, we never have a choice that will cause a string to be accepted if it isn't a palindrome.

A palindrome of odd length looks like xax^r or xbx^r, and one of even length looks like xx^r. Suppose we have read the string x, have pushed all the symbols of x onto the stack, and are still in the pushing-onto-the-stack state q_0. Suppose also that the next input symbol is a, so that we don't need to worry about the string we're working on being xbx^r. What we have to do is provide a choice of moves that will allow the string xax^r to be accepted; another choice that will allow xx^r to be accepted (if x^r begins with a); and other choices that will allow longer palindromes beginning with x to be accepted.

The moves that take care of these cases are: go to the state q_1 by reading the symbol a; go to q_1 without reading a symbol; and read the a but push it onto the stack and stay in q_0. In either of the first two cases, we will accept if and only if the symbols we read starting now are those in x^r.

Compare Table 5.8, for a PDA accepting *Pal*, to Table 5.6. In each of the first six lines of Table 5.8, the first move shown is identical to the move in that line of Table 5.6, and the last three lines of Table 5.8 are also the same as in Table 5.6. The tables differ only in the moves that cause the PDA to go from q_0 to q_1. In the earlier PDA, the only way to do it was to read a c. Here, the second move in each of the first three lines does it by reading an a; the second move in each of the next three lines does it by reading a b; and the moves in lines 7–9 do it by means of a Λ-transition.

The way the string *abbba* is accepted is for the machine to push the first two symbols onto the stack, so that the configuration is (q_0, bba, baZ_0). At that point the input symbol b is read and the state changes to q_1, but the stack is unchanged; the PDA guesses that the b is the middle symbol of an odd-length string and tests subsequent input symbols accordingly.

Table 5.8 | Transition Table for a PDA Accepting *Pal*

Move Number	State	Input	Stack Symbol	Move(s)
1	q_0	a	Z_0	$(q_0, aZ_0), (q_1, Z_0)$
2	q_0	a	a	$(q_0, aa), (q_1, a)$
3	q_0	a	b	$(q_0, ab), (q_1, b)$
4	q_0	b	Z_0	$(q_0, bZ_0), (q_1, Z_0)$
5	q_0	b	a	$(q_0, ba), (q_1, a)$
6	q_0	b	b	$(q_0, bb), (q_1, b)$
7	q_0	Λ	Z_0	(q_1, Z_0)
8	q_0	Λ	a	(q_1, a)
9	q_0	Λ	b	(q_1, b)
10	q_1	a	a	(q_1, Λ)
11	q_1	b	b	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)				none

In other words, after reading two symbols of the input string, it chooses to test whether the string is a palindrome of length 5 (if it is not, it will not be accepted). Once it makes this choice, its moves are the only possible ones. The complete sequence is

$$(q_0, abbba, Z_0) \vdash (q_0, bbba, aZ_0) \vdash (q_0, bba, baZ_0)$$
$$\vdash (q_1, ba, baZ_0) \vdash (q_1, a, aZ_0) \vdash (q_1, \Lambda, Z_0) \vdash (q_2, \Lambda, Z_0)$$

The presence of two moves in each of the first six lines is an obvious indication of nondeterminism. Lines 7–9 constitute nondeterminism of a slightly less obvious type. For the string $abba$, for example, the first two moves are the same as before, resulting in the configuration (q_0, ba, baZ_0). The PDA then guesses, by making a Λ-transition, that the input might be a palindrome of length 4. The moves are

$$(q_0, abba, Z_0) \vdash (q_0, bba, aZ_0) \vdash (q_0, ba, baZ_0)$$
$$\vdash (q_1, ba, baZ_0) \vdash (q_1, a, aZ_0) \vdash (q_1, \Lambda, Z_0) \vdash (q_2, \Lambda, Z_0)$$

Just as in Section 3.2, we can draw a computation tree for the PDA, showing the configuration and choice of moves at each step. Figure 5.9 shows the tree for the input

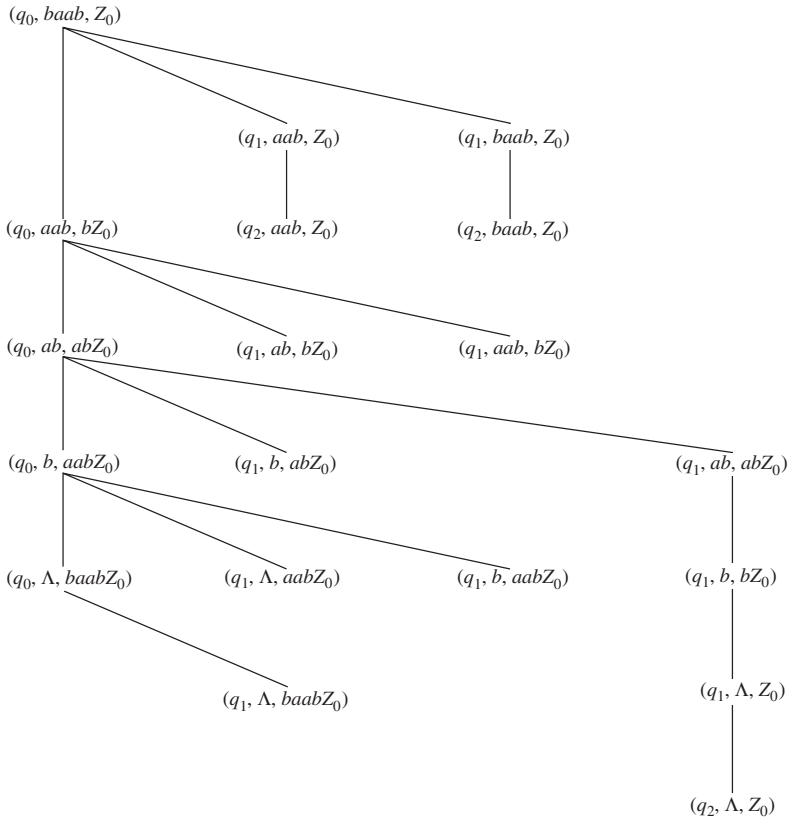


Figure 5.9
A computation tree for Table 5.8 and the input string *baab*.

string *baab*. In each configuration on the left, the state is q_0. As long as there is at least one unread input symbol, the PDA can choose from the three moves we have discussed. The three branches, left to right, represent the move that stays in q_0, the move that reads a symbol and moves to q_1, and the move to q_1 on a Λ-transition.

The path through the tree that leads to the string being accepted is the one that branches to the right with a Λ-transition after two symbols have been read. Paths that leave the vertical path too soon terminate before the PDA has finished reading the input; it either stops because it is unable to move or enters the accepting state prematurely, so that the string accepted is a palindrome of length 0 or 1. Paths that follow the vertical path too long terminate with input symbols left on the stack.

5.2 DETERMINISTIC PUSHDOWN AUTOMATA

The pushdown automaton in Example 5.3 accepting *SimplePal* never has a choice of moves. It is appropriate to call it deterministic if we are less strict than in Chapter 4 and allow a deterministic PDA to have no moves in some configurations. The one in Example 5.7 accepting *Pal* illustrates both ways a PDA can be nondeterministic: It can have more than one move for the same combination of state, input, and stack symbol, and it can have a choice, for some combination of state and stack symbol, between reading an input symbol and making a Λ-transition without reading one.

Definition 5.10 A Deterministic Pushdown Automaton

A pushdown automaton $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ is *deterministic* if it satisfies both of the following conditions.

1. For every $q \in Q$, every $\sigma \in \Sigma \cup \{\Lambda\}$, and every $X \in \Gamma$, the set $\delta(q, \sigma, X)$ has at most one element.
2. For every $q \in Q$, every $\sigma \in \Sigma$, and every $X \in \Gamma$, the two sets $\delta(q, \sigma, X)$ and $\delta(q, \Lambda, X)$ cannot both be nonempty.

A language L is a *deterministic context-free language* (DCFL) if there is a deterministic PDA (DPDA) accepting L.

We have not yet shown that context-free languages are precisely the languages that can be accepted by PDAs, and the last sentence in the definition anticipates the results in Sections 5.3 and 5.4. Later in this section, we will show that not every context-free language is a DCFL; it probably won't be a surprise that *Pal*, for which we constructed a PDA in Example 5.7, cannot be accepted by a deterministic PDA.

EXAMPLE 5.11 A DPDA Accepting *Balanced*

According to Example 1.25, a string of left and right parentheses is balanced if no prefix has more right than left and there are equal numbers of left and right. We present a very simple

deterministic PDA M and argue that it accepts the language *Balanced* of balanced strings. The only two states are q_0 and q_1, and the accepting state is q_0. For clarity of notation, we use [and] for our "parentheses." The transition table is shown in Table 5.12 below.

Table 5.12 | Transition Table for a DPDA Accepting *Balanced*

Move Number	State	Input	Stack Symbol	Move
1	q_0	[	Z_0	$(q_1, [Z_0)$
2	q_1	[	[	$(q_1, [[)$
3	q_1	]	[	(q_1, Λ)
4	q_1	Λ	Z_0	(q_0, Z_0)
	(all other combinations)			none

The PDA operates as follows. A left parenthesis in the input is always pushed onto the stack; a right parenthesis in the input is canceled with a left parenthesis on the stack; and once the machine enters state q_1, it stays there until the stack is empty except for Z_0, when it returns to q_0 with a Λ-transition.

It is not hard to see that every string accepted by M must be a balanced string of parentheses. Every prefix must have at least as many left parentheses as right, because for every right parenthesis that is read, some previous left parenthesis must have been pushed onto the stack and not yet canceled. And the total numbers of left and right parentheses must be equal, because the stack must be empty except for Z_0 by the time the string is accepted.

It is a little harder to show that every balanced string of parentheses is accepted by M. First we establish a preliminary result: For every balanced string x, the statement $P(x)$ is true, where $P(x)$ is

$$\text{For every } j \geq 1,\ (q_1, x, [^j Z_0) \vdash_M^* (q_1, \Lambda, [^j Z_0)$$

The proof is by mathematical induction on $|x|$. $P(\Lambda)$ is true, because for every j, the initial and final configurations are identical. Suppose that $k \geq 0$, that $P(x)$ is true for every balanced string x with $|x| \leq k$, and that x is a balanced string with $|x| = k + 1$.

According to the recursive definition of *Balanced* in Example 1.19, every nonnull balanced string is either the concatenation of two shorter balanced strings or of the form $[y]$ for some balanced string y. We consider the two cases separately. If $x = x_1x_2$, where x_1 and x_2 are both balanced strings shorter than x, then for every j, the induction hypothesis used twice (once for x_1, once for x_2) implies that

$$(q_1, x_1x_2, [^j Z_0) \vdash_M^* (q_1, x_2, [^j Z_0) \vdash_M^* (q_1, \Lambda, [^j Z_0)$$

In the other case, $x = [y]$ for some balanced string y. In this case, for every $j \geq 1$, the induction hypothesis tells us that

$$(q_1, y], [^{j+1} Z_0) \vdash^* (q_1,], [^{j+1} Z_0)$$

and it follows that for every $j \geq 1$,

$$(q_1, [y], [^j Z_0) \vdash (q_1, y], [^{j+1} Z_0) \vdash^* (q_1,], [^{j+1} Z_0) \vdash (q_1, \Lambda, [^j Z_0)$$

This preliminary result allows us to show, again using mathematical induction on the length of x, that every balanced string x is accepted by M. The basis step is easy: Λ is

accepted because q_0 is the accepting state. Suppose that $k \geq 0$ and that every balanced string of length k or less is accepted, and let x be a balanced string with $|x| = k + 1$. We consider the same two cases as before. If $x = x_1x_2$, where x_1 and x_2 are shorter balanced strings, then the induction hypothesis applied to x_1 and x_2 individually implies that

$$(q_0, x_1x_2, Z_0) \vdash^* (q_0, x_2, Z_0) \vdash^* (q_0, \Lambda, Z_0)$$

If $x = [\,y\,]$, where y is balanced, then

$$(q_0, [\,y\,], Z_0) \vdash (q_1, y\,], [\,Z_0) \vdash^* (q_1,], [\,Z_0) \vdash (q_1, \Lambda, Z_0) \vdash (q_0, \Lambda, Z_0)$$

Here the second statement follows from the preliminary result applied to the string y, and the other statements involve moves of M that are shown in the transition table.

EXAMPLE 5.13 Two DPDAs Accepting *AEqB*

In order to find a PDA accepting *AEqB*, the language $\{x \in \{a, b\}^* \mid n_a(x) = n_b(x)\}$, we can adapt the idea of the PDA M in Example 5.11: using the stack to save copies of one symbol if there is a temporary excess of that symbol, and canceling them with incoming occurrences of the opposite symbol. The only difference now is that we allow prefixes of the string to have either more a's than b's or more b's than a's, as long as the numbers are equal when we're done; before we finish processing the string we may have seen temporary excesses of both symbols. We can continue to use the state q_1 for the situation in which the string read so far has more of one symbol than the other; as before, whenever equality is restored, the PDA can return to the accepting state by a Λ-transition.

Here are the moves made by the PDA in accepting *abbaaabb*.

$$\begin{aligned}(q_0, abbaaabb, Z_0) &\vdash (q_1, bbaaabb, aZ_0) \vdash (q_1, baaabb, Z_0) \vdash (q_0, baaabb, Z_0)\\ &\vdash (q_1, aaabb, bZ_0) \vdash (q_1, aabb, Z_0) \vdash (q_0, aabb, Z_0) \vdash (q_1, abb, aZ_0)\\ &\vdash (q_1, bb, aaZ_0) \vdash (q_1, b, aZ_0) \vdash (q_1, \Lambda, Z_0) \vdash (q_0, \Lambda, Z_0)\end{aligned}$$

According to Definition 5.10, Λ-transitions are permissible in a DPDA, provided they do not allow the device a choice between reading a symbol and making a Λ-transition. Both the PDA in Table 5.14 and the one in Table 5.12 are deterministic, because the only Λ-transition is in q_1 with top stack symbol Z_0, and with that combination of state and stack symbol there are no other moves. If we wish, however, we can modify the DPDA in both

Table 5.14 | Transition Table for a DPDA Accepting *AEqB*

Move Number	State	Input	Stack Symbol	Move
1	q_0	a	Z_0	(q_1, aZ_0)
2	q_0	b	Z_0	(q_1, bZ_0)
3	q_1	a	a	(q_1, aa)
4	q_1	b	b	(q_1, bb)
5	q_1	a	b	(q_1, Λ)
6	q_1	b	a	(q_1, Λ)
7	q_1	Λ	Z_0	(q_0, Z_0)
(all other combinations)				none

Table 5.15 | A DPDA with No Λ-transitions Accepting *AEqB*

Move Number	State	Input	Stack Symbol	Move
1	q_0	a	Z_0	(q_1, AZ_0)
2	q_0	b	Z_0	(q_1, BZ_0)
3	q_1	a	A	(q_1, aA)
4	q_1	b	B	(q_1, bB)
5	q_1	a	a	(q_1, aa)
6	q_1	b	b	(q_1, bb)
7	q_1	a	b	(q_1, Λ)
8	q_1	b	a	(q_1, Λ)
9	q_1	a	B	(q_0, Λ)
10	q_1	b	A	(q_0, Λ)
(all other combinations)				none

cases so that there are no Λ-transitions at all. In order to do this for Table 5.14, we must provide a way for the device to determine whether the symbol currently on the stack is the only one other than Z_0; an easy way to do this to use special symbols, say A and B, to represent the first extra a or extra b, respectively.

In this modified DPDA, shown in Table 5.15, the moves made by the PDA in accepting *abbaaabb* are

$$(q_0, abbaaabb, Z_0) \vdash (q_1, bbaaabb, AZ_0) \vdash (q_0, baaabb, Z_0) \vdash (q_1, aaabb, BZ_0)$$
$$\vdash (q_0, aabb, Z_0) \vdash (q_1, abb, AZ_0) \vdash (q_1, bb, aAZ_0) \vdash (q_1, b, AZ_0) \vdash (q_0, \Lambda, Z_0)$$

As we mentioned at the beginning of this section, the language *Pal* can be accepted by a simple PDA but cannot be accepted by a DPDA. It may seem intuitively clear that the natural approach can't be used by a deterministic device (how can a PDA know, without guessing, that it has reached the middle of the string?), but the proof that no DPDA can accept *Pal* requires a little care.

Theorem 5.16

The language *Pal* cannot be accepted by a deterministic pushdown automaton.

Sketch of Proof Suppose for the sake of contradiction that M is a DPDA accepting *Pal*.

First, we can modify M if necessary (see Exercise 5.17) so that every move is either one of the form

$$\delta(p, \sigma, X) = \{(q, \Lambda)\}$$

or one of the form

$$\delta(p, \sigma, X) = \{(q, \alpha X)\}$$

The effect is that M can still remove a symbol from the stack or place another string on the stack, but it can't do both in the same move. If

the stack height does not decrease as a result of a move, then that move cannot have removed any symbols from the stack.

Next, we observe that for every input string $x \in \{a, b\}^*$, M must read every symbol of x. This feature is guaranteed by the fact that x is a prefix of the palindrome xx^r, and a palindrome must be read completely by M in order to be accepted. It follows in particular that no sequence of moves can cause M to empty its stack.

The idea of the proof is to find two different strings r and s such that for every suffix z, M treats both rz and sz the same way. This will produce a contradiction, because according to Example 2.27, no matter what r and s are, there is some z for which only one of the two strings rz and sz will be a palindrome.

Here is one more definition: For an arbitrary x, there is a string y_x such that of all the possible strings xy, xy_x is one whose processing by M produces a configuration with minimum stack height. The defining property of y_x and the modification of M mentioned at the beginning imply that if α_x represents the stack contents when xy_x has been processed, no symbols of α_x are ever removed from the stack as a result of processing subsequent input symbols.

Now we are ready to choose the strings r and s. There are infinitely many different strings of the form xy_x. Therefore, because there are only a finite number of combinations of state and stack symbol, there must be two different strings $r = uy_u$ and $s = vy_v$ so that the configurations resulting from M's processing r and s have both the same state and the same top stack symbol. It follows that although r and s are different, M can't tell the difference—the resulting states are the same, and the only symbols on the stack that M will ever be able to examine after processing the two strings are also the same. Therefore, for every z, the results of processing rz and sz must be the same.

5.3 | A PDA FROM A GIVEN CFG

The languages accepted by the pushdown automata we have studied so far have been generated by simple context-free grammars, but we have not referred to grammars in constructing the machines, relying instead on simple symmetry properties of the strings themselves. In this section, we will consider two ways of constructing a PDA from an arbitrary CFG.

In both cases, the device is nondeterministic. It attempts to simulate a derivation in the grammar, using the stack to hold portions of the current string, and terminates the computation if it determines at some point that the derivation-in-progress is not consistent with the input string. It is possible in both cases to visualize the simulation as an attempt to construct a derivation tree for the input string; the two PDAs we end up with are called *top-down* and *bottom-up* because of the different

approaches they take to building the tree. In both cases, we will describe the PDA obtained from a grammar G, indicate why the language accepted by the PDA is precisely the same as the one generated by G, and look at an example to see how the moves the PDA makes as it accepts a string correspond to the steps in a derivation of x.

The top-down PDA begins by placing the start symbol of the grammar, which is the symbol at the root of every derivation tree, on the stack. Starting at this point, each step in the construction of a derivation tree consists of replacing a variable that is currently on top of the stack by the right side of a grammar production that begins with that variable. If there are several such productions, one is chosen nondeterministically. This step corresponds to building the portion of the tree containing that variable-node and its children. The intermediate moves of the PDA, which prepare for the next step in the simulation, are to remove terminal symbols from the stack as they are produced and match them with input symbols. To the extent that they continue to match, the derivation being simulated is consistent with the input string. Because the variable that is replaced in each step of the simulation is the leftmost one in the current string, preceded only by terminal symbols that have been removed from the stack already, the derivation being simulated is a leftmost derivation.

Definition 5.17 The Nondeterministic Top-Down PDA *NT*(*G*)

Let $G = (V, \Sigma, S, P)$ be a context-free grammar. The nondeterministic top-down PDA corresponding to G is $NT(G) = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$, defined as follows:

$$Q = \{q_0, q_1, q_2\} \quad A = \{q_2\} \quad \Gamma = V \cup \Sigma \cup \{Z_0\}$$

The initial move of $NT(G)$ is the Λ-transition

$$\delta(q_0, \Lambda, Z_0) = \{(q_1, SZ_0)\}$$

and the only move to the accepting state is the Λ-transition

$$\delta(q_1, \Lambda, Z_0) = \{(q_2, Z_0)\}$$

The moves from q_1 are the following:

For every $A \in V$, $\delta(q_1, \Lambda, A) = \{(q_1, \alpha) \mid A \to \alpha \text{ is a production in } G\}$

For every $\sigma \in \Sigma$, $\delta(q_1, \sigma, \sigma) = \{(q_1, \Lambda)\}$

After the initial move and before the final move to q_2, the PDA stays in state q_1. In this state, the two types of moves are to replace a variable by the right side of a production and to match a terminal symbol on the stack with an input symbol and discard both. The nondeterminism in $NT(G)$ comes from the choice of moves when the top stack symbol is a variable for which there are several productions.

Theorem 5.18

If G is a context-free grammar, then the nondeterministic top-down PDA $NT(G)$ accepts the language $L(G)$.

Proof

Suppose first that $x \in L(G)$, and consider a leftmost derivation of x in G. We wish to show that there is a sequence of moves of $NT(G)$ that will simulate this derivation and cause x to be accepted.

For some $m \geq 0$, the leftmost derivation of x has the form

$$S = x_0 A_0 \alpha_0 \Rightarrow x_0 x_1 A_1 \alpha_1 \;\Rightarrow\; x_0 x_1 x_2 A_2 \alpha_2 \;\Rightarrow\; \ldots$$
$$\Rightarrow x_0 x_1 \ldots x_m A_m \alpha_m \;\Rightarrow\; x_0 x_1 \ldots x_m x_{m+1} = x$$

where each string x_i is a string of terminal symbols, each A_i is a variable, and each α_i is a string that may contain variables as well as terminals. (The strings x_0 and α_0 are both Λ.) For $i < m$, if the production applied to the variable A_i in the derivation is $A_i \rightarrow \beta_i$, then the variable A_{i+1} may be either a variable in the string β_i or a variable in α_i that was already present before the production was applied.

The first move of $NT(G)$ is to place S on top of the stack, and at that point, the string x_0 (i.e., Λ) of input symbols has already been read, and the stack contains the string $A_0 \alpha_0 Z_0$. If we ignore Z_0, the concatenation of these two strings is the first string in the derivation.

Now suppose that for some $i < m$, there is a sequence of moves, the result of which is that the prefix $x_0 x_1 \ldots x_i$ of x has been read and the string $A_i \alpha_i Z_0$ is on the stack. We observe that there are additional moves $NT(G)$ can make at that point that will simulate one more step of the derivation, so that $x_0 x_1 \ldots x_i x_{i+1}$ will have been read and $A_{i+1} \alpha_{i+1} Z_0$ will be on the stack. They are simply the moves that replace A_i by β_i on the stack, so that now the string on the stack has the prefix x_{i+1}, and then read the symbols in x_{i+1}, using them to cancel the identical ones on the stack.

The conclusion is that $NT(G)$ can make moves that will eventually cause $x_0 x_1 \ldots x_m$ to have been read and the stack to contain $A_m \alpha_m Z_0$. At that point, replacing A_m by β_m on the stack will cause the stack to contain $\beta_m \alpha_m$, which is the same as x_{m+1}, and $NT(G)$ can then read those symbols in the input string and match them with the identical symbols on the stack. At that point, it has read x and the stack is empty except for Z_0, so that it can accept.

In the other direction, the only way a string x can be accepted by $NT(G)$ is for all the symbols of x to be matched with terminal symbols on the stack; the only way this can happen, after the initial move placing S on the stack, is for it to make a sequence of moves that replace variables by strings on the stack, and between moves in this sequence to make intermediate moves matching input symbols with terminals from the stack.

We let X_0 be the string S, which is the first variable-occurrence to show up on top of the stack. After that, for each $i \geq 1$, the ith time a variable-occurrence appears on top of the stack, we let X_i be the string obtained from the terminals already read and the current stack contents (excluding Z_0). Then the sequence of X_i's, followed by x itself, constitutes a leftmost derivation of x.

The Language *Balanced* **EXAMPLE 5.19**

Let L be the language of balanced strings of parentheses (as in Example 5.11, we will use [and] instead of parentheses). Example 1.25 describes a balanced string by comparing the number of left parentheses and right parentheses in each prefix, but balanced strings can also be described as those strings of parentheses that appear within legal algebraic expressions. The CFG for algebraic expressions in Example 4.2 suggests that the language of balanced strings can be generated by the CFG G with productions

$$S \rightarrow [S] \mid SS \mid \Lambda$$

The nondeterministic top-down PDA $NT(G)$ corresponding to G is the one whose transitions are described in Table 5.21 on the next page.

We consider the balanced string [[][]]. Figure 5.20 shows a derivation tree corresponding to the leftmost derivation

$$S \Rightarrow [S] \Rightarrow [SS] \Rightarrow [[S]S] \Rightarrow [[]S] \Rightarrow [[][S]] \Rightarrow [[][]]$$

and we compare the moves made by $NT(G)$ in accepting this string to the leftmost derivation of the string in G.

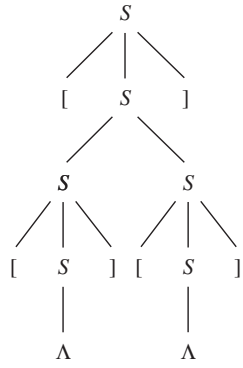


Figure 5.20
A derivation tree for the string [[][]], using $S \rightarrow [S] \mid SS \mid \Lambda$.

Table 5.21 | Transition Table for the Top-Down PDA $NT(G)$

Move Number	State	Input	Stack Symbol	Move
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	$(q_1, [S])$, (q_1, SS), (q_1, Λ)
3	q_1	[	[	(q_1, Λ)
4	q_1	]	]	(q_1, Λ)
5	q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)				none

To the right of each move that replaces a variable on top of the stack, we show the corresponding step in the leftmost derivation.

$(q_0, [[][]], Z_0)$
$\vdash (q_1, [[][]], SZ_0)$ S
$\vdash (q_1, [[][]], [S]Z_0)$ $\Rightarrow [S]$
$\vdash (q_1, [][]], S]Z_0)$
$\vdash (q_1, [][]], SS]Z_0)$ $\Rightarrow [SS]$
$\vdash (q_1, [][]], [S]S]Z_0)$ $\Rightarrow [[S]S]$
$\vdash (q_1,][]], S]S]Z_0)$
$\vdash (q_1,][]],]S]Z_0)$ $\Rightarrow [[]S]$
$\vdash (q_1, []], S]Z_0)$
$\vdash (q_1, []], [S]]Z_0)$ $\Rightarrow [[][S]]$
$\vdash (q_1,]], S]]Z_0)$
$\vdash (q_1,]],]]Z_0)$ $\Rightarrow [[][]]$
$\vdash (q_1,],]Z_0)$
$\vdash (q_1, \Lambda, Z_0)$ $\Rightarrow [[]][]$
$\vdash (q_2, \Lambda, Z_0)$

As the term *bottom-up* suggests, our second approach to accepting a string in $L(G)$ involves building a derivation tree from bottom to top, from leaf nodes to the root. The top-down PDA places S (the top of the tree) on the stack in the very first step of its computation. In the bottom-up version, S doesn't show up by itself on the stack until the very end of the computation, by which time the PDA has read all the input and can accept.

The way S ends up on the stack by itself is that it is put there to replace the symbols on the right side of the production $S \rightarrow \alpha$ that is the first step in a derivation of the string. We refer to a step like this as a *reduction*: a sequence of PDA moves (often more than one, because only one symbol can be removed from the stack in a single move) that remove from the stack the symbols representing the right side of a production and replace them by the variable on the left side. We are not really interested in the individual moves of a reduction, which are likely to be done using states that are not used except in that specific sequence of moves, but only in their combined effect. Each reduction "builds" the portion of a derivation tree that consists of several children nodes and their parent. The other moves the

bottom-up PDA performs to prepare the way for a reduction are *shift* moves, which simply read input symbols and push them onto the stack.

In going from top-down to bottom-up, there are several things that are reversed, the most obvious being the "direction" in which the tree is constructed. Another is the order of the steps of the derivation simulated by the PDA as it accepts a string; the PDA simulates the reverse of a derivation, starting with the last move and ending with the first. Yet another reversal is the order in which we read the symbols on the stack at the time a reduction is performed. For a simple example, suppose that a string abc has the simple derivation $S \Rightarrow abc$. The symbols a, b, and c get onto the stack by three shift moves, which have the effect of reversing their order. In general, a reduction corresponding to the production $A \rightarrow \beta$ is performed when the string β^r appears on top of the stack (the top symbol being the *last* symbol of β if $\beta \neq \Lambda$) and is then replaced by A. Finally, for reasons that will be more clear shortly, the derivation whose reverse the bottom-up PDA simulates is a *rightmost* derivation.

Definition 5.22 The Nondeterministic Bottom-Up PDA *NB*(*G*)

Let $G = (V, \Sigma, S, P)$ be a context-free grammar. The nondeterministic bottom-up PDA corresponding to G is $NB(G) = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$, defined as follows:

Q contains the initial state q_0, the state q_1, and the accepting state q_2, together with other states to be described shortly.

For every $\sigma \in \Sigma$ and every $X \in \Gamma$, $\delta(q_0, \sigma, X) = \{(q_0, \sigma X)\}$. This is a *shift* move.

For every production $B \rightarrow \alpha$ in G, and every nonnull string $\beta \in \Gamma^*$, $(q_0, \Lambda, \alpha^r \beta) \vdash^* (q_0, B\beta)$, where this *reduction* is a sequence of one or more moves in which, if there is more than one, the intermediate configurations involve other states that are specific to this sequence and appear in no other moves of $NB(G)$.

One of the elements of $\delta(q_0, \Lambda, S)$ is (q_1, Λ), and $\delta(q_1, \Lambda, Z_0) = \{(q_2, Z_0)\}$.

In the course of accepting a string x, the bottom-up PDA spends most of its time in the state q_0, leaving it if necessary within a reduction and in the last two moves that lead to the accepting state q_2 via q_1.

Theorem 5.23

If G is a context-free grammar, then the nondeterministic bottom-up PDA $NB(G)$ accepts the language $L(G)$.

Proof

As in the top-down case, we show that for a string x in $L(G)$, there is a sequence of moves $NB(G)$ can make that will simulate

a derivation of x in G (in reverse) and will result in x being accepted.

Consider a rightmost derivation of x, which has the form

$$S = \alpha_0 A_0 x_0 \Rightarrow \alpha_1 A_1 x_1 x_0 \Rightarrow \alpha_2 A_2 x_2 x_1 x_0 \Rightarrow \ldots$$
$$\Rightarrow \alpha_m A_m x_m x_{m-1} \ldots x_1 x_0 \Rightarrow x_{m+1} x_m \ldots x_1 x_0 = x$$

where, for each i, x_i is a string of terminals, A_i is a variable, and α_i is a string of variables and/or terminals. As before, x_0 and α_0 are both Λ. If the production that is applied to the indicated occurrence of A_i in this derivation is $A_i \rightarrow \gamma_i$, then for $i < m$ we may write

$$\alpha_{i+1} A_{i+1} x_{i+1} x_i \ldots x_0 = \alpha_i \gamma_i x_i \ldots x_0$$

(i.e., $\alpha_{i+1} A_{i+1} x_{i+1} = \alpha_i \gamma_i$, where the occurrence of A_{i+1} is within either α_i or γ_i), and

$$x_{m+1} x_m \ldots x_0 = \alpha_m \gamma_m x_m \ldots x_0$$

(i.e., $x_{m+1} = \alpha_m \gamma_m$).

From the initial configuration, $NB(G)$ can make a sequence of moves that shift the symbols of x_{m+1} onto the stack, so that the stack contains $x_{m+1}^r Z_0 = \gamma_m^r \alpha_m^r Z_0$. The string γ_m^r is now on top of the stack, so that $NB(G)$ can execute a reduction that leads to the configuration

$$(q_0,\ x_m x_{m-1} \ldots x_0,\ A_m \alpha_m^r Z_0)$$

Suppose in general that for some $i > 0$, there are moves leading to the configuration

$$(q_0,\ x_i x_{i-1} \ldots x_0,\ A_i \alpha_i^r Z_0)$$

$NB(G)$ can then shift the symbols of x_i onto the stack, resulting in the configuration

$$(q_0,\ x_{i-1} \ldots x_0,\ x_i^r A_i \alpha_i^r Z_0) = (q_0,\ x_{i-1} \ldots x_0,\ \gamma_{i-1}^r \alpha_{i-1}^r Z_0)$$

and reduce γ_{i-1} to A_{i-1}, so that the configuration becomes

$$(q_0,\ x_{i-1} \ldots x_0,\ A^{i-1} \alpha_{i-1}^r Z_0)$$

It follows that $NB(G)$ has a sequence of moves that leads to the configuration

$$(q_0,\ x_0,\ A_0 \alpha_0^r Z_0) = (q_0,\ \Lambda,\ S)$$

which allows it to accept x.

Now suppose that a string x is accepted by $NB(G)$. This requires that $NB(G)$ be able to reach the configuration

$$(q_0,\ \Lambda,\ S Z_0)$$

using moves that are either shifts or reductions. Starting in the initial configuration (q_0, x, Z_0), if we denote by x_{m+1} the prefix of x that is transferred to the stack before the first reduction, and assume that in this first step γ_m is reduced to A_m, then the configuration after the reduction looks like

$$(q_0,\ y,\ A_m\alpha_m^r Z_0)$$

for some string α_m, where y is the suffix of x for which $x = x_{m+1}y$. Continuing in this way, if we let the other substrings of x that are shifted onto the stack before each subsequent reduction be $x_m, x_{m-1}, \ldots, x_0$, and assume that for each $i > 0$ the reduction occurring after x_i has been shifted reduces γ_{i-1} to A_{i-1}, so that for some string α_{i-1} the configuration changes from

$$(q_0,\ x_{i-1}\ldots x_0,\ \gamma_{i-1}^r\alpha_{i-1}^r Z_0) \qquad \text{to} \qquad (q_0,\ x_{i-1}\ldots x_0,\ A_{i-1}\alpha_{i-1}^r Z_0)$$

then the sequence of strings

$$S = \alpha_0 A_0 x_0,\ \alpha_1 A_1 x_1 x_0,\ \alpha_2 A_2 x_2 x_1 x_0,\ \ldots,$$
$$\alpha_m A_m x_m x_{m-1}\ldots x_1 x_0,\ x_{m+1}x_m\ldots x_1 x_0 = x$$

constitutes a rightmost derivation of x in the grammar G.

Simplified Algebraic Expressions

EXAMPLE 5.24

We consider the nondeterministic bottom-up PDA $NB(G)$ for the context-free grammar G with productions

$$S \to S + T \mid T \qquad T \to T * a \mid a$$

This is essentially the grammar G_1 in Example 4.20, but without parentheses. We will not give an explicit transition table for the PDA. In addition to shift moves, it has the reductions corresponding to the four productions in the grammar. In Table 5.26 we trace the operation of $NB(G)$ as it processes the string $a + a * a$, for which the derivation tree is shown in Figure 5.25.

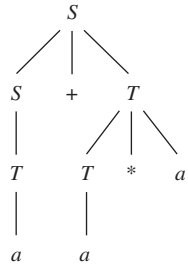


Figure 5.25 | A derivation tree for the string $a + a * a$ in Example 5.24.

In Table 5.26, the five numbers in parentheses refer to the order in which the indicated reduction is performed. In the table, we keep track at each step of the contents of the stack and the unread input. To make it easier to follow, we show the symbols on the stack in reverse order, which is their order when they appear in a current string of the derivation. Each reduction, along with the corresponding step in the rightmost derivation, is shown on the line with the configuration that results from the reduction; in the preceding line, the string that is replaced on the stack is underlined. Of course, as we have discussed, the steps in the derivation go from the bottom to the top of the table. Notice that for each step of the derivation, the current string is obtained by concatenating the reversed stack contents just before the corresponding reduction (excluding Z_0) with the string of unread input.

Table 5.26 | Bottom-Up Processing of $a + a * a$ by $NB(G)$

Reduction	Stack (reversed)	Unread Input	Derivation Step
	Z_0	$a + a * a$	
	$Z_0\ \underline{a}$	$+ a * a$	
(1)	$Z_0\ \underline{T}$	$+ a * a$	$\Rightarrow\ a + a * a$
(2)	$Z_0\ S$	$+ a * a$	$\Rightarrow\ T + a * a$
	$Z_0\ S +$	$a * a$	
	$Z_0\ S + \underline{a}$	$* a$	
(3)	$Z_0\ S + T$	$* a$	$\Rightarrow\ S + a * a$
	$Z_0\ S + T *$	a	
	$Z_0\ S + \underline{T * a}$		
(4)	$Z_0\ \underline{S + T}$		$\Rightarrow\ S + T * a$
(5)	$Z_0\ S$		$\Rightarrow\ S + T$
	(accept)		S

5.4 | A CFG FROM A GIVEN PDA

We will demonstrate in this section that from every pushdown automaton M, a context-free grammar can be constructed that accepts the language $L(M)$. A preliminary step that will simplify the proof is to show that starting with M, another PDA can be obtained that accepts the same language *by empty stack* (rather than the usual way, by final state).

Definition 5.27 Acceptance by Empty Stack

If M is a PDA with input alphabet Σ, initial state q_1, and initial stack symbol Z_1, then M accepts a language L by empty stack if $L = L_e(M)$, where

$$L_e(M) = \{x \in \Sigma^* \mid (q_1, x, Z_1) \vdash_M^* (q, \Lambda, \Lambda) \quad \text{for some state } q\}$$

Theorem 5.28

If $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ is a PDA, then there is another PDA M_1 such that $L_e(M_1) = L(M)$.

Sketch of Proof The idea of the proof is to let M_1 process an input string the same way that M does, except that when M enters an accepting state, and only when this happens, M_1 empties its stack.

If we make M_1 a duplicate of M but able to empty its stack automatically whenever M enters an accepting state, then we obtain part of what we want: Every time M accepts a string x, M_1 will accept x by empty stack. This is not quite what we want, however, because M might terminate its computation in a nonaccepting state by emptying

its stack, and if M_1 does the same thing, it will accept a string that M doesn't. To avoid this, we give M_1 a different initial stack symbol and let its first move be to push M's initial stack symbol on top of its own; this allows it to avoid emptying its stack until it should accept.

The way we allow M_1 to empty its stack automatically when M enters an accepting state is to provide it with a Λ-transition from each accepting state to a special "stack-emptying" state, from which there are Λ-transitions back to itself that pop every symbol off the stack until the stack is empty.

Now we must discuss how to start with a PDA M that accepts a language by empty stack, and find a CFG G generating $L_e(M)$. In Section 5.3, starting with a CFG, we constructed the nondeterministic top-down PDA so that in the process of accepting a string in $L(G)$, it simulated a leftmost derivation of the string. It will be helpful in starting with the PDA to try to construct a grammar in a way that will preserve as much as possible of this CFG-PDA correspondence—so that a sequence of moves by which M accepts a string can be interpreted as simulating the steps in a derivation of the string in the grammar.

When the top-down PDA corresponding to a grammar G makes a sequence of moves to accept x, the configuration of the PDA at each step is such that the string of input symbols read so far, followed by the string on the stack, is the current string in a derivation of x in G. One very simple way we might to try to preserve this feature, if we are starting with a PDA and trying to construct a grammar, is to ignore the states, let variables in the grammar be the stack symbols of the PDA, let the start symbol of the grammar be the initial stack symbol Z_0, and for each move that reads a and replaces A on the stack by $BC \ldots D$, introduce the production

$$A \rightarrow aBC \ldots D$$

An initial move $(q_0, ab \ldots, Z_0) \vdash (q_1, b \ldots, ABZ_0)$, which reads a and replaces Z_0 by ABZ_0, would correspond to the first step

$$Z_0 \Rightarrow aABZ_0$$

in a derivation; a second move $(q_1, b \ldots, ABZ_0) \vdash (q_2, \ldots, CBZ_0)$ that replaced A by C on the stack would allow the second step

$$aABZ_0 \Rightarrow abCBZ_0$$

and so forth. The states don't show up in the derivation steps at all, but the current string at each step is precisely the string of terminals read so far, followed by the symbols (variables) on the stack. The fact that productions in the grammar end in strings of variables would help to highlight the grammar-PDA correspondence: In order to produce a string of terminal symbols from $abCBZ_0$, for example, we need eventually to eliminate the variables C, B, and Z_0 from

the string, and in order to accept the input string starting with ab by empty stack, we need eventually to eliminate the stack symbols C, B, and Z_0 from the stack.

You may suspect that "ignore the states" sounds a little too simple to work in general. It does allow the grammar to generate all the strings that the PDA accepts, but it may also generate strings that are not accepted (see Exercise 5.35).

Although this approach must be modified, the essential idea can still be used. Rather than using the stack symbols themselves as variables, we try things of the form

$$[p, A, q]$$

where p and q are states. For the variable $[p, A, q]$ to be replaced directly by σ (either a terminal symbol or Λ), there must be a PDA move that reads σ, pops A from the stack, and takes M from state p to state q. More general productions involving $[p, A, q]$ represent *sequences* of moves that take M from state p to q and have the ultimate effect of removing A from the stack.

If the variable $[p, A, q]$ appears in the current string of a derivation, then completing the derivation requires that the variable be eliminated, perhaps replaced by Λ or a terminal symbol. This will be possible if there is a move that takes M from p to q and pops A from the stack. Suppose instead, however, that there is a move from p to p_1 that reads a and replaces A on the stack by $B_1 B_2 \ldots B_m$. It is appropriate to introduce σ into our current string at this point, since we want the prefix of terminals in our string to correspond to the input read so far. The new string in the derivation should reflect the fact that the new symbols $B_1, \ldots, B_m$ have been added to the stack. The most direct way to eliminate these new symbols is to start in p_1 and make moves ending in a new state p_2 that remove B_1 from the stack; then to make moves to p_3 that remove B_2; $\ldots$; to move from p_{m-1} to p_m and remove B_{m-1}; and finally to move from p_m to q and remove B_m. It doesn't matter what the states $p_2, p_3, \ldots, p_m$ are, and we allow every string of the form

$$\sigma[p_1, B, p_2][p_2, B_2, p_3] \ldots [p_m, B_m, q]$$

to replace $[p, A, q]$ in the current string. The actual moves of the PDA may not accomplish these steps directly (one or more of the intermediate steps represented by $[p_i, B_i, p_{i+1}]$ may need to be refined further), but this is an appropriate restatement at this point of what must eventually happen in order for the derivation to be completed. Thus, we will introduce into our grammar the production

$$[p, A, q] \rightarrow \sigma[p_1, B, p_2][p_2, B_2, p_3] \ldots [p_m, B_m, q]$$

for every possible sequence of states $p_2, \ldots, p_m$. Some such sequences will be dead ends, because there will be no moves following the particular sequence of states and having the ultimate effect represented by this sequence of variables. But no harm is done by introducing all these productions, because for every derivation in which one of the dead-end sequences appears, there will be

at least one variable that cannot be eliminated from the string, and the derivation will not produce a string of terminals. If we use S to denote the start symbol of the grammar, the productions that we need at the beginning are those of the form

$$S \to [q_0, Z_0, q]$$

where q_0 is the initial state. When we accept strings by empty stack, the final state is irrelevant, and we include a production of this type for every possible state q.

Theorem 5.29

If $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ is a pushdown automaton accepting L by empty stack, then there is a context-free grammar G such that $L = L(G)$.

Proof

We define $G = (V, \Sigma, S, P)$ as follows. V contains S as well as all possible variables of the form $[p, A, q]$, where $A \in \Gamma$ and $p, q \in Q$. P contains the following productions:

1. For every $q \in Q$, the production $S \to [q_0, Z_0, q]$ is in P.
2. For every $q, q_1 \in Q$, every $\sigma \in \Sigma \cup \{\Lambda\}$, and every $A \in \Gamma$, if $\delta(q, \sigma, A)$ contains (q_1, Λ), then the production $[q, A, q_1] \to \sigma$ is in P.
3. For every $q, q_1 \in Q$, every $\sigma \in \Sigma \cup \{\Lambda\}$, every $A \in \Gamma$, and every $m \geq 1$, if $\delta(q, \sigma, A)$ contains $(q_1, B_1B_2 \dots B_m)$ for some $B_1, B_2, \dots, B_m$ in Γ, then for every choice of $q_2, q_3, \dots, q_{m+1}$ in Q, the production

$$[q, A, q_{m+1}] \to \sigma[q_1, B_1, q_2][q_2, B_2, q_3] \dots [q_m, B_m, q_{m+1}]$$

is in P.

The idea of the proof is to characterize the strings of terminals that can be derived from a variable $[q, A, q']$—specifically, to show that for every $q, q' \in Q$, every $A \in \Gamma$, and every $x \in \Sigma^*$,

$$(1) \qquad [q, A, q'] \Rightarrow_G^* x \text{ if and only if } (q, x, A) \vdash_M^* (q', \Lambda, \Lambda)$$

If $x \in L_e(M)$, then formula (1) will imply that $(q_0, x, Z_0) \vdash_M^* (q, \Lambda, \Lambda)$ for some $q \in Q$, because it implies that $[q_0, Z_0, q] \Rightarrow_G^* x$ for some q, and we can start a derivation of x with a production of the first type. On the other hand, if $x \in L(G)$, then the first step of every derivation of x must be $S \Rightarrow [q_0, Z_0, q]$, for some $q \in Q$; this means that $[q_0, Z_0, q] \Rightarrow_G^* x$, and formula (1) then implies that $x \in L_e(M)$.

Both parts of formula (1) are proved using mathematical induction. First we show that for every $n \geq 1$,

$$(2) \qquad \text{If } [q, A, q'] \Rightarrow_G^n x, \text{ then } (q, x, A) \vdash^* (q', \Lambda, \Lambda)$$

In the basis step, the only production that allows this one-step derivation is one of the second type, and this is possible only if x is either Λ or an element of Σ and $\delta(q, x, A)$ contains (q', Λ). It follows that $(q, x, A) \vdash (q', \Lambda, \Lambda)$.

Suppose that $k \geq 1$ and that for every $n \leq k$, whenever $[q, A, q'] \Rightarrow^n x$, $(q, x, A) \vdash^* (q', \Lambda, \Lambda)$. Now suppose that $[q, A, q'] \Rightarrow^{k+1} x$. We wish to show that $(q, x, A) \vdash^* (q', \Lambda, \Lambda)$. Since $k \geq 1$, the first step in the derivation of x must be

$$[q, A, q'] \Rightarrow \sigma[q_1, B_1, q_2][q_2, B_2, q_3] \ldots [q_m, B_m, q']$$

for some some $m \geq 1$, some $\sigma \in \Sigma \cup \{\Lambda\}$, some sequence $B_1, B_2, \ldots, B_m$ in Γ, and some sequence $q_1, q_2, \ldots, q_m$ in Q, so that $\delta(q, \sigma, A)$ contains $(q_1, B_2 \ldots B_m)$. The remainder of the derivation takes each of the variables $[q_i, B_i, q_{i+1}]$ to a string x_i and the variable $[q_m, B_m, q']$ to a string x_m. The strings $x_1, \ldots, x_m$ satisfy the formula $\sigma x_1 \ldots x_m = x$, and each x_i is derived from its respective variable in k or fewer steps. The induction hypothesis implies that for each i with $1 \leq i \leq m - 1$,

$$(q_i, x_i, B_i) \vdash^* (q_{i+1}, \Lambda, \Lambda)$$

and that

$$(q_m, x_m, B_m) \vdash^* (q', \Lambda, \Lambda)$$

If M is in the configuration $(q, x, A) = (q, \sigma x_1 x_2 \ldots x_m, A)$, then because $\delta(q, \sigma, A)$ contains $(q_1, B_1 \ldots B_m)$, M can move in one step to the configuration

$$(q_1, x_1 x_2 \ldots x_m, B_1 B_2 \ldots B_m)$$

It can then move in a sequence of steps to

$$(q_2, x_2 \ldots x_m, B_2 \ldots B_m)$$

then to $(q_3, x_3 \ldots x_m, B_3 \ldots B_m)$, and ultimately to (q', Λ, Λ). Statement (2) then follows.

We complete the proof of statement (1) by showing that for every $n \geq 1$,

$$(3) \qquad \text{If } (q, x, A) \vdash^n (q', \Lambda, \Lambda), \text{ then } [q, A, q'] \Rightarrow^* x$$

In the case $n = 1$, a string x satisfying the hypothesis in statement (3) must be of length 0 or 1, and $\delta(q, x, A)$ must contain (q', Λ). We may then derive x from $[q, A, q']$ using a production of the second type.

Suppose that $k \geq 1$ and that for every $n \leq k$ and every combination of $q, q' \in Q$, $x \in \Sigma^*$, and $A \in \Gamma$, if $(q, x, A) \vdash^n (q', \Lambda, \Lambda)$, then $[q, A, q'] \Rightarrow^* x$. We wish to show that if $(q, x, A) \vdash^{k+1} (q', \Lambda, \Lambda)$, then $[q, A, q'] \Rightarrow^* x$. We know that for some $\sigma \in \Sigma \cup \{\Lambda\}$ and some $y \in \Sigma^*$, $x = \sigma y$ and the first of the $k + 1$ moves is

$$(q, x, A) = (q, \sigma y, A) \vdash (q_1, y, B_1 B_2 \dots B_m)$$

Here $m \geq 1$, since $k \geq 1$, and the B_i's are elements of Γ. In other words, $\delta(q, \sigma, A)$ contains $(q_1, B_1 \dots B_m)$. The k subsequent moves end in the configuration (q', Λ, Λ); therefore, for each i with $1 \leq i \leq m$ there must be intermediate points at which the stack contains precisely the string $B_i B_{i+1} \dots B_m$. For each such i, let q_i be the state M is in the first time the stack contains $B_i \dots B_m$, and let x_i be the portion of the input string that is consumed in going from q_i to q_{i+1} (or, if $i = m$, in going from q_m to the configuration (q', Λ, Λ)). Then

$$(q_i, x_i, B_i) \vdash^* (q_{i+1}, \Lambda, \Lambda)$$

for each i with $1 \leq i \leq m - 1$, and

$$(q_m, x_m, B_m) \vdash^* (q', \Lambda, \Lambda)$$

where each of the indicated sequences of moves has k or fewer. Therefore, by the induction hypothesis,

$$[q_i, B_i, q_{i+1}] \Rightarrow^* x_i$$

for each i with $1 \leq i \leq m - 1$, and

$$[q_m, B_m, q'] \Rightarrow^* x_m$$

Since $\delta(q, a, A)$ contains $(q_1, B_1 \dots B_m)$, we know that

$$[q, A, q'] \Rightarrow \sigma [q_1, B_1, q_2][q_2, B_2, q_3] \dots [q_m, B_m, q']$$

(this is a production of type 3), and we may conclude that

$$[q, A, q'] \Rightarrow^* \sigma x_1 x_2 \dots x_m = x$$

This completes the induction and the proof of the theorem.

A CFG from a PDA Accepting *SimplePal* **EXAMPLE 5.30**

We return to the language $SimplePal = \{xcx^r \mid x \in \{a, b\}^*\}$ from Example 5.3. The transition table in Table 5.31 is modified in that the letters on the stack are uppercase and the PDA accepts by empty stack.

In the grammar $G = (V, \Sigma, S, P)$ obtained from the construction in Theorem 5.29, V contains S as well as every variable of the form $[p, X, q]$, where X is a stack symbol

Table 5.31 | A PDA Accepting *SimplePal* by Empty Stack

Move Number	State	Input	Stack Symbol	Move(s)
1	q_0	a	Z_0	(q_0, AZ_0)
2	q_0	b	Z_0	(q_0, BZ_0)
3	q_0	a	A	(q_0, AA)
4	q_0	b	A	(q_0, BA)
5	q_0	a	B	(q_0, AB)
6	q_0	b	B	(q_0, BB)
7	q_0	c	Z_0	(q_1, Z_0)
8	q_0	c	A	(q_1, A)
9	q_0	c	B	(q_1, B)
10	q_1	a	A	(q_1, Λ)
11	q_1	b	B	(q_1, Λ)
12	q_1	Λ	Z_0	(q_1, Λ)
	(all other combinations)			none

and p and q can each be either q_0 or q_1. Productions of the following types are contained in P:

$$
\begin{array}{lcll}
(0) & S & \rightarrow & [q_0, Z_0, q] \\
(1) & [q_0, Z_0, q] & \rightarrow & a[q_0, A, p][p, Z_0, q] \\
(2) & [q_0, Z_0, q] & \rightarrow & b[q_0, B, p][p, Z_0, q] \\
(3) & [q_0, A, q] & \rightarrow & a[q_0, A, p][p, A, q] \\
(4) & [q_0, A, q] & \rightarrow & b[q_0, B, p][p, A, q] \\
(5) & [q_0, B, q] & \rightarrow & a[q_0, A, p][p, B, q] \\
(6) & [q_0, B, q] & \rightarrow & b[q_0, B, p][p, B, q] \\
(7) & [q_0, Z_0, q] & \rightarrow & c[q_1, Z_0, q] \\
(8) & [q_0, A, q] & \rightarrow & c[q_1, A, q] \\
(9) & [q_0, B, q] & \rightarrow & c[q_1, B, q] \\
(10) & [q_1, A, q_1] & \rightarrow & a \\
(11) & [q_1, B, q_1] & \rightarrow & b \\
(12) & [q_1, Z_0, q_1] & \rightarrow & \Lambda
\end{array}
$$

Allowing all combinations of p and q gives 35 productions in all.

The PDA accepts the string *bacab* by the sequence of moves

$$
\begin{aligned}
(q_0, bacab, Z_0) &\vdash (q_0, acab, BZ_0) \\
&\vdash (q_0, cab, ABZ_0) \\
&\vdash (q_1, ab, ABZ_0) \\
&\vdash (q_1, b, BZ_0) \\
&\vdash (q_1, \Lambda, Z_0) \\
&\vdash (q_1, \Lambda, \Lambda)
\end{aligned}
$$

The corresponding leftmost derivation in the grammar is

$$\begin{aligned} S &\Rightarrow [q_0, Z_0, q_1] \\ &\Rightarrow b[q_0, B, q_1][q_1, Z_0, q_1] \\ &\Rightarrow ba[q_0, A, q_1][q_1, B, q_1][q_1, Z_0, q_1] \\ &\Rightarrow bac[q_1, A, q_1][q_1, B, q_1][q_1, Z_0, q_1] \\ &\Rightarrow baca[q_1, B, q_1][q_1, Z_0, q_1] \\ &\Rightarrow bacab[q_1, Z_0, q_1] \\ &\Rightarrow bacab \end{aligned}$$

From the sequence of PDA moves, it may look as though there are several choices of leftmost derivations. For example, we might start with the production $S \rightarrow [q_0, Z_0, q_0]$. Remember, however, that $[q_0, Z_0, q]$ represents a sequence of moves from q_0 to q that has the ultimate effect of removing Z_0 from the stack. Since the PDA ends up in state q_1, it is clear that q should be q_1. Similarly, it may seem as if the second step could be

$$[q_0, Z_0, q_1] \Rightarrow b[q_0, B, q_0][q_0, Z_0, q_1]$$

However, the sequence of PDA moves that starts in q_0 and eliminates B from the stack ends with the PDA in state q_1, not q_0. In fact, because every move to state q_0 adds to the stack, no string of terminals can ever be derived from the variable $[q_0, B, q_0]$.

5.5 | PARSING

To *parse* a sentence or an expression means to determine its syntax, or its grammatical structure. To parse a string relative to a context-free grammar G means to determine an appropriate derivation for the string in G, or to determine that there is none. Parsing a sentence is necessary to understand it; parsing an expression makes it possible to evaluate it correctly; parsing a statement in a programming language makes it possible to execute it correctly or translate it correctly into machine language.

Section 5.3 described two general ways of obtaining a pushdown automaton from a grammar so that a sequence of moves by which a string is accepted corresponds to a derivation of the string in the grammar. In this section we consider two examples involving simple grammars. In the first case, taking advantage of the information available at each step from the input and the stack allows us to eliminate the inherent nondeterminism from the top-down PDA, so as to arrive at a rudimentary parser. In the second case, we can do the same thing with the bottom-up PDA.

Because not every CFL can be accepted by a deterministic PDA, this goal is not always achievable, and even for DCFLs the grammar that we start with may be inappropriate and the process is not always straightforward. This section is not intended to be a comprehensive survey of parsing techniques, but it may at least serve as a starting point for a discussion of the development of efficient parsers.

EXAMPLE 5.32 A Top-Down Parser for *Balanced*

In Section 5.3 we considered the CFG with productions $S \to [S] \mid SS \mid \Lambda$. The way we have formulated the top-down PDA involves inherent nondeterminism. Each time a variable appears on top of the stack, the machine replaces it by the right side of a production. These moves depend only on the variable; they are Λ-transitions and are chosen nondeterministically. The most obvious approach to eliminating the nondeterminism is to consider whether consulting the next input symbol as well would allow the PDA to choose the correct move.

The first few moves made for the input string `[[][]]` are enough to show that in this example, looking at the next input symbol is not enough.

1.	$(q_0, [[][]], Z_0)$		
2.	$\vdash (q_1, [[][]], SZ_0)$	S	
3.	$\vdash (q_1, [[][]], [S]Z_0)$	$\Rightarrow$	$[S]$
4.	$\vdash (q_1, [][]], S]Z_0)$		
5.	$\vdash (q_1, [][]], SS]Z_0)$	$\Rightarrow$	$[SS]$

In both lines 2 and 4, S is on top of the stack and the next input symbol is a left parenthesis, but S is replaced by different strings in these two places: $[S]$ in line 3 and SS in line 5.

We might have guessed in advance that we would have problems like this, for at least two reasons. First, the only variable in this grammar is S, and there are more productions than there are terminal symbols for the right sides to start with. Second, the grammar is ambiguous: Even if the stack and the next input symbol are enough to determine the next move for a particular leftmost derivation, there may be more than one LMD.

Let's try another grammar for this language. It can be shown (Exercises 4.44 and 4.45) that the CFG G with productions

$$S \to [S]S \mid \Lambda$$

is an unambiguous grammar that also generates the language *Balanced*. We begin by trying the nondeterministic top-down PDA $NT(G)$ on the input string `[]`; we will see that there is still a slight problem with eliminating the nondeterminism, but it will not be too serious.

1.	$(q_0, [], Z_0)$		
2.	$\vdash (q_1, [], SZ_0)$	S	
3.	$\vdash (q_1, [], [S]SZ_0)$	$\Rightarrow$	$[S]S$
4.	$\vdash (q_1,], S]SZ_0)$		
5.	$\vdash (q_1,],]SZ_0)$	$\Rightarrow$	$[]S$
6.	$\vdash (q_1, \Lambda, SZ_0)$		
7.	$\vdash (q_1, \Lambda, Z_0)$	$\Rightarrow$	$[]$

During these moves, there are three times when S is the top stack symbol. In the first two cases, the move is what we might have hoped: If the next input is `[`, the move is to replace S by $[S]S$, and if it is `]`, the move is to replace S by Λ. Although this string is hardly enough of a test, longer strings will result in the same behavior (see the discussion below). Unfortunately, S is also replaced by Λ at the end of the computation, when the input string has been completely read. This is unfortunate because the PDA must make a Λ-transition in this case, and if it can choose between reading an input symbol and making a Λ-transition when S is on the stack, then it cannot be deterministic.

Table 5.33 | Transition Table for $NT(G_1)$

Move Number	State	Input	Stack Symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	$(q_1, S_1\$)$
3	q_1	Λ	S_1	$(q_1, [\,S_1\,]\,S_1), (q_1, \Lambda)$
4	q_1	[	[	(q_1, Λ)
5	q_1	]	]	(q_1, Λ)
6	q_1	$	$	(q_1, Λ)
7	q_1	Λ	Z_0	(q_2, Z_0)

A similar problem arises for many CFGs. Even though at every step before the last one, looking at the next input symbol is enough to decide which production to apply to the variable on the stack, a Λ-transition at the end of the computation seems to require nondeterminism. We resolve the problem by introducing an *end-marker* \$ into the alphabet and changing the language slightly. Every string in the new language will be required to end with \$, and this symbol will be used only at the ends of strings in the language. Our new grammar G_1 has the productions

$$S \to S_1\$ \qquad S_1 \to [\,S_1\,]\,S_1 \mid \Lambda$$

Table 5.33 is a transition table for the nondeterministic top-down PDA $NT(G_1)$.
Line 3 is the only part of the table that needs to be changed in order to eliminate the nondeterminism. It is replaced by these lines:

State	Input	Stack Symbol	Move
q_1	[	S_1	$(q_{1,[}, [\,S_1\,]\,S_1)$
$q_{1,[}$	Λ	[	(q_1, Λ)
q_1	]	S_1	$(q_{1,]}, \Lambda)$
$q_{1,]}$	Λ	]	(q_1, Λ)
q_1	$	S_1	$(q_{1,\$}, \Lambda)$
$q_{1,\$}$	Λ	$	(q_1, Λ)

In the cases when S_1 is on the stack and the next input symbol is] or \$, replacing S_1 by Λ will be appropriate only if the symbol beneath it on the stack matches the input symbol. The corresponding states $q_{1,]}$ and $q_{1,\$}$ allow the PDA to pop S_1 and remember, when the stack symbol below it comes to the top, which input symbol it is supposed to match. In either state, the only correct move is to pop the appropriate symbol from the stack and return to q_1. For the sake of consistency, we use the state $q_{1,[}$ for the case when the top stack symbol is S_1 and the next input is [. Although in this case S_1 is replaced by $[\,S_1\,]\,S_1$, the move from $q_{1,[}$ is also to pop the [and return to q_1. The alternative, which would be more efficient, is to replace these two moves by a single one that doesn't change state and replaces S_1 by $S_1\,]\,S_1$.

The modified PDA is clearly deterministic. To check that it is equivalent to the original, it is enough to verify that our assumptions about the correct Λ-transition made by $NT(G_1)$

in each case where S_1 is on the stack are correct. If the next input symbol is either] or \$, it cannot be correct for the PDA to replace S_1 by $[\,S_1\,]\,S_1$, because the terminal symbol on top of the stack would not match the input. If the next input symbol is [, replacing S_1 by Λ could be correct only if the symbol below S_1 were either [or S_1. Here it is helpful to remember that at each step M makes in the process of accepting a string x, the string of input symbols already read, followed by the stack contents exclusive of Z_0, is the current string in a derivation of x. Since the current string in a leftmost derivation of x cannot contain either the substring $S_1\,[$ or $S_1 S_1$, neither of these two situations can occur.

Not only does the deterministic PDA accept the same strings as $NT(G_1)$, but it preserves the close correspondence between the moves made in accepting a string and the leftmost derivation of the string. We trace the moves for the simple input string []\$ and show the corresponding derivation.

$$
\begin{array}{lll}
(q_0, [\,]\$, Z_0) & & \\
\quad \vdash (q_1, [\,]\$, SZ_0) & S & \\
\quad \vdash (q_1, [\,]\$, S_1\$Z_0) & \Rightarrow & S_1\$ \\
\quad \vdash (q_{1,[},]\$, [\,S_1\,]\,S_1\$Z_0) & \Rightarrow & [\,S_1\,]\,S_1\$ \\
\quad \vdash (q_1,]\$, S_1\,]\,S_1\$Z_0) & & \\
\quad \vdash (q_{1,]}, \$,]\,S_1\$Z_0) & \Rightarrow & [\,]\,S_1\$ \\
\quad \vdash (q_1, \$, S_1\$Z_0) & & \\
\quad \vdash (q_{1,\$}, \Lambda, \$Z_0) & \Rightarrow & [\,]\$ \\
\quad \vdash (q_1, \Lambda, Z_0) & & \\
\quad \vdash (q_2, \Lambda, Z_0) & &
\end{array}
$$

This example illustrates the fact that we have a parser for this CFG. In order to obtain the leftmost derivation, or the derivation tree, for a string $x \in L(G_1)$, all we have to do is give x to the DPDA and watch the moves it makes.

G_1 is an example of an LL(1) grammar, one in which the nondeterministic top-down PDA corresponding to the grammar can be converted into a deterministic top-down parser, simply by allowing the PDA to "look ahead" to the next input symbol in order to decide what move to make. A grammar is LL(k) if looking ahead k symbols in the input is always enough to choose the next move. Such a grammar makes it possible to construct a deterministic top-down parser, and there are systematic methods for determining whether a grammar is LL(k) and for carrying out such a construction if it is. In some simple cases, a CFG that is not LL(1) may be transformed using straightforward techniques into one that is (Exercises 5.39-5.41).

EXAMPLE 5.34 A Bottom-Up Parser for *SimpleExpr*

In this example we return to the CFG in Example 5.24, with productions

$$S \to S + T \mid T \qquad T \to T * a \mid a$$

except that for the same reasons as in Example 5.11 we add the end-marker \$ to the end of every string in the language. G will be the modified grammar, with productions

$$S \to S_1\$ \qquad S_1 \to S_1 + T \mid T \qquad T \to T * a \mid a$$

In the nondeterministic bottom-up PDA $NB(G)$, the two types of moves, other than the initial move and the moves to accept, are shifts and reductions. A shift reads an input symbol and pushes it onto the stack, and a reduction removes a string α^r from the stack and replaces it by the variable A in a production $A \to \alpha$. There are five possible reductions in the PDA $NB(G)$, two requiring only one move and three requiring two or more. Nondeterminism arises in two ways: first, in choosing whether to shift or to reduce, and second, in making a choice of reductions if more than one is possible. There is only one correct move at each step, and G turns out to be a grammar for which the combination of input symbol and stack symbol will allow us to find it.

One basic principle is that if there is a reduction that should be made, then it should be made as soon as it is possible. For example, if a is on top of the stack, then a reduction, not a shift, is the move to make. Any symbol that went onto the stack now would have to be removed later in order to execute the reduction of a or $T * a$ to T; the only way to remove it is to do another reduction; and because the derivation tree is built from the bottom up, any subsequent reduction to construct the portion of the tree involving the a node needs its parent, not the node itself.

Another principle that is helpful is that in the moves made by $NB(G)$, the current string in the derivation being simulated contains the reverse of the stack contents (not including Z_0) followed by the remaining input. We can use this to answer the question of which reduction to execute if there seems to be a choice. If $a * T$ (the reverse of $T * a$) were on top of the stack, and we simply reduced a to T, then the current string in a rightmost derivation would contain the substring $T * T$, which is never possible. Similarly, if $T + S_1$, the reverse of $S_1 + T$, is on top, then reducing T to S_1 can't be correct. These two situations can be summarized by saying that when a reduction is executed, the longest possible string should be removed from the stack during the reduction.

Essentially the only remaining question is what to do when T is the top stack symbol: shift or reduce? We can answer this by considering the possibilities for the next input symbol. If it is $+$, then this $+$ should eventually be part of the expression $S_1 + T$ (in reverse) on the stack, so that $S_1 + T$ can be reduced to S_1; therefore, the T should eventually be reduced to S_1, and so the time to do it is now. Similarly, if the next input is \$, T should be reduced to S_1 so that $S_1\$$ can be reduced to S. The only other symbol that can follow T in a rightmost derivation is $*$, and $*$ cannot follow any other symbol; therefore, $*$ should be shifted onto the stack.

With these observations, we can formulate the rules the deterministic bottom-up PDA should follow in choosing its move.

1. If the top stack symbol is Z_0, S_1, $+$, or $*$, shift the next input symbol to the stack. (None of these four symbols is the rightmost symbol in the right side of a production.)
2. If the top stack symbol is \$, reduce $S_1\$$ to S.
3. If the top stack symbol is a, reduce $T * a$ to T if possible; otherwise reduce a to T.
4. If the top stack symbol is T, then reduce if the next input is $+$ or \$, and otherwise shift.

Table 5.35 | Bottom-Up Processing of $a + a * a\$$ by the DPDA

Rule	Stack (Reversed)	Unread Input	Derivation Step
	Z_0	$a + a * a\ \$$	
1	$Z_0\ a$	$+ a * a\ \$$	
3	$Z_0\ T$	$+ a * a\ \$$	$\Rightarrow\ a + a * a\$$
4 (red.)	$Z_0\ S_1\ +$	$a * a\ \$$	$\Rightarrow\ T + a * a\$$
1	$Z_0\ S_1 + a$	$* a\ \$$	
3	$Z_0\ S_1 + T$	$* a\ \$$	$\Rightarrow\ S_1 + a * a\$$
4 (shift)	$Z_0\ S_1 + T\ *$	$a\ \$$	
1	$Z_0\ S_1 + T * a$	$\$$	
3	$Z_0\ S_1 + T$	$\$$	$\Rightarrow\ S_1 + T * a\$$
4 (red.)	$Z_0\ S_1\ \$$		$\Rightarrow\ S_1 + T\$$
2	$Z_0\ S$		$\Rightarrow\ S_1\$$
5			S

5. If the top stack symbol is S, then pop it from the stack; if Z_0 is then the top symbol, accept, and otherwise reject.

All these rules except the fourth one are easily incorporated into a transition table for a deterministic PDA, but the fourth may require a little clarification. If the PDA sees the combination $(*, T)$ of next input symbol and stack symbol, then it shifts $*$ onto the stack. If it sees $(+, T)$ or $(\$, T)$, then the moves it makes are ones that carry out the appropriate reduction and *then* shift the input symbol onto the stack. The point is that "seeing" $(+, T)$, for example, implies reading the $+$, and the PDA then uses auxiliary states to remember that after it has performed a reduction, it should place $+$ on the stack.

We now have the essential specifications for a deterministic PDA; like $NB(G)$, the moves it makes in accepting a string simulate in reverse the steps in a rightmost derivation of that string, and in this sense our PDA serves as a bottom-up parser for G. The grammar G is an example of a *weak precedence grammar*, which is perhaps the simplest type of *precedence grammar*. The phrase refers to a precedence relation (a relation from Σ to Γ—i.e., a subset of $\Sigma \times \Gamma$), where Σ is the set of terminals and Γ is the set containing terminals and variables as well as Z_0. In grammars of this type, a precedence relation can be used to obtain a deterministic shift-reduce parser. In our example, the relation is simply the set of pairs $(a, X) \in \Sigma \times \Gamma$ for which a reduction is appropriate.

In Table 5.26 we traced the moves of the nondeterministic bottom-up parser corresponding to the original version of the grammar (without the end-marker), as it processed the string $a + a * a$. The moves for our DPDA and the input string $a + a * a\$$, shown in Table 5.35 above, are similar; this time, for each configuration, we show the rule (from the set of five above) that the PDA used to get there, and, in the case of rule 4, whether the move was a shift or a reduction.

EXERCISES

5.1. a. For the PDA in Table 5.4, trace the sequence of moves made for the input strings ab, aab, and abb.

b. For the PDA in Table 5.6, trace the sequence of moves made for the input strings *bacab* and *baca*.

5.2. For the PDA in Table 5.8, trace every possible sequence of moves for each of the two input strings *aba* and *aab*.

5.3. For an input string $x \in \{a, b\}^*$ with $|x| = n$, how many possible complete sequences of moves (sequences of moves that start in the initial configuration (q_0, x, Z_0) and terminate in a configuration from which no move is possible) can the PDA in Table 5.8 make? It is helpful to remember that once the PDA reaches the state q_1, there is no choice of moves.

5.4. Consider the PDA in Table 5.8, and for each of the following languages over $\{a, b\}$, modify it to obtain a PDA accepting the language.

a. The language of even-length palindromes.

b. The language of odd-length palindromes.

5.5. Give transition tables for PDAs accepting each of the following languages.

a. The language of all odd-length strings over $\{a, b\}$ with middle symbol a.

b. $\{a^n x \mid n \geq 0,\ x \in \{a, b\}^* \text{ and } |x| \leq n\}$.

c. $\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } j = i \text{ or } j = k\}$.

5.6. In both cases below, a transition table is given for a PDA with initial state q_0 and accepting state q_2. Describe in each case the language that is accepted.

Move Number	State	Input	Stack Symbol	Move(s)
1	q_0	a	Z_0	(q_1, aZ_0)
2	q_0	b	Z_0	(q_1, bZ_0)
3	q_1	a	a	$(q_1, a), (q_2, a)$
4	q_1	b	a	(q_1, a)
5	q_1	a	b	(q_1, b)
6	q_1	b	b	$(q_1, b), (q_2, b)$
(all other combinations)				none

Move Number	State	Input	Stack Symbol	Move(s)
1	q_0	a	Z_0	(q_0, XZ_0)
2	q_0	b	Z_0	(q_0, XZ_0)
3	q_0	a	X	(q_0, XX)
4	q_0	b	X	(q_0, XX)
5	q_0	c	X	(q_1, X)
6	q_0	c	Z_0	(q_1, Z_0)
7	q_1	a	X	(q_1, Λ)
8	q_1	b	X	(q_1, Λ)
9	q_1	Λ	Z_0	(q_2, Z_0)
(all other combinations)				none

5.7. What language (a subset of $\{a, b\}^*$) is accepted by the PDA whose transition table is shown below, if the only accepting state is q_3?

Move Number	State	Input	Stack Symbol	Move(s)
1	q_0	a	Z_0	$(q_0, xZ_0), (q_1, aZ_0)$
2	q_0	b	Z_0	$(q_0, xZ_0), (q_1, bZ_0)$
3	q_0	a	x	$(q_0, xx), (q_1, ax)$
4	q_0	b	x	$(q_0, xx)(q_1, bx)$
5	q_1	a	a	(q_1, a)
6	q_1	b	b	(q_1, b)
7	q_1	a	b	$(q_1, b), (q_2, \Lambda)$
8	q_1	b	a	$(q_1, a), (q_2, \Lambda)$
9	q_2	a	x	(q_2, Λ)
10	q_2	b	x	(q_2, Λ)
11	q_2	Λ	Z_0	(q_3, Z_0)
(all other combinations)				none

The PDA can stay in state q_0 by pushing x onto the stack for each input symbol read. From q_0 it also has the choice of entering q_1, by pushing onto the stack the symbol it has just read. In state q_1 there is always the option of ignoring the input symbol that is read and leaving the stack alone, but in order to reach the accepting state it must eventually be able to move from q_1 to q_2.

5.8. Give transition tables for PDAs accepting each of the following languages.

a. $\{a^i b^j \mid i \leq j \leq 2i\}$

b. $\{x \in \{a, b\}^* \mid n_a(x) < n_b(x) < 2n_a(x)\}$

5.9. Table 5.6 shows the transitions for a PDA accepting *SimplePal*. Draw a transition table for another PDA accepting this language and having only two states, the nonaccepting state q_0 and the accepting state q_2. (Use additional stack symbols.)

5.10. Show that every regular language can be accepted by a deterministic PDA M with only two states in which there are no Λ-transitions and no symbols are ever removed from the stack.

5.11. Show that if L is accepted by a PDA, then L is accepted by a PDA in which there are at most two stack symbols in addition to Z_0.

5.12. Show that if L is accepted by a PDA in which no symbols are ever removed from the stack, then L is regular.

5.13. Suppose $L \subseteq \Sigma^*$ is accepted by a PDA M, and that for some fixed k and every $x \in \Sigma^*$, no sequence of moves made by M on input x causes the stack to have more than k elements. Show that L is regular.

5.14. Suppose $L \subseteq \Sigma^*$ is accepted by a PDA M, and for some fixed k, and every $x \in \Sigma^*$, at least one choice of moves allows M to process x completely so that the stack never contains more than k elements. Does it follow that L is regular? Prove your answer.

5.15. Suppose $L \subseteq \Sigma^*$ is accepted by a PDA M, and for some fixed k, and every $x \in L$, at least one choice of moves allows M to accept x in such a way that the stack never contains more than k elements. Does it follow that L is regular? Prove your answer.

5.16. Show that if L is accepted by a PDA, then L is accepted by a PDA that never crashes (i.e., for which the stack never empties and no configuration is reached from which there is no move defined).

5.17. Show that if L is accepted by a PDA, then L is accepted by a PDA in which every move either pops something from the stack (i.e., removes a stack symbol without putting anything else on the stack); or pushes a single symbol onto the stack on top of the symbol that was previously on top; or leaves the stack unchanged.

5.18. For each of the following languages, give a transition table for a deterministic PDA that accepts that language.

a. $\{x \in \{a, b\}^* \mid n_a(x) < n_b(x)\}$

b. $\{x \in \{a, b\}^* \mid n_a(x) \neq n_b(x)\}$

c. $\{x \in \{a, b\}^* \mid n_a(x) = 2n_b(x)\}$

d. $\{a^n b^{n+m} a^m \mid n, m \geq 0\}$

5.19. Suppose M_1 and M_2 are PDAs accepting L_1 and L_2, respectively. For both the languages $L_1 L_2$ and L_1^*, describe a procedure for constructing a PDA accepting the language. In each case, nondeterminism will be necessary. Be sure to say precisely how the stack of the new machine works; no relationship is assumed between the stack alphabets of M_1 and M_2.

5.20. †Show that if L is accepted by a DPDA, then there is a DPDA accepting the language $\{x\#y \mid x \in L \text{ and } xy \in L\}$. (The symbol # is assumed not to occur in any of the strings of L.)

5.21. Prove the converse of Theorem 5.28: If there is a PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ accepting L by empty stack (that is, $x \in L$ if and only if $(q_0, x, Z_0) \vdash_M^* (q, \Lambda, \Lambda)$ for some state q), then there is a PDA M_1 accepting L by final state (i.e., the ordinary way).

5.22. Show that in Exercise 5.21, if M is a deterministic PDA, then M_1 can also be taken to be deterministic.

5.23. Show that if there are strings x and y in the language L such that x is a prefix of y and $x \neq y$, then no DPDA can accept L by empty stack.

5.24. Show that none of the following languages can be accepted by a DPDA. (Determine exactly what properties of the language *Pal* are used in the proof of Theorem 5.16, and show that these languages also have those properties.)

a. The set of even-length palindromes over $\{a, b\}$

b. The set of odd-length palindromes over $\{a, b\}$

c. $\{xx^{\sim} \mid x \in \{a, b\}^*\}$ (where $x^{\sim}$ means the string obtained from x by changing a's to b's and b's to a's)

d. $\{xy \mid x \in \{a, b\}^*$ and y is either x or $x^{\sim}\}$

5.25. A *counter automaton* is a PDA with just two stack symbols, A and Z_0, for which the string on the stack is always of the form $A^n Z_0$ for some $n \geq 0$. (In other words, the only possible change in the stack contents is a change in the number of A's on the stack.) For some context-free languages, such as *AnBn*, the obvious PDA to accept the language is in fact a counter automaton. Construct a counter automaton to accept the given language in each case below.

a. $\{x \in \{a, b\}^* \mid n_a(x) = n_b(x)\}$

b. $\{x \in \{0, 1\}^* \mid n_a(x) = 2n_b(x)\}$

5.26. Suppose that $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ is a deterministic PDA accepting a language L. If x is a string in L, then by definition there is a sequence of moves of M with input x in which all the symbols of x are read. It is conceivable, however, that for some strings $y \notin L$, no sequence of moves causes M to read all of y. This could happen in two ways: M could either crash by not being able to move, or it could enter a loop in which there were infinitely many repeated Λ-transitions. Find an example of a DCFL $L \subseteq \{a, b\}^*$, a string $y \notin L$, and a DPDA M accepting L for which M crashes on y by not being able to move. (Say what L is and what y is, and give a transition table for M.) Note that once you have such an M, it can easily be modified so that y causes it to enter an infinite loop of Λ-transitions.

5.27. a. Give a definition of "balanced string" involving two types of brackets, say `[]` and {}, corresponding to the definition in Example 1.25.

b. Write a transition table for a DPDA accepting the language of balanced strings of these two types of brackets.

5.28. In each case below, you are given a CFG G and a string x that it generates. For the top-down PDA $NT(G)$, trace a sequence of moves by which x is accepted, showing at each step the state, the unread input, and the stack contents. Show at the same time the corresponding leftmost derivation of x in the grammar. See Example 5.19 for a guide.

a. The grammar has productions

$$S \rightarrow S + T \mid T \qquad T \rightarrow T * F \mid F \qquad F \rightarrow [S] \mid a$$

and $x = [a + a * a] * a$.

b. The grammar has productions $S \rightarrow S + S \mid S * S \mid [S] \mid a$, and $x = [a * a + a]$.

c. The grammar has productions $S \rightarrow [S]S \mid \Lambda$, and $x = [][[][]]$.

5.29. Consider the CFG G with productions

$$S \to aB \mid bA \mid \Lambda \qquad A \to aS \mid bAA \qquad B \to bS \mid aBB$$

generating *AEqB*, the nondeterministic bottom-up PDA $NB(G)$, and the input string *aababb*. After the first few moves, the configuration of the PDA is $(q_0, abb, baaZ_0)$. There are two possible remaining sequences of moves that cause the string to be accepted. Write both of them.

5.30. For a certain CFG G, the moves shown below are those by which the nondeterministic bottom-up PDA $NB(G)$ accepts the input string *aabbbab*. Each occurrence of $\vdash^*$ indicates a sequence of moves constituting a reduction. Draw the derivation tree for *aabbbab* that corresponds to this sequence of moves.

$$\begin{aligned}(q_0, aabbbab, Z_0) &\vdash (q_0, abbbab, aZ_0) \vdash (q_0, bbbab, aaZ_0)\\ &\vdash (q_0, bab, baaZ_0) \vdash^* (q_0, bab, SaZ_0)\\ &\vdash (q_0, ab, bSaZ_0) \vdash^* (q_0, ab, SZ_0) \vdash (q_0, b, aSZ_0)\\ &\vdash (q_0, \Lambda, baSZ_0) \vdash^* (q_0, \Lambda, SSZ_0) \vdash^* (q_0, \Lambda, SZ_0)\\ &\vdash (q_1, \Lambda, Z_0) \vdash (q_2, \Lambda, Z_0)\end{aligned}$$

5.31. Let G be the CFG with productions $S \to S + T \mid T \qquad T \to [S] \mid a$. Both parts of the question refer to the moves made by the nondeterministic bottom-up PDA $NB(G)$ in the process of accepting the input string $[a + [a]]$.

a. If the configuration at some point is $(q_0, +[a]], S[Z_0)$, what is the configuration one move later?

b. If the configuration at some point is $(q_0, +[a]], T[Z_0)$, what is the configuration one move later?

5.32. Let M be the PDA in Example 5.7, except that move number 12 is changed to (q_2, Λ), so that M does in fact accept by empty stack. Let $x = ababa$. Find a sequence of moves of M by which x is accepted, and give the corresponding leftmost derivation in the CFG obtained from M as in Theorem 5.29.

5.33. Under what circumstances is the top-down PDA $NT(G)$ deterministic? (For what kind of grammar G, and what kind of language, could this happen?) Can the bottom-up PDA $NB(G)$ ever be deterministic? Explain.

5.34. In each case below, you are given a CFG G and a string x that it generates. For the nondeterministic bottom-up PDA $NB(G)$, trace a sequence of moves by which x is accepted, showing at each step the stack contents and the unread input. Show at the same time the corresponding rightmost derivation of x (in reverse order) in the grammar. See Example 5.24 for a guide.

a. The grammar has productions $S \to S[S] \mid \Lambda$, and $x = [][[]]$.

b. The grammar has productions $S \to [S]S \mid \Lambda$, and $x = [][[]]$.

5.35. Let M be the PDA whose transition table is given in Table 5.31, accepting *SimplePal*. Consider the simplistic preliminary approach to obtaining a CFG described in the discussion preceding Theorem 5.29. The states of M are ignored, the variables of the grammar are the stack symbols of M, and for every move that reads σ and replaces A on the stack by $BC \ldots D$, we introduce the production $A \to \sigma BC \ldots D$. Show that although the string aa is not accepted by M, it is generated by the resulting CFG.

5.36. If the PDA in Theorem 5.29 is deterministic, what nice property does the resulting grammar have? Can it have this property without the original PDA being deterministic?

5.37. Find the other useless variables in the CFG obtained in Example 5.30.

5.38. In each case, the grammar with the given productions satisfies the LL(1) property. For each one, give a transition table for the deterministic PDA obtained as in Example 5.32.

a. $S \to S_1\$ \quad S_1 \to AS_1 \mid \Lambda \quad A \to aA \mid b$

b. $S \to S_1\$ \quad S_1 \to aA \quad A \to aA \mid bA \mid \Lambda$

c. $S \to S_1\$ \quad S_1 \to aAB \mid bBA \quad A \to bS_1 \mid a \quad B \to aS_1 \mid b$

5.39. If G is the CFG with productions $S \to T\$$ and $T \to T[T] \mid \Lambda$, then you can see by considering an input string like `[][][]$`, which has the leftmost derivation

$$S \Rightarrow T\$ \Rightarrow T[T]\$ \Rightarrow T[T][T]\$ \Rightarrow T[T][T][T]\$ \Rightarrow^* [][][]\$$$

that the combination of next input symbol and top stack symbol does not determine the next move. The problem, referred to as *left recursion*, is the production using the variable T whose right side starts with T. In general, if a CFG has the productions $T \to T\alpha \mid \beta$, where β does not begin with T, the left recursion can be eliminated by noticing that the strings derivable from T using these productions are strings of the form $\beta\alpha^n$, where $n \geq 0$. The productions $T \to \beta U$ and $U \to \alpha U \mid \Lambda$ then generate the same strings with no left recursion. Use this technique to find an LL(1) grammar corresponding to the grammar G.

5.40. Another situation that obviously prevents a CFG from being LL(1) is several productions involving the same variable whose right sides begin with the same symbol. The problem can often be eliminated by factoring: For example, the productions $T \to a\alpha \mid a\beta$ can be replaced by $T \to aU$ and $U \to \alpha \mid \beta$. Use this technique (possibly more than

once) to obtain an LL(1) grammar from the CFG G having productions

$$S \to T\$ \qquad T \to [\,T\,] \mid [\,]T \mid [\,T\,]T \mid \Lambda$$

5.41. In each case, the grammar with the given productions does not satisfy the LL(1) property. Find an equivalent LL(1) grammar by factoring and eliminating left recursion (see Exercises 5.39 and 5.40).

a. $S \to S_1\$ \qquad S_1 \to aaS_1b \mid ab \mid bb$

b. $S \to S_1\$ \qquad S_1 \to S_1A \mid \Lambda \qquad A \to Aa \mid b$

c. $S \to S_1\$ \qquad S_1 \to S_1T \mid ab \qquad T \to aTbb \mid ab$

d. $S \to S_1\$ \qquad S_1 \to aAb \mid aAA \mid aB \mid bbA$
$A \to aAb \mid ab \qquad B \to bBa \mid ba$

5.42. Show that for the CFG in part (c) of the previous exercise, if the last production were $T \to a$ instead of $T \to ab$, the grammar obtained by factoring and eliminating left recursion would not be LL(1). (Find a string that doesn't work, and identify the point at which looking ahead one symbol in the input isn't enough to decide what move the PDA should make.)

5.43. Consider the CFG with productions

$$S \to S_1\$ \qquad S_1 \to S_1 + T \mid T \qquad T \to T * F \mid F$$
$$F \to [\,S_1\,] \mid a$$

a. Write the CFG obtained from this one by eliminating left recursion.

b. Give a transition table for a DPDA that acts as a top-down parser for this language.

5.44. Let G be the CFG with productions

$$S \to S_1\$ \qquad S_1 \to [\,S_1 + S_1\,] \mid [\,S_1 * S_1\,] \mid a$$

such that $L(G)$ is the language of all fully parenthesized algebraic expressions involving the operators $+$ and $*$ and the identifier a. Describe a deterministic bottom-up parser obtained from this grammar as in Example 5.34.

5.45. Let G have productions

$$S \to S_1\$ \qquad S_1 \to S_1\,[\,S_1\,] \mid S_1\,[\,] \mid [\,S_1\,] \mid [\,]$$

and let G_1 have productions

$$S \to S_1\$ \qquad S_1 \to [\,S_1\,]\,S_1 \mid [\,S_1\,] \mid [\,]\,S_1 \mid [\,]$$

a. Describe a deterministic bottom-up parser obtained from G as in Example 5.34.

b. Show that G_1 is not a weak precedence grammar.

5.46. a. Say exactly what the precedence relation is for the grammar in Example 5.34. In other words, for which pairs (X, σ), where X is a stack symbol and σ an input symbol, is it correct to reduce when X is on top of the stack and σ is the next input?

b. Answer the same question for the larger grammar (also a weak precedence grammar) with productions

$$S \to S_1\$ \qquad S_1 \to S_1 + T \mid S_1 - T \mid T$$
$$T \to T * F \mid T/F \mid F \qquad F \to [\, S_1 \,] \mid a$$

CHAPTER

6

Context-Free and Non-Context-Free Languages

Not all useful languages can be generated by context-free grammars, and a pushdown automaton is limited significantly by having to follow the rules of a stack in accessing its memory. The pumping lemma for context-free languages, like the pumping lemma for regular languages in Chapter 2, describes a property that every context-free language must have, and as a result it allows us to show that certain languages are not context-free. A slightly stronger result can be used in some cases where the pumping lemma doesn't apply. These results also suggest algorithms to answer certain decision problems involving context-free languages. Some of the languages that are shown not to be context-free illustrate differences between regular languages and context-free languages having to do with closure properties of certain set operations. In the second section of the chapter, some of these differences are explored and some partial results obtained.

6.1 THE PUMPING LEMMA FOR CONTEXT-FREE LANGUAGES

Once we define a finite automaton, it is not hard to think of languages that can't be accepted by one, even if proving it is a little harder. To accept $AnBn = \{a^n b^n \mid n \geq 0\}$, for example, the finiteness of the set of states makes it impossible after a while to remember how many a's have been read and therefore impossible to compare that number to the number of b's that follow.

We might argue in a similar way that neither

$$AnBnCn = \{a^n b^n c^n \mid n \geq 0\}$$

nor

$$XX = \{xx \mid x \in \{a, b\}^*\}$$

can be accepted by a PDA. The way a PDA processes an input string $a^i b^j c^k$ allows it to confirm that $i = j$ but not to remember that number long enough to compare it to k. In the second case, even if we use nondeterminism to begin processing the second half of the string, the symbol that we must compare to the next input is at the bottom of the stack and therefore inaccessible.

One way to *prove* that *AnBn* is not regular is to use the pumping lemma for regular languages. In this section we will establish a result for context-free languages that is a little more complicated but very similar. The easiest way to derive it is to look at context-free grammars rather than PDAs.

The principle behind the earlier pumping lemma is that if an input string is long enough, a finite automaton processing it will have to enter some state a second time. A corresponding principle for CFGs is that a sufficiently long derivation in a grammar G will have to contain a *self-embedded* variable A; that is, the derivation (or at least one with the same derivation tree) looks like

$$S \Rightarrow^* vAz \Rightarrow^* vwAyz \Rightarrow^* vwxyz$$

so that the string derived from the first occurrence of A is wAy and the string derived from the second occurrence is x. (All five of the strings v, w, x, y, and z are strings of terminals.) As a result,

$$S \Rightarrow^* vAz \Rightarrow^* vwAyz \Rightarrow^* vw^2Ay^2z \Rightarrow^* vw^3Ay^3z \Rightarrow^* \ldots$$

The string x can be derived from any one of these occurrences of A, so that each of the strings vxz, $vwxyz$, vw^2xy^2z, ... can be derived from S.

This observation will be useful if we can guarantee that the strings w and y are not both null, and even more useful if we can impose some other restrictions on the five strings of terminals. The easiest way to take care of both these objectives is to modify the grammar so that it has no Λ-productions or unit productions, and for added convenience we will use an equivalent grammar in Chomsky normal form.

Theorem 6.1 The Pumping Lemma for Context-Free Languages
Suppose L is a context-free language. Then there is an integer n so that for every $u \in L$ with $|u| \geq n$, u can be written as $u = vwxyz$, for some strings v, w, x, y, and z satisfying

1. $|wy| > 0$
2. $|wxy| \leq n$
3. for every $m \geq 0$, $vw^m xy^m z \in L$

Proof
According to Theorem 4.31, we can find a context-free grammar G so that $L(G) = L - \{\Lambda\}$ and G is in Chomsky normal form, so that the right side of every production is either a single terminal or a string of two variables. Every derivation tree in this grammar is then a binary tree. By the height of a derivation tree we will mean the number of edges (one less than the number of nodes) in the longest path from the root to a leaf node.

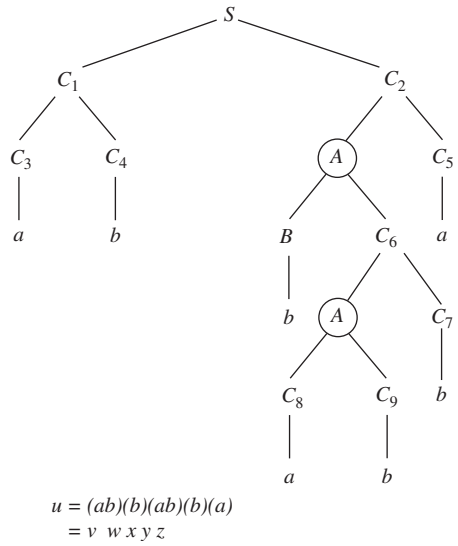


Figure 6.2 |
How the strings in the pumping lemma might be obtained.

A binary tree of height h has no more than 2^h leaf nodes. (This fact can be proved using mathematical induction on h. See Exercise 6.1.) Therefore, if $u \in L(G)$ and h is the height of a derivation tree for x, then $|u| \leq 2^h$.

Let n be 2^{p+1}, where p is the number of distinct variables in the grammar G, and suppose that u is a string in $L(G)$ of length at least n. Then $|u| > 2^p$, and it follows that every derivation tree for u must have height greater than p. In other words, in a derivation tree for u, there must be a path from the root to a leaf node with at least $p + 1$ interior nodes (nodes other than the leaf node).

Consider the portion of the longest path consisting of the leaf node and the $p + 1$ nodes just above it. Since every interior node corresponds to a variable, and there are only p distinct variables in the grammar, this portion of the path contains at least two occurrences of the same variable A. Let x be the substring of u derived from the occurrence of A farthest down on the path (closest to the leaf), and let w and y be the strings of terminals such that the substring of u derived from the occurrence of A farther from the leaf is wxy. Finally, let v and z be the prefix and the suffix of u that account for the remainder of the string, so that $u = vwxyz$. (Figure 6.1 illustrates what all of this might look like.)

If N is the node corresponding to the occurrence of A farther from the leaf node, the subtree with root node N has height $p + 1$ or less. It follows that $|wxy| \leq 2^{p+1} = n$. The leaf nodes corresponding to the

symbols of x are descendants of only one of the two children of N, and because G is in Chomsky normal form, the other child also has descendant nodes corresponding to terminal symbols. Therefore, w and y can't both be Λ. Finally, we have

$$S \Rightarrow^* vAz \Rightarrow^* vwAyz \Rightarrow^* vwxyz$$

where the two occurrences of A are the one farther from the leaf node and the one closer to it, respectively. We have already seen how the third conclusion of the theorem follows from this fact.

The comments we made in preparing to use the pumping lemma for regular languages still apply here. If we are using Theorem 6.1 to prove by contradiction that a language L is not a CFL, we start off assuming that it is, and we let n be the integer whose existence is then guaranteed by the pumping lemma. The pumping lemma makes an assertion about every string in L with length at least n, but the only strings that will do us any good in the proof are those that will produce a contradiction, and we try to select one of these to start with.

We can apply the pumping lemma to strings in L of length at least n, and all we know about n is that it is "the integer in the pumping lemma." For this reason, the string u we select must be defined in terms of n. Once we have chosen a string u that looks promising, the pumping lemma says that *some* ways of breaking up u into the five pieces v, w, x, y, z (not *all* ways) satisfy the three conditions. It's not sufficient to look at one choice of $v, \ldots, z$ and show that if those five strings satisfy the three conditions, they lead to a contradiction, because they may not satisfy them. The only way to be sure of a contradiction is to show that for the string u we have chosen, *every* choice of $v, \ldots, z$ satisfying the three conditions leads to a contradiction.

EXAMPLE 6.3 Applying the Pumping Lemma to *AnBnCn*

Suppose for the sake of contradiction that *AnBnCn* is a context-free language, and let n be the integer in the pumping lemma. Let u be the string $a^n b^n c^n$. Then $u \in AnBnCn$, and $|u| \geq n$; therefore, according to the pumping lemma, $u = vwxyz$ for some strings satisfying the conditions 1–3.

Condition 1 implies that the string wxy contains at least one symbol, and condition 2 implies that it can contain no more than two distinct symbols. If σ_1 is one of the three symbols that occurs in wy, and σ_2 is one of the three that doesn't, then the string vw^0xy^0z obtained from u by deleting w and y contains fewer than n occurrences of σ_1 and exactly n occurrences of σ_2. This is a contradiction, because condition 3 implies that vw^0xy^0z must have equal numbers of all three symbols. (We could also have obtained a contradiction by considering $m > 1$ in condition 3.)

As you can see, we have shown that the string $vxz = vw^0xy^0z$ is not only not an element of *AnBnCn*, but also not an element of the larger language $\{t \in \{a, b, c\}^* \mid n_a(t) =$

$n_b(t) = n_c(t)\}$, and so our argument also works as a proof that this larger language is not a CFL.

When we use the regular-language version of the pumping lemma, if n is the integer, the portion of the string that is pumped occurs within the prefix of length n. In Theorem 6.1, on the other hand, all we know about the location of w and y in the string is that they occur within *some* substring of length n. As Example 6.4 illustrates, it is often necessary to consider several possible cases and to show that in each case we can obtain a contradiction.

EXAMPLE 6.4 Applying the Pumping Lemma to *XX*

Suppose XX is a CFL, and let n be the integer in the pumping lemma. This time we will choose $u = a^n b^n a^n b^n$, which is an element of XX whose length is at least n. Suppose $u = vwxyz$ and that these five strings satisfy conditions 1–3. As in Example 6.3, the substring wxy contains at least one symbol and can overlap at most two of the four contiguous groups of symbols.

We consider first the case in which wy contains at least one a from the first group of a's. This means that w does, or y does, or both do. Then neither w nor y can contain any symbols from the second half of u. The string vw^0xy^0z then looks like

$$vw^0xy^0z = a^i b^j a^n b^n$$

for some i and j satisfying $i < n$ and $j \leq n$. (j is less than n if wy also contained at least one b.) If this string has odd length, it is certainly not an element of XX; if it has even length, its midpoint is between two occurrences of a in the substring a^n, so that its first half ends with a. In either case, we have a contradiction, because vw^0xy^0z cannot be of the form xx.

In the second case, suppose that neither w nor y contains an a from the first group of a's but that wy contains at least one b from the first group of b's. Then it may also contain a's from the second group, but it cannot contain b's from the second group. This time,

$$vw^0xy^0z = a^n b^i a^j b^n$$

for some i with $i < n$ and some j with $j \leq n$. If this string has even length, its midpoint occurs somewhere within the substring $b^i a^j$, and it is impossible for the first half of the string to end with b^n. Again this contradicts condition 3.

It is sufficient to consider two more cases, one in which wy contains no symbols from the first half of u but at least one a, and one in which wy contains only b's from the second group. In both these cases, we can obtain a contradiction by using $m = 0$ in statement 3, and the reasoning is almost the same as in the first two cases.

Just as in Example 6.3, this proof also shows that some other languages, such as

$$\{a^i b^i a^i b^i \mid i \geq 0\}$$

and

$$\{a^i b^j a^i b^j \mid i, j \geq 0\}$$

are not context-free.

Sometimes, as in the next example, different cases in the proof require different choices of m in condition 3 to obtain a contradiction.

EXAMPLE 6.5 Applying the Pumping Lemma to $\{x \in \{a, b, c\}^* \mid n_a(x) < n_b(x) \text{ and } n_a(x) < n_c(x)\}$

Let $L = \{x \in \{a, b, c\}^* \mid n_a(x) < n_b(x) \text{ and } n_a(x) < n_c(x)\}$, suppose that L is a CFL, and let n be the integer in the pumping lemma. Let $u = a^n b^{n+1} c^{n+1}$, and let v, w, x, y, and z be strings for which $u = vwxyz$ and conditions 1–3 are satisfied.

If wy contains an a, then because of condition 2 it cannot contain a c. It follows that vw^2xy^2z, obtained by adding one copy of w and y to u, has at least $n + 1$ a's and exactly $n + 1$ c's. On the other hand, if wy does not contain an a, then it contains either a b or a c; in this case, vw^0xy^0z contains exactly n a's, and either no more than n b's or no more than n c's. In both cases, we obtain a contradiction.

EXAMPLE 6.6 The Set of Legal C Programs Is Not a CFL

Although much of the syntax of programming languages like C can be described by context-free grammars, there are some rules that depend on context, such as the rule that a variable must be defined before it is used. In the simplest case, checking that this rule is obeyed involves testing whether a string has the form xyx, where x is a variable name and y is the portion of the program between the variable declaration and its use. We know from Example 6.4 that the language XX is not a CFL, and it is not surprising that a language whose strings must satisfy a similar restriction is not either.

Let L be the language of legal C programs, and suppose for the sake of contradiction that L is a CFL. Let n be the integer in the pumping lemma. We won't need to know much about the C language, other than the fact that the string

```
main(){int aaa...a;aaa...a;aaa...a;}
```

in which each of the three strings of a's has exactly $n + 1$ is an element of L. This will be our choice of u. It consists of a rudimentary header, a declaration of an integer variable, and two subsequent expressions, both consisting of just the variable name. The blank space after "`int`" is necessary as a separator. If the program were executed, the expression `aaa...a` would be "evaluated" twice, although the value would be meaningless in both cases because the variable is not initialized.

According to the pumping lemma, $u = vwxyz$ for some strings v, w, x, y, and z satisfying conditions 1–3. We will show that in every possible case, condition 3 with $m = 0$ produces a contradiction.

If wy contains any of the first six symbols of u, then vxz cannot be a program because it doesn't have a proper header. If it contains the left or right brace, then even if vxz has a legal header, it doesn't have the braces that must enclose the body of the program. If it contains any of the four characters after the left brace, then there is not enough left in vxz to qualify as a variable declaration. (Without the blank, "`int`" would be interpreted as part of an identifier.)

The only remaining cases are those in which wxy is a substring of

```
aaa...a;aaa...a;aaa...a;
```

If wy contains the final semicolon, then vxz is illegal, since there are still some of the a's in the last group remaining, and every statement (even if it is just an expression) must end with a semicolon. If wy contains one of the two preceding semicolons, and possibly portions of one or both of the identifiers on either side, then vxz has a variable declaration and another identifier; the two identifiers can't match, because one has length $n + 1$ and the other is longer. Finally, if wy contains only a portion of one of the identifiers (it can't contain all of it), then vxz contains a variable declaration and two subsequent expressions, but the three identifiers are not all the same. In all three of these cases, the declaration-before-use principle is violated.

The argument would almost work with u chosen to be the shorter program containing only two occurrences of the identifier. The case in which it fails is the one in which wy contains the first semicolon and nothing else. Deleting it would still leave a valid program consisting of a single variable declaration, and extra copies could be interpreted as harmless empty statements.

There are other examples of rules in C that cannot be tested by a PDA. For example, if a program defines two functions f and g having n and m formal parameters, respectively, and then makes calls on f and g, the numbers of parameters in the calls must agree with the numbers in the corresponding definitions. Testing this condition is enough like recognizing a string of the form $a^n b^m a^n b^m$ (see Example 6.4) to suggest that it would also be enough to keep L from being a CFL.

In the rest of this section, we present a result that is slightly stronger than the pumping lemma and is referred to as Ogden's lemma. Conditions 1 and 2 in the pumping lemma provide some information about the strings w and y that are pumped, but not much about their location in u. Ogden's lemma allows us to designate certain positions of u as "distinguished" and to guarantee that some of the distinguished positions appear in the pumped portions. It is sometimes more convenient than the pumping lemma and can occasionally be used when the pumping lemma fails.

Theorem 6.7 Ogden's Lemma

Suppose L is a context-free language. Then there is an integer n so that for every $u \in L$ with $|u| \geq n$, and every choice of n or more "distinguished" positions in the string u, there are strings v, w, x, y, and z so that $u = vwxyz$ and the following conditions are satisfied.

1. The string wy contains at least one distinguished position.
2. The string wxy contains n or fewer distinguished positions.
3. For every $m \geq 0$, $vw^m xy^m z \in L$.

Proof

As in the proof of the pumping lemma, we let G be a context-free grammar in Chomsky normal form generating $L - \{\Lambda\}$, and $n = 2^{p+1}$, where p

is the number of variables in G. Suppose $u \in L$, that n or more distinct positions in u are designated "distinguished," and that we have a derivation tree for u in G. We describe a path from the root of this tree to a leaf node that will give us the results we want.

The root node N_0 is the first node in the path. In general, if $i \geq 0$, the interior node N_i is the node most recently added to the path, and N_i has two children, then N_{i+1} is chosen to be the child with more distinguished descendants (leaf-node descendants of N_{i+1} that correspond to distinguished positions in u), or chosen arbitrarily if the two children have the same number.

We will call an interior node in the path a *branch point* if it has two children and both have at least one distinguished descendant. If we let $d(N)$ stand for the number of distinguished descendants of N, then the way we have constructed the path implies that

$$d(N_{i+1}) = d(N_i) \text{ if } N_i \text{ is not a branch point}$$
$$d(N_i)/2 \leq d(N_{i+1}) < d(N_i) \text{ if } N_i \text{ is a branch point}$$

From these two statements, it follows that if N_i and N_j are consecutive branch points on the path, with $j > i$, then

$$d(N_i)/2 \leq d(N_j) < d(N_i)$$

The induction argument required for the proof of Theorem 6.1 shows that if a path from the root has h interior nodes, then the total number of leaf-node descendants of the root node is no more than 2^h. A very similar argument shows that if our path has h branch points, then the total number of distinguished descendants of the root node is no more than 2^h.

As a result, our former argument can be used again, with distinguished descendants instead of descendants. Since there are more than 2^p distinguished leaf nodes, there must be more than p branch points in our path. If as before we choose the $p + 1$ branch points on the path that are closest to the leaf, then two of these must be labeled with the same variable A, and we obtain the five strings v, w, x, y, and z exactly as before. Condition 1 is true because the node labeled A that is higher up on the path is a branch point.

As you can see, the pumping lemma is simply the special case of Ogden's lemma in which all the positions of u are distinguished.

EXAMPLE 6.8 Ogden's Lemma Applied to $\{a^i b^j c^j \mid j \neq i\}$

Let L be the language $\{a^i b^j c^j \mid j \neq i\}$. Suppose L is a CFL, and let n be the integer in Ogden's lemma. For reasons that are not obvious now but will be shortly, we choose

$$u = a^n b^n c^{n+n!}$$

and we designate the first n positions of u (the positions of the a's) as distinguished. Suppose $u = vwxyz$ for some strings $v, \ldots, z$ satisfying conditions 1–3.

If either the string w or the string y contains two distinct symbols, then considering $m = 2$ in condition 3 produces a contradiction, because the string vw^2xy^2z no longer matches the regular expression $a^*b^*c^*$. Suppose neither w nor y contains two different symbols. Because of condition 1, one of these two strings is a^p for some $p > 0$.

We can easily take care of the following cases:

1. $w = a^p$ and $y = a^q$, where at least one of p and q is positive.
2. $w = a^p$ and $y = b^q$, where $p > 0$ and $p \neq q$.
3. $w = a^p$ and $y = c^q$, where $p > 0$.

In each of these cases, the string vw^mxy^mz will have different numbers of a's and b's as long as $m \neq 1$.

The only remaining case is the one in which $w = a^p$ and $y = b^p$ for some $p > 0$. In this case we will not be able to get a contradiction by finding an m so that vw^mxy^mz has different numbers of a's and b's. But we can choose *any* value of m, and the one we choose is $k + 1$, where k is the integer $n!/p$. The number of a's in vw^mxy^mz, which has $m - 1$ more copies of w than the original string u, is

$$n + (m - 1) * p = n + k * p = n + n!$$

and we have our contradiction, because the string has the same number of a's as c's.

Using Ogden's Lemma when the Pumping Lemma Fails — EXAMPLE 6.9

You might suspect that the ordinary pumping lemma is not enough to take care of Example 6.8. Without knowing that some of the a's are included in the pumped portion, there is no way to rule out the case in which both w and y contain only c's, and it doesn't seem likely that we can select m so that the number of c's in vw^mxy^mz is changed just enough to make it equal to the number of a's.

In this example we consider the language

$$L = \{a^pb^qc^rd^s \mid p = 0 \text{ or } q = r = s\}$$

and we suppose that L is a CFL. This time we can prove that the pumping lemma doesn't allow us to obtain a contradiction, because the conclusions of the pumping lemma are all true for this language. (This is the analogue for CFLs of Example 2.39.) To see this, let n be any positive integer and u any element of L with $|u| \geq n$, say $u = a^pb^qc^rd^s$. If $p = 0$, then we can take v to be Λ, w to be the first symbol of u, x and y to be Λ, and z to be the remaining suffix of u; it is clear that conditions 1–3 hold. If $p > 0$, on the other hand, then we can take w to be a, z to be $a^{p-1}b^qc^rd^s$, and the other three strings to be Λ; no matter what m is in condition 3, the string vw^mxy^mz still has equal numbers of b's, c's, and d's and is therefore an element of L.

Instead, we let n be the integer in Ogden's lemma, we choose $u = ab^nc^nd^n$, and we specify that all the positions of u except the first be distinguished. Suppose v, w, x, y, and z are strings satisfying $u = vwxyz$ and conditions 1–3. Then the string wy must contain at least one b, one c, or one d, and it can't contain all three symbols. Therefore, vw^2xy^2z

still contains at least one a, but it no longer has equal numbers of b's, c's, and d's. The conclusion is that this string cannot be in L, which is a contradiction.

6.2 INTERSECTIONS AND COMPLEMENTS OF CFLs

The set of context-free languages, like the set of regular languages, is closed under the operations of union, concatenation, and Kleene *. However, unlike the set of regular languages, the set of CFLs is not closed under intersection or difference, and we can demonstrate this by looking at the first two examples in Section 6.1.

EXAMPLE 6.10 Two CFLs Whose Intersection is Not a CFL

Example 6.3 uses the pumping lemma for context-free languages to show that *AnBnCn* is not a CFL. It is easy to write this language as the intersection of two others and not hard to see that the two others are both context-free.

AnBnCn can be written

$$\begin{aligned} AnBnCn &= \{a^i b^j c^k \mid i = j \text{ and } j = k\} \\ &= \{a^i b^i c^k \mid i, k \geq 0\} \cap \{a^i b^j c^j \mid i, j \geq 0\} \end{aligned}$$

The two languages involved in the intersection are similar to each other. The first can be written as the concatenation

$$\{a^i b^i \mid i \geq 0\}\{c^k \mid k \geq 0\} = L_1 L_2$$

and the second as the concatenation

$$\{a^i \mid i \geq 0\}\{b^j c^j \mid j \geq 0\} = L_3 L_4$$

and we know how to find a context-free grammar for the concatenation of two languages, given a CFG for each one.

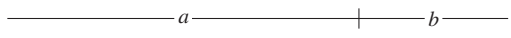
It would also not be difficult to construct pushdown automata directly for the two languages $L_1 L_2$ and $L_3 L_4$. In the first case, for example, a PDA could save a's on the stack, match them with b's, and remain in the accepting state while reading any number of c's.

EXAMPLE 6.11 A CFL Whose Complement Is Not a CFL

Surprisingly, although we showed in Example 6.4 that the language $XX = \{xx \mid x \in \{a, b\}^*\}$ is not a context-free language, its complement is.

Let L be the complement $\{a, b\}^* - XX$. L contains every string in $\{a, b\}^*$ of odd length. If $x \in L$ and $|x| = 2n$ for some $n \geq 1$, then for some k with $1 \leq k \leq n$, the kth and $(n + k)$th symbols of x are different (say a and b, respectively). There are $k - 1$ symbols before the a, $n - 1$ symbols between the two, and $n - k$ symbols after the b. But instead of thinking of the $n - 1$ symbols between the two as $n - k$ and then $k - 1$ (the remaining symbols in the first half, followed by the symbols in the second half that precede the b),

we can think of them as $k - 1$ and then $n - k$.



In other words, x is the concatenation of two odd-length strings, the first with a in the middle and $k - 1$ symbols on either side, and the second with b in the middle and $n - k$ symbols on either side. Conversely, every concatenation of two such odd-length strings is in L.

The conclusion is that L can be generated by the CFG with productions

$$S \to A \mid B \mid AB \mid BA \qquad A \to EAE \mid a \qquad B \to EBE \mid b \qquad E \to a \mid b$$

The variables A and B generate odd-length strings with middle symbol a and b, respectively, and together generate all odd-length strings. Therefore, L is a context-free language whose complement is not a CFL.

Another Example with Intersections and Complements — EXAMPLE 6.12

Suppose L_1, L_2, and L_3 are defined as follows:

$$L_1 = \{a^i b^j c^k \mid i \le j\}$$
$$L_2 = \{a^i b^j c^k \mid j \le k\}$$
$$L_3 = \{a^i b^j c^k \mid k \le i\}$$

For each i it is easy to find both a CFG generating L_i and a PDA accepting it. Therefore, $L_1 \cap L_2 \cap L_3$ represents another way of expressing the language *AnBnCn* as an intersection of context-free languages.

A string can fail to be in L_1 either because it fails to be in R, the regular language $\{a\}^*\{b\}^*\{c\}^*$, or because it is $a^i b^j c^k$ for some i, j, and k with $i > j$. Using similar arguments for L_2 and L_3, we obtain

$$L_1' = R' \cup \{a^i b^j c^k \mid i > j\}$$
$$L_2' = R' \cup \{a^i b^j c^k \mid j > k\}$$
$$L_3' = R' \cup \{a^i b^j c^k \mid k > i\}$$

The language R' is regular and therefore context-free, and the second language in each of these unions is a CFL, just as the languages L_1, L_2, and L_3 are. It follows that each of the languages L_i' is a CFL and that their union is. Therefore, because of the formula

$$(L_1' \cup L_2' \cup L_3')' = L_1 \cap L_2 \cap L_3$$

we conclude that $L_1' \cup L_2' \cup L_3'$ is another context-free language whose complement is not a CFL.

We can derive a similar result without involving the language R by considering

$$A_1 = \{x \in \{a, b, c\}^* \mid n_a(x) \le n_b(x)\}$$
$$A_2 = \{x \in \{a, b, c\}^* \mid n_b(x) \le n_c(x)\}$$
$$A_3 = \{x \in \{a, b, c\}^* \mid n_c(x) \le n_a(x)\}$$

The intersection of these three languages is $\{x \in \{a, b, c\}^* \mid n_a(x) = n_b(x) = n_c(x)\}$, which we saw in Example 6.3 is not a CFL. Therefore, $A'_1 \cup A'_2 \cup A'_3$ is a CFL whose complement is not.

We showed in Section 2.2 that if M_1 and M_2 are both finite automata, we can construct another FA that processes input strings so as to simulate the simultaneous processing by M_1 and M_2; we can simply take states to be ordered pairs (p, q), where p and q are states in M_1 and M_2, respectively. The construction works for any of the three languages $L(M_1) \cup L(M_2)$, $L(M_1) \cap L(M_2)$, and $L(M_1) - L(M_2)$, as long as we choose accepting states appropriately.

It is unreasonable to expect this construction to work for PDAs, because the current state of a PDA is only one aspect of the current configuration. We could define states (p, q) so that knowing the state of the new PDA would tell us the states of the two original PDAs, but there is no general way to use one stack so as to simulate two stacks.

There's nothing to stop us from using the Cartesian-product construction with just one stack, if it turns out that one stack is all we need. In the special case when one of the PDAs doesn't use its stack, because it's actually an FA, the construction works.

Theorem 6.13

If L_1 is a context-free language and L_2 is a regular language, then $L_1 \cap L_2$ is a CFL.

Sketch of Proof Let $M_1 = (Q_1, \Sigma, \Gamma, q_1, Z_0, A_1, \delta_1)$ be a PDA accepting L_1 and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ an FA accepting L_2. Then we define the PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ as follows.

$$Q = Q_1 \times Q_2 \qquad q_0 = (q_1, q_2) \qquad A = A_1 \times A_2$$

For $p \in Q_1$, $q \in Q_2$, and $Z \in \Gamma$,

1. For every $\sigma \in \Sigma$, $\delta((p, q), \sigma, Z)$ is the set of pairs $((p', q'), \alpha)$ for which $(p', \alpha) \in \delta_1(p, \sigma, Z)$ and $\delta_2(q, \sigma) = q'$.
2. $\delta((p, q), \Lambda, Z)$ is the set of pairs $((p', q), \alpha)$ for which $(p', \alpha) \in \delta_1(p, \Lambda, Z)$.

This PDA's computation simulates that of M_1, because for each move M consults the state of M_1 (the first component of the state-pair), the input, and the stack; it also simulates the computation of M_2, which requires only the state of M_2 and the input. M is nondeterministic if M_1 is, but the nondeterminism does not affect the second part of the state-pair. Similarly, if M_1 makes a Λ-transition, then so does M, but the second component of the state-pair is unchanged. The stack is used as if it were the stack of M_1.

The conclusion of the theorem follows from the equivalence of these two statements, for every two input strings y and z, every

state-pair (p, q), every string α of stack symbols, and every integer $n \geq 0$:

1. $(q_1, yz, Z_1) \vdash_{M_1}^{n} (p, z, \alpha)$ and $\delta_2^*(q_2, y) = q$.
2. $((q_1, q_2), yz, Z_1) \vdash_{M}^{n} ((p, q), z, \alpha)$.

Both directions can be proved by a straightforward induction argument. We will show only the induction step in the proof that statement 1 implies statement 2.

Suppose $k \geq 0$ and that for every $n \leq k$, statement 1 implies statement 2. Now suppose that

$$(q_1, yz, Z_1) \vdash_{M_1}^{k+1} (p, z, \alpha) \quad \text{and} \quad \delta_2^*(q_2, y) = q$$

We want to show that

$$((q_1, q_2), yz, Z_1) \vdash_{M}^{k+1} ((p, q), z, \alpha)$$

If the last move in the sequence of $k + 1$ moves of M_1 is a Λ-transition, then

$$(q_1, yz, Z_1) \vdash_{M_1}^{k} (p', z, \beta) \vdash_{M_1} (p, z, \alpha)$$

for some $p' \in Q_1$ and some $\beta \in \Gamma^*$. In this case, the induction hypothesis implies that

$$((q_1, q_2), yz, Z_1) \vdash_{M}^{k} ((p', q), z, \beta)$$

and from part 2 of the definition of δ we obtain

$$((p', q), z, \beta) \vdash_{M} ((p, q), z, \alpha)$$

which implies the result. Otherwise, $y = y'\sigma$, where $\sigma \in \Sigma$, and

$$(q_1, y'\sigma z, Z_1) \vdash_{M_1}^{k} (p', \sigma z, \beta) \vdash_{M_1} (p, z, \alpha)$$

If we let $q' = \delta_2^*(q_2, y')$, then the induction hypothesis implies that

$$((q_1, q_2), y'\sigma z, Z_1) \vdash_{M}^{k} ((p', q'), \sigma z, \beta)$$

and part 1 of the definition of δ implies

$$((p', q'), \sigma z, \beta) \vdash_{M} ((p, q), z, \alpha)$$

which gives us the conclusion we want.

We have convinced ourselves (see Examples 6.10 and 6.11) that there is no general way, given PDAs M_1 and M_2 that actually use their stacks, to construct a PDA simulating both of them simultaneously. One might still ask, since the construction for FAs worked equally well for unions and intersections, what makes it possible for a PDA to accept the union of two arbitrary CFLs. Perhaps the simplest answer is nondeterminism. If M_1 and M_2 are PDAs accepting L_1 and L_2, respectively, we can construct a new PDA M as follows: its first move, chosen

nondeterministically, is a Λ-transition to the initial state of either M_1 or M_2; after that, M's moves are exactly those of the PDA it has chosen in the first move. In other words, M chooses nondeterministically whether to test for membership in the first language or the second. Testing for both is obviously harder, and that's what is necessary in order to accept the intersection.

Thinking about nondeterminism also helps us to understand how it might be that no PDA can accept precisely the strings in L', even if there is a PDA that accepts precisely the string in L. For example, a PDA M might be able to choose between two sequences of moves on an input string x, so that both choices read all the symbols of x but only one causes M to end up in an accepting state. In this case, the PDA obtained from M by reversing the accepting and nonaccepting states will still have this property, and x will be accepted by both the original PDA and the modified one.

Even if M is a deterministic PDA that accepts L, the presence of Λ-transitions might prevent the PDA M' obtained from M by reversing the accepting and nonaccepting states from accepting L', in at least two ways.

Table 5.14 shows the transitions of a deterministic PDA M that accepts the language $AEqB = \{x \in \{a, b\}^* \mid n_a(x) = n_b(x)\}$. The string ab is accepted as a result of these moves:

$$(q_0, ab, Z_0) \vdash (q_1, b, aZ_0) \vdash (q_1, \Lambda, Z_0) \vdash (q_2, \Lambda, Z_0)$$

The modified PDA M' making the same sequence of moves would end in a nonaccepting state; unfortunately, M' enters q_2 by a Λ-transition from q_1, so that by the time it enters the nonaccepting state q_2, it's too late—ab has already been accepted.

Another potential complication is that M could allow an infinite sequence of consecutive Λ-transitions, which for a string x not in $L(M)$ could prevent x from being read completely. In this case, the PDA M' would still not accept such a string.

It is easy to see that for a DPDA M without Λ-transitions, the DPDA M' obtained this way accepts the complement of $L(M)$. (You can check that for the PDA M shown in Table 5.15, M' accepts the language of all strings in $\{a, b\}^*$ with unequal numbers of a's and b's.) It is not as easy to see, but still true, that for every language L accepted by a DPDA, L' can also be accepted by a DPDA. This result allows us to say in particular that if L is a context-free language for which L' is not a CFL, then L cannot be accepted by a DPDA. The language *Pal*, on the other hand, is not accepted by a DPDA (Theorem 5.1), even though both the language and its complement are CFLs (Example 4.3).

6.3 DECISION PROBLEMS INVOLVING CONTEXT-FREE LANGUAGES

We have shown in Chapter 5 that starting with a context-free grammar G, there are at least two simple ways of obtaining a pushdown automaton accepting $L(G)$, and that starting with a PDA M, there is a way to obtain a context-free grammar

G generating $L(M)$. Because of these algorithms, questions about context-free languages can be formulated by specifying either a CFG or a PDA.

One of the most basic questions about a context-free language is whether a particular string is an element of the language. The *membership problem* for CFLs is the decision problem

1. Given a context-free grammar G and a string x, is $x \in L(G)$?

For regular languages, it makes sense to ask the question by starting with a finite automaton and a string, because the way the FA processes a string makes the question easy to answer. We can trace the moves the FA makes on the input string and see whether it ends up in an accepting state. If we started with an NFA, we could still answer the question this way, once we applied the algorithm that starts with an NFA and produces an equivalent FA. Trying to use this approach with the membership problem for CFLs, however, would be more complicated, because a PDA might have nondeterminism that cannot be eliminated; we might have to consider a backtracking algorithm or some other way of determining whether there is a sequence of moves that causes the string to be accepted.

As we saw in Section 4.5, there is an algorithm to solve the membership problem if we have a CFG G that generates L. If $x = \Lambda$, deciding whether $x \in L$ means deciding whether the start variable S is nullable (Definition 4.26); otherwise, we have an algorithm to find a CFG G_1 with no Λ-productions or unit productions so that $L(G_1) = L(G) - \{\Lambda\}$, and we can decide whether $x \in L(G_1)$ by trying all possible derivations in G_1 with $2|x| - 1$ or fewer steps.

This algorithm is a theoretical possibility but not likely to be useful in practice. Because context-free grammars play such a significant role in programming languages and other real-world languages, the membership problem is a practical problem that needs a practical solution. Fortunately, there are algorithms that are considerably more efficient. A well-known one is due to Cocke, Younger, and Kasami and is described in the paper by Younger (1967). Another is an algorithm by Earley (1970).

There are two decision problems involving regular languages for which we found algorithms by using the pumping lemma for regular languages, and we can use the CFL version of the pumping lemma to find algorithms to answer the corresponding questions about context-free languages.

2. Given a context-free language L, is L nonempty?

3. Given a context-free language L, is L infinite?

The only significant difference from the regular-language case is that there we found the integer n in the statement of the pumping lemma by considering an FA accepting the language (see the discussion preceding Theorem 2.29), so that it was convenient to formulate the problem in terms of an FA; here we find n by considering a grammar generating the language (as in the proof of Theorem 6.1),

so that we would prefer to specify L in terms of a CFG. Once the integer n is obtained, the algorithms are identical.

With solutions to the regular-language versions of problems 2 and 3, we were able to solve other decision problems involving regular languages. The CFL versions of two of these are

4. Given CFGs G_1 and G_2, is $L(G_1) \cap L(G_2)$ nonempty?
5. Given CFGs G_1 and G_2, is $L(G_1) \subseteq L(G_2)$?

If we were able to take the same approach that we took with FAs, we would construct a context-free grammar G with $L(G) = L(G_1) \cap L(G_2)$, or with $L(G) = L(G_1) - L(G_2)$, and use the algorithm for problem 2 to determine whether $L(G)$ is nonempty. However, we know from the preceding section that in both cases there may not be such a G. It seems natural to look for another algorithm. Later, once we have described a more general model of computation, we will see that there is none, and that these are both examples of *undecidable* problems. Studying context-free grammars has already brought us to the point where we can easily formulate decision problems for which algorithmic solutions may be impossible.

EXERCISES

6.1. Show using mathematical induction that a binary tree of height h has no more than 2^h leaf nodes.

6.2. In each case below, show using the pumping lemma that the given language is not a CFL.

a. $L = \{a^i b^j c^k \mid i < j < k\}$
b. $L = \{a^{2^n} \mid n \geq 0\}$
c. $L = \{x \in \{a, b\}^* \mid n_b(x) = n_a(x)^2$
d. $L = \{a^n b^{2n} a^n \mid n \geq 0\}$
e. $L = \{x \in \{a, b, c\}^* \mid n_a(x) = \max\{n_b(x), n_c(x)\}\}$
f. $L = \{x \in \{a, b, c\}^* \mid n_a(x) = \min\{n_b(x), n_c(x)\}\}$
g. $\{a^n b^m a^n b^{n+m} \mid m, n \geq 0\}$

6.3. In the pumping-lemma proof in Example 6.4, give some examples of choices of strings $u \in L$ with $|u| \geq n$ that would not work.

6.4. In the proof given in Example 6.4 using the pumping lemma, the contradiction was obtained in each case by considering the string vw^0xy^0z. Would it have been possible instead to use vw^2xy^2z in each case? If so, give the proof in at least one case; if not, explain why not.

6.5. For each case below, decide whether the given language is a CFL, and prove your answer.

a. $L = \{a^n b^m a^m b^n \mid m, n \geq 0\}$
b. $L = \{xayb \mid x, y \in \{a, b\}^*$ and $|x| = |y|\}$

c. $L = \{xcx \mid x \in \{a, b\}^*\}$

d. $L = \{xyx \mid x, y \in \{a, b\}^* \text{ and } |x| \geq 1\}$

e. $L = \{x \in \{a, b\}^* \mid n_a(x) < n_b(x) < 2n_a(x)\}$

f. $L = \{x \in \{a, b\}^* \mid n_a(x) = 10n_b(x)\}$

g. $L =$ the set of non-balanced strings of parentheses

6.6. If L is a CFL, does it follow that $r(L) = \{x^r \mid x \in L\}$ is a CFL? Give either a proof or a counterexample.

6.7. State and prove theorems that generalize to context-free languages statements I and II, respectively, in Exercise 2.23. Then give an example to illustrate each of the following possibilities.

a. Theorem 6.1 can be used to show that the language is a CFL, but the generalization of statement I cannot.

b. The generalization of statement I can be used to show the language is not a CFL, but the generalization of statement II cannot.

c. The generalization of statement II can be used to show the language is not a CFL.

6.8. Show that if L is a DCFL and R is regular, then $L \cap R$ is a DCFL.

6.9. In each case below, show that the given language is a CFL but that its complement is not.

a. $\{a^i b^j c^k \mid i \geq j \text{ or } i \geq k\}$

b. $\{a^i b^j c^k \mid i \neq j \text{ or } i \neq k\}$

6.10. In Example 6.4, the pumping lemma proof began with the string

$$u = \text{“}\texttt{main()int aaa...a;aaa...a;aaa...a;}\text{”}$$

Redo the proof using Ogden's lemma and taking u to be “`main()int aaa...a;aaa...a;`” (where both strings have n a's).

6.11. †Use Ogden's lemma to show that the languages below are not CFLs.

a. $\{a^i b^{i+k} a^k \mid k \neq i\}$

b. $\{a^i b^i a^j b^j \mid j \neq i\}$

c. $\{a^i b^j a^i \mid j \neq i\}$

6.12. a. Show that if L is a CFL and F is finite, $L - F$ is a CFL.

b. Show that if L is not a CFL and F is finite, then $L - F$ is not a CFL.

c. Show that if L is not a CFL and F is finite, then $L \cup F$ is not a CFL.

6.13. For each part of Exercise 6.12, say whether the statement is true if “finite” is replaced by “regular”, and give reasons.

6.14. For each part of Exercise 6.12, say whether the statement is true if “CFL” is replaced by “DCFL”, and give reasons.

6.15. Verify that if M is a DPDA with no Λ-transitions, accepting a language L, then the DPDA obtained from M by reversing the accepting and nonaccepting states accepts L'.

6.16. †For each case below, decide whether the given language is a CFL, and prove your answer.

a. $L = \{x \in \{a, b\}^* \mid n_a(x)$ is a multiple of $n_b(x)\}$

b. Given a CFG L, the set of all prefixes of elements of L

c. Given a CFG L, the set of all suffixes of elements of L

d. Given a CFG L, the set of all substrings of elements of L

e. $\{x \in \{a, b\}^* \mid |x|$ is even and the first half of x has more a's than the second$\}$

f. $\{x \in \{a, b, c\}^* \mid n_a(x), n_b(x)$, and $n_c(x)$ have a common factor greater than 1$\}$

6.17. †Prove the following variation of Theorem 6.1. If L is a CFL, then there is an integer n so that for every $u \in L$ with $|u| \geq n$, and every choice of u_1, u_2, and u_3 satisfying $u = u_1u_2u_3$ and $|u_2| \geq n$, there are strings v, w, x, y, and z satisfying the following conditions:

1. $u = vwxyz$
2. $wy \neq \Lambda$
3. Either w or y is a nonempty substring of u_2
4. For every $m \geq 0$, $vw^ixy^iz \in L$

Hint: Suppose # is a symbol not appearing in strings in L, and let L_1 be the set of all strings that can be formed by inserting two occurrences of # into an element of L. Show that L_1 is a CFL, and apply Ogden's lemma to L_1.

This result is taken from Floyd and Beigel (1994).

6.18. Show that the result in Exercise 6.17 can be used in each part of Exercise 6.11.

6.19. Show that the result in Exercise 6.17 can be used in Examples 6.8 and 6.9 to show that the language is not context-free.

6.20. †The set of DCFLs is closed under the operation of complement, as discussed in Section 6.2. Under which of the following other operations is this set of languages closed? Give reasons for your answers.

a. Union

b. Intersection

c. Concatenation

d. Kleene *

e. Difference

6.21. Use Exercises 5.20 and 6.8 to show that the following languages are not DCFLs. This technique is used in Floyd and Beigel (1994), where the language in Exercise 5.20 is referred to as Double-Duty(L).

a. *Pal*, the language of palindromes over $\{a, b\}$ (Hint: Consider the regular language corresponding to $a^*b^*a^*\#b^*a^*$.)

b. $\{x \in \{a, b\}^* \mid n_b(x) = n_a(x)$ or $n_b(x) = 2n_a(x)\}$

c. $\{x \in \{a, b\}^* \mid n_b(x) < n_a(x)$ or $n_b(x) > 2n_a(x)\}$

6.22. Find an example of a CFL L so that $\{x\#y \mid x \in L \text{ and } xy \in L\}$ is not a CFL.

6.23. †Show that if $L \subseteq \{a\}^*$ is a CFL, then L is regular.

6.24. †Consider the language $L = \{x \in \{a, b\}^* \mid n_a(x) = f(n_b(x))\}$. In Exercise 2.31 you showed that L is regular if and only if f is ultimately periodic; in other words, L is regular if and only if there is a positive integer p so that for each r with $0 \leq r < p$, f is eventually constant on the set $S_{p,r} = \{jp + r \mid j \geq 0\}$. Show that L is a CFL if and only if there is a positive integer p so that for each r with $0 \leq r < p$, f is eventually linear on the set $S_{p,r}$. "Eventually linear" on $S_{p,r}$ means that there are integers N, c, and d so that for every $j \geq N$, $f(jp + r) = cj + d$. (Suggestion: for the "if" direction, show how to construct a PDA accepting L; for the converse, use the pumping lemma.)

CHAPTER

7

Turing Machines

A Turing machine is not just the next step beyond a pushdown automaton. It is, according to the Church-Turing thesis, a general model of computation, potentially able to execute any algorithm. This chapter presents the basic definitions as well as examples that illustrate the various modes of a Turing machine: accepting a language, computing a function, and carrying out other types of computations as well. Variants such as multitape Turing machines are discussed briefly. Nondeterministic Turing machines are introduced, as well as universal Turing machines, which anticipate the modern stored-program computer.

We adopt the Turing machine as our way of formulating an algorithm, and a few of the discussions in the chapter begin to suggest ideas about the limits of computation that will play an important part in Chapters 8 and 9.

7.1 | A GENERAL MODEL OF COMPUTATION

Both of the simple devices we studied earlier, finite automata and pushdown automata, are models of computation. A machine of either type can receive an input string and execute an algorithm to obtain an answer, following a set of rules specific to the machine type. In both cases, the machine can execute only very specialized algorithms: in the case of an FA, algorithms in which there is an absolute limit on the amount of information that can be remembered, and in the case of a PDA, algorithms in which it is sufficient to access information following the rules imposed by a stack.

It is easy to find examples of languages that cannot be accepted because of these limitations. A finite automaton cannot accept

$$SimplePal = \{xcx^r \mid x \in \{a, b\}^*$$

A device with only a stack can accept *SimplePal* but not

$$AnBnCn = \{a^n b^n c^n \mid n \in \mathcal{N}\} \qquad \text{or} \qquad L = \{xcx \mid x \in \{a, b\}^*\}$$

It is not hard to see that a PDA-like machine with *two* stacks can accept *AnBnCn*, and a finite automaton with a *queue* instead of a stack can accept L. Although in both cases it might seem that the machine is being devised specifically to handle one language, it turns out that both devices have substantially more computing power than either an FA or a PDA, and both are reasonable candidates for a model of general-purpose computation.

The abstract model we will study instead was designed in the 1930s by the English mathematician Alan Turing and is referred to today as a Turing machine. It was not obtained by adding data structures onto a finite automaton, and it predates the finite-automaton and pushdown-automaton models. Turing's objective was to demonstrate the inherent limitations of algorithmic methods. He began with the idea of formulating a model capable in principle of carrying out any algorithm whatsoever; this allowed him to assert that any deficiencies of the model were in fact deficiencies of the algorithmic method. The Turing machine was intended as a theoretical model, not as a practical way to carry out a computation. However, it anticipated in principle many of the features of modern electronic computers.

Turing considered a human computer working with a pencil and paper. He decided that without any loss of generality, the human computer could be assumed to operate under simple rules: First, the only things written on the paper are symbols from some fixed finite alphabet; second, each step taken by the computer depends only on the symbol he is currently examining and on his "state of mind" at the time; third, although his state of mind may change as a result of his examining different symbols, only a finite number of distinct states of mind are possible.

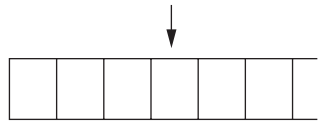
With these principles in mind, Turing isolated what he took to be the primitive steps that a human computer carries out during a computation:

1. Examining an individual symbol on the paper;

2. Erasing a symbol or replacing it by another;

3. Transferring attention from one symbol to another nearby symbol.

Some of these elements are reminiscent of our simpler abstract models. A Turing machine has a finite alphabet of symbols (actually two alphabets, an input alphabet and a possibly larger one for use during the computation), and a finite set of states, corresponding to the human computer's states of mind. A sheet of paper is two-dimensional but for simplicity Turing specified a linear *tape*, which has a left end and is potentially infinite to the right. The tape is marked off into squares, each of which can hold one symbol; if a square has no symbol on it, we say that it contains the *blank* symbol. For convenience, we sometimes think of the squares of the tape as being numbered left-to-right, starting with 0, although this numbering is not part of the official model and the numbers are not necessary in order to describe the operation of the machine. We visualize the reading and writing as being done by a *tape head*, which at any time is centered on one square of the tape.



In our version of a Turing machine, which is similar though not identical to the one proposed by Turing, a single move is determined by the current state and the current tape symbol, and consists of three parts:

1. Changing from the current state to another, possibly different state;
2. Replacing the symbol in the current square by another, possibly different symbol;
3. Leaving the tape head on the current square, or moving it one square to the right, or moving it one square to the left if it is not already on the leftmost square.

The tape serves as the input device (the input is the finite string of nonblank symbols on the tape originally), the memory available for use during the computation, and the output device. The output, if it is relevant, is the string of symbols left on the tape at the end of the computation.

One crucial difference between a Turing machine and the two simpler machines we have studied is that a Turing machine is not restricted to one left-to-right pass through the input, performing its computation as it reads. The input string is present initially, before the computation starts. Because the tape head will normally start out at the beginning of the tape, we think of the machine as "reading" its input from left to right, but it might move its tape head over the squares containing the input symbols simply in order to go to another portion of the tape. A human computer working in this format could examine part of the input, modify or erase part of it, take time out to execute some computations in a different area of the tape, return to re-examine the input, repeat any of these actions, and perhaps stop the computation before he has looked at all, or even any, of the input; a Turing machine can do these things as well.

Although it will be convenient to describe a computation of a Turing machine in a way that is general enough to include a variety of "subprograms" within some larger algorithm, the two primary objectives of a Turing machine that we will focus on are *accepting a language* and *computing a function*. The first, which we will discuss in more detail in the next section, is like the computation of a finite automaton or pushdown automaton, except that the languages can now be more general; the second makes sense because of the enhanced output capability of the Turing machine, and we will discuss it in Section 7.3.

A finite automaton requires more than one accepting state if for two strings x and y that are accepted, the information that must be remembered is different; x and y are both accepted, but the device must allow for the fact that there is more input to come, and what it does with the subsequent symbols depends on whether it started with x or with y. A Turing machine does not need to say whether each prefix

of the input string is accepted or not. It has the entire string to start with, and once the machine decides to accept or reject the input, the computation stops. For this reason, a Turing machine never needs more than two *halt* states, one that denotes acceptance and one that denotes rejection. If the computation is one of some other type, in which acceptance is not the issue, a normal end to the computation usually means halting in the accepting state, and the rejecting state usually signifies some sort of "crash," or abnormal termination.

With Turing machines there is also a new issue, which did not arise with finite automata and did not arise in any serious way with PDAs. If a Turing machine decides to accept or reject the input string, it stops. But it might not decide to do this, and so it might continue moving forever. This will turn out to be important!

Definition 7.1 Turing Machines

A Turing machine (TM) is a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$, where

Q is a finite set of states. The two *halt* states h_a and h_r are not elements of Q.

Σ, the input alphabet, and Γ, the tape alphabet, are both finite sets, with $\Sigma \subseteq \Gamma$. The *blank* symbol Δ is not an element of Γ.

q_0, the initial state, is an element of Q.

δ is the transition function:

$$\delta : Q \times (\Gamma \cup \{\Delta\}) \rightarrow (Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\}) \times \{\text{R, L, S}\}$$

For a state $p \in Q$, a state $q \in Q \cup \{h_a, h_r\}$, two symbols $X, Y \in \Gamma \cup \{\Delta\}$, and a "direction" $D \in \{\text{R, L, S}\}$, we interpret the formula

$$\delta(p, X) = (q, Y, D)$$

to mean that when T is in state p and the symbol on the current tape square is X, the TM replaces X by Y on that square, changes to state q, and either moves the tape head one square right, moves it one square left if the tape head is not already on square 0, or leaves it stationary, depending on whether D is R, L, or S, respectively. If the state q is either h_a or h_r, we say that this move causes T to *halt*. Once it halts, it cannot move, since δ is defined at (q, Y) only if $q \in Q$.

In this chapter we will return to drawing transition diagrams, similar to but more complicated than the diagrams for finite automata. The transition

$$\delta(p, X) = (q, Y, D)$$

will be represented by the diagram

If the TM attempts to execute the move $\delta(p, X) = (q, Y, \text{L})$ when the tape head is on square 0, we will say that it halts in the state h_r, leaving the tape head

in square 0 and leaving the symbol X unchanged. This is a way for the TM to halt that is not reflected by the transition function.

Normally a TM begins with an input string starting in square 1 of its tape and all other squares blank. We don't insist on this, because in Section 7.4 we will talk about a TM picking up where another one left off, so that the original tape may already have been modified. We do always assume, however, that the set of nonblank squares on the tape when a TM starts is finite, so that there must be only a finite number of nonblank squares at any stage of its operation. We can describe the current *configuration* of a TM by a single string

$$xqy$$

where q is the current state, x is the string of symbols to the left of the current square, y either is null or starts in the current square, and everything after xy on the tape is blank. The string x may be null, and if not, the symbols in x may include blanks as well as nonblanks. If the string y ends in a nonblank symbol, the strings xqy, $xqy\Delta$, $xqy\Delta\Delta$, ... all refer to the same configuration; normally in this case we will write the string with no blanks at the end. If all the symbols on and to the right of the current square are blanks, we will normally write the configuration as

$$xq\Delta$$

so as to identify the current symbol explicitly.

It is also useful sometimes to describe the tape (including the tape head position as well as the tape contents) without mentioning the current state. We will do this using the notation

$$x\underline{\sigma}y$$

where σ is the symbol in the current square, or

$$x\underline{y}$$

where the string y, which may contain more than one symbol, begins in the current square. In either case, x is the string to the left of the current square and all symbols to the right of the string y are blank. When $y = \Lambda$, instead of writing $x\underline{y}$ we will usually write $x\underline{\Delta}$.

We trace a sequence of TM moves by specifying the configuration at each step. For example, if q is a nonhalting state and r is any state, we write

$$xqy \vdash_T zrw \qquad \text{or} \qquad xqy \vdash_T^* zrw$$

to mean that the TM T moves from the first configuration to the second in one move or in zero or more moves, respectively. For example, if q is a nonhalting state and the current configuration of T is given by

$$aabqa\Delta a$$

and $\delta(q, a) = (r, \Delta, \text{L})$, we could write

$$aabqa\Delta a \vdash_T aarb\Delta\Delta a$$

The notations $\vdash_T$ and $\vdash_T^*$ are usually shortened to $\vdash$ and $\vdash^*$ if there is no ambiguity.

Finally, if T has input alphabet Σ and $x \in \Sigma^*$, the *initial configuration corresponding to input x* is given by

$$q_0 \Delta x$$

In this configuration T is in the initial state, the tape head is in square 0, that square is blank, and x begins in square 1. A TM starting in this configuration most often begins by moving its tape head one square right to begin reading the input.

It is widely believed that Turing was successful in formulating a completely general model of computation—roughly speaking, that any algorithmic procedure can be programmed on a TM. A statement referred to as Turing's thesis, or the Church-Turing thesis, makes this claim explicitly. Because at this stage we have not even said how a TM accepts a string or computes a function, we will discuss the Church-Turing thesis a little later, in Section 7.6. Before then, we will also consider a number of examples of TMs, which may make the thesis seem reasonable when it is officially presented.

7.2 TURING MACHINES AS LANGUAGE ACCEPTORS

Definition 7.2 Acceptance by a TM

If $T = (Q, \Sigma, \Gamma, q_0, \delta)$ is a TM and $x \in \Sigma^*$, x is accepted by T if

$$q_0 \Delta x \vdash_T^* w h_a y$$

for some strings $w, y \in (\Gamma \cup \{\Delta\})^*$ (i.e., if, starting in the initial configuration corresponding to input x, T eventually halts in the accepting state).

A language $L \subseteq \Sigma^*$ is accepted by T if $L = L(T)$, where

$$L(T) = \{x \in \Sigma^* \mid x \text{ is accepted by } T\}$$

Saying that L is accepted by T means that for any $x \in \Sigma^*$, $x \in L$ if and only if x is accepted by T. This does not imply that if $x \notin L$, then x is rejected by T. The two possibilities for a string x not in $L(T)$ are that T rejects x and that T never halts, or loops forever, on input x.

A TM Accepting a Regular Language — EXAMPLE 7.3

Figure 7.3a shows the transition diagram of a finite automaton accepting $L = \{a, b\}^*\{ab\}\{a, b\}^* \cup \{a, b\}^*\{ba\}$, the language of all strings in $\{a, b\}^*$ that either contain the substring ab or end with ba.

Using a TM instead of an FA to accept this language might seem like overkill, but the transition diagram in Figure 7.4b illustrates the fact that every regular language can

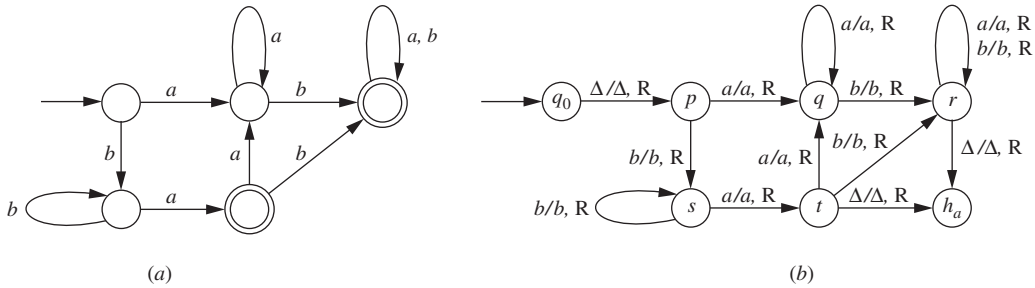


Figure 7.4 |
A finite automaton, and a Turing machine accepting the same language.

be accepted by a Turing machine, and that every FA can easily be converted into a TM accepting the same language.

Several comments about the TM are in order. First, it can process every input string in the way a finite automaton is forced to, moving the tape head to the right on each move and never modifying the symbols on the tape. This procedure will not work for languages that are not regular.

Second, the diagram in Figure 7.4b does not show moves to the reject state. There is one from each of the states p, q, and s (the nonhalting states that correspond to nonaccepting states in the FA), and they all look like this:

In each of these states, if the TM discovers by encountering a blank that it has reached the end of the input string, it can reject the input. From now on, we will usually omit these transitions in a diagram, so that very likely you will not see h_r at all; it should be understood that if for some combination of state and tape symbol, no move is shown explicitly, then the move is to h_r. What happens to the tape head and to the current symbol on the move is usually unimportant, because the computation is over, but we will say that both stay unchanged.

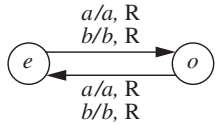
Finally, the general algorithm illustrated by Figure 7.4b for converting an FA to a TM requires the TM to read all the symbols of the input string, since there are no moves to either halt state until a blank is encountered. In this example, a string could be accepted as soon as an occurrence of ab is found, without reading the rest of the input. If we preferred this approach, we could eliminate the state r altogether and send the b-transitions from both q and t directly to h_a.

EXAMPLE 7.5 A TM Accepting $XX = \{xx \mid x \in \{a, b\}^*\}$

This example will demonstrate a little more of the computing power of Turing machines. Let

$$L = \{xx \mid x \in \{a, b\}^*\}$$

and let us start by thinking about what an algorithm for testing input strings might look like. For a string to be in L means first that its length is even. If we were thinking about finite automata, we could easily test this condition by using two states e (even) and o (odd), and transitions back and forth:



This allows us to reject odd-length strings quickly but is ultimately not very helpful, because if the length turns out to be even, we still need to find the middle of the string.

A human computer might do this by calculating the length of the string, dividing that number by 2, and counting again to find the spot that far in from the beginning. A Turing machine could do this too, but it would be laborious. Furthermore, in this problem arithmetic seems extraneous. A more alert human computer might position his fingers at both ends of the string and proceed to move both fingers toward the middle simultaneously, one symbol at a time. This is the approach we will take, and this type of computation is common for a TM. Of course, a device with only one tape head also has to do this rather laboriously, but this just reflects the fact that when Turing conceived of these machines, he was not primarily concerned with their efficiency.

As the TM works its way in from the ends, continuing to move its tape head back and forth from one side to the other, it will replace input symbols by the corresponding uppercase letters to keep track of its progress. Once it arrives at the middle of the string—and if it realizes at this point that the length is odd, it can reject—it changes the symbols in the first half back to their original lowercase form.

The second half of the processing starts at the beginning again and, for each lowercase symbol in the first half of the string, compares the lowercase symbol to the corresponding uppercase symbol in the second half. Here again we must keep track of our progress, and we do this by changing lowercase symbols to uppercase and simultaneously erasing the matching uppercase symbols.

The resulting TM is shown in Figure 7.6. For an even-length input string, the first phase of the computation is represented by the loop at the top of the diagram that includes q_1, q_2, q_3, and q_4. At the end of this loop, the string has been converted to uppercase and the current square contains the symbol that begins the second half of the string. (An odd-length string is discovered in state q_3 and rejected at that point.) The transition from q_1 to q_5 starts moving the tape head to the left, and by the time the state q_6 is reached, the first half of the string is again lowercase and the tape head is back at the beginning of the string. In the comparison of the first and second halves, an a in the first half that is not matched by the corresponding symbol in the second half is discovered in state q_8, and an unmatched b is discovered in q_7. If the comparison is successful, a blank is encountered in q_6 and the string is accepted.

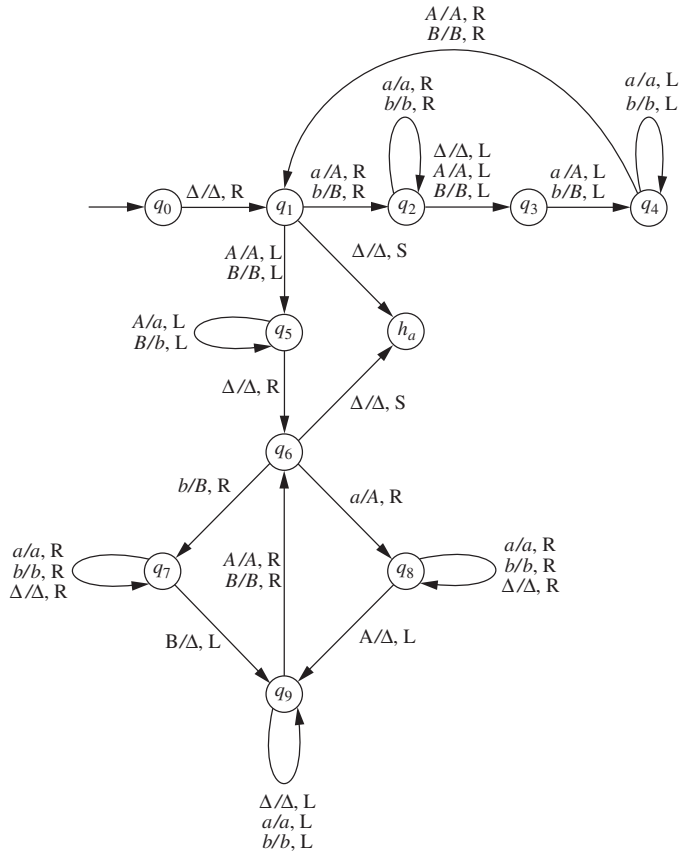


Figure 7.5
A Turing machine accepting $\{xx \mid x \in \{a, b\}^*\}$.

We trace the moves of the TM for three input strings: two that illustrate the two possible ways a string can be rejected, and one that is in L.

$q_0\Delta aba$	$\vdash \Delta q_1 aba$	$\vdash \Delta A q_2 ba$	$\vdash^* \Delta A b a q_2 \Delta$	$\vdash \Delta A b q_3 a$
	$\vdash \Delta A q_4 bA$	$\vdash \Delta q_4 AbA$	$\vdash \Delta A q_1 bA$	$\vdash \Delta AB q_2 A$
	$\vdash \Delta A q_3 BA$	$\vdash \Delta A h_r BA$	(reject)	
$q_0\Delta ab$	$\vdash \Delta q_1 ab$	$\vdash \Delta A q_2 b$	$\vdash \Delta A b q_2 \Delta$	$\vdash \Delta A q_3 b \Delta$
	$\vdash \Delta q_4 AB$	$\vdash \Delta A q_1 B$	$\vdash \Delta q_5 AB$	$\vdash q_5 \Delta aB$
	$\vdash \Delta q_6 aB$	$\vdash \Delta A q_8 B$	$\vdash \Delta A h_r B$	(reject)
$q_0\Delta aa$	$\vdash \Delta q_1 aa$	$\vdash \Delta A q_2 a$	$\vdash \Delta A a q_2 \Delta$	$\vdash \Delta A q_3 a \Delta$
	$\vdash \Delta q_4 AA$	$\vdash \Delta A q_1 A$	$\vdash \Delta q_5 AA$	$\vdash q_5 \Delta aA$
	$\vdash \Delta q_6 aA$	$\vdash \Delta A q_8 A$	$\vdash \Delta q_9 A$	$\vdash \Delta A q_6 \Delta$
	$\vdash \Delta A h_a \Delta$	(accept)		

EXAMPLE 7.7 Accepting $L = \{a^i b a^j \mid 0 \leq i < j\}$

If we were testing a string by hand for membership in L, there is at least one simple algorithm we could use that would be very easy to carry out on a Turing machine. The TM would be similar to the first phase of the one in Example 7.5, but even simpler: it could work its way in from the ends of the string to the b in the middle, in each iteration erasing an a in both portions. (In the previous example, erasing wasn't an option during the locating-the-middle phase, because we needed to remember what the symbols were.) When either or both portions had no a's left, it would be easy to determine whether the string should be accepted.

The TM we present will not be this one and will not be quite as simple; we'll say why in a minute. Ours will begin by making a preliminary pass through the entire string to make sure that it has the form a^*baa^*. If it doesn't, it will be rejected; if it does, this TM also carries out a sequence of iterations in which an a preceding the b and another one after the b are erased, but the a it erases in each portion will be the *leftmost* a.

The TM T that follows this approach is shown in Figure 7.8. The top part, involving states q_0 through q_4, is to handle the preliminary test described in the previous paragraph.

The subsequent passes begin in state q_5. Because of the preliminary check, if the first symbol we see on some pass is b, and if we then find an a somewhere to the right, perhaps after moving through blanks where a's have already been erased, we can simply accept. Otherwise, if the symbol we see first is a, we erase it, move the tape head to the right through the remaining a's until we hit the b, continue moving to the right through blanks until we hit another a, erase it, move the tape head back past any blanks and past the b, then past any remaining a's until we hit a blank; at that point we reverse course and prepare for another iteration. Each subsequent pass also starts with either the b (because we have erased all the a's that preceded it) or the leftmost unerased a preceding the b.

Here is a trace of the computation for the input string $abaa$:

$$
\begin{aligned}
q_0\Delta abaa &\vdash \Delta q_1 abaa \vdash \Delta a q_1 baa \vdash \Delta ab q_2 aa\Delta \vdash \Delta aba q_3 a \\
&\vdash \Delta aba a q_3 \Delta \vdash \Delta aba q_4 a \vdash^* q_4 \Delta aba a \vdash \Delta q_5 aba a \\
&\vdash \Delta\Delta q_6 baa \vdash \Delta\Delta b q_7 aa \vdash \Delta\Delta q_8 b\Delta a \vdash \Delta\Delta q_9 b\Delta a \\
&\vdash \Delta\Delta q_5 b\Delta a \vdash \Delta\Delta b q_{10}\Delta a \vdash \Delta\Delta b\Delta q_{10} a \vdash \Delta\Delta b h_a a \quad \text{(accept)}
\end{aligned}
$$

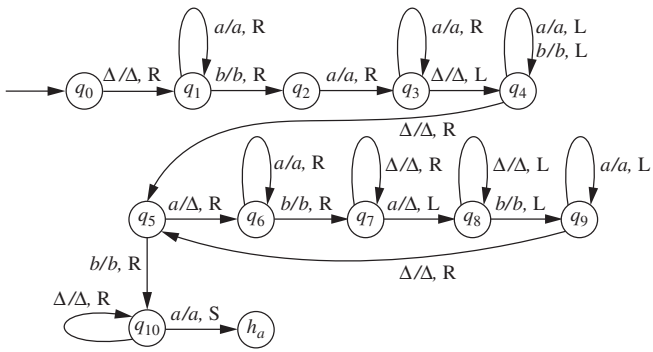


Figure 7.8 |
A Turing machine accepting $\{a^i b a^j \mid 0 \leq i < j\}$.

It is not hard to see that every string in L will be accepted by T. Every string that doesn't match the regular expression a^*baa^* will be rejected. The remaining strings are those of the form a^iba^j, where $0 < j \leq i$. Let's trace T on the simplest such string, aba.

$$
\begin{array}{llll}
q_0\Delta aba & \vdash \Delta q_1 aba & \vdash \Delta a q_1 ba & \vdash \Delta ab q_2 a & \vdash \Delta aba q_3 \Delta \\
 & \vdash \Delta ab q_4 a & \vdash^* q_4 \Delta aba & \vdash \Delta q_5 aba & \vdash \Delta\Delta q_6 ba \\
 & \vdash \Delta\Delta b q_7 a & \vdash \Delta\Delta q_8 b & \vdash \Delta q_9 \Delta b & \vdash \Delta\Delta q_5 b \\
 & \vdash \Delta\Delta b q_{10} \Delta & \vdash \Delta\Delta b \Delta q_{10} \Delta & \vdash \Delta\Delta b \Delta\Delta q_{10} \Delta & \vdash \ldots
\end{array}
$$

The TM is in an infinite loop—looking for an a that would allow it to accept, with no way to determine that it will never find one.

Obviously, given a choice, you'd pick the strategy we described at the beginning of the example, not this one. But the point is that this TM works! Accepting a language requires only that we accept every string in the language and not accept any string that isn't in the language.

This is perhaps at least a plausible example of a language L and an attempt to accept it that results in a TM with infinite loops for some input strings not in the language. In this case, the TM can be replaced by another one that accepts the same language without any danger of infinite loops. A question that is harder to answer is whether this is always possible. We will return to this issue briefly in the next section and discuss it in considerably more detail in Chapter 8.

7.3 TURING MACHINES THAT COMPUTE PARTIAL FUNCTIONS

A computer program whose purpose is to produce an output string for every legal input string can be interpreted as computing a function from one set of strings to another. A Turing machine T with input alphabet Σ that does the same thing computes a function whose domain D is a subset of Σ^*. Things will be a little simpler if we say instead that T computes a partial function on Σ^* with domain D. For every input string x in the domain of the function, T carries out a computation that ends with the output string $f(x)$ on the tape. The definition below considers a partial function on $(\Sigma^*)^k$, a function of k string variables; when $k > 1$, the k strings all appear initially on the tape, separated by blanks. The value of the function is always assumed to be a single string, and most of the time we will consider only $k = 1$.

The most significant issue for a TM T computing a function f is what output strings it produces for input strings in the domain of f. However, what T does for other input strings is not completely insignificant, because we want to say that T computes the partial function f, not some other partial function with a larger domain. For this reason, we say that T should end up in the accepting state for inputs in the domain of f and not for any other inputs. In other words, if T computes a partial function with domain D, then in particular, T accepts D, irrespective of the output it produces.

Definition 7.9 A Turing Machine Computing a Function

Let $T = (Q, \Sigma, \Gamma, q_0, \delta)$ be a Turing machine, k a natural number, and f a partial function on $(\Sigma^*)^k$ with values in Γ^*. We say that T computes f if for every $(x_1, x_2, \ldots, x_k)$ in the domain of f,

$$q_0 \Delta x_1 \Delta x_2 \Delta \ldots \Delta x_k \vdash_T^* h_a \Delta f(x_1, x_2, \ldots, x_k)$$

and no other input that is a k-tuple of strings is accepted by T.

A partial function $f : (\Sigma^*)^k \to \Gamma^*$ is Turing-computable, or simply computable, if there is a TM that computes f.

Ideally, as we have said, a TM should compute at most one function. This is still not quite the case, partly because of technicalities involving functions that have different codomains and are otherwise identical, but also because if T computes a partial function f on $(\Sigma^*)^2$, then T can also be said to compute the partial function f_1 on Σ^* defined by the formula $f_1(x) = f(x, \Lambda)$. We can at least say that once we specify a natural number k and a set $C \subseteq \Gamma^*$ for the codomain, a TM can't compute more than one partial function of k variables having codomain C.

We are free to talk about a TM computing a partial function whose domain and codomain are sets of numbers, once we adopt a way to represent numbers by strings. For our purposes it will be sufficient to consider partial functions on $\mathcal{N}^k$ with values in $\mathcal{N}$, and we will use *unary* notation to represent numbers: the natural number n is represented by the string $1^n = 11\ldots1$. The official definition is essentially the same as Definition 7.9, except that the input alphabet is {1}, the initial configuration looks like

$$q_0 \Delta 1^{n_1} \Delta 1^{n_2} \Delta \ldots \Delta 1^{n_k}$$

and the final configuration that results from an input in the domain of the partial function f is

$$h_a \Delta 1^{f(n_1, n_2, \ldots, n_k)}$$

EXAMPLE 7.10 The Reverse of a String

Figure 7.11 shows a transition diagram for a TM computing the reverse function $r : \{a, b\}^* \to \{a, b\}^*$. The TM moves from the ends of the string to the middle, making a sequence of swaps between symbols σ_1 in the first half and σ_2 in the second, and using uppercase letters to record progress. Each iteration starts in state q_1 with the TM looking at the symbol σ_1 in the first half. The tape head moves to the position of σ_2 in the second half, remembering σ_1 by using states q_2 and q_3 if it is a and q_4 and q_5 if it is b. When the tape head arrives at σ_2 (the rightmost lowercase symbol), the TM deposits the (uppercase) symbol in that position, but similarly follows different paths back to q_1, depending on whether σ_2 is a or b. The last iteration is cut short in the case of an odd-length string: When the destination position on the right is reached, there is already an uppercase symbol there. The last phase of the computation is to move to the right end of the string and make a pass toward the beginning of the tape, converting all the uppercase letters to their original form.

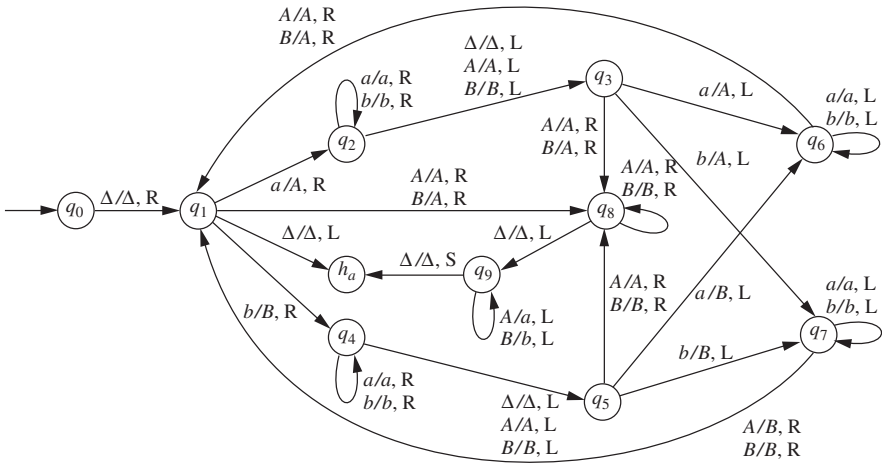


Figure 7.11 |
A Turing machine computing the reverse function.

We show the moves of the TM for the even-length string *baba*.

$$
\begin{array}{llllll}
q_0\Delta baba & \vdash \Delta q_1 baba & \vdash \Delta B q_4 aba & \vdash^* \Delta Babaq_4\Delta & \vdash \Delta Babq_5 a \\
 & \vdash \Delta Baq_6 bB & \vdash^* \Delta q_6 BabB & \vdash \Delta Aq_1 abB & \vdash \Delta AAq_2 bB \\
 & \vdash \Delta AAbq_2 B & \vdash \Delta AAq_3 bB & \vdash \Delta Aq_7 AAB & \vdash \Delta ABq_1 AB \\
 & \vdash \Delta ABAq_8 B & \vdash \Delta ABABq_8\Delta & \vdash \Delta ABAq_9 B & \vdash^* q_9\Delta abab \\
 & \vdash h_a\Delta abab & & &
\end{array}
$$

EXAMPLE 7.12 The Quotient and Remainder Mod 2

The two transition diagrams in Figure 7.13 show TMs that compute the quotient and remainder, respectively, when a natural number is divided by 2. The one in Figure 7.13a is another application of finding the middle of the string; This time, 1's in the first half are temporarily replaced by x's and 1's in the second half are erased. As usual, exactly what happens in the last iteration depends on whether the input n is even or odd. Here are traces illustrating both cases.

$$
\begin{array}{llllll}
q_0\Delta 1111 & \vdash \Delta q_1 1111 & \vdash \Delta x q_2 111 & \vdash \Delta x 1 q_3 11 & \vdash^* \Delta x 111 q_3\Delta \\
 & \vdash \Delta x 11 q_4 1 & \vdash \Delta x 1 q_5 1 & \vdash \Delta x q_5 11 & \vdash \Delta q_5 x 11 \\
 & \vdash \Delta x q_1 11 & \vdash \Delta x x q_2 1 & \vdash \Delta x x 1 q_3\Delta & \vdash \Delta x x q_4 1 \\
 & \vdash \Delta x q_5 x & \vdash \Delta x x q_1\Delta & \vdash \Delta x q_7 x & \vdash^* q_7\Delta 11 \\
 & \vdash h_a\Delta 11 & & &
\end{array}
$$

$$
\begin{array}{llllll}
q_0\Delta 111 & \vdash \Delta q_1 111 & \vdash \Delta x q_2 11 & \vdash \Delta x 1 q_3 1 & \vdash \Delta x 11 q_3\Delta \\
 & \vdash \Delta x 1 q_4 1 & \vdash \Delta x q_5 1 & \vdash \Delta q_5 x 1 & \vdash \Delta x q_1 1 \\
 & \vdash \Delta x x q_2\Delta & \vdash \Delta x q_6 x & \vdash \Delta q_7 x & \vdash q_7\Delta 1 \\
 & \vdash h_a\Delta 1 & & &
\end{array}
$$

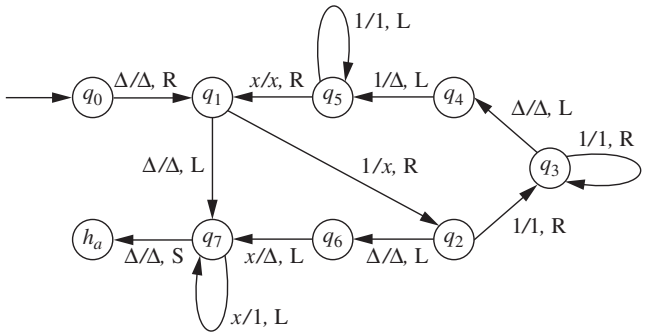


Figure 7.13 |
Computing the quotient and remainder mod 2.

The TM in Figure 7.13b moves the tape head to the end of the string, then makes a pass from right to left in which the 1's are counted and simultaneously erased. The final output is a single 1 if the input was odd and nothing otherwise.

The Characteristic Function of a Set

EXAMPLE 7.14

For a language $L \subseteq \Sigma^*$, the *characteristic function* of L is the function $\chi_L : \Sigma^* \to \{0, 1\}$ defined by

$$\chi_L(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L \end{cases}$$

Here we think of 0 and 1 as alphabet symbols, rather than numbers, so that when we use a TM to compute χ_L, unary notation is not involved.

Computing the function χ_L and accepting the language L are two approaches to the question of whether an arbitrary string is in L, and Turing machines that carry out these computations are similar in some respects. A TM computing χ_L indicates whether the input string is in L by producing output 1 or output 0, and one accepting L indicates the same thing by accepting or not accepting the input. If accepting a language L requires a complicated algorithm, then computing χ_L will require an algorithm that is at least as complicated. We

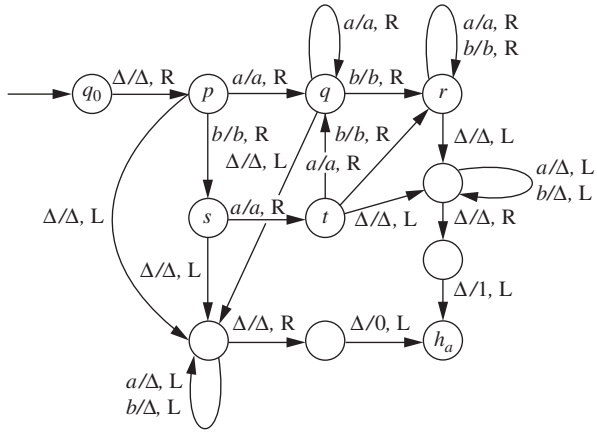


Figure 7.15 |
Computing χ_L, where $L = \{a, b\}^*(\{ab\}\{a, b\}^* \cup \{ba\})$.

can be a little more specific at this stage about the relationship between these two approaches, although in Chapter 8 we will consider the question more thoroughly.

It is easy to see that if we have a TM T computing χ_L, then we can construct another TM T_1 accepting L by modifying T in one simple way: each move of T to the accepting state is replaced by a sequence of moves that checks what output is on the tape, accepts if it is 1, and rejects if it is 0.

If we want a TM T_1 computing χ_L, however, it will have to accept every string, because $\chi_L(x)$ is defined for every $x \in \Sigma^*$. This means that if we are trying to obtain T_1 from a TM T that accepts L, it is important at the outset to know whether T halts for every input string. If we know that it does, then modifying it so that the halting configurations are the correct ones for computing χ_L can actually be done without too much trouble, as in the example pictured in Figure 7.15. Otherwise, there is at least some serious doubt as to whether what we want is even possible.

Figure 7.15 shows a transition diagram for a TM computing χ_L, where L is the regular language $L = \{a, b\}^*\{ab\}\{a, b\}^* \cup \{a, b\}^*\{ba\}$ discussed in Example 7.3. A transition diagram for a TM accepting L is shown in Figure 7.4b.

7.4 | COMBINING TURING MACHINES

A Turing machine represents an algorithm. Just as a typical large algorithm can be described as a number of subalgorithms working in combination, we can combine several Turing machines into a larger composite TM.

In the simplest case, if T_1 and T_2 are TMs, we can consider the composition

$$T_1T_2$$

whose computation can be described approximately by saying "first execute T_1, then execute T_2." In order to make sense of this, we think of the three TMs T_1,

T_2, and T_1T_2 as sharing a common tape and tape head, and the tape contents and tape head position when T_2 starts are assumed to be the same as when T_1 stops.

If $T = (Q, \Sigma, \Gamma, q_0, \delta)$ is the composite TM T_1T_2, we can obtain the state set Q (of nonhalting states) of T by taking the union of the state sets of T_1 and T_2, except that the states of T_2 are relabeled if necessary so that the two sets don't overlap. The initial state of T is the initial state of T_1. The transitions of T include all those of T_2, and all those of T_1 that don't go to h_a. A transition

in T_1 becomes

in T, where q_2 is the initial state of T_2. In other words, T begins in the initial state of T_1 and executes the moves of T_1 up to the point where T_1 would halt. If T_1 halts in the reject state, then so does T, but if T_1 halts in h_a, then at that point T_2 takes over, in its initial state. The moves that cause T to accept are precisely those that cause T_2 to accept.

We can usually avoid being too explicit about how the input alphabets and tape alphabets of T, T_1, and T_2 are related. It may be, for example, that T_1's job is to create an input configuration different from the one T_2 would normally have started with; or it may be that T_2 does not expect to be processing input at all, but only making some modification to the tape configuration in preparation for some other TM. It should be the case, at least, that any of T_1's tape symbols that T_2 is likely to encounter should be in T_2's input alphabet.

It may now be a little clearer why in certain types of computations we are careful to specify the final tape configuration. If T_1 is to be followed by another TM T_2, T_1 must finish up so that the tape looks like what T_2 expects when it starts. For example, if T_1 and T_2 compute the functions f and g, respectively, from $\mathcal{N}$ to $\mathcal{N}$, the output configuration left by T_1 is a legitimate input configuration for T_2, so that the output from T_1T_2 resulting from input n is $g(f(n))$. In other words, T_1T_2 computes the composite function $g \circ f$, defined by the formula $g \circ f(n) = g(f(n))$.

When we use a TM T as a component of a larger machine, then in order to guarantee that it accomplishes what we want it to, we must be sure that the tape has been prepared properly before T starts. For example, if T is designed to process an input string starting in the normal input configuration, then at the beginning of its operation the tape should look like $x\underline{\Delta}y$ for some string x and some input string y.

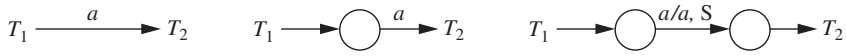
As long as T can process input y correctly starting with the configuration $q_0\Delta y$, it will work correctly starting with $xq_0\Delta y$, because it will never attempt to move its tape head to the left of the blank and will therefore never see any symbols in x. If the tape is not blank to the right of y, we can't be sure T will complete its processing properly.

In order to use the composite TM T_1T_2 in a larger context, and to draw transition diagrams without having to show the states of T_1 and T_2 explicitly, we can write

$$T_1 \rightarrow T_2$$

It is often helpful to use a mixed notation in which some but not all of the states of a TM are shown. For example, we might use any of these notations

to mean "in state p, if the current symbol is a, then execute the TM T." The third notation indicates most explicitly that this can be viewed as a composite TM formed by combining T with another very simple one. Similarly, the three notations



stand for "execute T_1, and if T_1 halts in h_a with current symbol a, then execute T_2." In all three, it is understood that if T_1 halts in h_a but the current symbol is one other than a for which no transition is indicated, then the TM rejects.

The TM pictured in Figure 7.16 should be interpreted as follows. If the current tape symbol is a, execute the TM T; if it is b, halt in the accepting state; and if it is anything else, reject. In the first case, if T halts in h_a, then as long as the current symbol is a, continue to execute T; if at some point T halts in h_a with current symbol b, halt and accept. The TM might reject during one of the iterations of T,

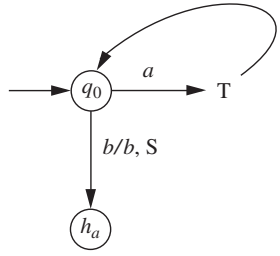


Figure 7.16

and it might loop forever, either because one of the iterations of T does or because T accepts with current symbol a every time.

There are too many potentially useful TM components to try to list them. It might not be worth trying to isolate the find-the-middle-of-the-string operation that we have used several times, because we almost always use some variation of it that not only finds the middle of the string but simultaneously does something else useful to one or both halves. Here are a few components that will come up repeatedly.

Finding the Next Blank or the Previous Blank — EXAMPLE 7.17

We will use *NB* to denote the Turing machine that moves the tape head to the *next blank* (the first blank square to the right of the current head position), and *PB* for the TM that moves to the *previous* blank (the rightmost blank square to the left of the current position). Neither of these is useful by itself—in particular, *PB* rejects if it begins with no blanks to the left of the current square—but both can often be used in composite TMs.

Copying a String — EXAMPLE 7.18

A *Copy* TM starts with tape $\underline{\Delta}x$, where x is a string of nonblank symbols, and ends up with $\underline{\Delta}x\Delta x$. Figure 7.19 shows a transition diagram for such a TM if the alphabet is $\{a, b\}$, and all that is needed in a more general situation is an "uppercase" version of each alphabet symbol.

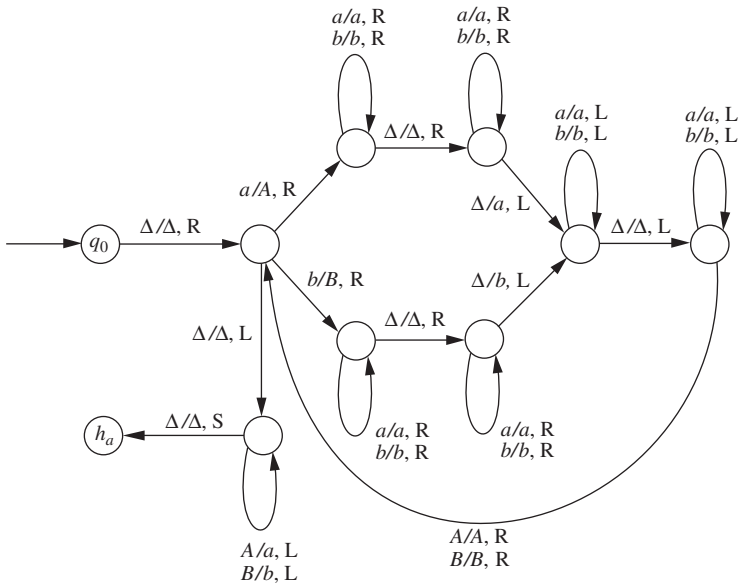


Figure 7.19 |
A Turing machine to copy strings.

EXAMPLE 7.20 Inserting and Deleting a Symbol

Deleting the current symbol means transforming the tape from $x\underline{\sigma}y$ to $x\underline{y}$, where σ is any symbol, including Δ, and y is a string of nonblank symbols. Inserting the symbol σ means starting with $x\underline{y}$ and ending up with $x\underline{\sigma}y$, where again y contains no blanks. A *Delete* TM for the alphabet $\{a, b\}$ is shown in Figure 7.22. The states labeled q_a and q_b allow the TM to remember a symbol between the time it is erased and the time it is copied in the next square to the left.

Inserting the symbol σ is done virtually the same way, except that the single pass goes from left to right, symbols are moved to the right instead of to the left, and the move that starts things off writes σ instead of Δ.

EXAMPLE 7.21 Erasing the Tape

There *is* no general "erase-the-tape" Turing machine! In particular, there is no TM E that is capable of starting in an arbitrary configuration, erasing all the nonblank symbols on or to the right of the current square, and halting in the square it started in. Doing this would require that it be able to find the rightmost nonblank symbol, which is not possible without special assumptions.

However, if T starts in the standard input configuration, we can effectively have it erase the portion of its tape to the right of its tape head after its operation. The trick is to have T perform its computation with a few modifications that will make the erasing feasible. T starts by placing an end-of-tape marker \$ in the square following the input string; during its computation, every time it moves its tape head as far right as the \$, it moves the marker one square further, to record the fact that one more square has been used in the computation. This complicates the transition diagram for T, but in a straightforward way. Except for the states involved in the preliminary moves, every state p is replaced by the transition shown in Figure 7.23.

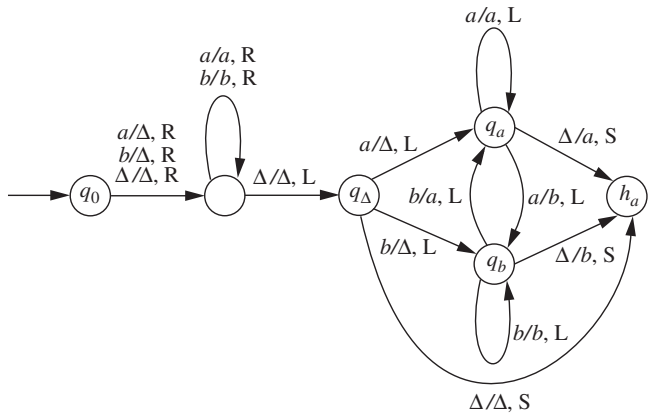


Figure 7.22 |
A Turing machine to delete a symbol.

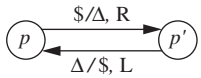


Figure 7.23 |

Any additional transitions to or from p are unchanged. Once T has finished its computation, it erases the tape as follows. It marks the square beyond which the tape is to be blank; moves to the right until it hits the \$, and erases it; then moves left back to the marked square, erasing each symbol as it goes.

The conclusion is that although there is no such thing as a general erase-the-tape TM, we can pretend that there is. We can include an erasing TM E in a larger composite machine, and we will interpret it to mean that the components that have executed previously have prepared the tape so that the end-of-tape marker is in place, and E will be able to use it and erase it.

Comparing Two Strings **EXAMPLE 7.24**

It is useful to be able to start with the tape $\underline{\Delta}x\Delta y$ and determine whether $x = y$. A comparison operation like this can be adapted and repeated in order to find out whether x occurs as a substring of y. In the next example we use a TM *Equal*, which begins with $\underline{\Delta}x\Delta y$ (where $x, y \in \{a, b\}^*$), accepts if they are equal, and rejects otherwise.

The last example in this section illustrates how a few of these operations can be used together.

Accepting the Language of Palindromes **EXAMPLE 7.25**

In the TM shown below, *NB* and *PB* are the ones in Example 7.17, *Copy* is the TM in Example 7.18, *R* is the TM in Example 7.10 computing the reverse function, and *Equal* is the TM mentioned in the previous example.

$$Copy \rightarrow NB \rightarrow R \rightarrow PB \rightarrow Equal$$

The Turing machine accepts the language of palindromes over $\{a, b\}$ by comparing the input string to its reverse and accepting if and only if the two are equal.

7.5 | MULTITAPE TURING MACHINES

Algorithms in which several different kinds of data are involved can be particularly unwieldy to implement on a Turing machine, because of the bookkeeping necessary to store and access the data. One feature that can simplify things is to have several different tapes, with independent tape heads, and to be able to move all the tape heads simultaneously in a single TM move.

This means that we are considering a different model of computation, a multitape Turing machine. Just as we showed in Chapter 3 that allowing nondeterminism

and Λ-transitions did not increase the computing power of finite automata, we wish to show now that allowing a Turing machine to have more than one tape does not make it any more powerful than an ordinary TM.

If we are comparing an ordinary TM and a multitape TM with regard to *power* (as opposed to convenience or efficiency), the question is whether the two machines can solve the same problems and get the same answers. A TM of any type produces an answer, first by accepting or rejecting a string, and second by producing an output string. What we must show, then, is that for every multitape TM T, there is a single-tape TM that accepts exactly the same strings as T, rejects the same strings as T, and produces exactly the same output as T for every input string it accepts. To simplify the discussion we restrict ourselves to Turing machines with two tapes, and it is easy to see that the principles are the same if there are more than two.

A 2-tape TM can also be described by a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$, where this time

$$\delta : Q \times (\Gamma \cup \{\Delta\})^2 \to (Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\})^2 \times \{\text{R,L,S}\}^2$$

A single move can change the state, the symbols in the current squares on both tapes, and the positions of the two tape heads. Let us describe a configuration of a 2-tape TM by a 3-tuple

$$(q, x_1\underline{a_1}y_1, x_2\underline{a_2}y_2)$$

where q is the current state and $x_i\underline{a_i}y_i$ represents the contents of tape i for each value of i.

We will define the initial configuration corresponding to input string x as

$$(q_0, \underline{\Delta}x, \underline{\Delta})$$

(that is, the input string is in the usual place on the first tape and the second tape is initially blank). Similarly, the output will appear on the first tape, and the contents of the second tape at the end of the computation will be irrelevant.

Theorem 7.26

For every 2-tape TM $T = (Q, \Sigma, \Gamma, q_0, \delta)$, there is an ordinary 1-tape TM $T_1 = (Q_1, \Sigma, \Gamma_1, q_1, \delta_1)$, with $\Gamma \subseteq \Gamma_1$, such that

1. For every $x \in \Sigma^*$, T accepts x if and only if T_1 accepts x, and T rejects x if and only if T_1 rejects x. (In particular, $L(T) = L(T_1)$.)
2. For every $x \in \Sigma^*$, if

$$(q_0, \underline{\Delta}x, \underline{\Delta}) \vdash_T^* (h_a, y\underline{a}z, u\underline{b}v)$$

for some strings $y, z, u, v \in (\Gamma \cup \{\Delta\})^*$ and symbols $a, b \in \Gamma \cup \{\Delta\}$, then

$$q_1\Delta x \vdash_{T_1}^* yh_aaz$$

Proof

We describe how to construct an ordinary TM T_1 that *simulates* the 2-tape TM T and satisfies the properties in the statement of the theorem. There are several possible ways an ordinary TM might simulate one with two tapes; the way we will describe is for it to divide the tape into two parts, as this diagram suggests.

\$	1	2	1	2	1	

In square 0 there is a marker to make it easy to locate the beginning of the tape. The odd-numbered squares represent the first tape and the remaining even-numbered squares represent the second. To make the operation easier to describe, we refer to the odd-numbered squares as the *first track* of the tape and the even-numbered squares beginning with square 2 as the second track.

It will also be helpful to have a marker at the other end of the nonblank portion of the tape, because if T accepts, then at the end of the simulation T_1 needs to delete all the symbols on the second track of the tape. Starting in the initial configuration $q_1 \Delta x$ with input $x = a_1 a_2 \dots a_n$, T_1 places the \$ marker in square 0, inserts blanks between consecutive symbols of x, and places the # marker after the last nonblank symbol, to produce the tape

$$\$\Delta\Delta a_1 \Delta a_2 \Delta \dots \Delta a_n \#$$

(so that the first track of the tape has contents Δx and the second is blank). From this point on, the # is moved if necessary to mark the farthest right that T has moved on either of its tapes (see Example 7.21).

The only significant complication in T_1's simulating the computation of T is that it must be able to keep track of the locations of both tape heads of T. We can handle this by including in T_1's tape alphabet an extra copy σ' of every symbol σ (including Δ) that can appear on T's tape. When T_1 has finished simulating a move, if the tape heads of T are on squares containing σ_1 and σ_2, respectively, then the two squares on T_1's tape that represent those squares will have the symbols σ_1' and σ_2' instead. At each step during the simulation, there will be one primed symbol on each of the two tracks of the tape.

By using extra states, T_1 can "remember" the current state of T, and it can also remember what the primed symbol on the first track is while it is looking for the primed symbol on the second track (or vice-versa). The steps T_1 makes in order to simulate a move of T are these:

1. Move the tape head left until the beginning-of-tape marker \$ is encountered, then right until a symbol of the form σ' is found on the first track of the tape. Remember σ and move back to the \$.
2. Move right until a symbol τ' is found on the second track. If the move of T that has now been determined is

$$\delta(p, \sigma, \tau) = (q, \sigma_1, \tau_1, D_1, D_2)$$

then reject if $q = h_r$, and otherwise change τ' to τ_1 and move in direction D_2 to the appropriate square on the second track.
3. If in moving the tape head this way, the \$ is encountered, reject, since T's move would have attempted to move its tape head off the tape. Otherwise (moving the # marker if necessary), change the symbol in the new square on track 2 to the corresponding primed symbol, and move back to the \$.
4. Locate σ' on the first track again, change it to σ_1, and move the tape head in direction D_1 to the appropriate square on the first track.
5. After allowing for encountering either marker, as in step 3, change the new symbol on the first track to the corresponding primed symbol.

The three lines below illustrate one iteration in a simple case. (Vertical lines are to make it easier to read.)

\$	Δ	0	$1'$	1	Δ	0	0	$0'$	1	#	Δ
\$	Δ	0	$1'$	1	Δ	0	0	1	1	Δ'	#
\$	Δ'	0	Δ	1	Δ	0	0	1	1	Δ'	#

Here we assume that σ and τ are 1 and 0, respectively, σ_1 and τ_1 are Δ and 1, and D_1 and D_2 are L and R. The move being simulated is

$$\delta(p, 1, 0) = (q, \Delta, 1, \mathrm{L}, \mathrm{R})$$

The second line represents the situation after step 3: the # has been moved one square to the right, the $0'$ on the second track has been replaced by 1, and the symbol after this 1 on the second track is now Δ'. The third line represents the situation after step 5.

As long as T has not halted, iterating these steps allows T_1 to simulate the moves of T correctly. If T finally accepts, then T_1 must carry out these additional steps in order to end up in the right configuration.

6. Delete every square in the second track to the left of #.
7. Delete both end-of-tape markers, so that the remaining symbols begin in square 0.
8. Move the tape head to the primed symbol, change it to the corresponding unprimed symbol, and halt in h_a with the tape head on that square.

Corollary 7.27 Every language that is accepted by a 2-tape TM can be accepted by an ordinary 1-tape TM, and every function that is computed by a 2-tape TM can be computed by an ordinary TM. ■

7.6 | THE CHURCH-TURING THESIS

To say that the Turing machine is a general model of computation means that any algorithmic procedure that can be carried out at all, by a human computer or a team of humans or an electronic computer, can be carried out by a TM. This statement was first formulated by Alonzo Church in the 1930s and is usually referred to as Church's thesis, or the Church-Turing thesis. It is not a mathematically precise statement that can be proved, because we do not have a precise definition of the term *algorithmic procedure*. By now, however, there is enough evidence for the thesis to have been generally accepted. Here is an informal summary of some of the evidence.

1. The nature of the model makes it seem likely that all the steps crucial to human computation can be carried out by a TM. Humans normally work with a two-dimensional sheet of paper, and a human computer may perhaps be able to transfer his attention to a location that is not immediately adjacent to the current one, but enhancements like these do not appear to change the types of computation that are possible. A TM tape could be organized so as to simulate two dimensions; one likely consequence would be that the TM would require more moves to do what a human could do in one.
2. Various enhancements of the TM model have been suggested in order to make the operation more like that of a human computer, or more convenient, or more efficient. The multitape TM discussed briefly in Section 7.5 is an example. In each case, it has been shown that the computing power of the device is unchanged.
3. Other theoretical models of computation have been proposed. These include abstract machines such as the ones mentioned in Section 7.1, with two stacks or with a queue, as well as machines that are more like modern computers. In addition, various notational systems (programming languages, grammars, and other formal mathematical systems) have been suggested as ways of describing or formulating computations. Again, in every case, the model has been shown to be equivalent to the Turing machine.
4. Since the introduction of the Turing machine, no one has suggested any type of computation that ought to be included in the category of "algorithmic procedure" and cannot be implemented on a TM.

So far in this chapter, we have considered several simple examples of Turing machines. None of them is all that complex, although as we suggested in Section 7.4, simple TMs can be combined to form complex ones. But you can already see in Example 7.5 that in executing an algorithm, a TM depends on low-level operations that can obviously be carried out in some form or other. What makes an algorithm complex is not that the low-level details are more complicated, but that there are more of them, and that they are combined in ways that involve sophisticated logic or complex bookkeeping strategies. Increasing the complexity of an algorithm poses challenges for someone trying to design a coherent and

understandable implementation, but not challenges that require more computing power than Turing machines provide.

We have said that the Church-Turing thesis is not mathematically precise because the term "algorithmic procedure" is not mathematically precise. Once we adopt the thesis, however, we are effectively giving a precise meaning to the term: An algorithm is a procedure that can be carried out by a TM. Such a definition provides a starting point for a discussion of which problems have algorithmic solutions and which don't. This discussion begins in Chapter 8 and continues in Chapter 9.

Another way we will use the Church-Turing thesis in the rest of this book is to describe algorithms in general terms, without giving all the details of Turing machines that can execute them. Even in the discussion so far, on several occasions we have sketched the operation of a TM without providing all the details; a full-fledged application of the Church-Turing thesis allows us to stop with a precise description of an algorithm, without referring to a TM implementation at all.

7.7 NONDETERMINISTIC TURING MACHINES

A *nondeterministic* Turing machine (NTM) T is defined the same way as an ordinary TM, except that for a state-symbol pair there might be more than one move. This means that if $T = (Q, \Sigma, \Gamma, q_0, \delta)$, and $(q, a) \in Q \times (\Gamma \cup \{\Delta\})$, then $\delta(q, a)$ is a finite subset, not an element, of $(Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\}) \times \{\text{R,L,S}\}$.

We can use the same notation for configurations of an NTM as in Section 7.1; if p is a nonhalting state, q is any state, a and b are tape symbols, and w, x, y, z are strings,

$$wpax \vdash_T yqbz$$

means that T has at least one move in the first configuration that leads to the second, and

$$wpax \vdash_T^* yqbz$$

means that there is at least one sequence of zero or more moves that takes T from the first to the second. It is still correct to say that a string x is accepted by T if

$$q_0 \Delta x \vdash_T^* wh_a y$$

for some strings $w, y \in (\Gamma \cup \{\Delta\})^*$.

The idea of an NTM that produces output will be useful, particularly one that is a component in a larger machine. We will not consider the idea of computing a function using a nondeterministic Turing machine, however, because a TM that computes a function should produce no more than one output for a given input string.

There are many types of languages for which nondeterminism makes it easy to construct a TM accepting the language. We give two examples, and there are others in the exercises.

The Set of Composite Natural Numbers **EXAMPLE 7.28**

Let L be the subset of $\{1\}^*$ of all strings whose length is a *composite* number (a nonprime bigger than 2); since a prime is a natural number 2 or higher whose only positive divisors are itself and 1, an element of L is a string of the form 1^n, where $n = p * q$ for some integers p and q with $p, q \geq 2$.

There are several possible strategies we could use to test an input $x = 1^n$ for membership in L. One approach is to try combinations $p \geq 2$ and $q \geq 2$ to see if any combination satisfies $p * q = n$ (i.e., if $1^{p*q} = 1^n$).

Nondeterminism allows a Turing machine to *guess*, rather than trying all possible combinations. This means guessing values of p and q, multiplying them, and accepting if and only if $p * q = n$. This sounds too simple to be correct, but it works. If n is nonprime, then there are $p \geq 2$ and $q \geq 2$ with $p * q = n$, and there is a sequence of steps our NTM could take that would cause the numbers generated nondeterministically to be precisely p and q; in other words, there is a sequence of steps that would cause it to accept the input. If $x \notin L$ (which means that $|x|$ is either prime or less than 2) then no matter what moves the NTM makes to generate $p \geq 2$ and $q \geq 2$, $p * q$ will not be n, and x will not be accepted.

We can describe the operation of the NTM more precisely. The nondeterminism will come from the component $G2$ shown in Figure 7.29, which does nothing but generate a string of two or more 1's on the tape.

The other components are the next-blank and previous-blank operations *NB* and *PB* introduced in Example 7.17; a TM M that computes the multiplication function from $\mathcal{N} \times \mathcal{N}$ to $\mathcal{N}$; and the *Equal* TM described in Example 7.24. The complete NTM is shown below:

$$NB \rightarrow G2 \rightarrow NB \rightarrow G2 \rightarrow PB \rightarrow M \rightarrow PB \rightarrow Equal$$

We illustrate one way this NTM might end up accepting the input string 1^{15}. The other possible way is for $G2$ to generate 111 first and then 11111.

$\underline{\Delta}$111111111111111	(original tape)
Δ111111111111111$\underline{\Delta}$	(after *NB*)
Δ111111111111111$\underline{\Delta}$11111	(after $G2$)
Δ111111111111111Δ11111$\underline{\Delta}$	(after *NB*)
Δ111111111111111Δ11111$\underline{\Delta}$111	(after $G2$)
Δ111111111111111$\underline{\Delta}$11111Δ111	(after *PB*)
Δ111111111111111$\underline{\Delta}$111111111111111	(after M)
$\underline{\Delta}$111111111111111Δ111111111111111	(after *PB*)
(accept)	(after *Equal*)

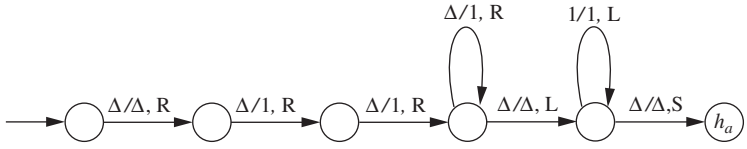


Figure 7.29
An NTM to generate a string of two or more 1's.

EXAMPLE 7.30 The Language of Prefixes of Elements of L

In this example we start with a language L that we assume is accepted by some Turing machine T, and we consider trying to find a TM that accepts the language $P(L)$ of prefixes of elements of L. $P(L)$ contains Λ as well as all the elements of L, and very likely (depending on L) many other strings as well.

If the input alphabet of T is Σ, a string $x \in \Sigma^*$ is a prefix of an element of L if there is a string $y \in \Sigma^*$ such that $xy \in L$. The set of candidates y we might have to test is infinite. (In the previous example there are easy ways to eliminate from consideration all but a finite set of possible factorizations.) This doesn't in itself rule out looking at the candidates one at a time, because an algorithm to accept $P(L)$ is allowed to loop forever on an input string not in $P(L)$.

There is another problem, however, that would also complicate things if we were looking for a deterministic algorithm. The only way we have of testing for membership in L is to use the Turing machine T, and T itself may loop forever on input strings not in L. This means that in order to test an input string x for membership in $P(L)$, simply executing T on all the strings xy, one at a time, is not sufficient: if T looped forever on xy_1, and there were another string y_2 that we hadn't tried yet for which $xy_2 \in L$, we would never find it.

This problem can also be resolved—see the proof of Theorem 8.9 for a similar argument. But for now this is just another way in which nondeterminism makes it easier to describe a solution. The NTM pictured below accepts $P(L)$. It uses a component G, which is identical to $G2$ in the previous example except for two features. First, it writes symbols in Σ, not just 1's (so that it is nondeterministic in two ways—with respect to which symbol it writes at each step, as well as when it stops); and second, it can generate strings of length 0 or 1 as well. Just as before, G is the only source of nondeterminism. The components NB and PB are from Example 7.17, and *Delete* was discussed in Example 7.20.

$$NB \rightarrow G \rightarrow Delete \rightarrow PB \rightarrow T$$

If the input string x is in $P(L)$, then there is a sequence of moves G can make that will cause it to generate a string y for which $xy \in L$. After the blank separating x and y is deleted, and the tape head is moved to the blank in square 0, T is executed as if its input were xy, and it will accept. On the other hand, if $x \notin P(L)$, then no matter what string y G generates, the string xy is not in L, and the NTM will not accept x because T will not accept xy.

Both these examples illustrate how nondeterminism can make it easy to describe an algorithm for accepting a language. But the algorithm being described is nondeterministic. The main result in this section, Theorem 7.31, asserts that a nondeterministic algorithm is sufficient, because once we have one, we can in principle replace it by an ordinary deterministic one. Turing machines share this feature with finite automata (where nondeterminism was helpful but not essential), whereas in the case of pushdown automata the nondeterminism could often not be eliminated.

Theorem 7.31

For every nondeterministic TM $T = (Q, \Sigma, \Gamma, q_0, \delta)$, there is an ordinary (deterministic) TM $T_1 = (Q_1, \Sigma, \Gamma_1, q_1, \delta_1)$ with $L(T_1) = L(T)$.

Proof

The idea of the proof is simple: we will describe an algorithm that can test, if necessary, every possible sequence of moves of T on an input string x. If there is a sequence of moves that would cause T to accept x, the algorithm will find it. It's only the details of the proof that are complicated, and we will leave some of them out.

There is an upper limit on the number of choices T might have in an arbitrary configuration: the maximum possible size of the set $\delta(q, \sigma)$, where $q \in Q$ and $\sigma \in \Gamma \cup \{\Delta\}$. Let us make the simplifying assumption that this upper bound is 2. Then we might as well assume that for every combination of nonhalting state and tape symbol, there are exactly two moves (which may be identical). We label these as move 0 and move 1, and the order is arbitrary.

These assumptions allow us to represent a sequence of moves of T on input x by a string of 0's and 1's. The string Λ represents the sequence of no moves; assuming that the moves represented by the string s take us to some nonhalting configuration C, the moves represented by the string $s0$ or the string $s1$ include the moves that take us to C, followed by move 0 or move 1, respectively, from that configuration.

The order in which we test sequences of moves corresponds to canonical order for the corresponding strings of 0's and 1's: We will test the sequence of zero moves, then all sequences of one move, then all sequences of two moves, and so forth. (This is likely to be a very inefficient algorithm, in the sense that testing a sequence of $n + 1$ moves requires us to start by repeating a sequence of n moves that we tried earlier.)

The essential requirement of the algorithm is that if there is a sequence of moves that allows T to accept x, the algorithm will find it and accept; and otherwise it will not accept. If x is never accepted and T can make arbitrarily long sequences of moves on x, the algorithm will never terminate. We will include a feature in the algorithm that *will* allow it to terminate if for some n, no matter what choices T makes, it reaches the reject state within n moves.

Our proof will not take full advantage of the Church-Turing thesis, because it will sketch some of the techniques a TM might use to organize the information it needs to carry out the algorithm. It will be a little easier to visualize if we allow T_1 to have four tapes:

1. The first tape holds the original input string x, and its contents never change.
2. The second tape contains the binary string that represents the sequence of moves we are currently testing.

3. The third tape will act as T's working tape as it makes the sequence of moves corresponding to the digits on tape 2.
4. We will describe the fourth tape's contents shortly; this will be the information that allows the algorithm to quit if it finds an n such that T always rejects the input within n moves.

Testing all possible sequences of zero moves is easy. The string on tape 2 is Λ, and T_1 doesn't have to do anything.

Now let us assume that T_1 has tested the sequence of moves corresponding to the string on tape 2, and that those moves did not result in x being accepted (if they did, then T_1 can quit and accept). Here is a summary of the steps T_1 takes to carry out the next iteration.

1. T_1 updates tape 2 so that it contains the next binary string in canonical order. If the current string is not a string of 1's, the next string will be the string, of the same length as the current one, that follows it in alphabetical order; if the current string is the string 1^n, then the next string is 0^{n+1}. There is one other operation performed by T_1 in this last case, which we will discuss shortly.
2. T_1 places the marker \$ in square 0 of tape 3, erases the rest of that tape, places a blank in square 1, and copies the input string x from tape 1, beginning in square 2. Tape 3 now looks like

   $$\$\underline{\Delta}x$$

   which if we ignore the \$ is the same as the initial tape of T. The purpose of the \$ is to allow T_1 to recover in case T would try to move its tape head off the tape during its computation.
3. T_1 then executes the moves of T corresponding to the binary digits on tape 2. At each step, it decides what move to make by consulting the current digit on tape 2 and the current symbol on tape 3. If at some point in this process T accepts, then T_1 accepts. If at some point T rejects, then T_1 copies all the binary digits on tape 2 onto tape 4, with a blank preceding them to separate that string from the ones already on tape 4.

Now we can explain the role played by tape 4 in the algorithm. If T_1 tries the sequence of moves corresponding to the binary string 1^n, the last string of length n in canonical order, and that sequence or some part of it causes T to reject, then T_1 writes 1^n on tape 4, and on the first step of the next iteration it examines tape 4 to determine whether *all* binary sequences of length n appear. If they do, then T_1 concludes that every possible sequence of moves causes T to reject within n steps, and at that point T_1 can reject.

7.8 | UNIVERSAL TURING MACHINES

The Turing machines we have studied so far have been special-purpose computers, capable of executing a single algorithm. Just as a modern computer is

a stored-program computer, which can execute any program stored in its memory, so a "universal" Turing machine, also anticipated by Alan Turing in a 1936 paper, can execute any algorithm, provided it receives an input string that describes the algorithm and any data it is to process.

Definition 7.32 Universal Turing Machines

A *universal* Turing machine is a Turing machine T_u that works as follows. It is assumed to receive an input string of the form $e(T)e(z)$, where T is an arbitrary TM, z is a string over the input alphabet of T, and e is an encoding function whose values are strings in $\{0, 1\}^*$. The computation performed by T_u on this input string satisfies these two properties:

1. T_u accepts the string $e(T)e(z)$ if and only if T accepts z.
2. If T accepts z and produces output y, then T_u produces output $e(y)$.

In this section we will first discuss a simple encoding function e, and then sketch one approach to constructing a universal Turing machine.

The idea of an encoding function turns out to be useful in itself, and in Chapter 8 we will discuss some applications not directly related to universal TMs. The crucial features of any encoding function e are these: First, it should be possible to decide algorithmically, for an arbitrary string $w \in \{0, 1\}^*$, whether w is a legitimate value of e (i.e., the encoding of a TM or the encoding of a string); second, a string w should represent at most one Turing machine, or at most one string; third, if w is of the form $e(T)$ or $e(z)$, there should be an algorithm for *decoding* w, to obtain the Turing machine T or the string z that it represents. Any function that satisfies these conditions is acceptable; the one we will use is not necessarily the best, but it is easy to describe.

We will make the assumption that in specifying a Turing machine, the labels attached to the states are irrelevant; in other words, we think of two TMs as being identical if their transition diagrams are identical except for the names of the states. This assumption allows us to number the states of an arbitrary TM and to base the encoding on the numbering.

The encoding of a string will also involve the assignment of numbers to alphabet symbols, but this is a little more complicated. As we have said, we want a string $w \in \{0, 1\}^*$ to encode no more than one TM. If two TMs T_1 and T_2 are identical except that one has tape alphabet $\{a, b\}$ and the other $\{a, c\}$, and if we agree that this difference is enough to make the two TMs different, then we can't assign b and c the same number. Similarly, we can't ever assign the same number to two different symbols that might show up in tape alphabets of TMs. This is the reason for adopting the convention stated below.

Convention

We assume that there is an infinite set $\mathcal{S} = \{a_1, a_2, a_3, \ldots\}$ of symbols, where $a_1 = \Delta$, such that the tape alphabet of every Turing machine T is a subset of $\mathcal{S}$.

The idea of the encoding we will use is simply that we represent a Turing machine as a set of moves, and each move

$$\delta(p, a) = (q, b, D)$$

is associated with a 5-tuple of numbers that represent the five components p, a, q, b, and D of the move. Each number is represented by a string of that many 1's, and the 5-tuple is represented by the string containing all five of these strings, each of the five followed by 0.

Definition 7.33 An Encoding Function

If $T = (Q, \Sigma, \Gamma, q_0, \delta)$ is a TM and z is a string, we define the strings $e(T)$ and $e(z)$ as follows.

First we assign numbers to each state, tape symbol, and tape head direction of T. Each tape symbol, including Δ, is an element a_i of $\mathcal{S}$, and it is assigned the number $n(a_i) = i$. The accepting state h_a, the rejecting state h_r, and the initial state q_0 are assigned the numbers $n(h_a) = 1$, $n(h_r) = 2$, and $n(q_0) = 3$. The other elements $q \in Q$ are assigned distinct numbers $n(q)$, each at least 4. We don't require the numbers to be consecutive, and the order is not important. Finally, the three directions R, L, and S are assigned the numbers $n(\mathrm{R}) = 1$, $n(\mathrm{L}) = 2$, and $n(\mathrm{S}) = 3$.

For each move m of T of the form $\delta(p, \sigma) = (q, \tau, D)$

$$e(m) = 1^{n(p)}01^{n(\sigma)}01^{n(q)}01^{n(\tau)}01^{n(D)}0$$

We list the moves of T in some order as $m_1, \ldots, m_k$, and we define

$$e(T) = e(m_1)0e(m_2)0 \ldots 0e(m_k)0$$

If $z = z_1 z_2 \ldots z_j$ is a string, where each $z_i \in \mathcal{S}$,

$$e(z) = 01^{n(z_1)}01^{n(z_2)}0 \ldots 01^{n(z_j)}0$$

EXAMPLE 7.34 A Sample Encoding of a TM

Let T be the TM shown in Figure 7.35, which transforms an input string of a's and b's by changing the leftmost a, if there is one, to b. We assume for simplicity that $n(a) = 2$ and $n(b) = 3$. By definition, $n(q_0) = 3$, and we let $n(p) = 4$ and $n(r) = 5$.

If m is the move determined by the formula $\delta(q_0, \Delta) = (p, \Delta, \mathrm{R})$, then

$$e(m) = 1^3 0101^4 01010 = 111010111101010$$

and if we encode the moves in the order they appear in the diagram, left to right,

$$e(T) = 111010111010100 \quad 1111011101111011110100 \quad 1111011011111011101100$$
$$1111010111110101100 \quad 1111101110111111011101100 \quad 11111010101011100$$

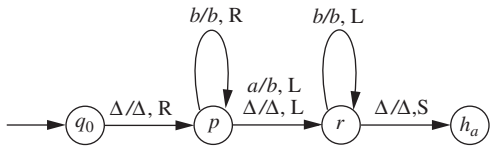


Figure 7.35

Because the states of a TM can be numbered in different ways, and the moves can be considered in any order, there are likely to be many strings of 0's and 1's that represent the same TM. The important thing, however, is that a string of 0's and 1's can't represent more than one.

Starting with a string such as the one in Example 7.34, it is easy to reconstruct the Turing machine it represents, because we can easily identify the individual moves. The only remaining question is whether we can determine, for an arbitrary string w of 0's and 1's, whether it actually represents a TM.

Every string of the form $e(T)$ must be a concatenation of one or more "5-tuples," each matching the regular expression $(11^*0)^50$. There are several ways in which a string having this form might still fail to be $e(T)$ for any Turing machine T. It might contain a 5-tuple that shows a move from one of the halt states; or it might contain more than one 5-tuple having the same first two parts (that is, either the same 5-tuple appears more than once, or there are 5-tuples representing more than one move for the same state-symbol combination); or it might contain a 5-tuple in which the last string of 1's has more than three. Theorem 7.36 asserts, however, that these are the only ways.

Theorem 7.36

Let $E = \{e(T) \mid T \text{ is a Turing machine}\}$. Then for every $x \in \{0, 1\}^*$, $x \in E$ if and only if all these conditions are satisfied:

1. x matches the regular expression $(11^*0)^50((11^*0)^50)^*$, so that it can be viewed as a sequence of one or more 5-tuples.
2. No two substrings of x representing 5-tuples can have the same first two parts (no move can appear twice, and there can't be two different moves for a given combination of state and tape symbol).
3. None of the 5-tuples can have first part 1 or 11 (there can be no moves from a halting state).
4. The last part of each 5-tuple must be 1, 11, or 111 (it must represent a direction).

These conditions do not guarantee that the string actually represents a TM that carries out a meaningful computation. There might be no transitions from the initial state, states that are unreachable for other reasons, and transitions to a state that doesn't show up in the encoding. (We can interpret missing transitions as rejecting

moves, just as we do for transition diagrams.) But if these conditions are satisfied, it is possible to draw a diagram that shows moves corresponding to each 5-tuple, and so the string encodes a TM.

In any case, testing a string $x \in \{0, 1\}^*$ to determine whether it satisfies these conditions is a straightforward process, and so we have verified that our encoding function e satisfies the minimal requirements for such a function.

In the last part of this section, we sketch how a universal TM T_u might be constructed. This time we will use three tapes. The first tape is for input and output and originally contains the string $e(T)e(z)$, where T is a TM and z is a string over the input alphabet of T. The second tape will correspond to the working tape of T, during the computation that simulates the computation of T on input z. The third tape will have only the encoded form of T's current state.

There is no confusion about where $e(T)$ stops and $e(z)$ starts on tape 1; in the first occurrence of 000, the first two 0's are the end of $e(T)$ and the third is the beginning of $e(z)$. T_u starts by transferring the string $e(z)$, except for the initial 0, from the end of tape 1 to tape 2, beginning in square 3. Since T begins with its leftmost square blank, T_u writes 10, the encoded form of Δ, in squares 1 and 2. Square 0 is left blank, and the tape head begins on square 1. The second step is for T_u to write 1110, the encoded form of the initial state, on tape 3, beginning in square 1.

As the simulation starts, the three tape heads are all on square 1. At each stage, the next move is determined by T's state, represented by the string on tape 3, and the current symbol on T's tape, whose encoding starts in the current position on tape 2. To simulate this move, T_u has to search tape 1 for the 5-tuple whose first two parts match this state-input combination; assuming it is found, the last three parts tell T_u how to carry out the move. To illustrate (returning to Example 7.34), suppose that before the search, the three tapes look like this:

$\Delta\underline{1}1101011101010011110111011110111010011110111011111011101100111101011\ldots$
$\Delta10111011101\underline{1}101110110\Delta$
$\Delta\underline{1}1111$

T_u searches for the 5-tuple that begins 11110110, which in this case is the third 5-tuple on tape 1. (5-tuples end in 00.) It indicates that T's current symbol should be changed from a (11) to b (111), that the state should change from p to r (from 1111 to 11111), and that the tape head should be moved left. After T_u simulates this move, the tapes look like

$\Delta\underline{1}1101011101010011110111011110111010011110111011111011101100111101011\ldots$
$\Delta10111011\underline{1}1011101110110\Delta$
$\Delta\underline{1}11111$

If T's computation on the input string z never terminates, then T_u will never halt. T might reject, because of an explicit transition to h_r, because no move is specified, or because the move calls for it to move its tape head off the tape. T_u can detect each of these situations (it detects the second by not finding on tape 1 the state-symbol combination it searches for), and in each case it rejects. Finally,

T might accept, which T_u will discover when it sees that the third part of the 5-tuple it is processing is 1; in this case, after T_u has modified tape 2 as the 5-tuple specifies, it erases tape 1, copies tape 2 onto tape 1, and accepts.

EXERCISES

7.1. Trace the TM in Figure 7.6, accepting the language $\{ss \mid s \in \{a, b\}^*\}$, on the string $aaba$. Show the configuration at each step.

7.2. Below is a transition table for a TM with input alphabet $\{a, b\}$.

q	σ	$\delta(q, \sigma)$	q	σ	$\delta(q, \sigma)$	q	σ	$\delta(q, \sigma)$
q_0	Δ	$(q_1, \Delta, \mathrm{R})$	q_2	Δ	$(h_a, \Delta, \mathrm{R})$	q_6	a	(q_6, a, R)
q_1	a	(q_1, a, R)	q_3	Δ	(q_4, a, R)	q_6	b	(q_6, b, R)
q_1	b	(q_1, b, R)	q_4	a	(q_4, a, R)	q_6	Δ	(q_7, b, L)
q_1	Δ	$(q_2, \Delta, \mathrm{L})$	q_4	b	(q_4, b, R)	q_7	a	(q_7, a, L)
q_2	a	$(q_3, \Delta, \mathrm{R})$	q_4	Δ	(q_7, a, L)	q_7	b	(q_7, b, L)
q_2	b	$(q_5, \Delta, \mathrm{R})$	q_5	Δ	(q_6, b, R)	q_7	Δ	$(q_2, \Delta, \mathrm{L})$

What is the final configuration if the TM starts with input string x?

7.3. Let $T = (Q, \Sigma, \Gamma, q_0, \delta)$ be a TM, and let s and t be the sizes of the sets Q and Γ, respectively. How many distinct configurations of T could there possibly be in which all tape squares past square n are blank and T's tape head is on or to the left of square n? (The tape squares are numbered beginning with 0.)

7.4. For each of the following languages, draw a transition diagram for a Turing machine that accepts that language.

a. $AnBn = \{a^n b^n \mid n \geq 0\}$

b. $\{a^i b^j \mid i < j\}$

c. $\{a^i b^j \mid i \leq j\}$

d. $\{a^i b^j \mid i \neq j\}$

7.5. For each part below, draw a transition diagram for a TM that accepts $AEqB = \{x \in \{a, b\}^* \mid n_a(x) = n_b(x)\}$ by using the approach that is described.

a. Search the string left-to-right for an a; as soon as one is found, replace it by X, return to the left end, and search for b; replace it by X; return to the left end and repeat these steps until one of the two searches is unsuccessful.

b. Begin at the left and search for either an a or a b; when one is found, replace it by X and continue to the right searching for the opposite symbol; when it is found, replace it by X and move back to the left end; repeat these steps until one of the two searches is unsuccessful.

7.6. Draw a transition diagram for a TM accepting *Pal*, the language of palindromes over $\{a, b\}$, using the following approach. Look at the

leftmost symbol of the current string, erase it but remember it, move to the rightmost symbol and see if it matches the one on the left; if so, erase it and go back to the left end of the remaining string. Repeat these steps until either the symbols are exhausted or the two symbols on the ends don't match.

7.7. Draw a transition diagram for a TM accepting *NonPal*, the language of nonpalindromes over $\{a, b\}$, using an approach similar to that in Exercise 7.6.

7.8. Refer to the transition diagram in Figure 7.8. Modify the diagram so that on each pass in which there are a's remaining to the right of the b, the TM erases the rightmost a. The modified TM should accept the same language but with no chance of an infinite loop.

7.9. Describe the language (a subset of $\{1\}^*$) accepted by the TM in Figure 7.37.

7.10. We do not define Λ-transitions for a TM. Why not? What features of a TM make it unnecessary or inappropriate to talk about Λ-transitions?

7.11. Given TMs $T_1 = (Q_1, \Sigma_1, \Gamma_1, q_1, \delta_1)$ and $T_2 = (Q_2, \Sigma_2, \Gamma_2, q_2, \delta_2)$, with $\Gamma_1 \subseteq \Sigma_2$, give a precise definition of the TM $T_1T_2 = (Q, \Sigma, \Gamma, q_0, \delta)$. Say precisely what Q, Σ, Γ, q_0, and δ are.

7.12. Suppose T is a TM accepting a language L. Describe how you would modify T to obtain another TM accepting L that never halts in the reject state h_r.

7.13. Suppose T is a TM that accepts every input. We might like to construct a TM R_T such that for every input string x, R_T halts in the accepting state

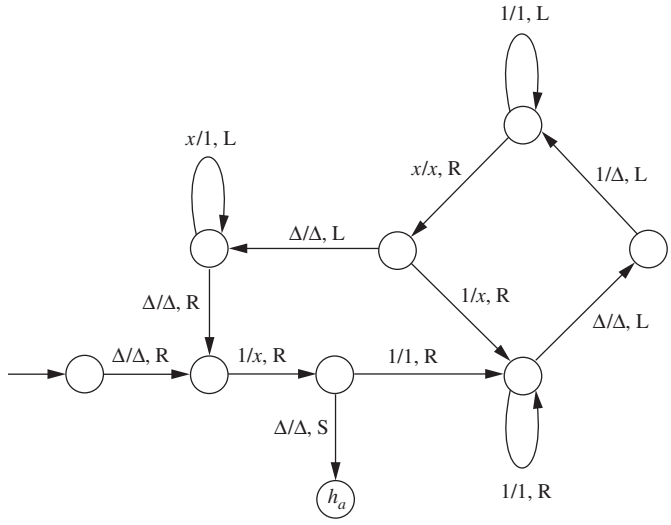


Figure 7.37

with exactly the same tape contents as when T halts on input x, but with the tape head positioned at the rightmost nonblank symbol on the tape. Show that there is no fixed TM T_0 such that $R_T = TT_0$ for every T. (In other words, there is no TM capable of executing the instruction "move the tape head to the rightmost nonblank tape symbol" in every possible situation.) Suggestion: Assume there is such a TM T_0, and try to find two other TMs T_1 and T_2 such that if $R_{T_1} = T_1T_0$ then R_{T_2} cannot be T_2T_0.

7.14. Draw the *Insert*(σ) TM, which changes the tape contents from $y\underline{z}$ to $y\underline{\sigma}z$. Here $y \in (\Sigma \cup \{\Delta\})^*$, $\sigma \in \Sigma \cup \{\Delta\}$, and $z \in \Sigma^*$. You may assume that $\Sigma = \{a, b\}$.

7.15. Draw a transition diagram for a TM *Substring* that begins with tape $\underline{\Delta}x\Delta y$, where $x, y \in \{a, b\}^*$, and ends with the same strings on the tape but with the tape head at the beginning of the first occurrence in y of the string x, if y contains an occurrence of x, and with the tape head on the first blank square following y otherwise.

7.16. Does every TM compute a partial function? Explain.

7.17. For each case below, draw a TM that computes the indicated function. In the first five parts, the function is from $\mathcal{N}$ to $\mathcal{N}$. In each of these parts, assume that the TM uses unary notation—i.e., the natural number n is represented by the string 1^n.

a. $f(x) = x + 2$

b. $f(x) = 2x$

c. $f(x) = x^2$

d. $f(x) =$ the smallest integer greater than or equal to $\log_2(x + 1)$ (i.e., $f(0) = 0$, $f(1) = 1$, $f(2) = f(3) = 2$, $f(4) = \ldots = f(7) = 3$, and so on).

e. E: $\{a, b\}^* \times \{a, b\}^* \to \{0, 1\}$ defined by $E(x, y) = 1$ if $x = y$, $E(x, y) = 0$ otherwise.

f. $p_l : \{a, b\}^* \times \{a, b\}^* \to \{0, 1\}$ defined by $p_l(x, y) = 1$ if $x < y$, $p_l(x, y) = 0$ otherwise. Here $<$ means with respect to "lexicographic," or alphabetical, order. For example, $a < aa$, $abab < abb$, etc.

g. p_c, the same function as in the previous part except this time $<$ refers to canonical order. That is, a shorter string precedes a longer one, and the order of two strings of the same length is alphabetical.

7.18. The TM shown in Figure 7.38 computes a function from $\{a, b\}^*$ to $\{a, b\}^*$. For any string $x \in \{a, b\}^*$, describe the string $f(x)$.

7.19. Suppose TMs T_1 and T_2 compute the functions f_1 and f_2 from $\mathcal{N}$ to $\mathcal{N}$, respectively. Describe how to construct a TM to compute the function $f_1 + f_2$.

7.20. Draw a transition diagram for a TM with input alphabet $\{0, 1\}$ that interprets the input string as the binary representation of a nonnegative integer and adds 1 to it.

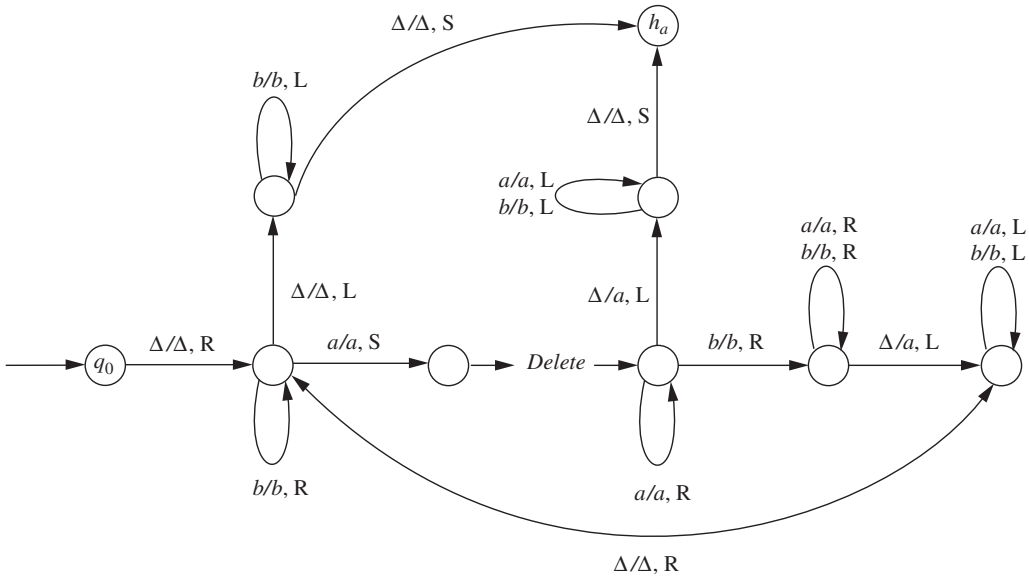


Figure 7.38 |

7.21. Draw a TM that takes as input a string of 0's and 1's, interprets it as the binary representation of a nonnegative integer, and leaves as output the unary representation of that integer (i.e., a string of that many 1's).

7.22. Draw a TM that does the reverse of the previous problem: accepts a string of n 1's as input and leaves as output the binary representation of n.

7.23. Draw a transition diagram for a three-tape TM that works as follows: starting in the configuration $(q_0, \underline{\Delta}x, \underline{\Delta}y, \underline{\Delta})$, where x and y are strings of 0's and 1's of the same length, it halts in the configuration $(h_a, \underline{\Delta}x, \underline{\Delta}y, \underline{\Delta}z)$, where z is the string obtained by interpreting x and y as binary representations and adding them.

7.24. In Example 7.5, a TM is given that accepts the language $\{ss \mid s \in \{a, b\}^*\}$. Draw a TM with tape alphabet $\{a, b\}$ that accepts this language.

7.25. We can consider a TM with a *doubly infinite* tape, by allowing the numbers of the tape squares to be negative as well as positive. In most respects the rules for such a TM are the same as for an ordinary one, except that now when we refer to the configuration $xq\sigma y$, including the initial configuration corresponding to some input string, there is no assumption about exactly where on the tape the strings and the tape head are. Draw a transition diagram for a TM with a doubly infinite tape that does the following: If it begins with the tape blank except for a single a somewhere on it, it halts in the accepting state with the head on the square with the a.

7.26. Let G be the nondeterministic TM described in Example 7.30, which begins with a blank tape, writes an arbitrary string x on the tape, and halts with tape $\underline{\Delta}x$. Let *NB*, *PB*, *Copy*, and *Equal* be the TMs described in Examples 7.17, 7.18, and 7.24. Consider the NTM

$$NB \rightarrow G \rightarrow Copy \rightarrow NB \rightarrow Delete \rightarrow PB \rightarrow PB \rightarrow Equal$$

which is nondeterministic because G is. What language does it accept?

7.27. Using the idea in Exercise 7.26, draw a transition diagram for an NTM that accepts the language $\{1^n \mid n = k^2 \text{ for some } k \geq 0\}$.

7.28. Suppose L is accepted by a TM T. For each of the following languages, describe informally how to construct a nondeterministic TM that will accept that language.

a. The set of all suffixes of elements of L

b. The set of all substrings of elements of L

7.29. Suppose L_1 and L_2 are subsets of Σ^* and T_1 and T_2 are TMs accepting L_1 and L_2, respectively. Describe how to construct a nondeterministic TM to accept L_1L_2.

7.30. Suppose T is a TM accepting a language L. Describe how to construct a nondeterministic TM accepting L^*.

7.31. Table 5.8 describes a PDA accepting the language *Pal*. Draw a TM that accepts this language by simulating the PDA. You can make the TM nondeterministic, and you can use a second tape to represent the stack.

7.32. Describe informally how to construct a TM T that enumerates the set of palindromes over $\{0, 1\}$ in canonical order. In other words, T loops forever, and for every positive integer n, there is some point at which the initial portion of T's tape contains the string

$$\Delta\Delta 0\Delta 1\Delta 00\Delta 11\Delta 000\Delta \ldots \Delta x_n$$

where x_n is the nth palindrome in canonical order, and this portion of the tape is never subsequently changed.

7.33. Suppose you are given a Turing machine T (you have the transition diagram), and you are watching T processing an input string. At each step you can see the configuration of the TM: the state, the tape contents, and the tape head position.

a. Suppose that for some n, the tape head does not move past square n while you are watching. If the pattern continues, will you be able to conclude at some point that the TM is in an infinite loop? If so, what is the longest you might need to watch in order to draw this conclusion?

b. Suppose that in each move you observe, the tape head moves right. If the pattern continues, will you be able to conclude at some point that the TM is in an infinite loop? If so, what is the longest you might need to watch in order to draw this conclusion?

7.34. In each of the following cases, show that the language accepted by the TM T is regular.

a. There is an integer n such that no matter what the input string is, T never moves its tape head to the right of square n.

b. For every $n \geq 0$ and every input of length n, T begins by making $n + 1$ moves in which the tape head is moved right each time, and thereafter T does not move the tape head to the left of square $n + 1$.

7.35. †Suppose T is a TM. For each integer $i \geq 0$, denote by $n_i(T)$ the number of the rightmost square to which T has moved its tape head within the first i moves. (For example, if T moves its tape head right in the first five moves and left in the next three, then $n_i(T) = i$ for $i \leq 5$ and $n_i(T) = 5$ for $6 \leq i \leq 10$.) Suppose there is an integer k such that no matter what the input string is, $n_i(T) \geq i - k$ for every $i \geq 0$. Does it follow that $L(T)$ is regular? Give reasons for your answer.

7.36. Suppose M_1 is a two-tape TM, and M_2 is the ordinary TM constructed in Theorem 7.26 to simulate M_1. If M_1 requires n moves to process an input string x, give an upper bound on the number of moves M_2 requires in order to simulate the processing of x. Note that the number of moves M_1 has made places a limit on the position of its tape head. Try to make your upper bound as sharp as possible.

7.37. Show that if there is a TM T computing the function $f : \mathcal{N} \to \mathcal{N}$, then there is another one, T', whose tape alphabet is $\{1\}$. Suggestion: Suppose T has tape alphabet $\Gamma = \{a_1, a_2, \ldots, a_n\}$. Encode Δ and each of the a_i's by a string of 1's and Δ's of length $n + 1$ (for example, encode Δ by $n + 1$ blanks, and a_i by $1^i\Delta^{n+1-i}$). Have T' simulate T, but using blocks of $n + 1$ tape squares instead of single squares.

7.38. Beginning with a nondeterministic Turing machine T_1, the proof of Theorem 7.31 shows how to construct an ordinary TM T_2 that accepts the same language. Suppose $|x| = n$, T_1 never has more than two choices of moves, and there is a sequence of n_x moves by which T_1 accepts x. Estimate as precisely as possible the number of moves that might be required for T_2 to accept x.

7.39. Formulate a precise definition of a *two-stack automaton*, which is like a PDA except that it is deterministic and a move takes into account the symbols on top of both stacks and can replace either or both of them. Describe informally how you might construct a machine of this type accepting $\{a^i b^i c^i \mid i \geq 0\}$. Do it in a way that could be generalized to $\{a^i b^i c^i d^i \mid i \geq 0\}$, $\{a^i b^i c^i d^i e^i \mid i \geq 0\}$, etc.

7.40. Describe how a Turing machine can simulate a two-stack automaton; specifically, show that any language that can be accepted by a two-stack machine can be accepted by a TM.

7.41. A *Post machine* is similar to a PDA, but with the following differences. It is deterministic; it has an auxiliary queue instead of a stack; and the input

is assumed to have been previously loaded onto the queue. For example, if the input string is abb, then the symbol currently at the front of the queue is a. Items can be added only to the rear of the queue, and deleted only from the front. Assume that there is a marker Z_0 initially on the queue following the input string (so that in the case of null input Z_0 is at the front). The machine can be defined as a 7-tuple $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$, like a PDA. A single move depends on the state and the symbol currently at the front of the queue; and the move has three components: the resulting state, an indication of whether or not to remove the current symbol from the front of the queue, and what to add to the rear of the queue (a string, possibly null, of symbols from the queue alphabet).

Construct a Post machine to accept the language $\{a^n b^n c^n \mid n \geq 0\}$.

7.42. We can specify a configuration of a Post machine (see Exercise 7.41) by specifying the state and the contents of the queue. If the original marker Z_0 is currently in the queue, so that the string in the queue is of the form $\alpha Z_0 \beta$, then the queue can be thought of as representing the tape of a Turing machine, as follows. The marker Z_0 is thought of, not as an actual tape symbol, but as marking the right end of the string on the tape; the string β is at the beginning of the tape, followed by the string α; and the tape head is currently centered on the first symbol of α—or, if $\alpha = \Lambda$, on the first blank square following the string β. In this way, the initial queue, which contains the string αZ_0, represents the initial tape of the Turing machine with input string α, except that the blank in square 0 is missing and the tape head scans the first symbol of the input.

Using this representation, it is not difficult to see how most of the moves of a Turing machine can be simulated by the Post machine. Here is an illustration. Suppose that the queue contains the string $abbZ_0ab$, which we take to represent the tape $ab\underline{a}bb$. To simulate the Turing machine move that replaces the a by c and moves to the right, we can do the following:

a. remove a from the front and add c to the rear, producing bbZ_0abc
b. add a marker, say \$, to the rear, producing $bbZ_0abc\$$
c. begin a loop that simply removes items from the front and adds them to the rear, continuing until the marker \$ appears at the front. At this point, the queue contains $\$bbZ_0abc$.
d. remove the marker, so that the final queue represents the tape $abc\underline{b}b$

The Turing machine move that is hardest to simulate is a move to the left. Devise a way to do it. Then give an informal proof, based on the simulation outlined in this discussion, that any language that can be accepted by a Turing machine can be accepted by a Post machine.

7.43. Show how a two-stack automaton can simulate a Post machine, using the first stack to represent the queue and using the second stack to help carry out the various Post machine operations. The first step in the simulation is

to load the input string onto stack 1, using stack 2 first in order to get the symbols in the right order. Give an informal argument that any language that can be accepted by a Post machine can be accepted by a two-stack automaton. (The conclusion from this exercise and the preceding ones is that the three types of machines—Turing machines, Post machines, and two-stack automata—are equivalent with regard to the languages they can accept.)

CHAPTER

8

Recursively Enumerable Languages

Recursively enumerable languages are the ones that can be accepted by Turing machines, and in the first section of the chapter we distinguish them from *recursive* languages, those that can be *decided* by Turing machines. Only if a language is recursive is there an algorithm guaranteed to determine whether an arbitrary string is an element. In the second and third sections of the chapter we examine two other ways of characterizing recursively enumerable languages, one in terms of algorithms to list the elements and one using *unrestricted grammars*. In the fourth section, we discuss the Chomsky hierarchy, which contains four classes of languages, each having a corresponding model of computation and a corresponding type of grammar. Two of the three classes other than recursively enumerable languages are the ones discussed earlier in the book. The remaining one, the class of context-sensitive languages, is also described briefly. In the last section we use a *diagonal* argument to demonstrate precisely that there are more languages than there are Turing machines. As a result, there must be some languages that are not recursively enumerable and others that are recursively enumerable but not recursive.

8.1 RECURSIVELY ENUMERABLE AND RECURSIVE

Definition 8.1 Accepting a Language and Deciding a Language

A Turing machine T with input alphabet Σ accepts a language $L \subseteq \Sigma^*$ if $L(T) = L$. T *decides* L if T computes the characteristic function $\chi_L : \Sigma^* \to \{0, 1\}$. A language L is *recursively enumerable* if there is a TM that accepts L, and L is *recursive* if there is a TM that decides L.

Recursively enumerable languages are sometimes referred to as Turing-acceptable, and recursive languages are sometimes called Turing-decidable, or simply decidable.

As the discussion in Section 7.3 suggested, trying to accept L and trying to decide L are two ways of approaching the membership problem for L:

$$\text{Given a string } x \in \Sigma^*, \text{ is } x \in L?$$

To decide L is to solve the problem conclusively, because for every $x \in \Sigma^*$, deciding whether $x \in L$ is exactly what must be done in order to determine the value of $\chi_L(x)$. Accepting L may be less conclusive. For a string $x \notin L$, an algorithm accepting L works correctly as long as it doesn't report that $x \in L$; but not reporting that $x \in L$ is less informative than reporting that $x \notin L$. At this stage, however, though the definitions are clearly different, it is not obvious whether there are really languages satisfying one and not the other.

We will be able to resolve these questions before this chapter is over. In this first section, we establish several facts that will help us understand the relationships between the two types of languages, starting in Theorem 8.2 with the fundamental relationship that we figured out in Section 7.3.

In proving the statements in this section, as well as in later parts of the chapter, we will take advantage of the Church-Turing thesis. For example, to show that a language L is recursively enumerable, it will be sufficient to describe an algorithm (without describing exactly how a TM might execute the algorithm) that answers yes if the input string is in L and either answers no or doesn't answer if the input string is not in L.

Theorem 8.2
Every recursive language is recursively enumerable.

Proof
Suppose T is a TM that decides $L \subseteq \Sigma^*$. An algorithm to accept L is the following: Given $x \in \Sigma^*$, execute T on input x. T will halt and produce an output; if the output is 1 (i.e., if $\chi_L(x) = 1$), accept, and if the output is 0, reject.

The reason the converse of Theorem 8.1 is not obviously true, and will turn out to be false, is that if T accepts L, there may be input strings not in L that cause T to loop forever. The closest we can come to the converse statement is Theorem 8.3.

Theorem 8.3
If $L \subseteq \Sigma^*$ is accepted by a TM T that halts on every input string, then L is recursive.

Proof
If T halts on every input string, then the following is an algorithm for deciding L: Given $x \in \Sigma^*$, if T accepts x, return 1, and if T rejects x, return 0.

A consequence of Theorem 8.3 is that if L is accepted by a nondeterministic TM T, and if there is no input string on which T can possibly loop forever, then L is recursive. The reason is that we can convert T to a deterministic TM T_1 as described in Section 7.7. The construction guarantees that if there are no inputs allowing T to loop forever, then T_1 halts on every input.

Theorem 8.4
If L_1 and L_2 are both recursively enumerable languages over Σ, then $L_1 \cup L_2$ and $L_1 \cap L_2$ are also recursively enumerable.

Proof
Suppose that T_1 is a TM that accepts L_1, and T_2 is a TM that accepts L_2. The algorithms we describe to accept $L_1 \cup L_2$ and $L_1 \cap L_2$ will both process an input string x by running T_1 and T_2 simultaneously on the string x. We might implement the algorithm by building a two-tape TM that literally simulates both TMs simultaneously, one on each tape, or we might simply use transition diagrams for both TMs to trace one move of each on the input string x, then two moves of each, and so forth.

To accept $L_1 \cup L_2$, the algorithm is simply to wait until one of the two TMs accepts x, and to accept only when that happens. If one rejects, we abandon it and continue with the other. If both eventually reject, we can reject and halt, though this is not required in order to accept $L_1 \cup L_2$. If both TMs loop forever, then we never receive an answer, but the only case in which this can occur is when $x \notin L_1 \cup L_2$.

To accept $L_1 \cap L_2$, the algorithm is to wait until both TMs accept x, and to accept only when that happens. This time, if one TM accepts, we continue with the other; if either TM ever rejects, we can reject and halt, though again this is not necessary.

Theorem 8.5
If L_1 and L_2 are both recursive languages over Σ, then $L_1 \cup L_2$ and $L_1 \cap L_2$ are also recursive.

Proof
See Exercise 8.1.

Theorem 8.6
If L is a recursive language over Σ, then its complement L' is also recursive.

Proof
If T is a TM that computes the characteristic function χ_L, the TM obtained from T by interchanging the two outputs computes the characteristic function of L'.

Theorem 8.7
If L is a recursively enumerable language, and its complement L' is also recursively enumerable, then L is recursive (and therefore, by Theorem 8.6, L' is recursive).

Proof
If T is a TM accepting L, and T_1 is another TM accepting L', then here is an algorithm to decide L: For a string x, execute T and T_1 simultaneously on input x, until one halts. (One will eventually accept, because either $x \in L$ or $x \in L'$.) If the one that halts first is T, and it accepts, or the one that halts first is T_1, and it rejects, return 1; otherwise return 0.

Theorem 8.7 implies that if there are any nonrecursive languages, then there must be languages that are not recursively enumerable. (Otherwise, for every language L, both L and L' would be recursively enumerable and therefore recursive.)

Suppose we have a TM T that accepts a language L. The possibility that T might loop forever on an input string x not in L might prevent us from using T to decide L. (This is why the converse of Theorem 8.2 is not obviously true, as we observed above.) The same possibility might also prevent us from using T in an algorithm to accept L' (this is why Theorem 8.6 is stated for recursive languages, not recursively enumerable languages), because accepting L' requires answering yes for strings in L', and these are the strings for which T might not return an answer.

For the language L accepted by T, according to Theorems 8.6 and 8.7, the two problems are equivalent: If we could find another TM to accept L', then we could find another TM to decide L, and conversely. We will see later in this chapter that the potential difficulty cannot always be eliminated. There *are* languages that can be accepted by TMs but not decided. These are the same languages that can be accepted by TMs but whose complements cannot; they are languages that can be accepted, but only by TMs that loop forever for some inputs not in the language.

8.2 | ENUMERATING A LANGUAGE

To enumerate a set means to list the elements, and we begin by saying precisely how a Turing machine enumerates a language L (or, informally, lists the elements

of L). The easiest way to formulate the definition is to use a multitape TM with one tape that operates exclusively as the output tape.

Definition 8.8 A TM Enumerating a Language

Let T be a k-tape Turing machine for some $k \geq 1$, and let $L \subseteq \Sigma^*$. We say T enumerates L if it operates such that the following conditions are satisfied.

1. The tape head on the first tape never moves to the left, and no nonblank symbol printed on tape 1 is subsequently modified or erased.
2. For every $x \in L$, there is some point during the operation of T when tape 1 has contents

$$x_1\#x_2\#\ldots\#x_n\#x\#$$

for some $n \geq 0$, where the strings $x_1, x_2, \ldots, x_n$ are also elements of L and $x_1, x_2, \ldots, x_n, x$ are all distinct. If L is finite, then nothing is printed after the # following the last element of L.

This idea leads to a characterization of recursively enumerable languages, and with an appropriate modification involving the order in which the strings are listed, it can also be used to characterize recursive languages.

Theorem 8.9

For every language $L \subseteq \Sigma^*$, L is recursively enumerable if and only if there is a TM enumerating L, and L is recursive if and only if there is a TM that enumerates the strings in L in canonical order (see Section 1.4).

Proof

We have to prove, for an alphabet Σ and a language $L \subseteq \Sigma^*$, these four things:

1. If there is a TM that accepts L, then there is a TM that enumerates L.
2. If there is a TM that enumerates L, then there is a TM that accepts L.
3. If there is a TM that decides L, then there is a TM that enumerates L in canonical order.
4. If there is a TM that enumerates L in canonical order, then there is a TM that decides L.

In all four of these proofs, just as in Section 8.1, we will take advantage of the Church-Turing thesis.

We start with statement 3, which is easier than statement 1. If T decides L, then for any string x, we can give x to T and wait for it to tell us whether $x \in L$. So here is an algorithm to list the elements of L in canonical order: Consider the strings of L in canonical order; for each

string x, give it to T and wait for the answer, and if the answer is yes, include x in the list. Then the order in which the elements of L are listed is the same as the order in which they are considered, which is canonical order.

For statement 1, this argument doesn't work without some modification, because if T is only assumed to accept L, "give it to T and wait for the answer" might mean waiting forever. The modification is that although we start by considering the elements x_0, x_1, x_2, ... of Σ^* in canonical order, we may not be able to decide whether to list a string x_i the first time we consider it. Instead, we make repeated passes. On each pass, we consider one additional string from the canonical-order list, and for each string x_i that we're still unsure about, we execute T on x_i for one more step than we did in the previous pass.

For $\{a, b\}^* = \{\Lambda, a, b, aa, \ldots\}$, here is a description of the computations of T that we may have to perform during the first four passes. (Whenever we decide that a string x is in L, we eliminate it from the canonical-order list, so that it is not included more than once in our enumeration.)

Pass 1: 1 step on input Λ

Pass 2: 2 steps on Λ, 1 step on a

Pass 3: 3 steps on Λ, 2 steps on a, 1 step on b

Pass 4: 4 steps on Λ, 3 steps on a, 2 steps on b, 1 step on aa

The enumeration that is produced will contain only strings that are accepted by T; and every string x that is accepted by T in exactly k steps will appear during the pass on which we first execute T for k steps on input x. The enumeration required in statement 1 does not need to be in canonical order, and we can't expect that it will be: There may be strings x and y such that even though x precedes y in canonical order, y is included in the enumeration on an earlier pass than x because T needs fewer steps to accept it.

Statement 2 is easy. If T enumerates L, then an algorithm to accept L is this: Given a string x, watch the computation of T, and accept x precisely if T lists the string x. (This algorithm illustrates in an extreme way the difference between accepting a language and deciding a language; at least if L is infinite, the only two possible outcomes for a string x are that x will be accepted and that no answer will ever be returned.)

Statement 4 is almost as easy, except for a slight subtlety in the case when L is finite, and allows us to use the same approach. This time, because T is guaranteed to enumerate L in canonical order, we will be able to decide whether a string x is in L as soon as one of these two things happens: (i) x appears in the enumeration (which means that $x \in L$); (ii) a string y that follows x in canonical order appears in the enumeration, and x has not yet appeared (which means that $x \notin L$). As long as L is infinite,

one of these two things will eventually happen. If L is finite, it may be that neither happens, because our rules for enumerating a language allow T to continue making moves forever after it has printed the last element of L. However, we do not need the assumption in statement 4 to conclude that a finite language L is recursive; in fact, every finite language is regular.

8.3 | MORE GENERAL GRAMMARS

In this section we introduce a type of grammar more general than a context-free grammar. These *unrestricted* grammars correspond to recursively enumerable languages in the same way that CFGs correspond to languages accepted by PDAs and regular grammars to those accepted by FAs.

The feature of context-free grammars that imposes such severe restrictions on the corresponding languages (the feature that makes it possible to prove the pumping lemma for CFLs) is precisely their context-freeness: every sufficiently long derivation must contain a "self-embedded" variable A (one for which the derivation looks like $S \Rightarrow^* vAz \Rightarrow^* vwAyz$), and any production having left side A can be applied in any place that A appears.

The grammars we are about to introduce will therefore allow productions involving a variable to depend on the context in which the variable appears. To illustrate: $Aa \rightarrow ba$ allows A to be replaced by b, but only if it is followed immediately by a. In fact, the easiest way to describe these more general productions is to drop the association of a production with a specific variable, and to think instead of a *string* being replaced by another string ($Aa \rightarrow b$, for example). We still want to say that a derivation terminates when the current string contains no more variables; for this reason, we will require the left side of every production to contain at least one variable. Context-free productions $A \rightarrow \alpha$ will certainly still be allowed, but in general they will not be the only ones.

Definition 8.10 Unrestricted Grammars

An *unrestricted* grammar is a 4-tuple $G = (V, \Sigma, S, P)$, where V and Σ are disjoint sets of variables and terminals, respectively. S is an element of V called the start symbol, and P is a set of productions of the form

$$\alpha \rightarrow \beta$$

where $\alpha, \beta \in (V \cup \Sigma)^*$ and α contains at least one variable.

We can continue to use much of the notation that was developed for CFGs. For example,

$$\alpha \Rightarrow_G^* \beta$$

means that β can be derived from α in G, in zero or more steps, and

$$L(G) = \{x \in \Sigma^* \mid S \Rightarrow_G^* x\}$$

One important difference is that because the productions are not necessarily context-free, the assumption that $S \Rightarrow^* xAy \Rightarrow^* z$, where $A \in V$ and $x, y, z \in \Sigma^*$, no longer implies that $z = xwy$ for some string w.

EXAMPLE 8.11 A Grammar Generating $\{a^{2^k} \mid k \in \mathcal{N}\}$

Let $L = \{a^{2^k} \mid k \in \mathcal{N}\}$. L can be defined recursively by saying that $a \in L$ and that for every $n \geq 1$, if $a^n \in L$, then $a^{2n} \in L$. Using this idea to obtain a grammar means finding a way to double the number of a's in the string obtained so far. The idea is to use a variable D that will act as a "doubling operator." D replaces each a by two a's, by means of the production $Da \rightarrow aaD$. At the beginning of each pass, D is introduced at the left end of the string, and we think of each application of the production as allowing D to move past an a, doubling it in the process. The complete grammar has the productions

$$S \rightarrow LaR \qquad L \rightarrow LD \qquad Da \rightarrow aaD \qquad DR \rightarrow R \qquad L \rightarrow \Lambda \qquad R \rightarrow \Lambda$$

Beginning with the string LaR, the number of a's will be doubled every time a copy of D is produced and moves through the string. Both variables L and R can disappear at any time. There is no danger of producing a string of a's in which the action of one of the doubling operators is cut short, because if R disappears when D is present, there is no way for D to be eliminated. The string $aaaa$ has the derivation

$$\begin{aligned} S \Rightarrow LaR &\Rightarrow LDaR \Rightarrow LaaDR \Rightarrow LaaR \Rightarrow LDaaR \\ &\Rightarrow LaaDaR \Rightarrow LaaaaDR \Rightarrow LaaaaR \Rightarrow aaaaR \Rightarrow aaaa \end{aligned}$$

EXAMPLE 8.12 A Grammar Generating $\{a^n b^n c^n \mid n \geq 1\}$

The previous example used the idea of a variable moving through the string and operating on it. In this example we use a similar left-to-right movement, although there is no explicit "operator" like the variable D, as well as another kind arising from the variables rearranging themselves. Like the previous example, this one uses the variable L to denote the left end of the string.

We begin with the productions

$$S \rightarrow SABC \mid LABC$$

which allow us to obtain strings of the form $L(ABC)^n$, where $n \geq 1$. Next, productions that allow the variables A, B, and C to arrange themselves in alphabetical order:

$$BA \rightarrow AB \qquad CB \rightarrow BC \qquad CA \rightarrow AC$$

Finally, productions that allow the variables to be replaced by the corresponding terminals, *provided* they are in alphabetical order:

$$LA \rightarrow a \qquad aA \rightarrow aa \qquad aB \rightarrow ab \qquad bB \rightarrow bb \qquad bC \rightarrow bc \qquad cC \rightarrow cc$$

Although nothing forces the variables to arrange themselves in alphabetical order, doing so is the only way they can ultimately be replaced by terminals.

None of the productions can rearrange existing terminal symbols. This means, for example, that if an a in the string, which must come from an A, has a terminal right before

it, that terminal must be an a, because no other terminal could have allowed the A to become an a. The combination cb cannot occur for the same reason. The final string has equal numbers of a's, b's, and c's, because originally there were equal numbers of the three variables, and the terminal symbols must be in alphabetical order.

These examples illustrate the fact that unrestricted grammars can generate non-context-free languages (see Exercises 6.2 and 6.3), although both these languages still involve simple repetitive patterns that make it relatively easy to find grammars. As we are about to see, unrestricted grammars must be able to generate more complicated languages, because every recursively enumerable language can be generated this way. We begin with the converse result.

Theorem 8.13

For every unrestricted grammar G, there is a Turing machine T with $L(T) = L(G)$.

Proof

We will describe a Turing machine T that accepts $L(G)$, and we can simplify the description considerably by making T nondeterministic. Its tape alphabet will contain all the variables and terminals in G, and perhaps other symbols, and it works as follows.

The first phase of T's operation is simply to move the tape head to the blank square following the input string x.

During the second phase of its operation, T treats this blank square as if it were the beginning of its tape, and the input string x is undisturbed. In this phase, T simulates a derivation in G nondeterministically, as follows. First, the symbol S is written on the tape in the square following the blank. Each subsequent step in the simulation can be accomplished by T carrying out these actions:

1. Choosing a production $\alpha \rightarrow \beta$ in the grammar G.
2. Selecting an occurrence of α, if there is one, in the string currently on the tape.
3. Replacing this occurrence of α by β (which, if $|\alpha| \neq |\beta|$, means moving the string immediately following this occurrence either to the left or to the right).

The nondeterminism shows up in both steps 1 and 2. This simulation phase of T's operation may continue forever, if for each production chosen in step 1 it is actually possible in step 2 to find an occurrence of the left side in the current string. Otherwise, step 2 will eventually fail because the left side of the production chosen in step 1 does not appear in the string. (One way this might happen, though not the only way, is that the string has no more variables and is therefore actually an element of $L(G)$.) In this case, the tape head moves back past the blank and past the original input string x to the blank at the beginning of the tape.

The final phase is to compare the two strings on the tape (x and the string produced by the simulation), and to accept if and only if they are equal.

If $x \in L(G)$, then there is a sequence of moves that causes the simulation to produce x as the second string on the tape, and therefore causes T to accept x. If $x \notin L(G)$, then it is impossible for x to be accepted by T; even if the second phase terminates and the second string on the tape contains only terminals, it *is* generated by productions in G, and so it will not be x.

Theorem 8.14

For every Turing machine T with input alphabet Σ, there is an unrestricted grammar G generating the language $L(T) \subseteq \Sigma^*$.

Proof

For simplicity, we assume that $\Sigma = \{a, b\}$. The grammar we construct will have three types of productions:

1. Productions that generate, for every string $x \in \{a, b\}^*$, a string containing two identical copies of x, which we will refer to as x_1 and x_2. The string also has additional variables that allow x_2 to represent the initial configuration of T corresponding to input x, and that prevent the derivation from producing a string of terminals before it has been determined whether T accepts x.
2. Productions that transform x_2 (the portion representing a configuration of T) so as to simulate the moves of T on input x, while keeping x_1 unchanged.
3. Productions that allow everything in the string except x_1 (the unchanged copy of x) to be erased, *provided* the computation of T that is being simulated reaches the accept state.

The initial configuration of T corresponding to input x includes the symbols Δ and q_0, as well as the symbols of x, and so these two symbols will be used as variables in the productions of type 1. In addition, using left and right parentheses as variables will make it easy to keep the two copies x_1 and x_2 separate. The productions of type 1 are

$$S \to S(\Delta\Delta) \mid T \qquad T \to T(aa) \mid T(bb) \mid q_0(\Delta\Delta)$$

For example, a derivation might begin

$$S \Rightarrow S(\Delta\Delta) \Rightarrow T(\Delta\Delta) \Rightarrow T(aa)(\Delta\Delta)$$
$$\Rightarrow T(bb)(aa)(\Delta\Delta) \Rightarrow (\Delta\Delta)q_0(bb)(aa)(\Delta\Delta)$$

where we think of the final string as representing the configuration $\Delta q_0 ba \Delta$. The actual string has two copies of each blank as well as each terminal symbol. The two copies of each blank are just to simplify things slightly, but the two copies of each terminal are necessary: The first copy

belongs to x_1 and the second to x_2. The TM configuration being represented can have any number of blanks at the end of the string, but this is the stage of the derivation in which we need to produce enough symbols to account for all the tape squares used by the TM as it processes the input string. The configuration we have shown that ends in one blank (so that the string ends in a pair of blanks) would be appropriate if the TM needs to move its tape head to the blank at the right of the input string, but no farther, in the process of accepting it.

The productions of types 2 and 3 will be easy to understand if we consider an example. Let's not worry about what language the TM accepts, but suppose that the transition diagram contains the transitions in Figure 8.1.

Then the computation that results in the string ba being accepted is this:

$$q_0\Delta ba\Delta \vdash \Delta q_1 ba\Delta \vdash \Delta bq_1 a\Delta \vdash \Delta q_2 b\$\Delta \vdash \Delta bq_2\$\Delta$$
$$\vdash \Delta b\$q_3\Delta \vdash \Delta b\$h_a\Delta$$

In the first step, which involves the TM transition $\delta(q_0, \Delta) = (q_1, \Delta, \mathrm{R})$, the initial portion $q_0\Delta$ of the configuration is transformed to Δq_1; the grammar production that will simulate this step is

$$q_0(\Delta\Delta) \rightarrow (\Delta\Delta)q_1$$

It is the second Δ in each of these two parenthetical pairs that is crucial, because it is interpreted to be the symbol on the TM tape at that step in the computation. (It may have started out as a different symbol, but now it is Δ.) The first symbol in each pair remains unchanged during the derivation. It is conceivable that for a different input string, the TM will encounter a Δ in state q_0 that was originally a or b, and for that reason the grammar will also have the productions

$$q_0(a\Delta) \rightarrow (a\Delta)q_1 \quad \text{and} \quad q_0(b\Delta) \rightarrow (b\Delta)q_1$$

In the computation we are considering, the second, fourth, and fifth steps also involve tape head moves to the right. In the fourth step, for example, the portion q_2b of the configuration is transformed to bq_2, and thus the grammar will have all three productions

$$q_2(\Delta b) \rightarrow (\Delta b)q_2 \qquad q_2(ab) \rightarrow (ab)q_2 \qquad q_2(bb) \rightarrow (bb)q_2$$

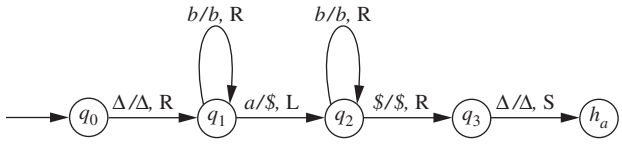


Figure 8.15 |
Part of a TM transition diagram.

although the derivation corresponding to this particular computation requires only the last one.

The grammar productions corresponding to a tape head move to the left are a little more complicated. In the third step of our computation, the portion bq_1a of the configuration is transformed to $q_2b\$$; the corresponding production is

$$(bb)q_1(aa) \rightarrow q_2(bb)(a\$)$$

and for the reasons we have just described, the grammar will contain every possible production of the form

$$(\sigma_1\sigma_2)q_1(\sigma_3 a) \rightarrow q_2(\sigma_1\sigma_2)(\sigma_3\$)$$

where σ_1 and σ_3 belong to the set $\{a, b, \Delta\}$ and σ_2 is one of these three or any other tape symbol of the TM.

The last step of the computation above involves a move in which the tape head is stationary, transforming $q_3\Delta$ to $h_a\Delta$, and the corresponding productions look like

$$q_3(\sigma\Delta) \rightarrow h_a(\sigma\Delta)$$

where σ is a, b, or Δ.

The appearance of the variable h_a in the derivation is what will make it possible for everything in the string except the copy x_1 of the original input string to disappear. So far, the derivation has produced a string containing pairs of the form $(\sigma_1\sigma_2)$ and one occurrence of the variable h_a. The productions of type 3 will first allow the h_a to propagate until one precedes each of the pairs, and then allow each pair to disappear unless the first symbol is a or b (i.e., a symbol in the unchanged copy x_1 of the string x). The "propagating" productions look like

$$(\sigma_1\sigma_2)h_a \rightarrow h_a(\sigma_1\sigma_2)h_a \quad \text{and} \quad h_a(\sigma_1\sigma_2) \rightarrow h_a(\sigma_1\sigma_2)h_a$$

and the "disappearing" productions look like

$$h_a(\sigma_1\sigma_2) \rightarrow \sigma_1$$

if σ_1 is either a or b, and

$$h_a(\Delta\sigma_2) \rightarrow \Lambda$$

This example already illustrates the sorts of productions that are required for an arbitrary Turing machine. The productions of type 1 allow two copies x_1 and x_2 of an arbitrary string x to be generated. The productions corresponding to TM moves allow the derivation to proceed by simulating the TM computation on x_2, and if x is accepted the last type of production permits the derivation to continue until only the string $x_1 = x$ of terminals is left. On the other hand, if x is not accepted by the TM, the variable h_a will never make an appearance in the derivation, and there is no other way for x to be obtained as the final product.

We conclude the argument by displaying the complete derivation of the string ba in our example, along with the TM moves corresponding to the middle phase of the derivation. At each step, we have underlined the left side of the production to be used next.

$$
\begin{array}{lcll}
\underline{S} & \Rightarrow & \underline{S}(\Delta\Delta) & \\
 & \Rightarrow & \underline{T}(\Delta\Delta) & \\
 & \Rightarrow & \underline{T}(aa)(\Delta\Delta) & \\
 & \Rightarrow & \underline{T}(bb)(aa)(\Delta\Delta) & \\
 & \Rightarrow & \underline{q_0(\Delta\Delta)}(bb)(aa)(\Delta\Delta) & q_0\Delta ba \\
 & \Rightarrow & (\Delta\Delta)\underline{q_1(bb)}(aa)(\Delta\Delta) & \vdash \quad \Delta q_1 ba \\
 & \Rightarrow & (\Delta\Delta)\underline{(bb)q_1(aa)}(\Delta\Delta) & \vdash \quad \Delta b q_1 a \\
 & \Rightarrow & (\Delta\Delta)\underline{q_2(bb)}(a\$)(\Delta\Delta) & \vdash \quad \Delta q_2 b\$ \\
 & \Rightarrow & (\Delta\Delta)(bb)\underline{q_2(a\$)}(\Delta\Delta) & \vdash \quad \Delta b q_2\$ \\
 & \Rightarrow & (\Delta\Delta)(bb)(a\$)\underline{q_3(\Delta\Delta)} & \vdash \quad \Delta b\$ q_3\Delta \\
 & \Rightarrow & (\Delta\Delta)(bb)\underline{(a\$)h_a}(\Delta\Delta) & \vdash \quad \Delta b\$ h_a\Delta \\
 & \Rightarrow & (\Delta\Delta)\underline{(bb)h_a}(a\$)h_a(\Delta\Delta) & \\
 & \Rightarrow & \underline{(\Delta\Delta)h_a}(bb)h_a(a\$)h_a(\Delta\Delta) & \\
 & \Rightarrow & \underline{h_a(\Delta\Delta)}h_a(bb)h_a(a\$)h_a(\Delta\Delta) & \\
 & \Rightarrow & \underline{h_a(bb)}h_a(a\$)h_a(\Delta\Delta) & \\
 & \Rightarrow & bh_a\underline{(a\$)}h_a(\Delta\Delta) & \\
 & \Rightarrow & bah_a\underline{(\Delta\Delta)} & \\
 & \Rightarrow & ba &
\end{array}
$$

8.4 CONTEXT-SENSITIVE LANGUAGES AND THE CHOMSKY HIERARCHY

Definition 8.16 Context-Sensitive Grammars

A *context-sensitive grammar* (CSG) is an unrestricted grammar in which no production is length-decreasing. In other words, every production is of the form $\alpha \rightarrow \beta$, where $|\beta| \geq |\alpha|$.

A language is a context-sensitive language (CSL) if it can be generated by a context-sensitive grammar.

In this section we will discuss these grammars and these languages briefly, and we will describe a type of abstract computing device that corresponds to CSLs in the same way that pushdown automata correspond to CFLs and Turing machines to

recursively enumerable languages. See Exercise 8.32 for another characterization of context-sensitive languages that may make it easier to understand why the phrase "context-sensitive" is appropriate.

EXAMPLE 8.17 A CSG Generating $L = \{a^n b^n c^n \mid n \geq 1\}$

The unrestricted grammar we used in Example 8.12 for this language is not context-sensitive, because of the production $LA \rightarrow a$, but using the same principle we can easily find a grammar that is. Instead of using the variable A as well as a separate variable to mark the beginning of the string, we introduce a new variable to serve both purposes. It is not hard to verify that the CSG with productions

$$S \rightarrow SABC \mid \mathcal{A}BC \qquad BA \rightarrow AB \qquad CA \rightarrow AC \qquad CB \rightarrow BC$$
$$\mathcal{A} \rightarrow a \qquad a\mathcal{A} \rightarrow aa \qquad a\mathcal{B} \rightarrow ab \qquad b\mathcal{B} \rightarrow bb \qquad b\mathcal{C} \rightarrow bc \qquad c\mathcal{C} \rightarrow cc$$

generates the language L.

Because a context-sensitive grammar cannot have Λ-productions, no language containing Λ can be a CSL; however, as Theorem 4.27 shows, every context-free language is either a CSL or the union of $\{\Lambda\}$ and a CSL. It is appropriate to think of context-sensitive languages as a generalization of context-free languages, and as Example 8.17 suggests, many CSLs are not context-free. In fact, just about any familiar language is context-sensitive; in particular, programming languages are—the non-context-free features of the C language mentioned in Example 6.6 can be handled by context-sensitive grammars. In any case, if we want to find a model of computation that corresponds exactly to CSLs, it will have to be potentially more powerful than a PDA.

Theorem 8.13 describes a way of constructing a Turing machine corresponding to a given unrestricted grammar G. One possible way of thinking about an appropriate model of computation is to refer to this construction and see how the TM's operations are restricted if the grammar is actually context-sensitive.

The TM described in Theorem 8.13 simulates a derivation in G, using the space on the tape to the right of the input string. The tape head never needs to move farther to the right than the blank at the end of the current string. If G is context-sensitive, the current string at any stage of the derivation is no longer than the string of terminals being derived. Therefore, for an input string of length n, the tape head never needs to move past square $2n$ (approximately) during the computation. This restriction on the space used in processing a string is the crucial feature of the following definition.

Definition 8.18 Linear-Bounded Automata

A *linear-bounded automaton* (LBA) is a 5-tuple $M = (Q, \Sigma, \Gamma, q_0, \delta)$ that is identical to a nondeterministic Turing machine, with the following

exception. There are two extra tape symbols, [and], assumed not to be elements of the tape alphabet Γ. The initial configuration of M corresponding to input x is $q_0[x]$, with the symbol [in the leftmost square and the symbol] in the first square to the right of x. During its computation, M is not permitted to replace either of these brackets or to move its tape head to the left of the [or to the right of the].

Theorem 8.19

If $L \subseteq \Sigma^*$ is a context-sensitive language, then there is a linear-bounded automaton that accepts L.

Proof

We can follow the proof of Theorem 8.13, except that instead of being able to use the space to the right of the input string on the tape, the LBA must use the space between the two brackets. It can do this by converting individual symbols into symbol-pairs so as to simulate two tape "tracks," the first containing the input string and the second containing the necessary computation space.

Suppose $G = (V, \Sigma, S, P)$ is a context-sensitive grammar generating L. In addition to other tape symbols, including elements of Σ, the LBA M that we construct will have in its tape alphabet symbols of the form (a, b), where a and b are elements of $\Sigma \cup V \cup \{\Delta\}$. Its first step will be to convert the tape contents

$$\underline{[}x_1 x_2 \ldots x_n]$$

to

$$[\underline{(x_1, \Delta)}(x_2, \Delta) \ldots (x_n, \Delta)]$$

From this point on, the LBA follows the same procedure as the TM in Theorem 8.8, using the second track of the tape as its working space. It places S in the first space on this track (replacing the first Δ) and begins the nondeterministic simulation of a derivation in G. As in the previous proof, the simulation terminates if, having chosen a production $\alpha \rightarrow \beta$, the LBA is unable to locate an occurrence of α in the current string; at that point, the original input string is accepted if it matches the string currently in the second track and rejected if it doesn't. The new feature of this computation is that the LBA also rejects the input if some iteration produces a string that is too long to fit into the n positions of the second track.

Because G is context-sensitive, no string appearing in the derivation of $x \in L$ can be longer than x. This means that if the LBA begins with input x, there is a sequence of moves that will cause it to accept. If $x \notin L$, no simulated derivation will be able to produce the string x, and the input will not be accepted.

The use of two tape tracks in the proof of Theorem 8.19 allows the LBA to get by with n squares on the tape instead of $2n$. Using k tracks instead of two would allow the LBA to simulate a computation that would ordinarily require kn squares. This idea is the origin of the phrase *linear-bounded*: an LBA can simulate the computation of a TM, provided the portion of the tape used by the TM is bounded by some linear function of the input length.

Theorem 8.20

If $L \subseteq \Sigma^*$ is accepted by a linear-bounded automaton $M = (Q, \Sigma, \Gamma, q_0, \delta)$, then there is a context-sensitive grammar G generating $L - \{\Lambda\}$.

Proof

We give only a sketch of the proof, which is similar to the proof of Theorem 8.14. The simple change described in Example 8.17 to the grammar in Example 8.12 helps to understand how the unrestricted grammar of Theorem 8.14 can be modified to make it context-sensitive.

In the grammar constructed in the proof of that theorem, the only variables were S, T, left and right parentheses, tape symbols of the TM that were not input symbols, and Δ. Productions in that grammar such as $h_a(b\Delta) \to b$ and $h_a(\Delta b) \to \Lambda$ violate the context-sensitive condition; the way to salvage the first one is to interpret $h_a(b\Delta)$ as a single variable, and the second one will be unnecessary because there are no blanks initially between the end markers on the tape of M.

Because of the tape markers, G may also have variables such as $(a[\Delta)$, $(a\Delta])$, and $(a\,[\,\Delta\,])$, corresponding to situations during the simulation when the current symbol Δ, in a square originally containing a, is the leftmost, rightmost, or only symbol between the markers, respectively. Finally, because the tape head of M can move to a tape square containing [or], we will also have variables such as $q(a\,[\,\Delta)^L$ or $q(a\,[\,\Delta\,])^R$. The first one signifies that M is in state q, its tape head is on the square containing [, and the next square contains Δ but originally contained a.

The first portion of a derivation in G produces a string of the form

$$q_0(\sigma_1\,[\,\sigma_1)^L(\sigma_2\sigma_2)\ldots(\sigma_{n-1}\sigma_{n-1})(\sigma_n\sigma_n\,]\,)$$

or

$$q_0(\sigma\,[\,\sigma\,]\,)^L$$

and the productions needed to generate these strings are straightforward.

Because of the more complicated variables, the productions needed to simulate the LBA's moves require that we consider more combinations. We will give the productions for the moves in which the tape head moves right, and the others are comparable.

Corresponding to the move

$$\delta(p, a) = (q, b, \mathrm{R})$$

we have these productions:

$$\begin{aligned} p(\sigma_1 a)(\sigma_2\sigma_3) &\to (\sigma_1 b)q(\sigma_2\sigma_3) \\ p(\sigma_1\,[\,a)(\sigma_2\sigma_3) &\to (\sigma_1\,[\,b)q(\sigma_2\sigma_3) \\ p(\sigma_1 a)(\sigma_2\sigma_3\,]\,) &\to (\sigma_1 b)q(\sigma_2\sigma_3\,]\,) \\ p(\sigma_1\,[\,a)(\sigma_2\sigma_3) &\to (\sigma_1\,[\,b)q(\sigma_2\sigma_3) \\ p(\sigma_1 a\,]\,) &\to q(\sigma_1 b\,]\,)^R \\ p(\sigma_1\,[\,a\,]\,) &\to q(\sigma_1\,[\,b\,]\,)^R \end{aligned}$$

for every combination of $\sigma_1, \sigma_2 \in \Sigma$ and $\sigma_3 \in \Gamma \cup \{\Delta\}$. In the first four lines both sides contain two variables, and in the last two lines both sides contain only one variable.

Corresponding to the move

$$\delta(p, [\,) = (q, [\,, \mathrm{R})$$

we have the two productions

$$\begin{aligned} p(\sigma_1\,[\sigma_2)^L &\to q(\sigma_1\,[\sigma_2) \\ p(\sigma_1\,[\sigma_2\,]\,)^L &\to q(\sigma_1\,[\sigma_2\,]\,) \end{aligned}$$

for each $\sigma_1 \in \Sigma$ and each $\sigma_2 \in \Gamma \cup \{\Delta\}$.

The productions used in the last portion of the derivation, in which copies of h_a proliferate and then things disappear, are similar to the ones in the proof of Theorem 8.14, except for the way we interpret them. A production like

$$(a\Delta)h_a(ba) \to h_a(a\Delta)h_a(ba)$$

involves the two variables $(a\Delta)$ and $h_a(ba)$ on the left side and two on the right, and one like

$$h_a(a\Delta) \to a$$

involves only one variable on the left.

The four classes of languages that we have studied—regular, context-free, context-sensitive, and recursively enumerable—make up what is often referred to as the *Chomsky hierarchy*. Chomsky himself designated the four types as type 3, type 2, type 1, and type 0, in order from the most restrictive to the most general. Each level of the hierarchy, which is summarized in Table 8.21, can be characterized by a class of grammars and by a specific model of computation.

The phrase *type 0 grammar* was originally used to describe a grammar in which the left side of every production is a string of variables. It is not difficult to see that every unrestricted grammar is equivalent to one with this property (Exercise 8.27).

The class of recursive languages does not show up explicitly in Table 8.21, because there is no known way to characterize these languages using grammars.

Table 8.21 | The Chomsky Hierarchy

Type	Languages (Grammars)	Form of Productions in Grammar	Accepting Device
3	Regular	$A \to aB$, $A \to \Lambda$ $(A, B \in V, a \in \Sigma)$	Finite automaton
2	Context-free	$A \to \alpha$ $(A \in V, \alpha \in (V \cup \Sigma)^*)$	Pushdown automaton
1	Context-sensitive	$\alpha \to \beta$ $(\alpha, \beta \in (V \cup \Sigma)^*, \lvert\beta\rvert \geq \lvert\alpha\rvert,$ α contains a variable)	Linear-bounded automaton
0	Recursively enumerable (unrestricted)	$\alpha \to \beta$ $(\alpha, \beta \in (V \cup \Sigma)^*,$ α contains a variable)	Turing machine

Theorem 8.22, however, allows us to locate this class with respect to the ones in Table 8.21.

Theorem 8.22

Every context-sensitive language L is recursive.

Proof

Let G be a context-sensitive grammar generating L. Theorem 8.19 says that there is an LBA M accepting L. According to the remark following Theorem 8.3, it is sufficient to show that there is a nondeterministic TM T_1 accepting L such that no input string can possibly cause T_1 to loop forever.

We may consider M to be a nondeterministic TM T, which begins by inserting the markers [and] in the squares where they would be already if we were thinking of T as an LBA. The TM T_1 is constructed as a modification of T. T_1 also begins by placing the markers on the tape, although it will not be prohibited from moving its tape head past the right marker, and it also transforms the tape between the two markers so that it has two tracks. Just like T, T_1 performs the first iteration in a simulated derivation in G by writing S in the first position of the second track. Before the second iteration, however, it moves to the blank portion of the tape after the right marker and records the string S obtained in the first iteration. In each subsequent iteration, it executes the following steps, the first three of which are the same as those of T:

1. It selects a production $\alpha \to \beta$ in G.
2. It attempts to select an occurrence of α in the current string of the derivation; if it is unable to, it compares the current string to the input x, accepts x if they are equal, and rejects x if they are not.

3. If it can select an occurrence of α, it attempts to replace it by β, and rejects if this would result in a string longer than the input.
4. If it successfully replaces α by β, it compares the new current string with the strings it has written in the portion of the tape to the right of], and rejects if the new one matches one of them; otherwise it writes the new string after the most recent entry, so that the strings on the tape correspond exactly to the steps of the derivation so far. Then it returns to the portion of the tape between the markers for another iteration.

For every string x generated by G, and only for an x of this type, there is a sequence of moves of T_1 that causes x to be accepted. A string may be rejected at several possible points during the operation of T_1: in step 2 of some iteration, or step 3, or step 4. If x is not accepted, then one of these three outcomes *must* eventually occur, because if the iterations continue to produce strings that are not compared to x and are no longer than x, eventually one of them will appear a second time.

Returning to the Chomsky hierarchy, we have the inclusions

$$\mathcal{S}_3 \subseteq \mathcal{S}_2 \subseteq \mathcal{S}_1 \subseteq \mathcal{R} \subseteq \mathcal{S}_0$$

where $\mathcal{R}$ is the set of recursive languages, $\mathcal{S}_i$ is the set of languages of type i for $i = 0$ and $i = 1$, and for $i = 2$ and $i = 3$ $\mathcal{S}_i$ is the set of languages of type i that don't contain Λ. (The last inclusion is Theorem 8.2.) We know already that the first two are strict inclusions: there are context-free languages that are not regular and context-sensitive languages that are not context-free. The last two inclusions are also strict: the third is discussed in Exercise 9.32, and in Chapter 9, Section 9.1, we will find an example of a recursively enumerable language that is not recursive.

Exercise 8.33 discusses the closure properties of the set of context-sensitive languages with respect to operations such as union and concatenation. The question of whether the complement of a CSL is a CSL is much harder and was not answered until 1987. Szelepcsényi and Immerman showed independently in 1987 and 1988 that if L can be accepted by an LBA, then L' can also be. There are still open questions concerning context-sensitive languages, such as whether or not every CSL can be accepted by a deterministic LBA.

8.5 NOT EVERY LANGUAGE IS RECURSIVELY ENUMERABLE

In this section, we will consider languages over an alphabet Σ, such as $\{0, 1\}$, and Turing machines with input alphabet Σ. We will show there are more languages than there are Turing machines to accept them, so that there must be many languages not accepted by any TMs.

Before we start, a word of explanation. In the next chapter, we will actually produce an example of a language $L \subseteq \{0, 1\}^*$ that is not recursively enumerable.

Why should we spend time now proving that such languages exist, if we will soon have an example? There are three reasons: (i) the technique we use is a remarkable one that caused considerable controversy in mathematical circles when it was introduced in the nineteenth century; (ii) it is almost exactly the same as the technique we will use in constructing our first example, and it will help in constructing others; and (iii) the example we finally obtain will not be an isolated one, because our discussion in this section will show that *most* languages are not recursively enumerable.

The first step is to explain how it makes sense to say that one infinite set, the set of languages over $\{0, 1\}$, is *larger than* another infinite set, the set of TMs with input alphabet $\{0, 1\}$.

The best way to do that will be to formulate two definitions: what it means for a set A to be the same size as a set B, and what it means for A to be larger than B. The most familiar case is when the sets are finite. The set $A = \{a, b, c, d\}$ is the same size as $B = \{3, 5, 7, 9\}$ and larger than $C = \{1, 15, 20\}$. Why? We might say that A and B both have four elements and C has only three. A more helpful answer, because it works with infinite sets too, is that A is the same size as B because the elements of A can be matched up with the elements of B in an exact correspondence (in other words, there is a *bijection* $f : A \to B$); and A is larger than C, because A has a subset the same size as C but A itself is not the same size as C.

These two approaches to comparing two finite sets are not really that different. One of the simplest ways to say what "A has four elements" means is to say that there is a bijection from A to the set $\{1, 2, 3, 4\}$. *Counting* the elements of A ("one, two, three, four") is shorthand for "this is the element that corresponds to 1, this is the element that corresponds to 2, ..., this is the element that corresponds to 4." In fact, the easiest way to say "A is finite" is to say that for some natural number k, there is a bijection from A to the set $\{1, 2, 3, \ldots, k\}$. If there are bijections from A to $\{1, 2, 3, 4\}$ and from B to $\{1, 2, 3, 4\}$, and if we just want to say that A and B have the same size, without worrying about what that size is, we can dispense with the set $\{1, 2, 3, 4\}$ and match up the elements of A directly with those of B.

Definition 8.23 A Set *A* of the Same Size as *B* or Larger Than *B*

Two sets A and B, either finite or infinite, are the same size if there is a bijection $f : A \to B$. A is larger than B if some subset of A is the same size as B but A itself is not.

As you might expect, the "same size as" terminology can be used in ways that are familiar intuitively, because the same-size relation is an equivalence relation on sets (see Exercise 1.27). For example, "A is the same size as B" and "B is the

same size as A" are interchangeable, and we can speak of several sets as being the same size. One would hope that the "larger-than" relation is transitive, and it is, but this is harder to demonstrate; see Exercise 8.48.

The distinguishing feature of any *infinite* set A is that there is an inexhaustible supply of elements. There is at least one element a_0; there is an element a_1 different from a_0; there is another element a_2 different from both a_0 and a_1; and so forth. For every natural number n, the elements $a_0, a_1, \ldots, a_n$ do not exhaust the elements of A, because it is infinite, and so there is an element a_{n+1} different from all these. Every infinite set A must have an infinite subset $A_1 = \{a_0, a_1, a_2, \ldots\}$, whose elements can be arranged in a *list* (so that every element of A_1 appears exactly once in the list).

An infinite set of this form, $A_1 = \{a_0, a_1, a_2, \ldots\}$, is the same size as the set $\mathcal{N}$ of natural numbers, because the function $f : \mathcal{N} \to A_1$ defined by the formula $f(i) = a_i$ is a bijection. The preceding paragraph allows us to say that every infinite set is either the same size as, or larger than, $\mathcal{N}$. In other words, $\mathcal{N}$ (or any other set the same size) is the smallest possible infinite set. We call such a set *countably infinite*, because as we have seen, starting to count the elements means starting to list them: the first, the second, and so on. We can at least *start* counting. We can never finish, because the set is infinite, but we can go as far as we want. For every element x in the set, we can continue down the list until we reach x.

Definition 8.24 Countably Infinite and Countable Sets

A set A is *countably infinite* (the same size as $\mathcal{N}$) if there is a bijection $f : \mathcal{N} \to A$, or a list $a_0, a_1, \ldots$ of elements of A such that every element of A appears exactly once in the list. A is *countable* if A is either finite or countably infinite.

Theorem 8.25

Every infinite set has a countably infinite subset, and every subset of a countable set is countable.

Proof

The first statement was discussed above. If A is countable and $B \subseteq A$, it is sufficient to consider the case when A and B are both infinite. In this case, $A = \{a_0, a_1, a_2, \ldots\}$, and the set $I = \{i \in \mathcal{N} \mid a_i \in B\}$ is infinite. If we let j_0 be the smallest number in I, j_1 the next smallest, and so on, then $B = \{b_{j_0}, b_{j_1}, \ldots\}$, so that B is countable.

Next we present several examples of countable sets, and we will see in the very first one that infinite sets can seem counter-intuitive or even paradoxical.

EXAMPLE 8.26 The Set $\mathcal{N} \times \mathcal{N}$ Is Countable

We can describe the set by drawing a two-dimensional array:

(0,0)	(0,1)	(0,2)	(0,3)	. . .
(1,0)	(1,1)	(1,2)	(1,3)	. . .
(2,0)	(2,1)	(2,2)	(2,3)	. . .
(3,0)	(3,1)	(3,2)	(3,3)	. . .
. . .				

The ordered pairs in the 0th row, the ones with first coordinate 0, are in an infinite list and form a countably infinite subset. So do the pairs in the first row, and for every $n \in \mathcal{N}$, so do the pairs in the nth row. There is a countably infinite set of pairwise disjoint subsets of $\mathcal{N} \times \mathcal{N}$, each of which is countably infinite. It probably seems obvious that $\mathcal{N} \times \mathcal{N}$ must be bigger than $\mathcal{N}$.

Obvious, maybe, but not true. As they appear in the array, the elements of $\mathcal{N} \times \mathcal{N}$ are enough to fill up infinitely many infinite lists. But they can also be arranged in a single list, as Figure 8.27 is supposed to suggest. The path starts with the ordered pair $(0, 0)$, which is the only one in which the two coordinates add up to 0. The first time the path spirals back through the array, it hits the ordered pairs $(0, 1)$ and $(1, 0)$, the two in which the coordinates add up to 1. The ordered pair (i, j) will be hit by the path during its nth pass back through the array, where $n = i + j$. Therefore, $\mathcal{N} \times \mathcal{N}$ is countable—the same size as, not bigger than, $\mathcal{N}$. (When this result was published by Georg Cantor in the 1870s, it seemed so counterintuitive that many people felt it could not be correct.)

For *finite* sets A and B, the assumption that A has a subset the same size as B and at least one additional element is enough to imply that A is larger than B. Example 8.26 illustrates why this assumption, or even one that seems much stronger, is not enough in the case of infinite sets.

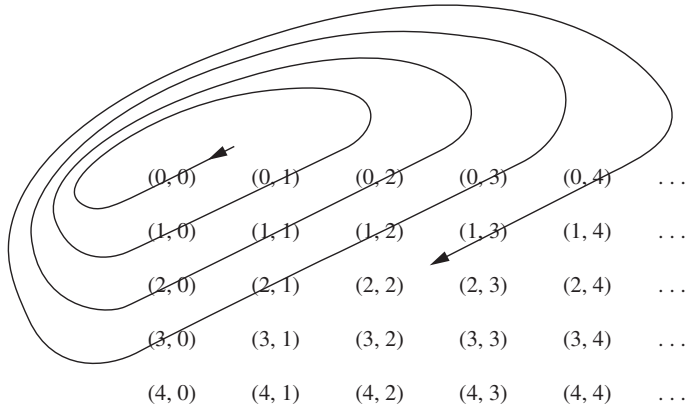


Figure 8.27
The set $\mathcal{N} \times \mathcal{N}$ is countable.

EXAMPLE 8.28 A Countable Union of Countable Sets Is Countable

If S_i is a countable set for every $i \in \mathcal{N}$, then

$$S = \bigcup_{i=0}^{\infty} S_i$$

is countable. The result is a generalization of the previous example and also follows from the idea underlying Figure 8.27. This time the ordered pair (i, j) in the figure stands for the jth element of S_i, so that the ith row of the two-dimensional array represents the elements of S_i. We must be a little more careful to specify the elements in the final list $S = \{a_0, a_1, a_2, \ldots\}$ (the list determined by the spiral path), for two reasons: first, the jth element of S_i may not exist, because S_i is allowed to be finite; and second, there may be elements of S that belong to S_i for more than one i, and we don't want any element to appear more than once in the list. The path in the figure still works, if we say first that a_0 is the first element of any S_i hit by the path (this makes sense provided not all the S_i's are empty, and of course if they are, then S is empty); and second that for each $n > 0$, a_n is the first element hit by the path that is not already included among $a_0, a_1, \ldots, a_{n-1}$. (Again, if there is no such element, then S is finite and therefore countable.)

EXAMPLE 8.29 Languages Are Countable Sets

For a finite alphabet Σ (such as $\{a, b\}$), the set Σ^* of all strings over Σ is countable. This follows from Example 8.28, because

$$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$$

and each of the sets Σ^i is countable. In fact, each Σ^i is finite, and a way of listing the elements of Σ^* that probably seems more natural than using the path in Figure 8.27 is to list the strings in Σ^0, then the ones in Σ^1, then the ones in Σ^2, and so forth. If strings in Σ^i are listed alphabetically, then the resulting infinite list is just what we get by listing all the strings in canonical order.

EXAMPLE 8.30 The Set of Turing Machines Is Countable

Let $\mathcal{T}$ represent the set of Turing machines. A TM T can be represented by the string $e(T) \in \{0, 1\}^*$, and a string can represent at most one TM. Therefore, the resulting function e from $\mathcal{T}$ to $\{0, 1\}^*$ is one-to-one, and we may think of it as a bijection from $\mathcal{T}$ to a subset of $\{0, 1\}^*$. Because $\{0, 1\}^*$ is countable, every subset is, and we can conclude that $\mathcal{T}$ is countable.

Saying that the set of Turing machines with input alphabet Σ is countable is approximately the same as saying that the set $\mathcal{RE}(\Sigma)$ of recursively enumerable languages over Σ is countable. A language $L \in \mathcal{RE}(\Sigma)$ can be accepted by a TM T with input alphabet Σ, and a TM can accept only one language over its input alphabet. Therefore, since $\mathcal{T}$ is countable, the same argument we have just used shows that $\mathcal{RE}(\Sigma)$ is countable. Finally, as long as we accept the idea that there are only countably many alphabets, we may say because of Example 8.28 that the set of all recursively enumerable languages is countable.

Example 8.30 provides us with a portion of the result we stated at the beginning of this section. The set of TMs with input alphabet $\{0, 1\}$ is countable, and therefore a relatively small infinite set. We have not yet shown, however, that there are any larger ones. There are actually *much* larger ones (Exercise 8.45 shows that for every set S, there is a larger set), but to complete the result we want, it will be enough to show that the set of all languages over $\{0, 1\}$ is not countable. In other words, the set of subsets of $\{0, 1\}^*$ is uncountable. Because $\{0, 1\}^*$ and $\mathcal{N}$ are both countably infinite, and because two countably infinite sets are identical, size-wise, except for the notation used to describe the elements, the next example is all we need.

EXAMPLE 8.31 The Set $2^{\mathcal{N}}$ Is Uncountable

We wish to show that there can be no list of subsets of $\mathcal{N}$ containing every subset of $\mathcal{N}$. In other words, every list $A_0, A_1, A_2, \ldots$ of subsets of $\mathcal{N}$ must leave out at least one.

The question is, without knowing anything about the subsets in the list, how can we hope to show that there's at least one subset missing? The answer is provided by an ingenious argument of Cantor's called a *diagonal* argument (we'll see why shortly). Here is a subset, constructed from the ones in the list, that cannot possibly be in the list:

$$A = \{i \in \mathcal{N} \mid i \notin A_i\}$$

The reason that A must be different from A_i for every i is that A and A_i differ because of the number i, which is in one but not both of the two sets: if $i \in A_i$, then by definition of A, i does not satisfy the defining condition of A, and so $i \notin A$; and if $i \notin A_i$, then (by definition of A) $i \in A$.

The formula for A, and the idea behind it, are a little cryptic and can seem baffling at first. An illustration may be more helpful. Suppose that the first few subsets in the list A_0, $A_1, \ldots$ are described below.

$A_0 = \{0, 2, 5, 9, \ldots\}$
$A_1 = \{1, 2, 3, 8, 12, \ldots\}$
$A_2 = \{0, 3, 6\}$
$A_3 = \emptyset$
$A_4 = \{4\}$
$A_5 = \{2, 3, 5, 7, 11, \ldots\}$
$A_6 = \{8, 16, 24, \ldots\}$
$A_7 = \mathcal{N}$
$A_8 = \{1, 3, 5, 7, 9, \ldots\}$
$A_9 = \{n \in \mathcal{N} \mid n > 12\}$
$\ldots$

In some cases the set A_i is not completely defined. (For example, there is nothing in the formula for A_0 to suggest what the smallest element larger than 9 is.) However, we can find the first few elements of A, because for each number from 0 through 9, we have enough information to determine whether it satisfies the condition for A.

$0 \in A_0$, and so $0 \notin A$.
$1 \in A_1$, and so $1 \notin A$.
$2 \notin A_2$, and so $2 \in A$.
$3 \notin A_3$, and so $3 \in A$.
$4 \in A_4$, and so $4 \notin A$.
$5 \in A_5$, and so $5 \notin A$.
$6 \notin A_6$, and so $6 \in A$.
$7 \in A_7$, and so $7 \notin A$.
$8 \notin A_8$, and so $8 \in A$.
$9 \notin A_9$, and so $9 \in A$.

So far, we have found five elements of A: 2, 3, 6, 8, and 9. It's very likely that there are subsets in the list whose first five elements are 2, 3, 6, 8, and 9. This might be true of A_{918}, for example. How can we be sure that A is different from A_{918}? Because the construction guarantees that A contains the number 918 precisely if A_{918} does not.

In order to understand what makes this a *diagonal* argument, let us consider a different way of visualizing a subset of $\mathcal{N}$. A subset $S \subseteq \mathcal{N}$ can be described by saying, for each $i \in \mathcal{N}$, whether $i \in S$. If we use 1 to denote membership and 0 nonmembership, we can think of S as an infinite sequence of 0's and 1's, where the ith entry in the sequence is 1 if $i \in S$ and 0 if $i \notin S$. This is like a Boolean array you might use to represent a subset in a computer program, except that here the subset might be infinite.

Using this representation, we can visualize the sequence A_0, A_1, ... in our example above as follows:

A_0:	$\underline{1}$	0	1	0	0	1	0	0	0	1	...
A_1:	0	$\underline{1}$	1	1	0	0	0	0	1	0	...
A_2:	1	0	$\underline{0}$	1	0	0	1	0	0	0	...
A_3:	0	0	0	$\underline{0}$	0	0	0	0	0	0	...
A_4:	0	0	0	0	$\underline{1}$	0	0	0	0	0	...
A_5:	0	0	1	1	0	$\underline{1}$	0	1	0	0	...
A_6:	0	0	0	0	0	0	$\underline{0}$	0	1	0	...
A_7:	1	1	1	1	1	1	1	$\underline{1}$	1	1	...
A_8:	0	1	0	1	0	1	0	1	$\underline{0}$	1	...
A_9:	0	0	0	0	0	0	0	0	0	$\underline{0}$	...
...											

In the first row, the 0th, 2nd, 5th, and 9th terms of the sequence are 1, because A_0 contains 0, 2, 5, and 9. The underlined terms in this two-dimensional array are the *diagonal* terms, and we obtain the sequence corresponding to the set A by reversing these: 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, ... Each entry we encounter as we make our way down the diagonal allows us to distinguish the set A from one more of the sets A_i. As before, $A = \{2, 3, 6, 8, 9, \ldots\}$.

In the first paragraph of this section, we promised to show that "there must be *many* languages over {0, 1} that cannot be accepted by TMs." This is stated more precisely in Theorem 8.32.

Theorem 8.32
Not all languages are recursively enumerable. In fact, the set of languages over $\{0, 1\}$ that are not recursively enumerable is uncountable.

Proof
In Example 8.31 we showed that the set of subsets of $\mathcal{N}$ is uncountable, and we observed that because $\{0, 1\}^*$ is the same size as $\mathcal{N}$, it follows that the set of languages over $\{0, 1\}$ is uncountable. According to Example 8.30, the set of recursively enumerable languages over $\{0, 1\}$ is countable. The theorem follows from the fact that if T is any countable subset of an uncountable set S, $S - T$ is uncountable (see Exercise 8.38).

Once we have an example of an uncountable set, there are simple ways of getting other examples. If A is uncountable and $A \subseteq B$, so that B is either the same size as A or larger than A, then B is uncountable. Theorem 8.32 uses the fact that if A is uncountable and C is countable, then $A - C$ is uncountable. Finally, if A is uncountable and there is a bijection from A to D, then D is uncountable. However, a diagonal argument or something comparable is the only method known for proving that a set is uncountable without having another set that is already known to be. The exercises contain a few more illustrations of diagonal arguments, and we have already said that we will use them again in the next chapter.

Finally, one possible source of confusion in this section is worth calling to your attention. We say informally that a set is uncountable if its elements "cannot be listed." Now from Theorem 8.32 we know there is a language $L \subseteq \{0, 1\}^*$ that, according to Section 8.2, cannot be enumerated (or listed) by a TM. It is important to understand that these are two very different ideas. The reason the elements of an uncountable set cannot be listed has nothing to do with what they are—there are just too many of them. The reason a language L fails to be recursively enumerable has nothing to do with its size: There is an easy way to list *all* the elements of $\{0, 1\}^*$, those of L and all the others. We can say that in some sense "there exists" a list of the elements of L (at least the existence of such a list does not imply any inherent contradiction), but there is no algorithm to tell us what strings are in it.

EXERCISES

8.1. Show that if L_1 and L_2 are recursive languages, then $L_1 \cup L_2$ and $L_1 \cap L_2$ are also.

8.2. Consider modifying the proof of Theorem 8.4 by executing the two TMs sequentially instead of simultaneously. Given TMs T_1 and T_2 accepting L_1 and L_2, respectively, and an input string x, we start by making a second copy of x. We execute T_1 on the second copy; if and when this computation stops, the tape is erased except for the original input, and T_2 is executed on it.

a. Is this approach feasible for accepting $L_1 \cup L_2$, thereby showing that the union of recursively enumerable languages is recursively enumerable? Why or why not?

b. Is this approach feasible for accepting $L_1 \cap L_2$, thereby showing that the intersection of recursively enumerable languages is recursively enumerable? Why or why not?

8.3. Is the following statement true or false? If $L_1, L_2, \ldots$ are any recursively enumerable subsets of Σ^*, then $\bigcup_{i=1}^{\infty} L_i$ is recursively enumerable. Give reasons for your answer.

8.4. Suppose $L_1, L_2, \ldots, L_k$ form a partition of Σ^*; in other words, their union is Σ^* and any two of them are disjoint. Show that if each L_i is recursively enumerable, then each L_i is recursive.

8.5. Suppose that in Definition 8.8 we require that a TM enumerating a finite language L eventually halt after printing the # following the last element of L. This makes it a little simpler to prove the last of the four statements listed in the proof of Theorem 8.9. Show how to resolve any complications introduced in the proofs of the other three.

8.6. Describe algorithms to enumerate these sets. (You do not need to discuss the mechanics of constructing Turing machines to execute the algorithms.)

a. The set of all pairs (n, m) for which n and m are relatively prime positive integers ("relatively prime" means having no common factor bigger than 1)

b. The set of all strings over $\{0, 1\}$ that contain a nonnull substring of the form www

c. $\{n \in \mathcal{N} \mid$ for some positive integers x, y, and z, $x^n + y^n = z^n\}$ (Answer this question without using Fermat's theorem, which says that the only elements of the set are 1 and 2.)

8.7. In Definition 8.8, the strings x_i appearing on the output tape of T are required to be distinct. Show that if L can be enumerated in the weaker sense, in which this requirement is dropped, then L is recursively enumerable.

8.8. Suppose L is recursively enumerable but not recursive. Show that if T is a TM accepting L, there must be infinitely many input strings for which T loops forever.

8.9. Suppose $L \subseteq \Sigma^*$. Show that L is recursively enumerable if and only if there is a computable partial function from Σ^* to Σ^* whose *range* is L.

8.10. Suppose $L \subseteq \Sigma^*$. Show that L is recursively enumerable if and only if there is a computable partial function from Σ^* to Σ^* whose *domain* is L.

8.11. Let $f : \{0, 1\}^* \to \{0, 1\}^*$ be a partial function. Let $g(f)$, the *graph* of f, be the language $\{x\#f(x) \mid x \in \{0, 1\}^*\}$. Show that f can be computed by a Turing machine if and only if the language $g(f)$ is recursively enumerable.

8.12. Show that a set $L \subseteq \Sigma^*$ is recursive if and only if there is an increasing computable total function $f : \Sigma^* \to \Sigma^*$ whose range is L. ("Increasing"

means that if x precedes y with respect to canonical order, then $f(x)$ precedes $f(y)$.)

8.13. Show that if $L \subseteq \Sigma^*$ and L is infinite and recursively enumerable, then L has an infinite recursive subset.

8.14. †Show that if $L \subseteq \Sigma^*$ and L is infinite and recursive, then L has an infinite subset that is not recursively enumerable and an infinite subset that is recursively enumerable but not recursive.

8.15. Show that if $L \subseteq \Sigma^*$ and L is infinite and recursively enumerable, then L has an infinite subset that is not recursively enumerable and an infinite subset that is recursively enumerable but not recursive.

8.16. Canonical order is a specific way of ordering the strings in Σ^*, and its use in Theorem 8.9 is somewhat arbitrary. By an ordering of Σ^*, we mean simply a bijection from the set of natural numbers to Σ^*. For any such bijection f, and any language $L \subseteq \Sigma^*$, let us say that "L can be enumerated in order f" means that there is a TM T enumerating L and for every i, the string $f(i)$ is the ith string listed by T. For an arbitrary ordering f of Σ^*, let $E(f)$ be the statement "For any $L \subseteq \Sigma^*$, L is recursive if and only if it can be enumerated in order f." For exactly which types of orderings f is $E(f)$ true? Prove your answer.

8.17. In each case below, describe the language generated by the unrestricted grammar with the given productions. The symbols a, b, and c are terminals, and all other symbols are variables.

a.
$$S \to ABCS \mid ABC$$
$$AB \to BA \qquad AC \to CA \qquad BC \to CB$$
$$BA \to AB \qquad CA \to AC \qquad CB \to BC$$
$$A \to a \qquad B \to b \qquad C \to c$$

b. $S \to LaR \qquad L \to LD \mid LT \mid \Lambda \qquad Da \to aaD \qquad Ta \to aaaT$
$$DR \to R \qquad TR \to R \qquad R \to \Lambda$$

c.
$$S \to LaMR \qquad L \to LT \mid E$$
$$Ta \to aT \qquad TM \to aaMT \qquad TR \to aMR$$
$$Ea \to aE \qquad EM \to E \qquad ER \to \Lambda$$

8.18. Consider the unrestricted grammar with the following productions.
$$S \to TD_1D_2 \qquad T \to ABCT \mid \Lambda$$
$$AB \to BA \qquad BA \to AB \qquad CA \to AC \qquad CB \to BC$$
$$CD_1 \to D_1C \qquad CD_2 \to D_2a \qquad BD_1 \to D_1b$$
$$A \to a \qquad D_1 \to \Lambda \qquad D_2 \to \Lambda$$

a. Describe the language generated by this grammar.

b. Find a single production that could be substituted for $BD_1 \to D_1b$ so that the resulting language would be
$$\{xa^n \mid n \geq 0,\ |x| = 2n,\ \text{and } n_a(x) = n_b(x) = n\}$$

8.19. For each of the following languages, find an unrestricted grammar that generates the language.

a. $\{a^n b^n a^n b^n \mid n \geq 0\}$

b. $\{a^n x b^n \mid n \geq 0, x \in \{a, b\}^*, |x| = n\}$

c. $\{sss \mid s \in \{a, b\}^*\}$

d. $\{ss^r s \mid s \in \{a, b\}^*\}$

8.20. For each of the following languages, find an unrestricted grammar that generates the language.

a. $\{x \in \{a, b, c\}^* \mid n_a(x) < n_b(x) \text{ and } n_a(x) < n_c(x)\}$

b. $^\dagger\{x \in \{a, b, c\}^* \mid n_a(x) < n_b(x) < 2n_c(x)\}$

c. $\{a^n \mid n = j(j+1)/2 \text{ for some } j \geq 1\}$ (Suggestion: if a string has j groups of a's, the ith group containing i a's, then you can create $j+1$ groups by adding an a to each of the j groups and adding a single extra a at the beginning.)

8.21. Suppose G is an unrestricted grammar with start symbol T that generates the language $L \subseteq \{a, b\}^*$. In each part below, another unrestricted grammar is described. Say (in terms of L) what language it generates.

a. The grammar containing all the variables and all the productions of G, two additional variables S (the start variable) and E, and the additional productions

$$S \to ET \qquad E \to \Lambda \qquad Ea \to E \qquad Eb \to E$$

b. The grammar containing all the variables and all the productions of G, four additional variables S (the start variable), F, R, and E, and the additional productions

$$S \to FTR \qquad Fa \to aF \qquad Fb \to bF \qquad F \to E$$
$$Ea \to E \qquad Eb \to E \qquad ER \to \Lambda$$

8.22. Figure 7.6 shows the transition diagram for a TM accepting $XX = \{xx \mid x \in \{a, b\}^*\}$. In the grammar obtained from this TM as in the proof of Theorem 8.14, give a derivation for the string $abab$.

8.23. Find a context-sensitive grammar generating the language in Example 8.11.

8.24. Find a context-sensitive grammar generating the language $XX - \{\Lambda\} = \{xx \mid x \in \{a, b\}^* \text{ and } x \neq \Lambda\}$.

8.25. Example 8.17 provides a context-sensitive grammar that generates $\{a^n b^n c^n \mid n > 0\}$. Find another CSG generating this language that has only three variables and six productions. Suggestion: omit the variables A and $\mathcal{A}$.

8.26. Find CSGs equivalent to the grammars in Exercise 8.17, parts (b) and (c).

8.27. Show that if L is any recursively enumerable language, then L can be generated by a grammar in which the left side of every production is a string of one or more variables.

8.28. Show by examples that the constructions in the proof of Theorem 4.9 do not work to show that the class of recursively enumerable languages is closed under concatenation and Kleene *, or that the class of CSLs is closed under concatenation.

8.29. Show that if for some positive integer k, there is a nondeterministic TM accepting L such that for any input x, the tape head never moves past square $k|x|$, then $L - \{\Lambda\}$ is a context-sensitive language.

8.30. Show that if G is an unrestricted grammar generating L, and there is an integer k such that for any $x \in L$, every string appearing in a derivation of x has length $\leq k|x|$, then L is recursive.

8.31. In the proof of Theorem 8.20, the CSG productions corresponding to an LBA move of the form $\delta(p, a) = (q, b, \mathrm{R})$ are given. Give the productions corresponding to the move $\delta(p, a) = (q, b, \mathrm{L})$ and those corresponding to the move $\delta(p, a) = (q, b, \mathrm{S})$.

8.32. †Suppose G is a context-sensitive grammar. In other words, for every production $\alpha \to \beta$ of G, $|\beta| \geq |\alpha|$. Show that there is a grammar G', with $L(G') = L(G)$, in which every production is of the form

$$\alpha A \beta \to \alpha X \beta$$

where A is a variable and α, β, and X are strings of variables and/or terminals, with X not null.

8.33. Use the LBA characterization of context-sensitive languages to show that the class of CSLs is closed under union, intersection, and concatenation, and that if L is a CSL, then so is LL^*.

8.34. †Suppose G_1 and G_2 are unrestricted grammars generating L_1 and L_2 respectively.

a. Using G_1 and G_2, describe an unrestricted grammar generating $L_1 L_2$.

b. Using G_1, describe an unrestricted grammar generating L_1^*.

8.35. Show that every subset of a countable set is countable.

8.36. By definition, a set S is finite if for some natural number n, there is a bijection from S to $\{i \in \mathcal{N} \mid 1 \leq i \leq n\}$. An infinite set is a set that is not finite. Show that a set S is infinite if and only if there is a bijection from S to some proper subset of S. (Theorem 8.25 might be helpful.)

8.37. Saying that a property is *preserved under bijection* means that if a set S has the property and $f : S \to T$ is a bijection, then T also has the property. Show that both countability and uncountability are preserved under bijection.

8.38. Show that if S is uncountable and T is countable, then $S - T$ is uncountable. Suggestion: proof by contradiction.

8.39. Let Q be the set of all rational numbers, or fractions, negative as well as nonnegative. Show that Q is countable by describing explicitly a bijection from $\mathcal{N}$ to Q—in other words, a way of creating a list of rational numbers that contains every rational number exactly once.

8.40. For each part below, use a diagonal argument to show that the given set is uncountable.

a. The set $\mathcal{S}$ of infinite sequences of 0's and 1's. This part is essentially done for you, in Example 8.31, but write it out carefully.

b. The set $I = \{x \in \mathcal{R} \mid 0 \leq x < 1\}$. Do this by using the one-to-one correspondence between elements of I and infinite decimal expansions $0.a_0a_1\ldots$ that do not end with an infinite sequence of consecutive 9's. Every element of I has exactly one such expansion, and every such expansion represents exactly one element of I. For example, 0.5 could be written using the infinite decimal expansion $0.5000\ldots$ or the expansion $0.49999\ldots$, but only the first is relevant in this problem, because the second ends with an infinite sequence of 9's. In your proof, you just have to make sure that the decimal expansion you construct, in order to get a number not in the given list, doesn't end with an infinite sequence of 9's.

8.41. For each case below, determine whether the given set is countable or uncountable. Prove your answer.

a. The set of all three-element subsets of $\mathcal{N}$

b. The set of all finite subsets of $\mathcal{N}$

c. The set of all partitions of $\mathcal{N}$ into a finite number of subsets A_1, A_2, $\ldots$, A_k (such that any two are disjoint and $\bigcup_{i=1}^{k} A_i = \mathcal{N}$). The order of the A_i's is irrelevant—i.e., two partitions $A_1, \ldots, A_k$ and $B_1, \ldots, B_k$ are considered identical if each A_i is B_j for some j.

d. The set of all functions from $\mathcal{N}$ to $\{0, 1\}$

e. The set of all functions from $\{0, 1\}$ to $\mathcal{N}$

f. The set of all functions from $\mathcal{N}$ to $\mathcal{N}$

g. The set of all nondecreasing functions from $\mathcal{N}$ to $\mathcal{N}$

h. The set of all regular languages over $\{0, 1\}$

i. The set of all context-free languages over $\{0, 1\}$

8.42. We know that $2^{\mathcal{N}}$ is uncountable. Give an example of a set $S \subseteq 2^{\mathcal{N}}$ such that both S and $2^{\mathcal{N}} - S$ are uncountable.

8.43. Show that the set of languages L over $\{0, 1\}$ such that neither L nor L' is recursively enumerable is uncountable.

8.44. This exercise is taken from Dowling (1989). It has to do with an actual computer, which is assumed to use some fixed operating system under which all its programs run. A "program" can be thought of as a function that takes one string as input and produces another string as output. On the other hand, a program written in a specific language can be thought of as a string itself.

By definition, a program P spreads a *virus* on input x if running P with input x causes the operating system to be altered. It is safe on input x if this doesn't happen, and it is safe if it is safe on every input string. A *virus tester* is a program IsSafe that when given the input Px, where P is

a program and x is a string, produces the output "YES" if P is safe on input x and "NO" otherwise. (We make the assumption that in a string of the form Px, there is no ambiguity as to where the program P stops.) Prove that if there is the actual possibility of a virus—i.e., there is a program and an input that would cause the operating system to be altered—then there can be no virus tester that is both safe and correct. Hint: Suppose there is such a virus tester IsSafe. Then it is possible to write a program D (for "diagonal") that operates as follows when given a program P as input. It evaluates IsSafe(PP); if the result is "NO," it prints "XXX", and otherwise it alters the operating system. Now consider what D does on input D.

8.45. The two parts of this exercise show that for every set S (not necessarily countable), 2^S is larger than S.

a. For every S, describe a simple bijection from S to a subset of 2^S.

b. Show that for every S, there is no bijection from S to 2^S. (You can copy the proof in Example 8.31, as long as you avoid trying to list the elements of S or making any reference to the countability of S.)

8.46. †For each part below, use a diagonal argument to show that the given set is uncountable.

a. The set I in Exercise 8.40b, but this time using the one-to-one correspondence between elements of I and infinite binary expansions $0.a_0a_1\ldots$ (where each a_i is 0 or 1) that do not end in an infinite sequence of 1's. This is slightly harder than Exercise 8.40b, because if $x_0, x_1, \ldots$ is a list of elements of I, and x_i corresponds to the infinite binary expansion $0.a_{i,0}a_{i,1}\ldots$, it may turn out that the binary expansion $0.a_0a_1\ldots$ constructed using the diagonal argument *does* end with an infinite sequence of 1's. You can get around this problem by constructing $a = 0.a_0a_1\ldots$ that is different from each x_i because $a_{2i+1} \neq a_{i,2i+1}$.

b. The set of bijections from $\mathcal{N}$ to $\mathcal{N}$. If $f_0, f_1, \ldots$ is a list of bijections, you can construct a bijection f that doesn't appear in the list, by making $f(2n)$ different from $f_n(2n)$ and specifying both $f(2n)$ and $f(2n+1)$ so that f is guaranteed to be a bijection.

8.47. In each case below, determine whether the given set is countable or uncountable. Prove your answer.

a. The set of all real numbers that are roots of integer polynomials; in other words, the set of real numbers x such that, for some nonnegative integer n and some integers $a_0, a_1, \ldots, a_n$, x is a solution to the equation

$$a_0 + a_1x + a_2x^2 + \ldots a_nx^n = 0$$

b. The set of all nondecreasing functions from $\mathcal{N}$ to $\mathcal{N}$.

c. The set of all functions from $\mathcal{N}$ to $\mathcal{N}$ whose range is finite.

d. The set of all nondecreasing functions from $\mathcal{N}$ to $\mathcal{N}$ whose range is finite (i.e, all "step" functions).

e. The set of all periodic functions from $\mathcal{N}$ to $\mathcal{N}$. (A function $f : \mathcal{N} \to \mathcal{N}$ is periodic if, for some positive integer P_f, $f(x + P_f) = f(x)$ for every x.)

f. The set of all eventually constant functions from $\mathcal{N}$ to $\mathcal{N}$. (A function $f : \mathcal{N} \to \mathcal{N}$ is eventually constant if, for some C and for some N, $f(x) = C$ for every $x \geq N$.)

8.48. †We have said that a set A is larger than a set B if there is a bijection from B to a subset of A, but no bijection from B to A, and we have proceeded to use this terminology much as we use the $<$ relation on the set of numbers. What we have not done is to show that this relation satisfies the same essential properties that $<$ does.

a. The *Schröder-Bernstein theorem* asserts that if A and B are sets and there are bijections f from A to a subset of B and g from B to a subset of A, then there is a bijection from A to B. Prove this statement. Here is a suggested approach. An *ancestor* of $a \in A$ is a point $b \in B$ such that $g(b) = a$, or a point $a_1 \in A$ such that $g(f(a_1)) = a$, or a point $b_1 \in B$ such that $g(f(g(b_1))) = a$, etc. In other words, an ancestor of $a \in A$ is a point x in A or B such that by starting at x and continuing to evaluate the two functions alternately, we arrive at a. If $g(f(g(b))) = a$ and b has no ancestors, for example, we will say that a has the three ancestors $f(g(b))$, $g(b)$, and b. Note that we describe the number of ancestors as three, even in the case that $f(g(b)) = b$ or $g(b) = a$. In this way, A can be partitioned into three sets A_0, A_1, and A_∞; A_0 is the set of elements of A having an even (finite) number of ancestors, A_1 is the set of elements having an odd number, and A_∞ is the set of points having an infinite number of ancestors. Ancestors of elements of B are defined the same way, and we can partition B similarly into B_0, B_1, and B_∞. Show that there are bijections from A_0 to B_1, from A_1 to B_0, and from A_∞ to B_∞.

b. Show that the "larger than" relation on sets is transitive. In other words, if there is a bijection from A to a subset of B but none from A to B, and a bijection from B to a subset of C but none from B to C, then there is a bijection from A to a subset of C but none from A to C.

c. Show that the "larger-than" relation is asymmetric. In other words, if A is larger than B, then B cannot be larger than A.

d. Show that countable sets are the smallest infinite sets in both possible senses: Not only are uncountable sets larger than countable sets, but no infinite set can be smaller than a countable set.

8.49. Let I be the unit interval $[0, 1]$, the set of real numbers between 0 and 1. Let $S = I \times I$, the unit square. Use the Schröder-Bernstein theorem (see Exercise 8.48) to show that there is a bijection from I to S. One way is to use infinite decimal expansions as in Exercise 8.40b.

8.50. Show that A is bigger than B if and only if there is a one-to-one function from B to A but none from A to B. (One way is easy; for the other, Exercise 8.48 will be helpful.)

CHAPTER

9

Undecidable Problems

Using a diagonal argument that is logically almost identical to the one at the end of Chapter 8, we can produce an example of a language not accepted by any Turing machine. The way the language is defined also makes it easy to formulate a decision problem (a general question, each instance of which has a yes-or-no answer) that is *undecidable*—that is, for which no algorithm can produce the correct answer in every instance.

In general, we have a choice between considering whether a decision problem is decidable and considering whether the language corresponding to it, the one containing strings that represent "yes-instances" of the problem, is recursive. Using either approach, once we have one example of an undecidable problem P, we can obtain others by constructing *reductions* from P. In this chapter we describe several other undecidable problems involving Turing machines, including the well-known halting problem, as well as a general method for obtaining a large class of unsolvable problems. In the last two sections of this chapter, we consider another well-known undecidable problem, Post's correspondence problem. Constructing reductions from it is one of two ways we use to show that several simple-to-state questions about context-free grammars are also undecidable.

9.1 A LANGUAGE THAT CAN'T BE ACCEPTED, AND A PROBLEM THAT CAN'T BE DECIDED

We know from Chapter 8, Section 8.5, that there are uncountably many languages, too many for them all to be recursively enumerable. This conclusion by itself doesn't make it obvious that we will be able to identify and describe a non-recursively-enumerable language. The set of *descriptions*, which are strings over some alphabet Σ, is countable, the same size as the set of recursively enumerable languages; maybe whenever there is a precise finite description of a language L, there is an algorithm to accept L.

As it turns out, not only can we describe a language that is not recursively enumerable, but we can do it by using the same diagonal argument that we used for the uncountability result. We showed that for every list $A_0, A_1, \ldots$ of subsets of $\mathcal{N}$, there is a subset A of $\mathcal{N}$ that is not in the list. The fact that the sets A_i were in a *list* was not crucial; here are the crucial steps.

1. We started with a collection of subsets A_i of $\mathcal{N}$, each one associated with a specific *element* of $\mathcal{N}$ (namely, i).
2. We defined another subset A, containing the elements i of $\mathcal{N}$ that do not belong to the subset A_i associated with i.

The conclusion was that for each i, $A \neq A_i$, because $i \in A \Leftrightarrow i \notin A_i$.

Now we want to find a language $L \subseteq \{0, 1\}^*$ that cannot be accepted by a Turing machine—in other words, a language that is different from $L(T)$, for every Turing machine T with input alphabet $\{0, 1\}$. The string $e(T)$ introduced in Section 7.8 is "associated with" $L(T)$, just as i is associated with A_i, and we can repeat the argument as follows:

1. We have a collection of subsets $L(T)$ of $\{0, 1\}^*$, each one associated with a specific element of $\{0, 1\}^*$ (namely, $e(T)$).
2. We define another subset L, containing the elements $e(T)$ of $\{0, 1\}^*$ that do not belong to the subset $L(T)$ associated with $e(T)$.

The conclusion this time is that for each T, $L \neq L(T)$, because $e(T) \in L \Leftrightarrow e(T) \notin L(T)$.

Definition 9.1 The Languages *NSA* and *SA*

Let

$$NSA = \{e(T) \mid T \text{ is a TM, and } e(T) \notin L(T)\}$$
$$SA = \{e(T) \mid T \text{ is a TM, and } e(T) \in L(T)\}$$

(*NSA* and *SA* stand for "non-self-accepting" and "self-accepting.")

Theorem 9.2

The language *NSA* is not recursively enumerable. The language *SA* is recursively enumerable but not recursive.

Proof

The language *NSA* is the language L described above, and the conclusion we obtained is that it is not recursively enumerable.

Let E be the language $\{e(T) \mid T \text{ is a TM}\}$. It follows from Theorem 7.36 that E is recursive. (This is one of the crucial features we identified

in Section 7.8 for any encoding function e—there should be an algorithm to determine, for any string z, whether $z = e(T)$ for a TM T.)

For every string z of 0's and 1's, exactly one of these three statements is true: (i) z does not represent a TM; (ii) z represents a TM that accepts z; (iii) z represents a TM that does not accept z. In other words,

$$\{0, 1\}^* = NSA \cup SA \cup E'$$

and the three sets on the right are disjoint. Therefore,

$$NSA = (SA \cup E')' = SA' \cap E$$

If SA were recursive, then SA' would be recursive, by Theorem 8.6, and then NSA would be recursive, by Theorem 8.5. But NSA is not recursively enumerable, and so by Theorem 8.2 it can't be recursive.

To finish the proof, we need to show that SA is recursively enumerable. Here is an algorithm to accept SA. Given a string $x \in \{0, 1\}^*$, determine whether $x \in E$. If not, then $x \notin SA$. If $x \in E$, then $x = e(T)$ for some TM T, and as we observed in Section 7.8, we can reconstruct T from the string $e(T)$. Then execute T on the input x, and accept x precisely if T accepts x.

The statement of the theorem says that there is no algorithm to determine whether a given string represents a TM that accepts its own encoding. Because we *can* determine whether a string represents a TM, there is no algorithm to determine whether a given TM accepts its own encoding. There is one aspect of this result that deserves a little more discussion.

It may seem from the statement that when we are considering strings that a TM T might or might not accept, the string $e(T)$ presents particular problems—perhaps for most strings x we can easily decide whether T accepts x, but for some reason, deciding whether T accepts $e(T)$ is more challenging. However, you can see if you go back to the preliminary discussion that this is not the right interpretation. All we needed for the diagonal argument was a string "associated with" T. The string $e(T)$ was an obvious choice, but many other choices would have worked just as well. For example, we could list all TMs by arranging the strings $e(T)$ in canonical order, and we could list all elements of $\{0, 1\}^*$ in canonical order, and we could associate the ith string with the ith Turing machine.

The appropriate conclusion seems to be, not so much that it's hard to decide what a Turing machine T does with the string $e(T)$, but that it's hard to answer questions about Turing machines and the strings they accept. This is the way to understand the significance of Theorem 9.2. We can probably agree that for most Turing machines T, whether T happens to accept its own encoding is not very interesting, or at least not very important. The general question of whether an arbitrary T accepts $e(T)$ is interesting, not because the answer is, but because even though the question is easy to state, there is no algorithm to answer it; no algorithm

can provide the right answer for every T. In the next section we will be able to use this result to find other, more useful problems that are also "undecidable" in the same way. So, even though you may not feel that this particular question provides an example of the following principle, we will have better examples soon.

> There are precise, easy-to-formulate questions for which it would be useful to have algorithms to answer them, but for which there are no such algorithms.

We can approach other decision problems the same way we approached this one. For a decision problem, we use the term "yes-instances" to refer to instances of the problem for which the answers are yes, and "no-instances" for the others. The three languages *SA*, *NSA*, and E' contain, respectively, the strings that represent yes-instances, the strings that represent no-instances, and the strings that don't represent instances at all, of the decision problem

Self-Accepting: Given a TM T, does T accept the string $e(T)$?

In discussing a general decision problem P, we start the same way, by choosing an encoding function e that allows us to represent instances I by strings $e(I)$ over some alphabet Σ. We give the names $Y(P)$ and $N(P)$ to the sets of strings in Σ^* that represent yes-instances and no-instances, respectively, of P. If $E(P) = Y(P) \cup N(P)$, then just as in our example, we have the third subset $E(P)'$ of strings in Σ^* that don't represent instances of P.

The requirements for a function to encode instances of P are the same as those for a function to encode Turing machines. The function must be one-to-one, so that a string can't represent more than one instance; it must be possible to *decode* a string $e(I)$ and recover the instance I; and there must be an algorithm to decide whether a given string in Σ^* represents an instance of P (in other words, the language $E(P)$ should be recursive). We will refer to a function satisfying these conditions as a *reasonable* encoding of instances of P.

Solving, or deciding, a decision problem P means starting with an arbitrary instance I of P and deciding whether I is a yes-instance or a no-instance. The way that a Turing machine must decide P, however, is to start with a *string*, and decide whether the string represents a yes-instance of P, represents a no-instance, or does not represent an instance. It sounds at first as though T's job is harder: before it can attempt to decide whether an instance is a yes-instance, it must decide whether a string represents an instance at all. However, the extra work simply reflects the fact that a TM must work with an input string. As long as the encoding function is reasonable, there is an algorithm for deciding P if and only if there is a Turing machine that decides $Y(P)$. Furthermore, whether this statement is true does not depend on which encoding function we use to define $Y(P)$; if one reasonable encoding makes the statement true, then every one does.

Definition 9.3 Decidable Problems

If P is a decision problem, and e is a reasonable encoding of instances of P over the alphabet Σ, we say that P is *decidable* if $Y(P) = \{e(I) \mid I$ is a yes-instance of $P\}$ is a recursive language.

Theorem 9.4

The decision problem *Self-Accepting* is undecidable.

Proof

Because $SA = Y(\textit{Self-Accepting})$, the statement follows immediately from Theorem 9.2.

A decision problem has the general form

Given _____, is it true that _____?

For every decision problem P, there is a *complementary* problem P', obtained by changing "true" to "false" in the statement of the problem. For example, the complementary problem to *Self-Accepting* is

Non-Self-Accepting: Given a TM T, does T fail to accept $e(T)$?

For every P, yes-instances of P are no-instances of P', and vice-versa. P and P' are essentially just two ways of asking the same question, and the result in Theorem 9.5 is not surprising.

Theorem 9.5

For every decision problem P, P is decidable if and only if the complementary problem P' is decidable.

Proof

For exactly the same reasons as in the proof of Theorem 9.2, we have

$$N(P) = Y(P)' \cap E(P)$$

$E(P)$ is assumed to be recursive. If $Y(P)$ is recursive, then so is $Y(P)'$, and therefore so is $N(P) = Y(P')$. The other direction is similar.

Theorem 9.5 helps make it clear why $Y(P)$ is required to be recursive, not just recursively enumerable, if we are to say that P is decidable. In our example, although *SA* is recursively enumerable and *NSA* is not, *Self-Accepting* is no closer to being decidable than its complementary problem is. An algorithm attempting to solve either one would eventually hit an instance I that it couldn't handle; the only difference is that if it were trying to solve *Self-Accepting*, I would be a no-instance, and if it were trying to solve *Non-Self-Accepting*, I would be a yes-instance.

Finally, you should remember that even for an undecidable problem, there may be many instances for which answers are easy to obtain. If T is a TM that rejects every input on the first move, then T obviously does not accept the string $e(T)$; and there are equally trivial TMs that obviously do accept their own encodings. What makes *Self-Accepting* undecidable is that there is no algorithm guaranteed to produce the right answer for every instance.

9.2 REDUCTIONS AND THE HALTING PROBLEM

Very often in computer science, one problem can be solved by reducing it to another, simpler one. Recursive algorithms usually rely on the idea of a reduction in size: starting with an instance of size n and reducing it to a smaller instance of the same problem. For example, in the binary search algorithm, a single comparison reduces the number of elements to be searched from n to $n/2$.

In this section we consider reducing one decision problem to a different decision problem. Suppose we want to determine whether an NFA M accepts a string x. The nondeterminism in M means that we can't simply "run M on input x and see what happens." However, we have an algorithm (see the proofs of Theorems 3.17 and 3.18) that allows us to find an ordinary FA M_1 accepting the same strings as M. The algorithm allows us to reduce the original problem to the simpler problem of determining whether an FA accepts a given string.

Here's another example. Let P_1 be the problem: Given two FAs M_1 and M_2, is $L(M_1) \subseteq L(M_2)$? Once we realize that for two sets A and B, $A \subseteq B$ if and only if $A - B = \emptyset$, we can use the algorithm described in the proof of Theorem 2.15 to reduce P_1 to the problem P_2: Given an FA M, is $L(M) = \emptyset$? We use the algorithm to find an FA M such that $L(M) = L(M_1) - L(M_2)$, and the answer to the original question (is $L(M_1) \subseteq L(M_2)$?) is the same as the answer to the second question (is $L(M) = \emptyset$?).

In both these examples, we started with an instance I of the original problem, and we applied an algorithm to obtain an instance $F(I)$ of the second problem having the same answer. In the first example, I is a pair (M, x), where M is an NFA and x is a string; $F(I)$ is a pair (M_1, y), where M_1 is an FA and y is a string. M_1 is chosen to be an FA accepting the same language as M, and y is chosen simply to be x. In the second example, I is a pair (M_1, M_2) of FAs and $F(I)$ is the single FA M constructed such that $L(M) = L(M_1) - L(M_2)$.

The two crucial features in a reduction are: First, for every instance I of the first problem that we start with, we must be able to obtain the instance $F(I)$ of the second problem algorithmically; and second, the answer to the second question for the instance $F(I)$ must be the same as the answer to the original question for I.

If the two problems P_1 and P_2 happen to be the membership problems for languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, respectively, then an instance of P_1 is a string $x \in \Sigma_1^*$, and an instance of P_2 is a string $y \in \Sigma_2^*$. In this case, a reduction from P_1 to P_2 means an algorithm to compute for each $x \in \Sigma_1^*$ a string $F(x) \in \Sigma_2^*$ such that for every x, $x \in L_1$ if and only if $F(x) \in L_2$. Here, because instances are

strings, "algorithm" literally means "Turing machine"—the function F should be computable. We refer to F as a reduction from L_1 to L_2; it is sometimes called a *many-one* reduction.

Definition 9.6 Reducing One Decision Problem to Another, and Reducing One Language to Another

Suppose P_1 and P_2 are decision problems. We say P_1 is reducible to P_2 ($P_1 \leq P_2$) if there is an algorithm that finds, for an arbitrary instance I of P_1, an instance $F(I)$ of P_2, such that for every I, the answers for the two instances are the same, or I is a yes-instance of P_1 if and only if $F(I)$ is a yes-instance of P_2.

If L_1 and L_2 are languages over alphabets Σ_1 and Σ_2, respectively, we say L_1 is reducible to L_2 ($L_1 \leq L_2$) if there is a Turing-computable function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ such that for every $x \in \Sigma_1^*$,

$$x \in L_1 \text{ if and only if } f(x) \in L_2$$

Theorem 9.7

Suppose $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$, and $L_1 \leq L_2$. If L_2 is recursive, then L_1 is recursive.

Suppose P_1 and P_2 are decision problems, and $P_1 \leq P_2$. If P_2 is decidable, then P_1 is decidable.

Proof

To prove the first statement, let $f : \Sigma_1^* \rightarrow \Sigma_2^*$ be a function that reduces L_1 to L_2, and let T_f be a TM computing f. Let T_2 be another TM that decides L_2. Let T be the composite TM $T_f T_2$. On input x, T first computes $f(x)$ and then decides, by returning output 1 or 0, whether $f(x) \in L_2$. Because f is a reduction, this is the same as deciding whether $x \in L_1$. Therefore, T decides the language L_1.

Suppose now that F is an algorithm reducing P_1 to P_2, and that we have encoding functions e_1 and e_2 for the instances of P_1 and P_2, respectively. Our assumption is that $Y(P_2)$, the subset of Σ_2^* containing the strings that represent yes-instances of P_2, is recursive, and we wish to show that $Y(P_1)$ is also recursive.

For an arbitrary string $x \in \Sigma_1^*$, we can decide, assuming that e_1 is a reasonable encoding, whether x represents an instance of P_1. If not, then $x \notin Y(P_1)$. If $x \in E(P_1)$, then x represents an instance I of P_1, and we have this sequence of equivalences:

$$\begin{aligned} x = e_1(I) \in Y(P_1) &\Leftrightarrow I \text{ is a yes-instance of } P_1 \\ &\Leftrightarrow F(I) \text{ is a yes-instance of } P_2 \\ &\Leftrightarrow e_2(F(I)) \in Y(P_2) \end{aligned}$$

The statement in the second line is true because F is a reduction from P_1 to P_2. Because $Y(P_2)$ is recursive, we have an algorithm to decide whether $e_2(F(I)) \in Y(P_2)$, and so we have an algorithm to decide whether $x \in Y(P_1)$.

In discussing the decidability of a problem P, we are identifying P with the language $Y(P)$; deciding whether an instance I of P is a yes-instance is essentially the same problem as deciding whether a string x represents a yes-instance of P. It would seem reasonable that the statement $P_1 \leq P_2$ should be equivalent to the statement $Y(P_1) \leq Y(P_2)$. There are slight technical complications, because the first statement involves only instances of the two problems, whereas the second involves *all* strings over the appropriate alphabets, whether or not they represent instances of P_1 or P_2. However, with minor qualifications, this equivalence does hold (see Exercises 9.16–9.18). When we discuss reductions, we will usually use the slightly more informal definition involving "algorithms" unless instances of the decision problems are actually strings.

The most obvious reason for discussing a reduction from P_1 to P_2 is that if we have one, and if we can decide P_2, then we have a way of deciding P_1. However, it is important to separate the idea of reducing P_1 to P_2 from the question of whether either problem can be decided. The statement "if P_2 is decidable, then P_1 is decidable" is equivalent to the statement "if P_1 is not decidable, then P_2 is not decidable." In this chapter, we will use Theorem 9.7 primarily to obtain more examples of *un*decidable problems, or *non*recursive languages.

Let us consider two more decision problems: the general membership problem for recursively enumerable languages, and a problem usually referred to as the *halting problem*.

Accepts: Given a TM T and a string w, is $w \in L(T)$?

Halts: Given a TM T and a string w, does T halt on input w?

In both cases, instances of the problem are ordered pairs (T, w), where T is a TM and w is a string. Remember why these problems don't both have trivial decision algorithms: We can't just say "Execute T on input w and see what happens," because T might loop forever on input w.

Theorem 9.8

Both *Accepts* and *Halts* are undecidable.

Proof

By Theorems 9.4 and 9.7, in order to show *Accepts* is undecidable it is sufficient to show

$$\textit{Self-Accepting} \leq \textit{Accepts}$$

An instance of *Self-Accepting* is a TM T. A reduction to *Accepts* means finding a pair $F(T) = (T_1, y)$ such that T accepts $e(T)$ if and only if

T_1 accepts y. Letting $T_1 = T$ and $y = e(T)$ gives us such a pair, and $F(T) = (T, e(T))$ can be obtained algorithmically from T; therefore, F is a reduction from *Self-Accepting* to *Accepts*.

In order to prove that *Halts* is undecidable, we will show that

$$Accepts \leq Halts$$

Suppose we start with an arbitrary instance (T, x) of *Accepts* where T is a TM and x is a string. We must describe the pair $F(T, x) = (T_1, y)$, an instance of *Halts* such that the two answers are the same: T accepts x if and only if T_1 halts on input y.

How can we do it? It seems clear that T_1 should somehow be defined in terms of T (and maybe in terms of x as well?), and y should be defined in terms of x (and maybe somehow depend on T as well?). We cannot simply say "let T_1 be T and let $y = x$," because this choice doesn't force the two answers to be the same. (Obviously, if T accepts x, then T halts on x, but T could also halt on x by rejecting x.)

Let us choose y to be x, since there don't seem to be any natural alternatives. Now what we want is a TM T_1 such that for every x, T accepts x if and only if T_1 halts on x. A reformulation of this statement is: (i) if T accepts x, then T_1 halts on x, and (ii) if T doesn't accept x, then T_1 doesn't halt on x. Saying it this way may make it easier to see what to do. Statement (i) will be true if T_1 makes the same moves as T for any input that is accepted. Statement (ii) will also be true for every string on which T never halts, if T_1 makes the same moves as T on every such string. The only strings for which T_1 needs to do something different from T are the strings that cause T to reject; for these strings x, we want T_1 not to halt on x. Therefore, we can construct T_1 so that it makes the same moves as T, except that every move of T to the reject state is replaced by a sequence of moves causing T_1 to loop forever. Replacing every move of the form $\delta(p, a) = (h_r, b, D)$ by the move $\delta(p, a) = (p, a, \mathrm{S})$ almost accomplishes this, because if T_1 ever arrives in the state p with the current symbol a, then instead of rejecting, it will continue making moves that leave the state, symbol, and tape head position unchanged.

The other way T might reject, however, is to try to move its tape head left, starting in square 0. This case requires one additional modification to T. T_1 will begin by inserting a new symbol \$ in square 0, moving everything else one square to the right, and then move to q_0 with the tape head in square 1. There will now be additional moves $\delta(q, \$) = (q, \$, \mathrm{S})$ for every state q, such that if T ever tries to move its tape head off the tape, T_1 enters an infinite loop with its tape head on square 0.

If T_1 is the TM T with these modifications incorporated, then T_1 will accept every string T does, and T_1 will loop forever on every string T does not accept. Therefore, the function F defined by $F(T, x) = (T_1, x)$ is a reduction of *Accepts* to *Halts*.

Often you can test for infinite loops in a computer program by just verifying that you haven't made careless errors, such as initializing a variable incorrectly or failing to increment a variable within a loop. In other cases, however, a simple test is too much to expect. Here is an algorithm that would be easy to implement on a Turing machine:

```
n = 4
while (n is the sum of two primes)
  n = n+2
```

If you could decide whether this program continues forever, and prove your answer, you would be famous! The statement that every even integer greater than 2 is the sum of two primes is known as Goldbach's conjecture, named after a mathematician who made a similar conjecture in 1742; mathematicians have been interested in it ever since, but in spite of a $1,000,000 prize offered in 2000 to anyone who could prove or disprove it by 2002, no one has determined whether it is true or false. It is clear, however, that it is true precisely if the program above loops forever.

The halting problem is a famous undecidable problem. As the above three-line program might suggest, it is also a good example of a problem for which a decision algorithm, though impossible, would be useful.

9.3 MORE DECISION PROBLEMS INVOLVING TURING MACHINES

Of the three decision problems we have shown are undecidable, two, *Accepts* and *Halts* have instances that are pairs (T, x), where T is a TM and x is a string. The special case of *Accepts* in which an algorithm is still required to consider an arbitrary TM but can restrict its attention to the string $e(T)$, is also undecidable; this is *Self-Accepting*. If we restrict the problem in the opposite direction, by fixing the TM and only requiring an algorithm to consider arbitrary strings, there are TMs for which the resulting problem is undecidable: for example, if T_u is a universal TM, the decision problem

$$\text{Given } x, \text{ does } T_u \text{ accept } x?$$

is undecidable, because if we could answer the question for every string of the form $e(T)e(z)$, we would have a decision algorithm for *Accepts*.

In this section we extend the list of undecidable problems involving TMs. We start in Theorem 9.9 with a list of five problems, for each of which we will construct an appropriate reduction to show that the problem is undecidable.

Theorem 9.9

The following five decision problems are undecidable.

1. *Accepts*-Λ: Given a TM T, is $\Lambda \in L(T)$?
2. *AcceptsEverything*: Given a TM T with input alphabet Σ, is $L(T) = \Sigma^*$?
3. *Subset*: Given two TMs T_1 and T_2, is $L(T_1) \subseteq L(T_2)$?

4. *Equivalent*: Given two TMs T_1 and T_2, is $L(T_1) = L(T_2)$?
5. *WritesSymbol*: Given a TM T and a symbol a in the tape alphabet of T, does T ever write an a if it starts with an empty tape?

Proof

In each case, we will construct a reduction to the given problem from another decision problem that we already know is undecidable.

1. *Accepts* $\leq$ *Accepts*-Λ

An instance of *Accepts* is a pair (T, x) containing a TM and a string. We must show how to construct for every pair like this a TM $F(T, x) = T_1$ (an instance of *Accepts*-Λ) such that for every pair (T, x), T accepts x if and only if T_1 accepts Λ.

We have to construct T_1 such that even though we don't know what the outcome will be when T processes x, the outcome when T_1 processes Λ will be the same—or at least if the first outcome is acceptance, then the second will be, and if the first is not, then the second is not. This seems very confusing if you're thinking about what a TM might "normally" do on input Λ, because it probably seems to have nothing to do with x! But keep in mind that as long as Λ is the *original* input, then no matter what other string T_1 writes on the tape before starting a more "normal" computation, it's Λ that will be accepted or not accepted. If we want T_1 to perform a computation on Λ that has something to do with the computation of T on x, then we can simply make the computations identical, except that first we have to put x on the tape before the rest of the computation proceeds.

That's the key to the reduction. We let $T_1 = \mathit{Write}(x)T$ the composite TM that first writes x on the tape, beginning in square 1, and then executes T. If T accepts x, then when T is called in the computation of T_1, it will process x as if it were the original input, and Λ will be accepted; otherwise the second phase of T_1's computation will not end in acceptance, and Λ will not be accepted.

It may not be clear what T_1 is supposed to do if the original input is not Λ. It doesn't matter, because whether or not the algorithm that comes up with T_1 is a correct reduction depends only on what T_1 does with the input Λ.

2. *Accepts*-Λ $\leq$ *AcceptsEverything*

Both problems have instances that are TMs. Starting with an arbitrary TM T, we must find another TM $F(T) = T_1$ such that if T accepts Λ then T_1 accepts every string over its input alphabet, and if T doesn't accept Λ then there is at least one string not accepted by T_1.

It may be a little easier after the first reduction to see what to do here. This time, if T_1 starts with an input string x, it's x that is accepted if T_1 eventually reaches the accepting state, even if the computation resembles

the one that T performs with input Λ. We can define T_1 to be the composite Turing machine *Erase* T where *Erase* erases the input string and leaves the tape head on square 0. For every input, whether T_1 reaches h_a depends only on whether T accepts Λ. If T accepts Λ, T_1 accepts everything, and otherwise T_1 accepts nothing.

3. *AcceptsEverything* $\leq$ *Subset*

For an arbitrary instance T of *AcceptsEverythingy*, with input alphabet Σ, we must construct a pair of TMs (T_1, T_2) such that $L(T) = \Sigma^*$ if and only if $L(T_1) \subseteq L(T_2)$.

We start by specifying in advance that T_1 and T_2 will both have input alphabet Σ too, so that all the sets in the statements $L(T) = \Sigma^*$ and $L(T_1) \subseteq L(T_2)$ are subsets of Σ^*. At this point, rather than starting to think about what T_1 and T_2 should do, it's appropriate to ask what subsets of Σ^* we would like $L(T_1)$ and $L(T_2)$ to be, in order for the statement

$$L(T) = \Sigma^* \Leftrightarrow L(T_1) \subseteq L(T_2)$$

to be true. There's an easy answer: Saying that $L(T) = \Sigma^*$ is the same as saying that the subset $L(T)$ of Σ^* is big enough to contain all of Σ^*. In other words,

$$L(T) = \Sigma^* \Leftrightarrow \Sigma^* \subseteq L(T)$$

Therefore, we can choose T_1 such that $L(T_1) = \Sigma^*$, and we can let $T_2 = T$. Any choice of T_1 for which $L(T_1) = \Sigma^*$ will work; the easiest choice is to make T_1 enter the state h_a on its first move without looking at the input.

4. *Subset* $\leq$ *Equivalent*

For this reduction, we must start with two arbitrary TMs T_1 and T_2, and construct two other TMs T_3 and T_4 such that

$$L(T_1) \subseteq L(T_2) \Leftrightarrow L(T_3) = L(T_4)$$

Just as in the previous reduction, we can figure out what T_3 and T_4 should be by just thinking about the languages we want them to accept. If A and B are two sets, how can we express the statement $A \subseteq B$ by another statement that says two sets are equal? One answer is

$$A \subseteq B \Leftrightarrow A \cap B = A$$

This will work. We can choose T_3 to be a TM accepting $L(T_1) \cap L(T_2)$ (see Theorem 8.4), and we can choose T_4 to be T_1.

5. *Accepts-*Λ $\leq$ *WritesSymbol*

An instance of *Accepts-*Λ is a TM T, and an instance of *WritesSymbol* is a pair (T_1, σ), where T_1 is a TM and σ is a tape symbol of T_1. For an arbitrary T, we want a pair (T_1, σ) such that T accepts Λ if and only if T_1 eventually writes the symbol σ, if it starts with an empty tape.

In both cases (the TM T we start with, and the TM T_1 we are trying to find), we are interested only in what the TM does when it starts with a blank tape. We can make T_1 do exactly what T does, except that if and when T accepts, T_1 writes the symbol σ, and T_1 doesn't do that unless T accepts.

With that in mind, we can define (T_1, σ) as follows. T_1 will be similar to T, with just two differences: it will have an additional tape symbol σ, not in the tape alphabet of T, and any move of T that looks like $\delta(p, \sigma_1) = (h_a, \sigma_2, D)$ will be replaced by the move $\delta(p, \sigma_1) = (h_a, \sigma, D)$. Then T_1 has the property that it accepts Λ precisely if T does, but more to the point, this happens precisely if T_1 writes the symbol σ.

You might be concerned about the special case in which a move of T that looks like an accepting move is actually a rejecting move, because it causes T to try to move the tape head off the tape. Because of the way we have modified T to obtain T_1, however, T_1 does not actually write the symbol σ in this case; if T tries to move its tape head left from square 0, then T_1 will also, and our convention in this case is that the symbol in square 0 before the move will remain unchanged (see the discussion following Definition 7.1).

There are actually decision problems having to do with Turing machines that are decidable! Here is a simple example: Given a TM T, does T make at least ten moves on input Λ? Being "given" a TM allows us to trace as many of its moves as we wish. Here is another example that is not quite as simple, and might surprise you in the light of the fifth problem in the list above.

WritesNonblank: Given a TM T, does T ever write a nonblank symbol on input Λ?

Theorem 9.10

The decision problem *WritesNonblank* is decidable.

Proof

Suppose T is a TM with n nonhalting states. Then within n moves, either it halts or it enters some nonhalting state q for the second time. We can certainly decide whether T writes a nonblank symbol within the first n moves. If it reaches q the second time without having written a nonblank symbol, the tape head is at least as far to the right as it was the first time q was reached, and the tape is still blank; in this case, T will continue to repeat the finite sequence of moves that brought it from q back to q, and the tape will remain blank forever. Therefore, an algorithm to decide *WritesNonblank* is to trace T for n moves, or until it halts, whichever comes first; if by that time no nonblank symbol has been written, none will ever be.

In both of these last two cases, as well as in the first two decision problems in Theorem 9.9, the problem has the form: Given a TM T, . . . ? However, there is an important difference between the two problems in the theorem and these last two: The undecidable problems involve questions about the strings accepted by a TM, and the two decidable problems ask questions about the way the TM operates, or about the steps of a computation. The discouraging news, as we shall see, is that the first type of problem is almost always undecidable.

Definition 9.11 A Language Property of TMs

A property R of Turing machines is called a *language property* if, for every Turing machine having property R, and every other TM T_1 with $L(T_1) = L(T)$, T_1 also has property R. A language property of TMs is *nontrivial* if there is at least one TM that has the property and at least one that doesn't.

Some properties of TMs are clearly language properties: for example, "accepts at least two strings" and "accepts Λ." (If T accepts at least two strings, and $L(T_1) = L(T)$, then T_1 also accepts at least two strings.) Some properties are clearly not language properties: for example, "writes a nonblank symbol when started on a blank tape."

In some cases you have to be a little more careful. For example, "accepts its own encoding" might sound at first like a language property, but it is not. Knowing exactly what set of strings T accepts may not help you decide whether T accepts $e(T)$ unless you also know something about the string $e(T)$ (see Exercise 9.23).

Theorem 9.12 Rice's Theorem

If R is a nontrivial language property of TMs, then the decision problem

$$P_R: \text{ Given a TM } T, \text{ does } T \text{ have property } R?$$

is undecidable.

Proof

We will show that the undecidable problem *Accepts*-Λ can be reduced, either to P_R or to its complementary problem P_{not-R}. This is sufficient to prove the theorem, because by Theorem 9.5, if P_{not-R} is undecidable, then so is P_R.

Because R is assumed to be a nontrivial language property of TMs, there are two TMs T_R and T_{not-R}, the first satisfying property R, the second not satisfying it (i.e., satisfying property *not-R*). We may not even need the second; let us describe the algorithm we will use to obtain our reduction, and then consider exactly what it accomplishes.

We start with an arbitrary TM T, an instance of *Accepts*-Λ, and construct a TM $T_1 = F(T)$ to work as follows. T_1 begins by moving its tape head to the first blank after the input string and executing T as if its

input was Λ. If this computation causes T to reject, then T_1 rejects; if it causes T to accept, then T_1 erases the portion of the tape following the original input, moves its tape head back to square 0, and executes T_R on the original input.

If T accepts Λ, then T_1 accepts the language $L(T_R)$, and because T_R has the language property R, so does T_1. If T doesn't accept Λ, either because it rejects or because it loops forever, then T accepts the empty language: the language $L(T_0)$, where T_0 is a trivial TM that immediately rejects every input. We don't know whether T_0 has property R or not, but if it happens *not* to have it, our construction of T_1 has given us a reduction from *Accepts*-Λ to P_R.

Otherwise, if T_0 has property R, we use exactly the same algorithm, but substituting T_{not-R} for T_R. This allows us to say that if T accepts Λ, T_1 has property *not*-R, and otherwise T_1 has property R, so that we have a reduction from *Accepts*-Λ to P_{not-R}.

Now it should be more clear that our tentative conclusion in Section 9.1—that it's difficult to answer questions about Turing machines and the strings they accept—is correct. No matter what general question we ask about the language a TM accepts, there is no algorithm that will always give us the answer. The only exceptions to this rule are for *trivial* questions, to which the answer is always yes or always no.

Here are a few more decision problems that Rice's theorem implies are undecidable. This list is certainly not exhaustive.

1. (Assuming L is a recursively enumerable language): *AcceptsL*: Given a TM T, is $L(T) = L$?
2. *AcceptsSomething*: Given a TM T, is there at least one string in $L(T)$?
3. *AcceptsTwoOrMore*: Given a TM T, does $L(T)$ have at least two elements?
4. *AcceptsFinite*: Given a TM T, is $L(T)$ finite?
5. *AcceptsRecursive*: Given a TM T, is $L(T)$ (which by definition is recursively enumerable) recursive?

In order to use Rice's theorem directly to show that a decision problem P is undecidable, P must be a problem involving the language accepted by *one* TM. However, other problems might be proved undecidable by using Rice's theorem indirectly. For example, to show that the problem *Equivalent* mentioned in Theorem 9.9 is undecidable, we could argue as follows. Let $L = \{\Lambda\}$. Then by Rice's theorem, *AcceptsL*, problem 1 in the list just above, is undecidable. If T_1 is a TM with $L(T_1) = \{\Lambda\}$, then the function $F(T) = (T, T_1)$ is a reduction from *AcceptsL* to *Equivalent*, because for every TM T, $L(T) = \{\Lambda\}$ if and only if $L(T) = L(T_1)$. Therefore, *Equivalent* is undecidable.

Decision problems involving the operation of a TM, as opposed to the language it accepts, cannot be proved undecidable by using Rice's theorem directly. As the

examples *WritesNonblank* and *WritesSymbol* show, some of these are decidable and some are not.

Finally, Rice's theorem has nothing to say about trivial decision problems, which are decidable. It may not always be easy to tell whether a problem is trivial. The problem

$$\text{Given a TM } T, \text{ is } L(T) = NSA?$$

is trivial, because there is no TM that accepts *NSA*, and the answer is always no. If we didn't know *NSA* was not recursively enumerable, however, we might very well guess that the problem was undecidable.

9.4 | POST'S CORRESPONDENCE PROBLEM

The decision problem that we discuss in this section is a combinatorial problem involving pairs of strings, and is, for a change, not obviously related to Turing machines. Figure 9.13a shows a sample instance.

Think of each of the five rectangles in Figure 9.13a as a domino, and assume that there is an unlimited supply of each of the five. The question is whether it is possible to make a horizontal line of one or more dominoes, with duplicates allowed, so that the string obtained by reading across the top halves matches the one obtained by reading across the bottom.

In this example, the answer is yes. One way to do it is shown in Figure 9.13b.

10	01	0	100	1
101	100	10	0	010

(*a*)

10	1	01	0	100	100	0	100
101	010	100	10	0	0	10	0

(*b*)

Figure 9.13 |

Definition 9.14 Post's Correspondence Problem

An instance of Post's correspondence problem (*PCP*) is a set $\{(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_n, \beta_n)\}$ of pairs, where $n \geq 1$ and the α_i's and β_i's are all nonnull strings over an alphabet Σ. The decision problem is this: Given an instance of this type, do there exist a positive integer k and a sequence of integers $i_1, i_2, \ldots, i_k$, with each i_j satisfying $1 \leq i_j \leq n$, satisfying

$$\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$$

An instance of the *modified* Post correspondence problem (*MPCP*) looks exactly like an instance of *PCP*, but now the sequence of integers is required to start with 1. The question can be formulated this way: Do there exist a positive integer k and a sequence $i_2, i_3, \ldots, i_k$ such that

$$\alpha_1\alpha_{i_2}\ldots\alpha_{i_k} = \beta_1\beta_{i_2}\ldots\beta_{i_k}$$

Instances of *PCP* and *MPCP* are called *correspondence systems* and *modified correspondence systems*, respectively. For an instance of either type, if it is a yes-instance we will say that there is a *match* for the instance, or that the sequence of subscripts is a match, or that the string formed by the α_{i_j}'s represents a match.

Just as in many of the other decision problems we have discussed, there are simple trial-and-error algorithms which, for every yes-instance of *PCP*, can confirm that it is a yes-instance. For no-instances, however, algorithms like this don't always work, and there is no other algorithm that can do much better; in this section we will show that

$$Accepts \leq MPCP \leq PCP$$

which implies that both problems are undecidable, and that neither is completely unrelated to Turing machines after all.

Theorem 9.15
$MPCP \leq PCP$.

Proof
We will construct, for an arbitrary modified correspondence system

$$I = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_n, \beta_n)\}$$

a correspondence system

$$J = \{(\alpha'_1, \beta'_1), (\alpha'_2, \beta'_2), \ldots, (\alpha'_n, \beta'_n), (\alpha'_{n+1}, \beta'_{n+1})\}$$

as follows. The symbols used in J will be those in I plus two additional symbols # and \$. For $1 \leq i \leq n$, if

$$(\alpha_i, \beta_i) = (a_1a_2\ldots a_r, b_1b_2\ldots b_s)$$

we let

$$(\alpha'_i, \beta'_i) = (a_1\#a_2\#\ldots a_r\#, \#b_1\#b_2\#\ldots\#b_s)$$

except that the string α'_1 has an additional # at the beginning. The last pair in J is

$$(\alpha'_{n+1}, \beta'_{n+1}) = (\$, \#\$)$$

The pairs of I correspond to the first n pairs of J, and it is easy to check that if $\alpha_1\alpha_{i_2}\ldots\alpha_{i_k} = \beta_1\beta_{i_2}\ldots\beta_{i_k}$, then

$$\alpha'_1\alpha'_{i_2}\ldots\alpha'_{i_k}\alpha'_{n+1} = \beta'_1\beta'_{i_2}\ldots\beta'_{i_k}\beta'_{n+1}$$

On the other hand, because of the way the pairs of J are constructed, the only possible match must start with pair 1; in fact, if

$$\alpha'_{i_1}\alpha'_{i_2}\ldots\alpha'_{i_k} = \beta'_{i_1}\beta'_{i_2}\ldots\beta'_{i_k}$$

for some sequence $i_1, i_2, \ldots, i_k$, then i_1 must be 1 and i_k must be $n+1$, because the strings of the first pair must begin with the same symbol and those of the last pair must end with the same symbol. It is conceivable that some earlier i_j is also $n+1$, but if i_m is the first i_j to equal $n+1$, then $1, i_2, \ldots, i_m$ is a match for J, and $1, i_2, \ldots, i_{m-1}$ is a match for I.

Theorem 9.16
Accepts $\leq$ *MPCP*.

Proof
Let (T, w) be an arbitrary instance of *Accepts*, where $T = (Q, \Sigma, \Gamma, q_0, \delta)$ is a TM and w is a string over Σ. We will construct a modified correspondence system $F(T, w) = (\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_n, \beta_n)$ such that T accepts w if and only if there is a match for $F(T, w)$.

It will be convenient to assume that T never halts in the reject state, and we can modify T if necessary so that it has this property and still accepts the same strings (see the proof of Theorem 9.8). Some additional terminology will be helpful: For an instance $(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_n, \beta_n)$ of *MPCP*, we call the sequence $1, i_2, \ldots, i_j$ a *partial match* if $\alpha = \alpha_1\alpha_{i_2}\ldots\alpha_{i_j}$ is a prefix of $\beta = \beta_1\beta_{i_2}\ldots\beta_{i_j}$. We will also say in this case that the string α obtained this way is a partial match, or that the two strings α and β represent a partial match.

To simplify the notation, we will assume that $w \neq \Lambda$. This assumption will not play an essential part in the proof, and we will indicate later how to take care of the case $w = \Lambda$.

Here is a rough outline of the proof. The symbols involved in our pairs will be the ones that appear in configurations of T, together with the additional symbol #. We will try to specify pairs (α_i, β_i) in the modified correspondence system so that these four conditions are satisfied:

1. For every j, if $q_0\Delta w$, $x_1q_1y_1$, ... , $x_jq_jy_j$ are successive configurations through which T moves as it processes w, then a partial match can be obtained that looks like

$$\alpha = \#q_0\Delta w\#x_1q_1y_1\#x_2q_2y_2\#\ldots\#x_jq_jy_j$$

2. Every partial match is a prefix of one like this—or at least starts deviating from this form only after an occurrence of an accepting configuration $xh_a y$ in the string.
3. If $\alpha = \#q_0\Delta w\#x_1q_1y_1\#\ldots\#x_jh_ay_j$ is a partial match (which will imply, because of statement 2, that T accepts w), then there will be additional pairs (α_i, β_i) that will allow the prefix α to "catch up with" β, so that there will actually be a match for the modified correspondence system.
4. Statement 3 represents the *only* way that the modified correspondence system will have a match.

If we can successfully do this, the conclusions we want will not be hard to obtain.

Here is a slightly more detailed outline of how the pairs will be constructed. The first pair (α_1, β_1), with which every match must start, is

$$\alpha_1 = \#$$
$$\beta_1 = \#q_0\Delta w\#$$

Now suppose, for example, that $\delta(q_0, \Delta) = (q_1, a, \mathrm{R})$. Corresponding to this move, we will have a pair

$$(q_0\Delta, aq_1)$$

and we will make sure that this is the only choice for the next pair $(\alpha_{i_2}, \beta_{i_2})$ in a partial match. The resulting partial match will be

$$\alpha = \alpha_1\alpha_{i_2} = \#q_0\Delta$$
$$\beta = \beta_1\beta_{i_2} = \#q_0\Delta w\#aq_1$$

At this point, the only correct way to add to α is to add the symbols of w. Moreover, it is appropriate to add the same symbols to β, because they appear in the configuration that follows $q_0\Delta w$. Therefore, we include in our modified correspondence system pairs (σ, σ) for every $\sigma \in \Gamma \cup \{\Delta\}$, as well as the pair (#, #). These allow us to obtain the partial match

$$\alpha = \#q_0\Delta w\#$$
$$\beta = \#q_0\Delta w\#aq_1w\#$$

Now α has caught up to the *original* string β, and the new β has a second portion representing the configuration of T after the first move. We continue to extend the partial match in the same way. At each step, as long as the accepting state h_a has not appeared, the string β is one step ahead of α, and the next portion to be added to α is determined. The pairs (α_i, β_i) are such that every time α_i's are added to α to obtain the configuration of T required next, the corresponding β_i's that are added to β specify the configuration of T one move later. In this way, we guarantee that every partial match is of the right form. Roughly speaking, the only

thing left to do is to include pairs (α_i, β_i) that allow α to catch up to β once the configuration includes the state h_a.

Here is a precise description of the pairs (α_i, β_i) included in the modified correspondence system $F(T, w)$. They are grouped into several types; the order is not significant, and no subscripts are specified except for the first pair $(\alpha_1, \beta_1) = (\#, \#q_0\Delta w\#)$.

Pairs of type 1: (a, a) (for every $a \in \Gamma \cup \{\Delta\}$), and $(\#, \#)$

Pairs of type 2: For every choice of $q \in Q$, $p \in Q \cup \{h_a\}$, and $a, b, c \in \Gamma \cup \{\Delta\}$,

$$\begin{aligned}
(qa, pb) &\text{ if } \delta(q, a) = (p, b, \mathrm{S}) \\
(qa, bp) &\text{ if } \delta(q, a) = (p, b, \mathrm{R}) \\
(cqa, pcb) &\text{ if } \delta(q, a) = (p, b, \mathrm{L}) \\
(q\#, pa\#) &\text{ if } \delta(q, \Delta) = (p, a, \mathrm{S}) \\
(q\#, ap\#) &\text{ if } \delta(q, \Delta) = (p, a, \mathrm{R}) \\
(cq\#, pca\#) &\text{ if } \delta(q, \Delta) = (p, a, \mathrm{L})
\end{aligned}$$

Pairs of type 3: For every choice of $a, b \in \Gamma \cup \{\Delta\}$, the pairs

$$(h_a a, h_a), \quad (a h_a, h_a), \quad (a h_a b, h_a)$$

One pair of type 4: $(h_a\#\#, \#)$

The proof of the theorem depends on this claim: If

$$\begin{aligned}
\alpha &= \gamma\# \\
\beta &= \gamma\#z\#
\end{aligned}$$

is any partial match for the modified correspondence system, where z represents a nonhalting configuration of T, then we can extend it to a partial match

$$\begin{aligned}
\alpha' &= \gamma\#z\# \\
\beta &= \gamma\#z\#z'\#
\end{aligned}$$

where z' represents the configuration of T one move later. Furthermore, the string β' shown is the only one that can correspond to this α' in a partial match.

We examine the claim in the case where

$$\begin{aligned}
\alpha &= \gamma\# \\
\beta &= \gamma\#a_1 \ldots a_k q a_{k+1} \ldots a_{k+m}\#
\end{aligned}$$

and $m > 0$ and $\delta(q, a_{k+1}) = (p, b, \mathrm{R})$. The other cases are similar. We can extend the partial match α by using first the pairs $(a_1, a_1), \ldots, (a_k, a_k)$ of type 1; then the pair (qa_{k+1}, bp) of type 2; then any remaining pairs

(a_{k+2}, a_{k+2}), ..., (a_{k+m}, a_{k+m}); and finally the pair (#, #). The partial match we get is

$$\alpha = \gamma\#a_1 \ldots a_k q a_{k+1} \ldots a_{k+m}\#$$
$$\beta = \gamma\#a_1 \ldots a_k q a_{k+1} \ldots a_{k+m}\#a_1 \ldots a_k b p a_{k+2} \ldots a_{k+m}\#$$

The substring of β between the last two #'s is the configuration of T resulting from the move indicated, and at no point in this sequence of steps is there any choice as to which pair to use next. Thus the claim is correct in this case.

Suppose that T accepts the string w. Then there is a sequence of successive configurations

$$z_0 = q_0 \Delta w,\ z_1,\ z_2,\ \ldots, z_j$$

that ends with an accepting configuration. The claim above implies that there is a partial match

$$\alpha = \#z_0\#z_1\# \ldots \#z_{j-1}\#$$
$$\beta = \#z_0\#z_1\# \ldots \#z_{j-1}\#z_j\#$$

to the modified correspondence system, where $z_j = u h_a v$ for two strings u and v, either or both of which may be null. If at least one of these two strings is nonnull, we can extend the partial match by using one pair of type 3 and others of type 1, to obtain

$$\alpha' = \alpha z_j\#$$
$$\beta' = \alpha z_j\#z'_j\#$$

where z'_j still contains h_a but has at least one fewer symbol than z_j. We can continue to extend the partial match, decreasing the lengths of the strings between successive #'s, until we obtain a partial match of the form

$$\alpha''\#$$
$$\alpha''\#h_a\#$$

at which point the pair of type 4 produces a match for the modified correspondence system.

If T does not accept w, on the other hand, then our assumption is that it loops forever. For every partial match of the form

$$\alpha = \#z_0\#z_1\# \ldots \#z_k\#$$
$$\beta = \#z_0\#z_1\# \ldots \#z_k\#z_{k+1}\#$$

(where no string z_i contains #), the second part of the claim implies that the strings z_i must represent consecutive configurations of T. It follows that h_a can never appear in any partial match, which means that the pair

of type 4 is never used. Because α_1 and β_1 have different numbers of #'s, and the pair of type 4 is the only other one with this property, the modified correspondence system can have no match.

In the case $w = \Lambda$, the only change needed in the proof is that the initial pair is $(\#, \#q_0\#)$ instead of $(\#, \#q_0\Delta w\#)$.

Theorem 9.17
Post's correspondence problem is undecidable.

Proof
The theorem is an immediate consequence of Theorems 9.15 and 9.16.

EXAMPLE 9.18 A Modified Correspondence System for a TM

Let T be the TM shown in Figure 9.19 that accepts all strings in $\{a, b\}^*$ ending with b.

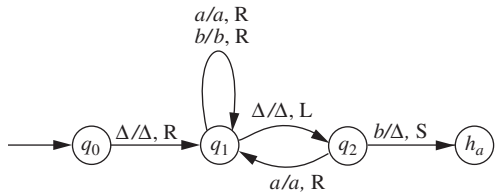


Figure 9.19

In the modified correspondence system constructed as in the proof of Theorem 9.10, the pairs of type 2 are these:

$(q_0\Delta, \Delta q_1)$	$(q_0\#, \Delta q_1\#)$	(q_1a, aq_1)	(q_1b, bq_1)
$(aq_1\Delta, q_2a\Delta)$	$(bq_1\Delta, q_2b\Delta)$	$(aq_1\#, q_2a\Delta\#)$	$(bq_1\#, q_2b\Delta\#)$
$(\Delta q_1\Delta, q_2\Delta\Delta)$	$(\Delta q_1\#, q_2\Delta\Delta\#)$	(q_2a, aq_1)	$(q_2b, h_a\Delta)$

We consider three possible choices for pair 1, corresponding to two strings w that are not accepted by T and one that is. For the input Λ, pair 1 is $(\#, \#q_0\#)$, and the partial match below is the only one, except for smaller portions of it.

#	$q_0\#$	$\Delta q_1\#$
$\#q_0\#$	$\Delta q_1\#$	$q_2\Delta\Delta\#$

The input string a causes T to loop forever. In this case, pair 1 is $(\#, \#q_0\Delta a\#)$. In the partial match shown, the pair that appears last is appearing for the second time, and

longer partial matches would simply involve repetitions of the portion following the first occurrence.

#	$q_0\Delta$	a	#	Δ	q_1a	#	Δ	$aq_1\#$	Δ	q_2a	Δ	#
$\#q_0\Delta a\#$	Δq_1	a	#	Δ	aq_1	#	Δ	$q_2a\Delta\#$	Δ	aq_1	Δ	#

Δ	$aq_1\Delta$	#	Δ	q_2a
Δ	$q_2a\Delta$	#	Δ	aq_1

Finally, for the input string b, which is accepted by T, pair 1 is $(\#, \#q_0\Delta b\#)$, and the match is shown below.

#	$q_0\Delta$	b	#	Δ	q_1b	#	Δ	$bq_1\#$	Δ	q_2b	Δ	#
$\#q_0\Delta b\#$	Δq_1	b	#	Δ	bq_1	#	Δ	$q_2b\Delta\#$	Δ	$h_a\Delta$	Δ	#

$\Delta h_a\Delta$	Δ	#	$h_a\Delta$	#	$h_a\#\#$
h_a	Δ	#	h_a	#	#

9.5 UNDECIDABLE PROBLEMS INVOLVING CONTEXT-FREE LANGUAGES

In this section we will look at two approaches to obtaining undecidability results involving context-free grammars and CFLs. The first uses the fact that Post's Correspondence Problem is undecidable.

For an instance $(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_n, \beta_n)$ of *PCP*, where the α_i's and β_i's are strings over Σ, we construct two context-free grammars G_α and G_β whose properties are related to those of the correspondence system. Let $c_1, c_2, \ldots, c_n$ be symbols that are not in Σ. G_α will be the CFG with start symbol S_α and the $2n$ productions

$$S_\alpha \rightarrow \alpha_i S_\alpha c_i \mid \alpha_i c_i \quad (1 \leq i \leq n)$$

and G_β is constructed the same way from the strings β_i. Then for every string $c = c_{i_1}c_{i_2}\ldots c_{i_k}$, where $k \geq 1$, there is exactly one string x in $L(G_\alpha)$ and exactly one string y in $L(G_\beta)$ that is a string in Σ^* followed by c. The string x is $\alpha_{i_k}\ldots\alpha_{i_2}\alpha_{i_1}c_{i_1}c_{i_2}\ldots c_{i_k}$, and the string y is $\beta_{i_k}\ldots\beta_{i_2}\beta_{i_1}c_{i_1}c_{i_2}\ldots c_{i_k}$. Both x and y

have unique derivations in their respective grammars, and both derivations are determined by c.

Theorem 9.20

These two problems are undecidable:

1. *CFGNonemptyIntersection*: Given two CFGs G_1 and G_2, is $L(G_1) \cap L(G_2)$ nonempty?
2. *IsAmbiguous*: Given a CFG G, is G ambiguous?

Proof

We will reduce *PCP* both to *CFGNonemptyIntersection* and to *IsAmbiguous*, and the two reductions will be very similar.

Starting with an arbitrary instance I of *PCP*, involving α_i's and β_i's as above, we construct the two context-free grammars G_α and G_β as we have described. For the first reduction, we let $F_1(I)$ be the instance (G_α, G_β) of *CFGNonemptyIntersection*, and for the second we let $F_2(I)$ be the grammar G constructed in the usual way to generate $L(G_\alpha) \cup L(G_\beta)$: The start symbol of G is S, and the productions include those of G_α, those of G_β, and the two additional ones $S \rightarrow S_\alpha \mid S_\beta$.

If I is a yes-instance of *PCP*, then for some sequence of integers i_1, i_2, . . . , i_k,

$$\alpha_{i_k}\alpha_{i_{k-1}} \ldots \alpha_{i_1} = \beta_{i_k}\beta_{i_{k-1}} \ldots \beta_{i_1}$$

so that

$$\alpha_{i_k}\alpha_{i_{k-1}} \ldots \alpha_{i_1}c_{i_1} \ldots c_{i_k} = \beta_{i_k}\beta_{i_{k-1}} \ldots \beta_{i_1}c_{i_1} \ldots c_{i_k}$$

It follows that this string is an element of the intersection $L(G_\alpha) \cap L(G_\beta)$, and that it has two derivations in G, one starting with $S \Rightarrow S_\alpha$ and the other with $S \Rightarrow S_\beta$. Therefore, both $F_1(I)$ and $F_2(I)$ are yes-instances of their respective decision problems.

On the other hand, if either $F_1(I)$ or $F_2(I)$ is a yes-instance (the two statements are equivalent), then for some $x \in \Sigma^*$ and some sequence of integers i_1, . . . , i_k, there is a string $xc_{i_1}c_{i_2} \ldots c_{i_k}$ that is in the intersection $L(G_\alpha) \cap L(G_\beta)$ and can therefore be derived from either S_α or S_β. As we observed above, this implies that

$$x = \alpha_{i_k}\alpha_{i_{k-1}} \ldots \alpha_{i_1}c_{i_1} \ldots c_{i_k} = \beta_{i_k}\beta_{i_{k-1}} \ldots \beta_{i_1}c_{i_1} \ldots c_{i_k}$$

which implies that I is a yes-instance of *PCP*.

The other approach in this section is to return to Turing machines and to consider, for a TM T, some aspects of the computations of T that are represented by context-free languages, so that being able to answer certain questions about CFLs would allow us to answer questions about TMs.

Definition 9.21 Valid Computations of a TM

Let $T = (Q, \Sigma, \Gamma, q_0, \delta)$ be a Turing machine. A *valid computation* of T is a string of the form

$$z_0\#z_1^r\#z_2\#z_3^r \ldots \#z_{n-1}^r\#z_n\#$$

if n is even, or

$$z_0\#z_1^r\#z_2\#z_3^r \ldots \#z_{n-1}\#z_n^r\#$$

if n is odd, where in either case, # is a symbol not in Γ, and the strings z_i represent successive configurations of T on some input string x, starting with the initial configuration z_0 and ending with an accepting configuration. The set of valid computations of T will be denoted by C_T.

Reversing every other configuration in these strings is the trick that allows us to involve CFLs, because a substring of the form $z_i\#z_{i+1}^r\#$ differs in only a few positions from a palindrome.

Theorem 9.22

For a TM $T = (Q, \Sigma, \Gamma, q_0, \delta)$, the set C_T of valid computations of T is the intersection of two context-free languages, and its complement C_T' is a context-free language.

Proof

The two context-free languages that we will use to obtain C_T can be described in terms of these five languages:

$L_1 = \{z\#(z')^r\# \mid z \text{ and } z' \text{ represent configurations of } T \text{ for which } z \vdash z'\}$

$L_2 = \{z^r\#z'\# \mid z \text{ and } z' \text{ represent configurations of } T \text{ for which } z \vdash z'\}$

$I = \{z\# \mid z \text{ is an initial configuration of } T\}$

$A = \{z\# \mid z \text{ is an accepting configuration of } T\}$

$A_1 = \{z^r\# \mid z \text{ is an accepting configuration of } T\}$

First we will express C_T as the intersection of two languages that can be obtained from these five, and then we will show that the two languages are actually CFLs.

A valid computation c of T in which the total number of configurations is even can be written both as

$$c = z_0\#(z_1^r\#z_2\#) \ldots (z_{n-2}^r\#z_{n-1}\#)z_n^r\#$$

where z_0 is an initial configuration and z_n is an accepting configuration, and as

$$c = (z_0\#z_1^r\#)(z_2\#z_3^r\#) \ldots (z_{n-1}\#z_n^r\#)$$

In other words, $c \in IL_2^*A_1 \cap L_1^*$. Similarly, if the number of configurations involved in the string c is odd, then $c \in IL_2^* \cap L_1^*A$. These two statements imply that

$$C_T \subseteq L_3 \cap L_4$$

where $L_3 = IL_2^*(A_1 \cup \{\Lambda\})$ and $L_4 = L_1^*(A \cup \{\Lambda\})$.

On the other hand, consider a string z that is either of the form

$$z = z_0\#z_1^r\#z_2\#z_3^r \ldots \#z_{n-1}^r\#z_n\#$$

or of the form

$$z_0\#z_1^r\#z_2\#z_3^r \ldots \#z_{n-1}\#z_n^r\#$$

where the z_i's represent configurations of T (not necessarily consecutive), z_0 represents an initial configuration, and z_n represents an accepting configuration. If $z \in L_3$, then for each odd i, $z_i \vdash z_{i+1}$, and if $z \in L_4$, the same is true for each even i. It follows that

$$L_3 \cap L_4 \subseteq C_T$$

Therefore, we can complete the proof of the first statement in the theorem by showing that L_3 and L_4 are CFLs. Because of the way L_3 and L_4 are defined, it is sufficient to show that the languages L_1 and L_2 are both CFLs.

We will prove that L_1 is a CFL by describing a PDA M that accepts it, and the proof for L_2 is similar. M will have in its input alphabet both states of T and tape symbols of T, including Δ, and the two types of symbols are assumed not to overlap.

One aspect of M's computation, which can be accomplished without the stack, is to check that the input string is of the form $z\#z'\#$, where both z and z' are elements of $\Gamma^*Q\Gamma^*$. If the input is not of this form, M will reject.

The step that is crucial for the rest of the argument is to show that M can operate so that if the first portion z of the input is a configuration of T, then the stack contents when this portion has been processed is simply $(z')^r Z_0$, where $z \vdash_T z'$ and Z_0 is the initial stack symbol of M. This feature will allow M to process the input after the first # by simply matching input symbols with stack symbols.

We consider the case in which

$$z = xapby$$

where $x, y \in (\Gamma \cup \{\Delta\})^*$, $p \in Q$, $a, b \in \Gamma \cup \Delta$, and

$$\delta(p, b) = (q, c, \mathrm{L})$$

The other cases are similar. T moves from the configuration $xapby$ to $xqacy$. M can operate by pushing input symbols onto its stack until it sees a state of T; when this happens, the stack contains $ax^r Z_0$, and the next input symbol is b. It pops the a, replaces it by caq, and continues to

read symbols and push them onto the stack until it encounters #. At this point, z has been processed completely, the stack contents are

$$y^r caqx^r Z_0 = (xqacy)^r Z_0$$

and M can continue its operation through the remainder of the string in the way we described.

For the second statement of the theorem, we must consider the complement C'_T of C_T in the set $(\Gamma \cup \{\Delta\} \cup Q \cup \{h_a\} \cup \{\#\})^*$.

We can use the definition of a valid computation of T to show that there are seven ways a string x over this alphabet can fail to be in C_T. The first and simplest way is for x not to end in #. The remaining six ways, for

$$x = z_0\#z_1\#\ldots\#z_k\#$$

where no z_i contains #, are:

2. For some even i, z_i does not represent a configuration of T.
3. For some odd i, z_i^r does not represent a configuration of T.
4. The string z_0 does not represent an initial configuration of T.
5. z_k is neither an accepting configuration of T nor the reverse of one.
6. For some even i, z_i and z_{i+1}^r represent configurations of T but z_{i+1}^r is not the configuration to which T moves from z_i.
7. For some odd i, z_i^r and z_{i+1} represent configurations of T but z_{i+1} is not the configuration to which T moves from z_i^r.

Each of these seven conditions can be tested individually by a PDA, and in some cases an FA would be enough. For the last two conditions, nondeterminism can be used to select an i, and from that point on the argument is similar to the one we used for the language L_1 above.

The conclusion is that C'_T is the union of seven CFLs, which means that it is a CFL itself.

Not only are there context-free grammars generating the languages L_3 and L_4 described in the proof of Theorem 9.22, but they can be constructed algorithmically from the Turing machine T. This provides an alternative proof that the decision problem *CFGNonemptyIntersection* is undecidable (see Exercise 9.26). The last theorem in this section is another undecidability result that uses the second part of Theorem 9.22.

Theorem 9.23

The decision problem

CFGGeneratesAll: Given a CFG G with terminal alphabet Σ, is $L(G) = \Sigma^*$?

is undecidable.

Proof
If T is a TM, the last part of the proof of Theorem 9.22 refers to seven CFLs whose union is C'_T; we could formulate an algorithm to find a CFG for each one in terms of T, and therefore to find a CFG $F(T) = G$ generating C'_T. Let Σ be the terminal alphabet of G. Saying that T accepts at least one string is equivalent to saying that C_T is not empty, or that C'_T is not Σ^*. As a result, the algorithm defines a reduction from the problem

AcceptsNothing: Given a TM T, is $L(T) = \emptyset$?

to *CFGGeneratesAll*, and *AcceptsNothing* is undecidable by Rice's theorem.

EXERCISES

9.1. Show that the relation $\leq$ on the set of languages (or on the set of decision problems) is reflexive and transitive. Give an example to show that it is not symmetric.

9.2. Describe how a universal Turing machine could be used in the proof that *SA* is recursively enumerable.

9.3. Show that if L_1 and L_2 are languages over Σ and L_2 is recursively enumerable and $L_1 \leq L_2$, then L_1 is recursively enumerable.

9.4. Show that if $L \subseteq \Sigma^*$ is neither empty nor all of Σ^*, then every recursive language over Σ can be reduced to L.

9.5. *Fermat's last theorem*, until recently one of the most famous unproved statements in mathematics, asserts that there are no integer solutions (x, y, z, n) to the equation $x^n + y^n = z^n$ satisfying $x, y > 0$ and $n > 2$. Ignoring the fact that the theorem has now been proved, explain how a solution to the halting problem would allow you to determine the truth or falsity of the statement.

9.6. Show that every recursively enumerable language can be reduced to the language $Acc = \{e(T)e(w) \mid T \text{ is a TM and } T \text{ accepts input } w\}$.

9.7. As discussed at the beginning of Section 9.3, there is at least one TM T such that the decision problem "Given w, does T accept w?" is unsolvable. Show that every TM accepting a nonrecursive language has this property.

9.8. Show that for every $x \in \Sigma^*$, the problem *Accepts* can be reduced to the problem: Given a TM T, does T accept x? (This shows that, just as *Accepts*-Λ is unsolvable, so is *Accepts*-x, for every x.)

9.9. Construct a reduction from *Accepts*-Λ to the problem *Accepts*-$\{\Lambda\}$: Given a TM T, is $L(T) = \{\Lambda\}$?

9.10. a. Given two sets A and B, find two sets C and D, defined in terms of A and B, such that $A = B$ if and only if $C \subseteq D$.

b. Show that the problem *Equivalent* can be reduced to the problem *Subset*.

9.11. Construct a reduction from the problem *AcceptsEverything* to the problem *Equivalent*.

9.12. For each decision problem below, determine whether it is decidable or undecidable, and prove your answer.

a. Given a TM T, does it ever reach a state other than its initial state if it starts with a blank tape?

b. Given a TM T and a nonhalting state q of T, does T ever enter state q when it begins with a blank tape?

c. Given a TM T and a nonhalting state q of T, is there an input string x that would cause T eventually to enter state q?

d. Given a TM T, does it accept the string Λ in an even number of moves?

e. Given a TM T, is there a string it accepts in an even number of moves?

f. Given a TM T and a string w, does T loop forever on input w?

g. Given a TM T, are there any input strings on which T loops forever?

h. Given a TM T and a string w, does T reject input w?

i. Given a TM T, are there any input strings rejected by T?

j. Given a TM T, does T halt within ten moves on every string?

k. Given a TM T, is there a string on which T halts within ten moves?

l. †Given a TM T, does T eventually enter every one of its nonhalting states if it begins with a blank tape?

m. Given a TM T, is there an input string that causes T to enter every one of its nonhalting states?

9.13. In this problem TMs are assumed to have input alphabet $\{0, 1\}$. For a finite set $S \subseteq \{0, 1\}^*$, P_S denotes the decision problem: Given a TM T, is $S \subseteq L(T)$?

a. Show that if $x, y \in \{0, 1\}^*$, then $P_{\{x\}} \leq P_{\{y\}}$.

b. Show that if $x, y, z \in \{0, 1\}^*$, then $P_{\{x\}} \leq P_{\{y,z\}}$.

c. Show that if $x, y, z \in \{0, 1\}^*$, then $P_{\{x,y\}} \leq P_{\{z\}}$.

d. Show that for every two finite subsets S and U of $\{0, 1\}^*$, $P_S \leq P_U$.

9.14. †Repeat the previous problem, but this time letting P_S denote the problem: Given a TM T, is $L(T) = S$?

9.15. Let us make the informal assumption that Turing machines and computer programs written in the C language are equally powerful, in the sense that anything that can be programmed on one can be programmed on the other. Give a convincing argument that both of the following decision problems are undecidable:

a. Given a C program, a statement s in the program, and a specific set I of input data, is s ever executed when the program is run on input I?

b. Given a C program and a statement s in the program, is there a set I of input data such that s is executed when the program runs on input I?

9.16. Suppose P_1 and P_2 are decision problems, and $Y(P_1) \subseteq \Sigma_1^*$ and $Y(P_2) \subseteq \Sigma_2^*$ are the corresponding languages (that is, the languages of strings representing yes-instances of P_1 and P_2, respectively, with respect to some reasonable encoding functions e_1 and e_2). Assume there is at least one string in Σ_2^* that is not an instance of P_2. Suppose the function f defines a reduction from P_1 to P_2; in other words, for every instance I of P_1, $f(I)$ is an instance of P_2 having the same answer. Show that $Y(P_1) \leq Y(P_2)$. Describe a function from Σ_1^* to Σ_2^* that gives a reduction.

9.17. Suppose P_1, P_2, $Y(P_1)$, and $Y(P_2)$ are as in Exercise 9.16. Suppose also that there is at least one no-instance of P_2. Show that if there is a function $t : \Sigma^* \to \Sigma^*$ reducing $Y(P_1)$ to $Y(P_2)$, then there is another (computable) function t' reducing $Y(P_1)$ to $Y(P_2)$ and having the property that for every $x \in \Sigma^*$ that corresponds to an instance of P_1, $t'(x)$ corresponds to an instance of P_2.

9.18. Let P_1, P_2, $Y(P_1)$, and $Y(P_2)$ be as in Exercise 9.16. Suppose $t : \Sigma_1^* \to \Sigma_2^*$ is a reduction of $Y(P_1)$ to $Y(P_2)$. According to Exercise 9.17, we may assume that for every string x in Σ_1^* representing an instance of P_1, $t(x)$ represents an instance of P_2. Show that $P_1 \leq P_2$. Describe a function f that gives a reduction. (In other words, for an instance I of P_1, say how to calculate an instance $f(I)$ of P_2.)

9.19. Show that the following decision problems involving unrestricted grammars are undecidable.

a. Given a grammar G and a string w, does G generate w?

b. Given a grammar G, does it generate any strings?

c. Given a grammar G with terminal alphabet Σ, does it generate every string in Σ^*?

d. Given grammars G_1 and G_2, do they generate the same language?

9.20. †This exercise presents an example of a language L such that neither L nor L' is recursively enumerable. Let Acc and AE be the languages over $\{0, 1\}$ defined as follows.

$$Acc = \{e(T)e(w) \mid T \text{ is a TM that accepts the input string } w\}$$
$$AE = \{e(T) \mid T \text{ is a TM accepting every string in its input alphabet}\}$$

(Acc and AE are the sets of strings representing yes-instances of the problems *Accepts* and *AcceptsEverything*, respectively.) Acc' and AE' denote the complements of these two languages.

a. Show that $Acc \leq AE$.

b. Show that $\text{Acc}' \leq AE'$.

c. Show that AE' is not recursively enumerable.

d. Show that $Acc' \leq AE$. (If $x = e(T)e(z)$, let $f(x) = e(S_{T,z})$, where $S_{T,z}$ is a TM that works as follows. On input w, $S_{T,z}$ simulates the computation performed by T on input z for up to $|w|$ moves. If this computation would cause T to accept within $|w|$ moves, $S_{T,z}$ enters an

infinite loop; otherwise $S_{T,z}$ accepts. Show that if $f(x)$ is defined appropriately for strings x not of the form $e(T)e(z)$, then f defines a reduction from Acc' to AE.)

e. Show that AE is not recursively enumerable.

9.21. If AE is the language defined in the previous exercise, show that if L is any language whose complement is not recursively enumerable, then $L \leq AE$.

9.22. Find two undecidable decision problems, neither of which can be reduced to the other, and prove it.

9.23. Show that the property "accepts its own encoding" is not a language property of TMs.

9.24. In parts (a), (d), (e), (g), (i), (j), (k), (l), and (m) of Exercise 9.12, an instance of the decision problem is a TM. Show in each part that the TM property involved is not a language property.

9.25. Rice's theorem can be extended to decision problems whose instances are pairs (T_1, T_2) of Turing machines, problems of the form: Given TMs T_1 and T_2, do T_1 and T_2 satisfy some property? Following the original version of the theorem, we call a 2-TM property (i.e., a property of pairs of TMs) a *language property* if, whenever T_1 and T_2 have the property and $L(T_3) = L(T_1)$ and $L(T_4) = L(T_2)$, T_3 and T_4 also have the property; and it is a *nontrivial* language property if there is a pair (T_1, T_2) of TMs having the property and another pair (T_3, T_4) not having the property. By adapting the proof of Rice's theorem, prove that if R is a nontrivial language property of two TMs, then the decision problem "Given TMs T_1 and T_2, do they have property R?" is undecidable.

9.26. The decision problem *NonemptyIntersection*: "Given two TMs T_1 and T_2, is $L(T_1) \cap L(T_2)$ nonempty?" is undecidable, as a result of what was proved in Exercise 9.25. Prove that this problem is undecidable without using Rice's theorem, by reducing another undecidable problem to it. Do the same for the problem: Given two TMs T_1 and T_2, is $L(T_1) \cup L(T_2)$ empty?

9.27. In each case below, either find a match for the correspondence system or show that none exists.

a.

100	101	110
10	01	1010

b.

1	01	0	001
10	101	101	0

9.28. Show that the special case of *PCP* in which the alphabet has only two symbols is still undecidable.

9.29. Show that the special case of *PCP* in which the alphabet has only one symbol is decidable.

9.30. Show that the problem *CSLIsEmpty*: given a linear-bounded automaton, is the language it accepts empty? is undecidable. Suggestion: use the fact that Post's correspondence problem is undecidable, by starting with an arbitrary correspondence system and constructing an LBA that accepts precisely the strings α representing solutions to the correspondence system.

9.31. This exercises establishes the fact that there is a recursive language over $\{a, b\}$ that is not context-sensitive. A diagonal argument is suggested. (At this point, a diagonal argument or something comparable is the only technique known for constructing languages that are not context-sensitive.)

a. Describe a way to enumerate explicitly the set of context-sensitive grammars generating languages over $\{a, b\}$. You may make the assumption that for some set $A = \{A_1, A_2, \dots\}$, every such grammar has start symbol A_1 and only variables that are elements of A.

b. If $G_1, G_2, \dots$ is the enumeration in part (a), and $x_1, x_2, \dots$ are the nonnull elements of $\{a, b\}^*$ listed in canonical order, let $L = \{x_i \mid x_i \notin L(G_i)\}$. Show that L is recursive and not context-sensitive.

9.32. Show that each of the following decision problems for CFGs is undecidable.

a. Given two CFGs G_1 and G_2, is $L(G_1) = L(G_2)$?

b. Given two CFGs G_1 and G_2, is $L(G_1) \subseteq L(G_2)$?

c. Given a CFG G and a regular language R, is $L(G) = R$?

9.33. †Is the decision problem. "Given a CFG G, and a string x, is $L(G) = \{x\}$?" decidable or undecidable? Give reasons for your answer.

9.34. Is the decision problem. "Given a CFG G and a regular language R, is $L(G) \subseteq R$?" decidable or undecidable? Give reasons for your answer.

9.35. †Show that the problem. "Given a CFG G with terminal alphabet Σ, is $L(G) \neq \Sigma^*$?" is undecidable by directly reducing *PCP* to it. Suggestion: if G_α and G_β are the CFGs constructed from an instance of *PCP* as in Section 9.5, show that there is an algorithm to construct a CFG generating $(L(G_\alpha) \cap L(G_\beta))'$.

CHAPTER

10

Computable Functions

In the same way that most languages over an alphabet Σ are not decidable, most partial functions on Σ^* are not computable. In this chapter we consider an approach to computability due to Kleene, in which we try to say more explicitly what sorts of computations we (or a Turing machine) can actually carry out. We start by considering only numeric functions, although this is not as much of a restriction as it might seem. We discuss how to characterize the computable functions by describing a set of "initial" functions and a set of operations that preserve the property of computability. Using a numbering system introduced by Gödel to "arithmetize" a Turing machine, we can demonstrate that the resulting "μ-recursive" functions are the same as the Turing-computable functions.

10.1 | PRIMITIVE RECURSIVE FUNCTIONS

For the rest of this chapter, the functions we discuss will be partial functions from $\mathcal{N}^k$ to $\mathcal{N}$, for some $k \geq 1$. We will generally use a lowercase letter for a number and an uppercase one for a vector (a k-tuple of natural numbers).

Definition 10.1 Initial Functions

The initial functions are the following:

1. *Constant* functions: For each $k \geq 1$ and each $a \geq 0$, the constant function $C_a^k : \mathcal{N}^k \to \mathcal{N}$ is defined by the formula

$$C_a^k(X) = a \qquad \text{for every } X \in \mathcal{N}^k$$

2. The *successor* function $s : \mathcal{N} \to \mathcal{N}$ is defined by the formula

$$s(x) = x + 1$$

3. *Projection* functions: For each $k \geq 1$ and each i with $1 \leq i \leq k$, the projection function $p_i^k : \mathcal{N}^k \to \mathcal{N}$ is defined by the formula

$$p_i^k(x_1, x_2, \ldots, x_i, \ldots, x_k) = x_i$$

The first operations by which new functions will be obtained are composition, which in the simplest case gives us functions with formulas like $f \circ g(x) = f(g(x))$, and an operation that produces a function defined recursively in terms of given functions. For the second one, a few preliminaries may be helpful.

The simplest way to define a function f from $\mathcal{N}$ to $\mathcal{N}$ recursively is to define $f(0)$ first, and then for every $k \geq 0$ to define $f(k+1)$ in terms of $f(k)$. A familiar example is the factorial function:

$$0! = 1 \qquad (k+1)! = (k+1) * k!$$

In the recursive step, the expression for $f(k+1)$ involves both k and $f(k)$. We can generalize this by substituting any expression of the form $h(k, f(k))$, where h is a function of two variables. In order to use this approach for a function f of more than one variable, we simply restrict the recursion to the last coordinate. In other words, we start by saying what $f(x_1, x_2, \ldots, x_n, 0)$ is, for any choice of $(x_1, \ldots, x_n)$. In the recursive step, we say what $f(x_1, x_2, \ldots, x_n, k+1)$ is, in terms of $f(x_1, \ldots, x_n, k)$. If X represents the n-tuple $(x_1, \ldots, x_n)$, then in the most general case, $f(X, k+1)$ may depend on X and k directly, in addition to $f(X, k)$, just as $(k+1)!$ depended on k as well as on $k!$. A reasonable way to formulate this recursive step is to say that

$$f(X, k+1) = h(X, k, f(X, k))$$

for some function h of $n+2$ variables.

Definition 10.2 The Operations of Composition and Primitive Recursion

1. Suppose f is a partial function from $\mathcal{N}^k$ to $\mathcal{N}$, and for each i with $1 \leq i \leq k$, g_i is a partial function from $\mathcal{N}^m$ to $\mathcal{N}$. The partial function obtained from f and $g_1, g_2, \ldots, g_k$ by composition is the partial function h from $\mathcal{N}^m$ to $\mathcal{N}$ defined by the formula

$$h(X) = f(g_1(X), g_2(X), \ldots, g_k(X)) \text{ for every } X \in \mathcal{N}^m$$

2. Suppose $n \geq 0$, and g and h are functions of n and $n+2$ variables, respectively. (By "a function of 0 variables," we mean simply a constant.) The function obtained from g and h by the operation of *primitive recursion*

is the function $f : \mathcal{N}^{n+1} \to \mathcal{N}$ defined by the formulas

$$f(X, 0) = g(X)$$
$$f(X, k + 1) = h(X, k, f(X, k))$$

for every $X \in \mathcal{N}^n$ and every $k \geq 0$.

A function obtained by either of these operations from total functions is a total function. If $f : \mathcal{N}^{n+1} \to \mathcal{N}$ is obtained from g and h by primitive recursion, and $g(X)$ is undefined for some $X \in \mathcal{N}^n$, then $f(X, 0)$ is undefined, $f(X, 1) = h(X, 0, f(X, 0))$ is undefined, and similarly $f(X, k)$ is undefined for each k. For the same reason, if $f(X, k)$ is undefined for *some* k, say $k = k_0$, then $f(X, k)$ is undefined for every $k \geq k_0$. This is the same as saying that if $f(X, k_1)$ is defined, then $f(X, k)$ is defined for every $k \leq k_1$.

Definition 10.3 Primitive Recursive Functions

The set *PR* of *primitive recursive* functions is defined as follows.

1. All initial functions are elements of *PR*.
2. For every $k \geq 0$ and $m \geq 0$, if $f : \mathcal{N}^k \to \mathcal{N}$ and $g_1, g_2, \ldots, g_k : \mathcal{N}^m \to \mathcal{N}$ are elements of *PR*, then the function $f(g_1, g_2, \ldots, g_k)$ obtained from f and $g_1, g_2, \ldots, g_k$ by composition is an element of *PR*.
3. For every $n \geq 0$, every function $g : \mathcal{N}^n \to \mathcal{N}$ in *PR*, and every function $h : \mathcal{N}^{n+2} \to \mathcal{N}$ in *PR*, the function $f : \mathcal{N}^{n+1} \to \mathcal{N}$ obtained from g and h by primitive recursion is in *PR*.

In other words, the set *PR* is the smallest set of functions that contains all the initial functions and is closed under the operations of composition and primitive recursion.

It is almost obvious that the initial functions are computable, and the set of computable functions is also closed under the operations of composition and primitive recursion. It is possible to describe in detail Turing machines that can compute the functions obtained by these operations, assuming that the functions we start with are computable; the algorithms are straightforward, and we will simply cite the Church-Turing thesis as the reason for Theorem 10.4.

Theorem 10.4

Every primitive recursive function is total and computable.

Not all total computable functions are primitive recursive, and Exercise 10.28 discusses a way to show this using a diagonal argument. In the next section we will consider other operations on functions that preserve the property of computability but not that of primitive recursiveness.

EXAMPLE 10.5 Addition, Multiplication, and Subtraction

The functions *Add* and *Mult*, both functions from $\mathcal{N} \times \mathcal{N}$ to $\mathcal{N}$, are defined by the formulas

$$Add(x, y) = x + y \qquad Mult(x, y) = x * y$$

In general, we can show that a function f is primitive recursive by constructing a *primitive recursive derivation*: a sequence of functions $f_0, f_1, \ldots, f_j$ such that $f_j = f$ and each f_i in the sequence is an initial function, or obtained from earlier functions in the sequence by composition, or obtained from earlier functions by primitive recursion. For both *Add* and *Mult*, we can obtain a derivation in reverse, by identifying simpler functions from which the function we want can be obtained by primitive recursion.

$$Add(x, 0) = x = p_1^1(x)$$
$$Add(x, k + 1) = (x + k) + 1 = s(Add(x, k)) = s(p_3^3(x, k, Add(x, k)))$$

A primitive recursive derivation for *Add*, therefore, can consist of the three initial functions p_1^1, s, and p_3^3, the function f_3 obtained from s and p_3^3 by composition, and *Add*, which is obtained from p_1^1 and f_3 by primitive recursion.

$$Mult(x, 0) = 0 = C_0^1(x)$$
$$Mult(x, k + 1) = x * k + x = Add(x, Mult(x, k))$$

The first of the two arguments in the last expression is $p_1^3(x, k, Mult(x, k))$ and the second is $p_3^3(x, k, Mult(x, k))$. Using this approach, we might begin our derivation with the derivation of *Add*, continue with C_0^1 and the function f obtained from *Add*, p_1^3, and p_3^3 by composition, and finish with *Mult*, obtained from C_0^1 and f by primitive recursion.

The "proper subtraction" function *Sub* is defined by

$$Sub(x, y) = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$$

Just as we showed that *Add* is primitive recursive by using the successor function, we show *Sub* is by using a predecessor function *Pred*:

$$Pred(x) = \begin{cases} 0 & \text{if } x = 0 \\ x - 1 & \text{if } x \geq 1 \end{cases}$$

Pred is primitive recursive because of the formulas

$$Pred(0) = 0 \qquad Pred(k + 1) = k$$

and now we can use the formula

$$Sub(x, 0) = x$$
$$Sub(x, k + 1) = Pred(Sub(x, k))$$

to show that *Sub* is primitive recursive. The proper subtraction operation is often written $\dot{-}$, with a dot above the minus sign, and another name for it is the *monus* operation.

The two functions *Pred* and *Sub* have both been defined by considering two cases. *Sub(x,y)*, for example, is defined one way if some condition $P(x, y)$ is true and another way if it is false. In general an *n-place* predicate P is a function, or partial function, from $\mathcal{N}^n$ to { true, false }, and the corresponding numeric function is the *characteristic function* χ_P defined by

$$\chi_P(X) = \begin{cases} 1 & \text{if } P(X) \text{ is true} \\ 0 & \text{if } P(X) \text{ is false} \end{cases}$$

It makes sense to say that the predicate P is primitive recursive, or that it is computable, if the function χ_P is.

Theorem 10.6 asserts that the set of primitive recursive predicates includes the common relational predicates, such as $<$ and $\neq$, and is closed under the logical operations AND, OR, and NOT. Theorem 10.7 says that more general definitions by cases, involving two or more primitive recursive predicates, also produce primitive recursive functions.

Theorem 10.6

The two-place predicates *LT*, *EQ*, *GT*, *LE*, *GE*, and *NE* are primitive recursive. (*LT* stands for "less than," and the other five have similarly intuitive abbreviations.) If P and Q are any primitive recursive n-place predicates, then $P \wedge Q$, $P \vee Q$, and $\neg P$ are primitive recursive.

Proof

The second statement follows from the equations

$$\chi_{P_1 \wedge P_2} = \chi_{P_1} * \chi_{P_2}$$
$$\chi_{P_1 \vee P_2} = \chi_{P_1} + \chi_{P_2} \dot{-} \chi_{P_1 \wedge P_2}$$
$$\chi_{(\neg P_1)} = 1 \dot{-} \chi_{P_1}$$

For the first statement, we introduce the function $Sg: \mathcal{N} \to \mathcal{N}$ defined by

$$Sg(0) = 0 \qquad Sg(k+1) = 1$$

This function takes the value 0 if $x = 0$ and the value 1 otherwise, and its definition makes it clear that it is primitive recursive. We may write

$$\chi_{LT}(x, y) = Sg(y \dot{-} x)$$

which implies that χ_{LT} is obtained from primitive recursive functions by composition. The result for the equality predicate follows from the formula

$$\chi_{EQ}(x, y) = 1 \dot{-} (Sg(x \dot{-} y) + Sg(y \dot{-} x))$$

(If $x < y$ or $x > y$, then one of the terms $x \dot{-} y$ and $y \dot{-} x$ is nonzero, and the expression in parentheses is nonzero, causing the final result to be 0. If $x = y$, both terms in the parenthesized expression are 0, and the final result is 1.)

The other four relational predicates can be handled similarly, but we can also use the second statement in the theorem along with the formulas

$$LE = LT \vee EQ$$
$$GT = \neg LE$$
$$GE = \neg LT$$
$$NE = \neg EQ$$

If P is an n-place predicate and $f_1, f_2, \ldots, f_n : \mathcal{N}^k \to \mathcal{N}$, we can form the k-place predicate $Q = P(f_1, \ldots, f_n)$, and the characteristic function χ_Q is obtained from χ_P and $f_1, \ldots, f_n$ by composition. Using Theorem 10.6, we see that arbitrarily complicated predicates constructed using relational and logical operators, such as

$$(f_1 = (3f_2)^2 \wedge (f_3 < f_4 + f_5)) \vee \neg(P \vee Q)$$

are primitive recursive as long as the basic constituents (in this case, the functions $f_1, \ldots, f_5$ and the predicates P and Q) are.

Theorem 10.7

Suppose $f_1, f_2, \ldots, f_k$ are primitive recursive functions from $\mathcal{N}^m$ to $\mathcal{N}$, $P_1, P_2, \ldots, P_k$ are primitive recursive n-place predicates, and for every $X \in \mathcal{N}^n$, exactly one of the conditions $P_1(X), \ldots, P_k(X)$ is true. Then the function $f : \mathcal{N}^n \to \mathcal{N}$ defined by

$$f(X) = \begin{cases} f_1(X) & \text{if } P_1(X) \text{ is true} \\ f_2(X) & \text{if } P_2(X) \text{ is true} \\ \cdots & \\ f_k(X) & \text{if } P_k(X) \text{ is true} \end{cases}$$

is primitive recursive.

Proof

The last of the three assumptions guarantees that f is unambiguously defined and that

$$f = f_1 * \chi_{P_1} + f_2 * \chi_{P_2} + \cdots + f_k * \chi_{P_k}$$

The result follows from the fact that all the functions appearing on the right side of this formula are primitive recursive.

The *Mod* and *Div* Functions **EXAMPLE 10.8**

For natural numbers x and y with $y > 0$, we denote by $Div(x, y)$ and $Mod(x, y)$ the integer quotient and remainder, respectively, when x is divided by y. For example, $Div(8, 5) = 1$, $Mod(8, 5) = 3$, and $Mod(12, 4) = 0$. Unless we allow division by 0, these are not total functions; let us say that for any x, $Div(x, 0) = 0$ and $Mod(x, 0) = x$. Then the usual formula

$$x = y * Div(x, y) + Mod(x, y)$$

still holds for every x and y, and

$$0 \leq Mod(x, y) < y$$

is true as long as $y > 0$.

We begin by showing that *Mod* is primitive recursive. The derivation involves recursion in the *first* variable, and for this reason we let

$$R(x, y) = Mod(y, x)$$

In order to show that *Mod* is primitive recursive, it is sufficient to show that R is, because *Mod* can be obtained from R, p_2^2, and p_1^2 by composition. The following formulas can be verified easily.

$$\begin{aligned} R(x, 0) &= Mod(0, x) \;=\; 0 \\ R(x, k+1) &= Mod(k+1, x) \\ &= \begin{cases} R(x, k) + 1 & \text{if } x \neq 0 \text{ and } R(x, k) + 1 < x \\ 0 & \text{if } x \neq 0 \text{ and } R(x, k) + 1 = x \\ k + 1 & \text{if } x = 0 \end{cases} \end{aligned}$$

For example,

$$R(5, 6 + 1) = Mod(7, 5) = Mod(6, 5) + 1$$

since $5 \neq 0$ and $Mod(6, 5) + 1 = 1 + 1 < 5$, and

$$R(5, 9 + 1) = Mod(10, 5) = 0$$

since $5 \neq 0$ and $Mod(9, 5) + 1 = 4 + 1 = 5$. The function h defined by

$$h(x_1, x_2, x_3) = \begin{cases} x_3 + 1 & \text{if } x_1 \neq 0 \text{ and } x_3 + 1 < x_1 \\ 0 & \text{if } x_1 \neq 0 \text{ and } x_3 + 1 = x_1 \\ x_2 + 1 & \text{if } x_1 = 0 \end{cases}$$

is not a total function, since it is undefined if $x_1 \neq 0$ and $x_3 + 1 > x_1$. However, the modification

$$h(x_1, x_2, x_3) = \begin{cases} x_3 + 1 & \text{if } x_1 \neq 0 \text{ and } x_3 + 1 < x_1 \\ 0 & \text{if } x_1 \neq 0 \text{ and } x_3 + 1 \geq x_1 \\ x_2 + 1 & \text{if } x_1 = 0 \end{cases}$$

works just as well. The function R is obtained by primitive recursion from C_0^1 and this modified h, and Theorem 10.7 implies that h is primitive recursive. Therefore, so are R and *Mod*.

The function *Div* can now be handled in a similar way. If we define $Q(x, y)$ to be $Div(y, x)$, then it is not hard to check that Q is obtained by primitive recursion from C_0^1

and the primitive recursive function h_1 defined by

$$h_1(x_1, x_2, x_3) = \begin{cases} x_3 & \text{if } x_1 \neq 0 \text{ and } Mod(x_2, x_1) + 1 < x_1 \\ x_3 + 1 & \text{if } x_1 \neq 0 \text{ and } Mod(x_2, x_1) + 1 = x_1 \\ 0 & \text{if } x_1 = 0 \end{cases}$$

(Note that for any choice of (x_1, x_2, x_3), precisely one of the predicates appearing in this definition is true.)

10.2 QUANTIFICATION, MINIMALIZATION, AND μ-RECURSIVE FUNCTIONS

The operations that can be applied to predicates to produce new ones include not only logical operations such as AND, but also universal and existential quantifiers. A simple example is provided by the two-place predicate Sq defined as follows:

$$Sq(x, y) = (y^2 = x)$$

Applying the quantifier "there exists" to the second variable produces the one-place predicate *PerfectSquare*, defined by

$$PerfectSquare(x) = (\text{there exists } y \text{ with } y^2 = x)$$

For a second example, suppose that for $x \in \mathcal{N}$, s_x denotes the xth element of $\{0, 1\}^*$ with respect to canonical order, and let T_u be a universal Turing machine. We can consider the two-place predicate $H(x, y)$ defined by

$$H(x, y) = (T_u \text{ halts after exactly } y \text{ moves on input } s_x)$$

and its existential quantification

$$Halts(x) = (\text{there exists } y \text{ such that } T_u \text{ halts after } y \text{ moves on input } s_x)$$

The predicate Sq is primitive recursive, and we will see later in this chapter that H is too. In any case, both are computable. The predicate *Halts* is certainly not computable, because if we could decide for every x whether $Halts\ (x)$ is true, we would be able to decide the halting problem (see Section 9.2). Therefore, the operation of quantification does not preserve either computability or primitive recursiveness.

However, the two-place predicate E_H defined by

$$E_H(x, k) = (\text{there exists } y \leq k \text{ such that } T_u \text{ halts after } y \text{ moves on input } s_x)$$

is computable, and we will see shortly that this restricted type of quantification also preserves primitive recursiveness. In a similar way, we let

$$E_{Sq}(x, k) = (\text{there exists } y \leq k \text{ such that } y^2 = x)$$

E_{Sq} is also primitive recursive.

There is an important difference between these two examples. For every natural number n, $n \leq n^2$. If we determine that $Sq(x, y)$ is false for every y with $y \leq x$, then any y satisfying $y^2 = x$ would have to be larger than its square; therefore,

PerfectSquare(x) is false. In other words, the predicate *PerfectSquare* is exactly the same as the primitive recursive predicate B defined by $B(x) = E_{Sq}(x, x)$. We have already noticed that *Halts* is not computable. There is no k, or even a function $k(x)$, such that for every x, T_u halts on input s_x if and only if it halts on input s_x within k moves. The predicate *Halts* illustrates the fact that if the simple algorithm that comes with such a bound is not available, there may be no algorithm at all.

Definition 10.9 Bounded Quantifications

Let P be an $(n + 1)$-place predicate. The *bounded existential quantification* of P is the $(n + 1)$-place predicate E_P defined by

$$E_P(X, k) = \text{(there exists } y \text{ with } 0 \leq y \leq k \text{ such that } P(X, y) \text{ is true)}$$

The *bounded universal quantification* of P is the $(n + 1)$-place predicate A_P defined by

$$A_P(X, k) = \text{(for every } y \text{ satisfying } 0 \leq y \leq k,\ P(X, y) \text{ is true)}$$

Theorem 10.10

If P is a primitive recursive $(n + 1)$-place predicate, both the predicates E_P and A_P are also primitive recursive.

Proof

Introducing two other "bounded operations" will help to simplify the proof. If $n \geq 0$ and $g : \mathcal{N}^{n+1} \to \mathcal{N}$ is primitive recursive, then we define the functions $f_1, f_2 : \mathcal{N}^{n+1} \to \mathcal{N}$, obtained from g by *bounded sums* and *bounded products*, respectively, as follows. For every $X \in \mathcal{N}^n$ and $k \geq 0$,

$$f_1(X, k) = \sum_{i=0}^{k} g(X, i)$$

$$f_2(X, k) = \prod_{i=0}^{k} g(X, i)$$

The bounded product is a natural generalization of the factorial function, which is obtained by taking $n = 0$ and by letting the function g have the value 1 when $i = 0$ and the value i when $i > 0$. We could also consider more general sums and products that begin with the $i = i_0$ term, for any fixed i_0 (see Exercise 10.27).

We can write

$$f_1(X, 0) = g(X, 0)$$

$$f_1(X, k + 1) = f_1(X, k) + g(X, k + 1)$$

Therefore, f_1 is obtained by primitive recursion from the two primitive recursive functions g_1 and h, where $g_1(X) = g(X, 0)$ and $h(X, y, z) = z + g(X, y + 1)$. A very similar argument shows that f_2 is also primitive recursive.

By definition of bounded universal quantification, $A_P(X, k)$ is true if and only if $P(X, i)$ is true for every i with $0 \leq i \leq k$. Therefore, $\chi_{A_P}(X, k) = 1$ if and only if all the terms $\chi_P(X, i)$ are also 1. This equivalence implies that

$$\chi_{A_P}(X, k) = \prod_{i=0}^{k} \chi_P(X, i)$$

and therefore that A_P is primitive recursive.

Saying that there is an i such that $0 \leq i \leq k$ and $P(X, i)$ is true is the same as saying that $P(X, i)$ is not always false for these i's. In other words,

$$E_P(X, k) = \neg A_{\neg P}(X, k)$$

It follows from this formula that E_P is also primitive recursive.

The bounded quantifications in this section preserve primitive recursiveness and computability, and the unbounded versions don't. In order to characterize the computable functions as those obtained by starting with initial functions and applying certain operations, we need at least one operation that preserves computability but not primitive recursiveness—because the initial functions are primitive recursive, and not all computable functions are. The operation of *minimalization* turns out to have this feature. It also has a bounded version as well as an unbounded one.

For an $(n + 1)$-place predicate P, and a given $X \in \mathcal{N}^n$, we may consider the smallest value of y for which $P(X, y)$ is true. To turn this operation into a bounded one, we specify a value of k and ask for the smallest value of y that is less than or equal to k and satisfies $P(X, y)$. There may be no such y (whether or not we bound the possible choices by k); therefore, because we want the bounded version of our function to be total, we introduce an appropriate default value for the function in this case.

Definition 10.11 Bounded Minimalization

For an $(n + 1)$-place predicate P, the *bounded minimalization* of P is the function $m_P : \mathcal{N}^{n+1} \to \mathcal{N}$ defined by

$$m_P(X, k) = \begin{cases} \min \{y \mid 0 \leq y \leq k \text{ and } P(X, y)\} & \text{if this set is not empty} \\ k + 1 & \text{otherwise} \end{cases}$$

The symbol μ is often used for the minimalization operator, and we sometimes write

$$m_P(X, k) = \overset{k}{\mu}\, y[P(X, y)]$$

An important special case is that in which $P(X, y)$ is $(f(X, y) = 0)$, for some $f : \mathcal{N}^{n+1} \to \mathcal{N}$. In this case m_P is written m_f and referred to as the bounded minimalization of f.

Theorem 10.12

If P is a primitive recursive $(n + 1)$-place predicate, its bounded minimalization m_P is a primitive recursive function.

Proof

We show that m_P can be obtained from primitive recursive functions by the operation of primitive recursion. For $X \in \mathcal{N}^n$, $m_P(X, 0)$ is 0 if $P(X, 0)$ is true and 1 otherwise. In order to evaluate $m_P(X, k + 1)$, we consider three cases. First, if there exists $y \leq k$ for which $P(X, y)$ is true, then $m_P(X, k + 1) = m_P(X, k)$. Second, if there is no such y but $P(X, k + 1)$ is true, then $m_P(X, k + 1) = k + 1$. Third, if neither of these conditions holds, then $m_P(X, k + 1) = k + 2$. It follows that the function m_P can be obtained by primitive recursion from the functions g and h, where

$$g(X) = \begin{cases} 0 & \text{if } P(X, 0) \text{ is true} \\ 1 & \text{otherwise} \end{cases}$$

$$h(X, y, z) = \begin{cases} z & \text{if } E_P(X, y) \text{ is true} \\ y + 1 & \text{if } \neg E_P(X, y) \wedge P(X, y + 1) \text{ is true} \\ y + 2 & \text{if } \neg E_P(X, y) \wedge \neg P(X, y + 1) \text{ is true} \end{cases}$$

Because P and E_P are primitive recursive predicates, the functions g and h are both primitive recursive.

The nth Prime Number **EXAMPLE 10.13**

For $n \geq 0$, let $PrNo(n)$ be the nth prime number: $PrNo(0) = 2$, $PrNo(1) = 3$, $PrNo(2) = 5$, and so on. Let us show that the function $PrNo$ is primitive recursive.

First we observe that the one-place predicate $Prime$, defined by

$$Prime(n) = (n \geq 2) \wedge \neg(\text{there exists } y \text{ such that } y \geq 2 \wedge y \leq n \dot{-} 1 \wedge Mod(n, y) = 0)$$

is primitive recursive, and $Prime(n)$ is true if and only if n is a prime.

For every k, $PrNo(k + 1)$ is the smallest prime greater than $PrNo(k)$. Therefore, if we can just place a bound on the set of integers greater than $PrNo(k)$ that may have to be tested in order to find a prime, then we can use the bounded minimalization operator to obtain

PrNo by primitive recursion. The number-theoretic fact that makes this possible was proved in Example 1.4: For every positive integer m, there is a prime greater than m and no larger than $m! + 1$. (If we required m to be 3 or bigger, we could say there is a prime between m and $m!$.)

With this in mind, let

$$P(x, y) = (y > x \wedge Prime(y))$$

Then

$$PrNo(0) = 2$$
$$PrNo(k + 1) = m_P(PrNo(k),\ PrNo(k)! + 1)$$

We have shown that *PrNo* can be obtained by primitive recursion from the two functions C_2^0 and h, where

$$h(x, y) = m_P(y, y! + 1)$$

Therefore, *PrNo* is primitive recursive.

In defining $m_P(X, k)$, the value of the bounded minimalization of a predicate P at the point (X, k), we specify the default value $k + 1$ if there are no values of y in the range $0 \leq y \leq k$ for which $P(X, y)$ is true. Something like this is necessary if P is a total function and we want m_P to be total. If we want the minimalization to be unbounded, and we want this operator to preserve computability, a default value is no longer appropriate: If there *is* a value k such that $P(X, k)$ is true, there is no doubt that we can find the smallest one, but it might be impossible to determine that $P(X, y)$ is false for every y and that the default value is therefore the right one. Forcing the minimalization to be a total function might make it uncomputable; allowing its domain to be smaller guarantees that it will be computable.

Definition 10.14 Unbounded Minimalization

If P is an $(n + 1)$-place predicate, the *unbounded minimalization* of P is the partial function $M_P : \mathcal{N}^n \to \mathcal{N}$ defined by

$$M_P(X) = min\ \{y \mid P(X, y) \text{ is true}\}$$

$M_P(X)$ is undefined at any $X \in \mathcal{N}^n$ for which there is no y satisfying $P(X, y)$.

The notation $\mu y[P(X, y)]$ is also used for $M_P(X)$. In the special case in which $P(X, y) = (f(X, y) = 0)$, we write $M_P = M_f$ and refer to this function as the unbounded minimalization of f.

The fact that we want M_P to be a computable partial function for any computable predicate P also has another consequence. Suppose the algorithm we are relying on for computing $M_P(X)$ is simply to evaluate $P(X, y)$ for increasing values of y, and that $P(X, y_0)$ is undefined. Although there might be a value $y_1 > y_0$

for which $P(X, y_1)$ is true, we will never get around to considering $P(X, y_1)$ if we get stuck in an infinite loop while trying to evaluate $P(X, y_0)$. In order to avoid this problem, we stipulate that unbounded minimalization should be applied only to total predicates or total functions.

Unbounded minimalization is the last of the operations we need in order to characterize the computable functions. In the definition below, this operator is applied only to predicates defined by some numeric function being zero.

Definition 10.15 μ-Recursive Functions

The set $\mathcal{M}$ of μ-recursive, or simply *recursive*, partial functions is defined as follows.

1. Every initial function is an element of $\mathcal{M}$.
2. Every function obtained from elements of $\mathcal{M}$ by composition or primitive recursion is an element of $\mathcal{M}$.
3. For every $n \geq 0$ and every total function $f : \mathcal{N}^{n+1} \to \mathcal{N}$ in $\mathcal{M}$, the function $M_f : \mathcal{N}^n \to \mathcal{N}$ defined by

$$M_f(X) = \mu y[f(X, y) = 0]$$

is an element of $\mathcal{M}$.

Just as in the case of primitive recursive functions, a function is in the set $\mathcal{M}$ if and only if it has a finite, step-by-step derivation, where at each step either a new initial function is introduced or one of the three operations is applied to initial functions, to functions obtained earlier in the derivation, or to both. As long as unbounded minimalization is not used, the function obtained at each step in such a sequence is primitive recursive. Once unbounded minimization appears in the sequence, the functions may cease to be primitive recursive or even total. If f is obtained by composition or primitive recursion, it is possible for f to be total even if not all the functions from which it is obtained are. Therefore, it is conceivable that in the derivation of a μ-recursive function, unbounded minimalization can be used more than once, even if its first use produces a nontotal function. However, in the proof of Theorem 10.20 we show that every μ-recursive function actually has a derivation in which unbounded minimalization is used only once.

Theorem 10.16

All μ-recursive partial functions are computable.

Proof

For a proof using structural induction, it is sufficient to show that if $f : \mathcal{N}^{n+1} \to \mathcal{N}$ is a computable total function, then its unbounded minimalization M_f is a computable partial function.

If T_f is a Turing machine computing f, a TM T computing M_f operates on an input X by computing $f(X, 0)$, $f(X, 1)$, ..., until it discovers an i for which $f(X, i) = 0$, and halting in h_a with i as its output. If there is no such i, the computation continues forever, which is acceptable because $M_f(X)$ is undefined in that case.

10.3 | GÖDEL NUMBERING

In the 1930s, the logician Kurt Gödel developed a method of "arithmetizing" a formal axiomatic system by assigning numbers to statements and formulas, so as to be able to describe relationships between objects in the system using relationships between the corresponding numbers. His ingenious use of these techniques led to dramatic and unexpected results about logical systems; Gödel's *incompleteness theorem* says, roughly speaking, that any formal system comprehensive enough to include the laws of arithmetic must, if it is consistent, contain true statements that cannot be proved within the system.

The numbering schemes introduced by Gödel have proved to be useful in a number of other settings. We will use this approach to arithmetize Turing machines—to describe operations of TMs, and computations performed by TMs, in terms of purely numeric operations. As a result, we will be able to show that every Turing-computable function is μ-recursive.

Most Gödel-numbering techniques depend on a familiar fact about positive integers: Every positive integer can be factored as a product of primes (1 is the *empty* product), and this factorization is unique except for differences in the order of the factors. Because describing a Turing machine requires describing a sequence of tape symbols, we start by assigning Gödel numbers to sequences of natural numbers.

Definition 10.17 The Gödel Number of a Sequence of Natural Numbers

For every $n \geq 1$ and every finite sequence $x_0, x_1, \ldots, x_{n-1}$ of n natural numbers, the *Gödel number* of the sequence is the number

$$gn(x_0, x_1, \ldots, x_{n-1}) = 2^{x_0}3^{x_1}5^{x_2} \ldots (PrNo(n-1))^{x_{n-1}}$$

where $PrNo(i)$ is the ith prime (Example 10.13).

The Gödel number of every sequence is positive, and every positive integer is the Gödel number of a sequence. The function gn is not one-to-one; for example,

$$gn(0, 2, 1) = gn(0, 2, 1, 0, 0) = 2^0 3^2 5^1$$

However, if $gn(x_0, x_1, \ldots, x_m) = gn(y_0, y_1, \ldots, y_m, y_{m+1} \ldots y_{m+k})$, then

$$\prod_{i=0}^{m} PrNo(i)^{x_i} = \prod_{i=0}^{m} PrNo(i)^{y_i} \prod_{i=m+1}^{m+k} PrNo(i)^{y_i}$$

and because a number can have only one prime factorization, we must have $y_i = x_i$ for $0 \leq i \leq m$ and $y_i = 0$ for $i > m$. Therefore, two sequences having the same Gödel number and ending with the same number of 0's are identical. In particular, for every $n \geq 1$, every positive integer is the Gödel number of at most one sequence of n integers.

For every n, the Gödel numbering we have defined for sequences of length n determines a function from $\mathcal{N}^n$ to $\mathcal{N}$. We will be imprecise and use the name gn for any of these functions. All of them are primitive recursive.

If we start with a positive integer g, we can *decode* g to find a sequence x_0, $x_1, \ldots, x_n$ whose Gödel number is g by factoring g into primes. For each i, x_i is the number of times $PrNo(i)$ appears as a factor of g. For example, the number 59895 has the prime factorization

$$59895 = 3^2 5^1 11^3 = 2^0 3^2 5^1 7^0 11^3$$

and is therefore the Gödel number of the sequence 0,2,1,0,3 or any other sequence obtained from this by adding extra 0's. The prime number 31 is the Gödel number of the sequence 0,0,0,0,0,0,0,0,0,0,1, since $31 = PrNo(10)$. This type of calculation will be needed often enough that we introduce a function just for this purpose.

The Power to Which a Prime Is Raised in the Factorization of x — EXAMPLE 10.18

The function $Exponent: \mathcal{N}^2 \to \mathcal{N}$ is defined as follows:

$$Exponent(i, x) = \begin{cases} \text{the exponent of } PrNo(i) \text{ in } x\text{'s prime factorization} & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

For example, $Exponent(4, 59895) = 3$, because the fourth prime, 11, appears three times as a factor of 59895. (Remember, 2 is the *zero*th prime.)

We can show that *Exponent* is primitive recursive by expressing $Exponent(i, x)$ in a way that involves bounded minimalization. In order to make the formulas look less intimidating, let us temporarily use the notation

$$M(x, i, y) = Mod(x, PrNo(i)^y)$$

Saying that $PrNo(i)^y$ divides x evenly is the same as saying that $M(x, i, y) = 0$, and this is true if and only if $y \leq Exponent(i, x)$. Therefore, if $y = Exponent(i, x) + 1$, then $M(x, i, y) > 0$, and this is the smallest value of y for which $M(x, i, y) > 0$. In other words,

$$Exponent(i, x) + 1 = \mu y[M(x, i, y) > 0]$$

or

$$Exponent(i, x) = \mu y[M(x, i, y) > 0] \dot{-} 1$$

However, if we want to compute $Exponent(i, x)$ this way, by searching for the smallest y satisfying $M(x, i, y) > 0$, we don't have to consider any values bigger than x, because x is already such a value: We know that

$$PrNo(i)^x > x$$

so that

$$M(x, i, y) = Mod(x, PrNo(i)^y) = x > 0$$

Therefore,

$$Exponent(i, x) = \overset{x}{\mu}\, y[Mod(x, PrNo(i)^y) > 0] \dot{-} 1$$

All the operations involved in the right-hand expression preserve primitive recursiveness, and it follows that *Exponent* is primitive recursive.

For many functions that are defined recursively, the definition does not make it obvious that the function is obtained by using primitive recursion. For example, the right side of the formula

$$f(n+1) = f(n) + f(n-1)$$

is not of the form $h(n, f(n))$, since it also depends on $f(n-1)$. In general $f(n+1)$ might depend on several, or even all, of the terms $f(0), f(1), \ldots, f(n)$. This type of recursion is known as *course-of-values* recursion, and it is related to primitive recursion in the same way that strong induction (Example 1.24) is related to ordinary mathematical induction.

Gödel numbering provides us with a way to reformulate definitions like these. Suppose $f(n+1)$ depends on some or all of the numbers $f(0), \ldots, f(n)$, and possibly also directly on n. In order to obtain f by using primitive recursion, we would like another function f_1 for which

1. Knowing $f_1(n)$ would allow us to calculate $f(n)$.
2. $f_1(n+1)$ depends only on n and $f_1(n)$.

If we could relax the requirement that $f_1(n)$ be a *number*, we could consider the entire sequence $f_1(n) = (f(0), f(1), \ldots, f(n))$. Condition 1 is satisfied, because $f(n)$ is simply the last term of the sequence $f_1(n)$. Since $f(n+1)$ can be expressed in terms of n and $f(0), \ldots, f(n)$, the sequence $(f(0), \ldots, f(n), f(n+1))$ can be said to depend only on n and the sequence $(f(0), \ldots, f(n))$, so that we also have condition 2. To make this intuitive idea work, all we need to do is to use the Gödel numbers of the sequences, instead of the sequences themselves. Rather than saying that $f(n+1)$ depends on $f(0), \ldots, f(n)$, we say that $f(n+1)$ depends on the single value $gn(f(0), \ldots, f(n))$. The two versions are intuitively equivalent, since there is enough information in $gn(f(0), \ldots, f(n))$ to find each number $f(i)$.

Theorem 10.19

Suppose that $g : \mathcal{N}^n \to \mathcal{N}$ and $h : \mathcal{N}^{n+2} \to \mathcal{N}$ are primitive recursive functions, and $f : \mathcal{N}^{n+1} \to \mathcal{N}$ is obtained from g and h by course-of-values recursion; that is,

$$f(X, 0) = g(X)$$
$$f(X, k+1) = h(X, k, gn(f(X, 0), \ldots, f(X, k)))$$

Then f is primitive recursive.

Proof

First we define $f_1 : \mathcal{N}^{n+1} \to \mathcal{N}$ by the formula

$$f_1(X, k) = gn(f(X, 0), f(X, 1), \ldots, f(X, k))$$

Then f can be obtained from f_1 by the formula

$$f(X, k) = Exponent(k, f_1(X, k))$$

Therefore, it will be sufficient to show that f_1 is primitive recursive.

We have

$$\begin{aligned} f_1(X, 0) &= gn(f(X, 0)) = 2^{f(X,0)} = 2^{g(X)} \\ f_1(X, k+1) &= \prod_{i=0}^{k+1} PrNo(i)^{f(X,i)} \\ &= \prod_{i=0}^{k} PrNo(i)^{f(X,i)} * PrNo(k+1)^{f(X,k+1)} \\ &= f_1(X, k) * PrNo(k+1)^{h(X,k,f_1(X,k))} \\ &= h_1(X, k, f_1(X, k)) \end{aligned}$$

where the function h_1 is defined by

$$h_1(X, y, z) = z * PrNo(y+1)^{h(X,y,z)}$$

Because 2^g and h_1 are primitive recursive and f_1 is obtained from these two by primitive recursion, f_1 is also primitive recursive.

Now we are ready to apply our Gödel numbering techniques to Turing machines. A TM move can be interpreted as a transformation of one TM configuration to another, and we need a way of characterizing a Turing machine configuration by a number.

We begin by assigning a number to each state. The halt states h_a and h_r are assigned the numbers 0 and 1, and the elements of the state set Q will be assigned the numbers 2, 3, . . . , s, with 2 representing the initial state. The tape head position is simply the number of the current tape square, and we assign numbers to the tape symbols by using 0 for the blank symbol Δ, and 1, 2, . . . , t for the distinct nonblank tape symbols. Now we can define the current *tape number* of the TM to be the Gödel number of the sequence of symbols currently on the tape. Because we are identifying Δ with 0, the tape number does not depend on how many blanks we include at the end of this sequence. The number of a blank tape is 1.

The configuration of a TM is determined by the state, tape head position, and current contents of the tape, and we define the *configuration number* to be the number

$$gn(q, P, tn)$$

where q is the number of the current state, P is the current head position, and tn is the current tape number. The most important feature of the configuration number is that from it we can reconstruct all the details of the configuration; we will be more explicit about this in the next section.

10.4 ALL COMPUTABLE FUNCTIONS ARE μ-RECURSIVE

Suppose that $f : \mathcal{N}^n \to \mathcal{N}$ is a partial function computed by the TM T. In order to compute $f(X)$ for an n-tuple X, T starts in the standard initial configuration corresponding to X, carries out a sequence of moves, and ends up in an accepting configuration, if X is in the domain of f, with a string representing $f(X)$ on the tape. At the end of the last section we introduced configuration numbers for T, which are Gödel numbers of configurations; we can interpret each move of the computation as transforming one configuration number to another, and if the computation halts, we can interpret the entire computation as transforming an initial configuration number c to the resulting halting-configuration number $f_T(c)$.

We can therefore write

$$f(X) = Result_T(f_T(InitConfig^{(n)}(X)))$$

where f_T is the function described in the first paragraph; $InitConfig^{(n)}(X)$ is the Gödel number of the initial configuration corresponding to the n-tuple X; and $Result_T : \mathcal{N} \to \mathcal{N}$ assigns to j the number y if j represents an accepting configuration of T in which output corresponding to y is on the tape, and an appropriate default value otherwise.

In order to show that f is μ-recursive, it will be sufficient to show that each of the three functions on the right side of this formula is. The two outer ones, $Result_T$ and $InitConfig^{(n)}$, are actually primitive recursive. That is too much to expect of f_T, which is defined in a way that involves unbounded minimalization and is defined only at configuration numbers corresponding to n-tuples in the domain of f.

The list below itemizes these three, as well as several auxiliary functions that will be useful in the description of f_T.

1. The function $InitConfig^{(n)} : \mathcal{N}^n \to \mathcal{N}$ does not actually depend on the Turing machine, because of our assumption that the initial state of a TM is always given the number 2. The Gödel number of the initial configuration depends on the Gödel number of the initial tape; in order to show that $InitConfig^{(n)}$ is primitive recursive, it is sufficient to show that $t^{(n)} : \mathcal{N}^n \to \mathcal{N}$ is, where $t^{(n)}(x_1, \ldots, x_n)$ is the tape number of the tape containing the string

$$\Delta 1^{x_1} \Delta 1^{x_2} \Delta \ldots \Delta 1^{x_n}$$

The proof is by mathematical induction on n and is left to the exercises.

The function $Result_T$ is one of several in this discussion whose value at m will depend on whether m is the number of a configuration of T. The next step, therefore, is to consider two related predicates.

2. $IsConfig_T$ is the one-place predicate defined by

$$IsConfig_T(n) = (n \text{ is a configuration number for } T)$$

and $IsAccepting_T$ is defined by

$$IsAccepting_T(m) = \begin{cases} 0 & \text{if } IsConfig_T(m) \wedge Exponent(0, m) = 0 \\ 1 & \text{otherwise} \end{cases}$$

$IsAccepting_T(m)$ is 0 if and only if m is the number of an accepting configuration for T, and this predicate will be primitive recursive if $IsConfig_T$ is.

Let s_T be one more than the number of nonhalting states of T, which are numbered starting with 2, and let ts_T be the number of nonblank tape symbols of T. A number m is a configuration number for T if and only if

$$m = 2^q 3^p 5^{tn}$$

where $q \leq s_T$, p is arbitrary, and tn is the Gödel number of a sequence of natural numbers, each one between 0 and ts_T.

The statement that m is of the general form $2^a 3^b 5^c$ can be expressed by saying that

$$(m \geq 1) \wedge (\text{for every } i,\ i \leq 2 \vee Exponent(i, m) = 0)$$

For numbers m of this form, the conditions on q and tn are equivalent to the statement

$$(Exponent(0, m) \leq s_T) \wedge (Exponent(2, m) \geq 1) \\ \wedge (\text{for every } i, Exponent(i, tn) \leq ts_T)$$

In order to show that the conjunction of these two predicates is primitive recursive, it is sufficient to show that both of the universal quantifications can be replaced by *bounded* universal quantifications. This is true because $Exponent(i, n) = 0$ when $i > n$; the first occurrence of "for every i" can be replaced by "for every $i \leq m$" and the second by "for every $i \leq tn$." Therefore, $IsConfig_T$ is primitive recursive.

3. Next, we check that the function $Result_T : \mathcal{N} \to \mathcal{N}$ is primitive recursive. Because the tape number for the configuration represented by n is $Exponent(2, n)$ and the prime factors of the tape number correspond to the squares with nonblank symbols, we may write

$$Result_T(n) = \begin{cases} HighestPrime(Exponent(2, n)) & \text{if } IsConfig_T(n) \\ 0 & \text{otherwise} \end{cases}$$

where for any positive k, $HighestPrime(k)$ is the number of the largest prime factor of k, and $HighestPrime(0) = 0$ (e.g., $HighestPrime(2^3 5^5 19^2) = 7$, because 19 is $PrNo(7)$). It is not hard to see that the function $HighestPrime$ is primitive recursive (Exercise 10.22), and it follows that $Result_T$ is.

We are ready to consider f_T, the numeric function that represents the processing done by T. Without loss of generality, we can make the simplifying assumption that T never attempts to move its tape head left from square 0.

4. The current state, tape head position, tape symbol, and tape number can all be calculated from the configuration number. The formulas are

$$\begin{aligned} State(m) &= Exponent(0, m) \\ Posn(m) &= Exponent(1, m) \\ TapeNumber(m) &= Exponent(2, m) \\ Symbol(m) &= Exponent(Posn(m)), TapeNumber(m)) \end{aligned}$$

for every m that is a configuration number for T, and 0 otherwise. Because $IsConfig_T$ is a primitive recursive predicate, all four functions are primitive recursive.

A single move is determined by considering cases, and may result in new values for these four quantities. We have the corresponding functions *NewState*, *NewPosn*, *NewTapeNumber*, and *NewSymbol*. $NewState(m)$, for example, is 0 if m is not a configuration number; it is the same as $State(m)$ if m represents a configuration from which T cannot move; and otherwise, it has a possibly different value that is determined by the ordered pair $(State(m), Symbol(m))$. All the cases can be described by primitive recursive predicates, and the resulting function is therefore primitive recursive.

The same argument applies to *NewSymbol*, and to *NewPosn* as well except that when there actually is a move, the new position may be obtained from the old by adding or subtracting 1. The *NewTapeNumber* function is slightly more complicated, partly because the new value depends on $Posn(m)$, $NewSymbol(m)$, and $Symbol(m)$ as well as the old $TapeNumber(m)$, and partly because the numeric operations required to obtain it are more complicated. The details are left to Exercise 10.35, and the conclusion is still that all four of these *New* functions are primitive recursive.

5. The function $Move_T : \mathcal{N} \to \mathcal{N}$ is defined by

$$Move_T(m) = \begin{cases} gn(NewState(m), NewPosn(m), NewTapeNum(m)) \\ \quad \text{if } IsConfig_T(m) \\ \\ 0 \quad \text{otherwise} \end{cases}$$

If m represents a configuration of T from which T can move, then $Move_T(m)$ represents the configuration after the next move; if m represents a configuration of T from which no move is possible, then $Move_T(m) = m$; and if m does not represent a configuration of T, $Move_T(m) = 0$. The function $Move_T$ is primitive recursive.

6. We can go from a single move to a sequence of k moves by considering the function $Moves_T : \mathcal{N} \to \mathcal{N}$ defined by

$$Moves_T(m, 0) = \begin{cases} m & \text{if } IsConfig_T(m) \\ 0 & \text{otherwise} \end{cases}$$

$$Moves_T(m, k+1) = \begin{cases} Move_T(Moves(m, k)) & \text{if } IsConfig_T(m) \\ 0 & \text{otherwise} \end{cases}$$

$Moves_T$ is obtained by primitive recursion from two primitive recursive functions and is therefore primitive recursive. For a configuration number m, we may describe $Moves_T(m, k)$ as the configuration number after k moves, if T starts in configuration m—or, if T is unable to make as many as k moves from configuration m, as the number of the last configuration T reaches starting from configuration m.

7. Finally, we define $NumberOfMovesToAccept_T : \mathcal{N} \to \mathcal{N}$ by the formula

$$NumberOfMovesToAccept_T(m) = \mu k[Accepting_T(Moves_T(m, k)) = 0]$$

and $f_T : \mathcal{N} \to \mathcal{N}$ by

$$f_T(m) = Moves_T(m, NumberOfMovesToAccept_T(m))$$

If m is a configuration number for T and T eventually accepts when starting from configuration m, then $NumberOfMovesToAccept_T(m)$ is the number of moves from that point before T accepts, and $f_T(m)$ is the number of the accepting configuration that is eventually reached. For any other m, both functions are undefined. $NumberOfMovesToAccept_T$ is μ-recursive because it is obtained from a primitive recursive (total) function by unbounded minimalization, and f_T is μ-recursive because it is obtained by composition from μ-recursive functions.

We have essentially proved Theorem 10.20.

Theorem 10.20

Every Turing-computable partial function from $\mathcal{N}^n$ to $\mathcal{N}$ is μ-recursive.

The Rest of the Proof Suppose the TM T computes $f : \mathcal{N}^n \to \mathcal{N}$. If $f(X)$ is defined, then when T begins in the configuration $InitConfig^{(n)}(X)$, it eventually accepts. Therefore, $f_T(InitConfig^{(n)}(X))$ is the configuration number of the accepting configuration $(h_a, \underline{\Delta}1^{f(X)})$, and when $Result_T$ is applied to this configuration number it produces $f(X)$. On the other hand, suppose that $f(X)$ is undefined. Then T fails to accept input X, and this means that $f_T(InitConfig^{(n)}(X))$ is undefined. Therefore, f is identical to the μ-recursive function

$$Result_T \circ f_T \circ InitConfig^{(n)}$$

Theorem 10.20 can be generalized. Gödel numbering lets us extend the definitions of primitive recursive and μ-recursive to functions involving strings, and it

is relatively straightforward to derive analogues of Theorems 10.16 and 10.20 in this more general setting.

10.5 OTHER APPROACHES TO COMPUTABILITY

In the last section of this chapter we mention briefly two other approaches to computable functions that turn out to be equivalent to the two approaches we have studied.

Unrestricted grammars provide a way of generating languages, and they can also be used to compute functions. If $G = (V, \Sigma, S, P)$ is a grammar, and f is a partial function from Σ^* to Σ^*, G is said to compute f if there are variables A, B, C, and D in the set V such that for every x and y in Σ^*,

$$f(x) = y \text{ if and only if } AxB \Rightarrow_G^* CyD$$

It can be shown, using arguments comparable to the ones in Section 8.3, that the functions computable in this way are the same as the ones that can be computed by Turing machines.

Computer programs can be viewed as computing functions from strings to strings. Ignoring limitations imposed by a particular programming language or implementation, and assuming that there is an unlimited amount of memory, no limit to the size of integers, and so on, functions that can be computed this way include all the Turing-computable functions.

We don't need all the features present in modern high-level languages like C to compute these functions. For numeric functions, we need to be able to perform certain basic operations: assignment statements; algebraic operations including addition and subtraction; statements that transfer control, depending on the values of certain variables; and, depending on the conventions we adopt regarding input and output, perhaps "read" and "write" statements. Even with languages this simple, it is possible to compute all Turing-computable functions. One approach to proving this would be to write a program in such a language to simulate an arbitrary TM. This might involve some sort of arithmetization of TMs similar to Gödel numbering: One integer variable would represent the state, another the head position, a third the tape contents, and so on. Another approach, comparable to Theorems 10.4 and 10.16, would be to show that the set of functions computable using the language contains the initial functions and is closed under all the operations permitted for μ-recursive functions.

The Church-Turing thesis asserts that every function that can be computed using a high-level programming language is Turing-computable. A direct proof might be carried out by building a TM that can simulate each feature of the language and can therefore execute a program written in the language. Another approach would be along the lines of Theorem 10.20. Program configurations comparable to TM configurations can be described by specifying a statement (the next statement to be executed) and the current values of all variables. Configuration numbers can be defined, and each step in the execution of the program can be viewed as

a transformation of one configuration number to another, just as in the proof of Theorem 10.20. As a result, every function computed by the program is μ-recursive.

EXERCISES

10.1. Let F be the set of partial functions from $\mathcal{N}$ to $\mathcal{N}$. Then $F = C \cup U$, where the functions in C are computable and the ones in U are not. Show that C is countable and U is not.

10.2. The *busy-beaver function* $b : \mathcal{N} \to \mathcal{N}$ is defined as follows. The value $b(0)$ is 0. For $n > 0$, there are only a finite number of Turing machines having n nonhalting states $q_0, q_1, \ldots, q_{n-1}$ and tape alphabet $\{0, 1\}$. Let $T_0, T_1, \ldots, T_m$ be the TMs of this type that eventually halt on input 1^n, and for each i, let n_i be the number of 1's that T_i leaves on its tape when it halts after processing the input string 1^n. The number $b(n)$ is defined to be the maximum of the numbers $n_0, \ldots, n_m$.

Show that the total function $b : \mathcal{N} \to \mathcal{N}$ is not computable. Suggestion: Suppose for the sake of contradiction that T_b is a TM that computes b. Then we can assume without loss of generality that T_b has tape alphabet $\{0, 1\}$.

10.3. Let $f : \mathcal{N} \to \mathcal{N}$ be the function defined as follows: $f(0) = 0$, and for $n > 0$, $f(n)$ is the maximum number of moves a TM with n non-halting states and tape alphabet $\{0, 1\}$ can make if it starts with input 1^n and eventually halts. Show that f is not computable.

10.4. Define $f : \mathcal{N} \to \mathcal{N}$ by letting $f(0)$ be 0, and for $n > 0$ letting $f(n)$ be the maximum number of 1's that a TM with n states and no more than n tape symbols can leave on the tape, assuming that it starts with input 1^n and always halts. Show that f is not computable.

10.5. Show that the uncomputability of the busy-beaver function (Exercise 10.2) implies that the halting problem is undecidable.

10.6. †Suppose we define $bb(0)$ to be 0, and for $n > 0$ we define $bb(n)$ to be the maximum number of 1's that can be printed by a TM with n states and tape alphabet $\{0, 1\}$, assuming that it starts with a blank tape and eventually halts. Show that bb is not computable.

10.7. Let $b : \mathcal{N} \to \mathcal{N}$ be the busy-beaver function in Exercise 10.2. Show that b is eventually larger than every computable function; in other words, for every computable total function $g : \mathcal{N} \to \mathcal{N}$, there is an integer k such that $b(n) > g(n)$ for every $n \geq k$.

10.8. Suppose we define $b_2(0)$ to be 0, and for $n > 0$ we define $b_2(n)$ to be the largest number of 1's that can be left on the tape of a TM with two states and tape alphabet $\{0, 1\}$, if it starts with input 1^n and eventually halts.

a. Give a convincing argument that b_2 is computable.

b. Is the function b_k (identical to b_2 except that "two states" is replaced by "k states") computable for every $k \geq 2$? Why or why not?

10.9. Show that if $f : \mathcal{N} \to \mathcal{N}$ is a total function, then f is computable if and only if the decision problem. "Given natural numbers n and C, is $f(n) > C$?" is solvable.

10.10. Suppose that instead of including all constant functions in the set of initial functions, C_0^0 is the only constant function included. Describe what the set *PR* obtained by Definition 10.3 would be.

10.11. Suppose that in Definition 10.3 the operation of composition is allowed but that of primitive recursion is not. What functions are obtained?

10.12. If $g(x) = x$ and $h(x, y, z) = z + 2$, what function is obtained from g and h by primitive recursion?

10.13. Here is a primitive recursive derivation. $f_0 = C_1^0$; $f_1 = C_0^2$; f_2 is obtained from f_0 and f_1 by primitive recursion; $f_3 = p_2^2$; f_4 is obtained from f_2 and f_3 by composition; $f_5 = C_0^0$; f_6 is obtained from f_5 and f_4 by primitive recursion; $f_7 = p_1^1$; $f_8 = p_3^3$; $f_9 = s$; f_{10} is obtained from f_9 and f_8 by composition; f_{11} is obtained from f_7 and f_{10} by primitive recursion; $f_{12} = p_1^2$; f_{12} is obtained from f_6 and f_{12} by composition; f_{14} is obtained from f_{11}, f_{12}, and f_3 by composition; and f_{15} is obtained from f_5 and f_{14} by primitive recursion.

Give simple formulas for f_2, f_6, f_{14}, and f_{15}.

10.14. Find two functions g and h such that the function f defined by $f(x) = x^2$ is obtained from g and h by primitive recursion.

10.15. Give complete primitive recursive derivations for each of the following functions.

a. $f : \mathcal{N}^2 \to \mathcal{N}$ defined by $f(x, y) = 2x + 3y$

b. $f : \mathcal{N} \to \mathcal{N}$ defined by $f(n) = n!$

c. $f : \mathcal{N} \to \mathcal{N}$ defined by $f(n) = 2^n$

d. $f : \mathcal{N} \to \mathcal{N}$ defined by $f(n) = n^2 \dot{-} 1$

e. $f : \mathcal{N}^2 \to \mathcal{N}$ defined by $f(x, y) = |x - y|$

10.16. Show that for any $n \geq 1$, the functions Add_n and $Mult_n$ from $\mathcal{N}^n$ to $\mathcal{N}$, defined by

$$Add_n(x_1, \ldots, x_n) = x_1 + x_2 + \cdots + x_n$$
$$Mult_n(x_1, \ldots, x_n) = x_1 * x_2 * \cdots * x_n$$

respectively, are both primitive recursive.

10.17. Show that if $f : \mathcal{N} \to \mathcal{N}$ is primitive recursive, $A \subseteq \mathcal{N}$ is a finite set, and g is a total function agreeing with f at every point not in A, then g is primitive recursive.

10.18. Show that if $f : \mathcal{N} \to \mathcal{N}$ is an *eventually periodic* total function, then f is primitive recursive. "Eventually periodic" means that for some n_0 and some $p > 0$, $f(x + p) = f(x)$ for every $x \geq n_0$.

10.19. Show that each of the following functions is primitive recursive.

a. $f : \mathcal{N}^2 \to \mathcal{N}$ defined by $f(x, y) = max\{x, y\}$

b. $f : \mathcal{N}^2 \to \mathcal{N}$ defined by $f(x, y) = min\{x, y\}$

c. $f : \mathcal{N} \to \mathcal{N}$ defined by $f(x) = \lfloor \sqrt{x} \rfloor$ (the largest natural number less than or equal to $\sqrt{x}$)

d. $f : \mathcal{N} \to \mathcal{N}$ defined by $f(x) = \lfloor \log_2(x + 1) \rfloor$

10.20. Suppose P is a primitive recursive $(k + 1)$-place predicate, and f and g are primitive recursive functions of one variable. Show that the predicates $A_{f,g}P$ and $E_{f,g}P$ defined by

$$A_{f,g}P(X, k) = (\text{for every } i \text{ with } f(k) \leq i \leq g(k),\ P(X, i))$$
$$E_{f,g}P(X, k) = (\text{there exists } i \text{ with } f(k) \leq i \leq g(k) \text{ such that } P(X, i))$$

are both primitive recursive.

10.21. Show that if $g : \mathcal{N}^2 \to \mathcal{N}$ is primitive recursive, then $f : \mathcal{N} \to \mathcal{N}$ defined by

$$f(x) = \sum_{i=0}^{x} g(x, i)$$

is primitive recursive.

10.22. Show that the function *HighestPrime* introduced in Section 10.4 is primitive recursive.

10.23. In addition to the bounded minimalization of a predicate, we might define the bounded maximalization of a predicate P to be the function m^P defined by

$$m^P(X, k) = \begin{cases} \max\{y \leq k \mid P(X, y) \text{ is true}\} & \text{if this set is not empty} \\ 0 & \text{otherwise} \end{cases}$$

a. Show m^P is primitive recursive by finding two primitive recursive functions from which it can be obtained by primitive recursion.

b. Show m^P is primitive recursive by using bounded minimalization.

10.24. †Consider the function f defined recursively as follows:

$$f(0) = f(1) = 1;\ \text{for } x > 0,\ f(x) = 1 + f(\lfloor \sqrt{x} \rfloor)$$

Show that f is primitive recursive.

10.25. a. Show that the function $f : \mathcal{N}^2 \to \mathcal{N}$ defined by $f(x, y) = (\text{the number of integer divisors of } x \text{ less than or equal to } y)$ is primitive recursive. Use this to show that the one-place predicate *Prime* (see Example 10.13) is primitive recursive.

b. Show that the function $f : \mathcal{N}^3 \to \mathcal{N}$ defined by $f(x, y, z) = (\text{the number of integers less than or equal to } z \text{ that are divisors of both } x \text{ and } y)$ is primitive recursive. Use this to show that the two-place predicate P defined by $P(x, y) = (x \text{ and } y \text{ are relatively prime})$ is primitive recursive.

10.26. †Show that the following functions from $\mathcal{N}$ to $\mathcal{N}$ are primitive recursive.

a. $f(n) =$ the leftmost digit in the decimal representation of 2^x

b. $f(n) =$ the nth digit of the infinite decimal expansion of $\sqrt{2} = 1.414212\ldots$ (i.e., $f(0) = 1$, $f(1) = 4$, and so on)

10.27. †Show that if $g : \mathcal{N}^{n+1} \to \mathcal{N}$ is primitive recursive, and $l, m : \mathcal{N} \to \mathcal{N}$ are both primitive recursive, then the functions f_1 and f_2 from $\mathcal{N}^{n+1}$ to $\mathcal{N}$ defined by

$$f_1(X, k) = \prod_{i=l(k)}^{m(k)} g(X, i) \qquad f_2(X, k) = \sum_{i=l(k)}^{m(k)} g(X, i)$$

are primitive recursive.

10.28. Suppose Σ is an alphabet containing all the symbols necessary to describe numeric functions, so that a primitive recursive derivation is a string over Σ. Suppose in addition that there is an algorithm capable of deciding, for every string x over Σ, whether x is a legitimate primitive recursive derivation of a function of one variable. Then in principle we can consider the strings in Σ^* in canonical order, and for each one that is a primitive recursive derivation of a function f from $\mathcal{N}$ to $\mathcal{N}$, we can include f in a list of primitive recursive functions of one variable. The resulting list $f_0, f_1, \ldots$, will contain duplicates, because functions can have more than one primitive recursive derivation, but it contains *every* primitive recursive function of one variable.

a. With these assumptions, show that there is a computable total function from $\mathcal{N}$ to $\mathcal{N}$ that is not primitive recursive.

b. Every μ-recursive function from $\mathcal{N}$ to $\mathcal{N}$ has a "μ-recursive derivation." What goes wrong when you try to adapt your argument in part (a) to show that there is a computable function from $\mathcal{N}$ to $\mathcal{N}$ that is not μ-recursive?

10.29. Give an example to show that the unbounded universal quantification of a computable predicate need not be computable.

10.30. Show that the unbounded minimalization of any predicate can be written in the form $\mu y[f(X, y) = 0]$, for some function f.

10.31. The set of μ-recursive functions was defined to be the smallest set that contains the initial functions and is closed under the operations of composition, primitive recursion, and unbounded minimalization (applied to total functions). In the definition, no explicit mention is made of the bounded operators (universal and existential quantification, bounded minimalization). Do bounded quantifications applied to μ-recursive predicates always produce μ-recursive predicates? Does bounded minimalization applied to μ-recursive predicates or functions always produce μ-recursive functions? Explain.

10.32. †Consider the following problem: Given a Turing machine T computing some partial function f, is f a total function? Is this problem decidable? Explain.

10.33. Suppose that $f : \mathcal{N} \to \mathcal{N}$ is a μ-recursive total function that is a bijection from $\mathcal{N}$ to $\mathcal{N}$. Show that its inverse f^{-1} is also μ-recursive.

10.34. Show using mathematical induction that if $t^{(n)}(x_1, \ldots, x_n)$ is the tape number of the tape containing the string

$$\Delta 1^{x_1} \Delta 1^{x_2} \Delta \ldots \Delta 1^{x_n}$$

then $t^{(n)} : \mathcal{N}^n \to \mathcal{N}$ is primitive recursive. Suggestion: In the induction step, show that

$$tn^{(k+1)}(X, x_{k+1}) = tn^{(k)}(X) * \prod_{j=1}^{x_{k+1}} PrNo(k + \sum_{i=1}^{k} x_i + j)$$

10.35. Show that the function *NewTapeNumber* discussed in Section 10.4 is primitive recursive. Suggestion: Determine the exponent e such that $TapeNumber(m)$ and $NewTapeNumber(m)$ differ by the factor $PrNo(Posn(m))^e$, and use this to express $NewTapeNumber(m)$ in terms of $TapeNumber(m)$.

CHAPTER

11

Introduction to Computational Complexity

A decision problem is decidable if there is an algorithm that can answer it in principle. In this chapter, we try to identify the problems for which there are *practical* algorithms that can answer reasonable-size instances in a reasonable amount of time. These aren't necessarily the same thing. The *satisfiability problem* is decidable, but the known algorithms aren't much of an improvement on the brute-force approach, in which exponentially many cases are considered one at a time.

The set P is the set of problems, or languages, that can be decided (by a Turing machine, or by any comparable model of computation) in *polynomial time*, as a function of the instance size. NP is defined similarly, except that the restrictions on the decision algorithms are relaxed so as to allow *nondeterministic* polynomial-time algorithms. Most people assume that NP is a larger set—that being able to guess a solution and verify it in polynomial time does not guarantee an ordinary polynomial-time algorithm—but no one has been able to demonstrate that $P \neq NP$.

Using the idea of a polynomial-time reduction, we discuss NP-complete problems, which are *hardest* problems in NP, and prove the Cook-Levin theorem, which asserts that the satisfiability problem is one of these. In the last section, we look at a few of the many other decision problems that are now known to be NP-complete.

11.1 THE TIME COMPLEXITY OF A TURING MACHINE, AND THE SET *P*

A Turing machine deciding a language $L \subseteq \Sigma^*$ can be thought of as solving a decision problem: Given $x \in \Sigma^*$, is $x \in L$? A natural measure of the size of the problem instance is the length of the input string. In the case of a decision problem with other kinds of instances, we usually expect that any integer we might use to

measure the size of the instance will be closely related to the length of the string that encodes that instance—although we will return to this issue later in this section to pin down this relationship a little more carefully.

The first step in describing and categorizing the complexity of computational problems is to define the time complexity function of a Turing machine.

Definition 11.1 The Time Complexity of a Turing Machine

Suppose T is a Turing machine with input alphabet Σ that eventually halts on every input string. The time complexity of T is the function $\tau_T : \mathcal{N} \to \mathcal{N}$, where $\tau_T(n)$ is defined by considering, for every input string of length n in Σ^*, the number of moves T makes on that string before halting, and letting $\tau_T(n)$ be the maximum of these numbers. When we refer to a TM with a certain time complexity, it will be understood that it halts on every input.

Calculating the Time Complexity of a Simple TM **EXAMPLE 11.2**

Figure 11.3 shows the transition diagram also shown in Figure 7.6, for the TM T in Example 7.5 that accepts the language $L = \{xx \mid x \in \{a, b\}^*\}$. We will derive a formula for $\tau_T(n)$ in the case when n is even, and in the other case it is smaller.

The computation for a string of length $n = 2k$ has three parts. First, T finds the middle, changing the symbols to uppercase as it goes; second, it changes the first half back to lowercase as it moves the tape head back to the beginning; finally, it compares the two halves.

In the first part, it takes one move to move the tape head to the first input symbol, $2k + 1$ moves to change the first symbol and find the rightmost symbol, and $2k$ more to change the rightmost symbol and return the tape head to the leftmost lowercase symbol. Each subsequent pass requires four fewer moves, and the total number of moves is therefore

$$1 + \sum_{i=0}^{k}(4i + 1) = 1 + (k + 1) + 4\sum_{i=0}^{k} i = k + 2 + 4\frac{k(k + 1)}{2} = 2k^2 + 3k + 2$$

The number of moves in the second part is $k + 1$ and also depends only on the length of the input string. The length of the third part depends on the actual string, but the strings requiring the most moves are the ones in L. The last part of the computation for a string in L involves one pass for each of the k symbols in the first half, and each pass contains k moves to the right, k to the left, and one more to the right. The maximum number of moves in this part is $2k^2 + k$, and we conclude that

$$\tau_T(n) = \tau_T(2k) = 4k^2 + 5k + 4 = n^2 + 5n/2 + 4$$

For each symbol of the input string, there are one or two moves that change the symbol to the opposite case, and one move that compares it to another symbol. The formula for $\tau_T(n)$ is quadratic because of the other moves, which involve repeated back-and-forth movements from one end of the string to the other. It is possible to reduce the number of these by adding more states. If we had the TM remember two symbols at a time, we could come

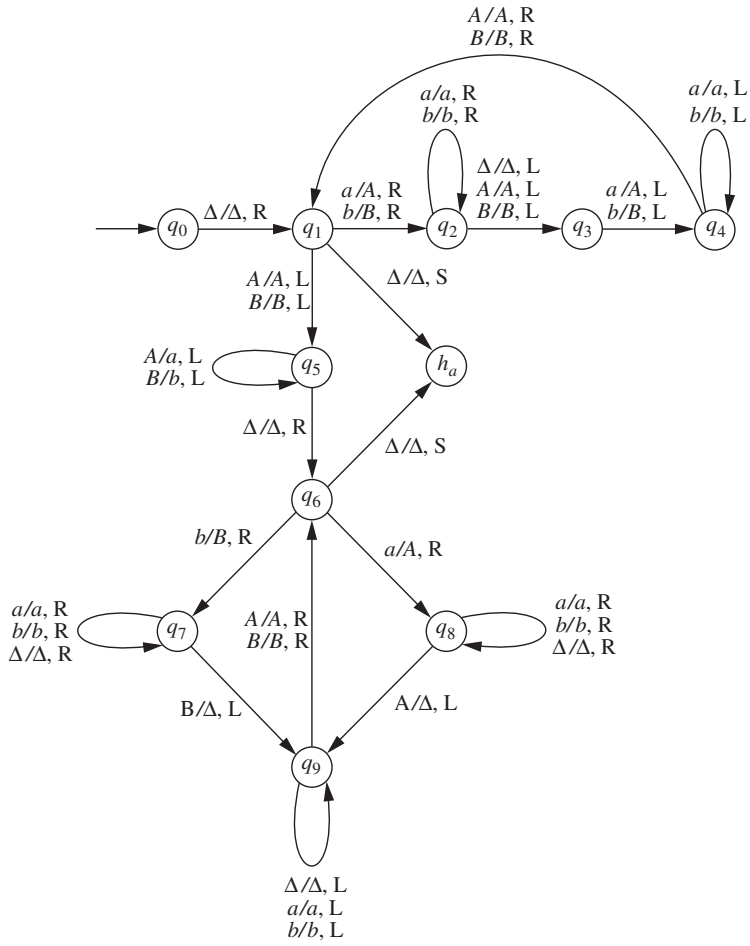


Figure 11.3 |
A Turing machine accepting $\{xx \mid x \in \{a, b\}^*\}$.

close to halving the number of back-and-forth passes, and going from two to a larger number would improve the performance even more, at least for large values of n. Techniques like these can, in fact, be used to reduce the time complexity of an arbitrary TM, by as large a constant factor as we wish. However, the back-and-forth movement in this example can't be eliminated completely, and it is possible to show that every TM accepting this language has time complexity that is at least quadratic.

In this example, the significant feature of τ_T is the fact that the formula is quadratic. The next definition makes it a little easier, when discussing the time complexity of a TM, to focus on the general growth rate of the function rather than the value at any specific point.

Definition 11.4 Big-Oh Notation

If f and g are partial functions from $\mathcal{N}$ to $\mathcal{R}^+$ (that is, both functions have values that are nonnegative real numbers wherever they are defined), we say that $f = O(g)$, or $f(n) = O(g(n))$ (which we read "f is big-oh of g," or "$f(n)$ is big-oh of $g(n)$"), if for some positive numbers C and N,

$$f(n) \leq Cg(n) \text{ for every } n \geq N$$

We can paraphrase the statement $f = O(g)$ by saying that except perhaps for a few values of n, f is no larger than g except for some constant factor. The statement $f(n) = O(g(n))$ starts out looking like a statement about the number $f(n)$, but it is not. (The notation is standard but is confusing at first.) It means the same thing as $f = O(g)$. Even though it tells us nothing about the values of $f(n)$ and $g(n)$ for any specific n, it is a way of comparing the long-term growth rates of the two functions.

If f and g are total functions with $f = O(g)$, and $g(n)$ is positive for every $n \in \mathcal{N}$, then it is possible to say that for some positive C, $f(n) \leq Cg(n)$ for *every* n; the way the definition is stated allows us to write $f = O(g)$ even if f or g is undefined in a few places, or if $g(n) = 0$ for a few values of n. For example,

$$n^2 + 5n/2 + 4 = O(n^2)$$

As a reason in this case, we could use the fact that for every $n \geq 1$, $n^2 + 5n/2 + 4 \leq n^2 + 5n^2/2 + 4n^2 = 7.5n^2$. Or, if we preferred, we could say that $n^2 + 5n/2 + 4 \leq 2n^2$ for every $n \geq 4$, because $5n/2 + 4 \leq n^2$ for those values of n.

Similarly, we can consider a polynomial f of degree k, with the formula

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$$

where the coefficients a_i are real numbers. As long as the leading coefficient a_k is positive, $f(n)$ will be positive for all sufficiently large values of n (and everywhere it's negative we can simply think of it as undefined). An argument similar to the one for the quadratic polynomial shows that $f(n) = O(n^k)$.

Although we haven't gotten very far yet toward our goal of categorizing decision problems in terms of their complexity, we would probably say that the problem of deciding the language in Example 11.2 is relatively simple. There is no shortage of decision problems that appear to be much harder. An easy way to find examples is to consider combinatorial problems for which finding the answer might involve considering a large number of cases. Here are two examples.

An instance of the *satisfiability problem* is an expression involving Boolean variables $x_1, x_2, \ldots, x_n$ (whose values are *true* or *false*) and the logical connectives $\wedge$, $\vee$, and $\neg$. The question is whether there is an assignment of truth values to the variables that satisfies the expression, or makes its value true. There is no doubt that this problem is decidable, because a brute-force algorithm will work: Try possible assignments of truth values, until one has been found that satisfies the expression or until all 2^n assignments have been tried.

The *traveling salesman problem* considers n cities that a salesman must visit, with a distance specified for every pair of cities. It's simplest to formulate this as an optimization problem, and to ask in which order the salesman should visit the cities in order to minimize the total distance traveled. We can turn the problem into a decision problem by introducing another number k that represents a constraint on the total distance: Is there an order in which the salesman can visit the cities so that the total distance traveled is no more than k? There is a brute-force solution here also, to consider possible orders one at a time. The decision problem might be answered before all $n!$ permutations have been considered, but the brute-force approach for either formulation might require looking at all of them.

For every decision problem, if the number of steps required to decide an instance of size n is proportional to n, computer hardware currently available makes it possible to perform the computation for instances that are very large indeed. The TM in Example 11.2 has time complexity that is roughly proportional to n^2, and that computation is still feasible for large values of n; many standard algorithms involving matrices or graphs take time proportional to n^2 or n^3. On the other hand, a decision problem for which the required time grows exponentially with n is undecidable in practice except for small instances. If an instance of size n required time 2^n, and if we could just manage an instance of size 20, doubling the computing speed would only allow us to go to $n = 21$, and a thousand-fold increase would still not be quite sufficient for $n = 30$. An instance of size 100 would be out of reach. It's not hard to see that if the time is proportional to $n!$, the situation is considerably worse.

Showing that a brute-force approach to solving a problem takes a long time does not in itself mean that the problem is complex. The satisfiability problem and the traveling salesman problem are assumed to be hard, not because the brute-force approach takes at least exponential time in both cases, but because no one has found a way of solving either problem that doesn't take at least exponential time. These two are typical of a class of problems that will be important in this chapter, because no one has been able to determine for sure just how complex they are, and because an answer to that question would help to clarify many long-standing open questions in complexity theory.

If we have determined that problems, or languages, requiring only quadratic time to decide should be considered relatively easy, and those that require exponential time or more should be considered very hard, what is a reasonable criterion that can be used to distinguish the *tractable* problems, those for which reasonable-size instances can be decided in a reasonable amount of time? The most common answer, though it is not perfect, is that the tractable problems are those with a *polynomial-time* solution; in particular, the problem of deciding a language L may be considered tractable if L can be accepted by a Turing machine T for which $\tau_T(n) = O(n^k)$ for some $k \in \mathcal{N}$.

One reason this criterion is convenient is that the class of tractable problems seems to be invariant among various models of computation. There is obviously a difference between the length of a computation on a Turing machine and the length of a corresponding computation on another model, whether it is a multitape

TM or an electronic computer, but the difference is likely to be no more than a polynomial factor. As a general rule, if a problem can be solved in polynomial time on *some* computer, then the same is true on any other computer. Similarly, using this criterion makes it unnecessary to distinguish between "the time required to answer an instance of the problem," as a function of the size of the instance, and the number of steps a Turing machine needs to make on an input string representing that instance. We must qualify this last statement only in the sense that the encoding we use should be *reasonable*, and that another requirement for an encoding method to be reasonable is that both encoding an instance and decoding a string should take no more than polynomial time (in the first case as a function of the instance size, and in the second as a function of the string length).

Not only is the polynomial-time criterion convenient in these ways, but to a large extent it seems to agree with people's ideas of which problems are tractable. Most interesting problems with polynomial-time solutions seem to require time proportional to n^2, or n^3, or n^4, rather than n^{50}; and in the case of problems for which no polynomial-time solutions are known, the only solutions that *are* known typically require time proportional to 2^n or worse.

Definition 11.5 The Set *P*

P is the set of languages L such that for some Turing machine T deciding L and some $k \in \mathcal{N}$, $\tau_T(n) = O(n^k)$.

Because of our comments above about decision problems and reasonable encodings, we can speak informally about a decision problem being in P if for some reasonable way of encoding instances, the resulting language of encoded yes-instances is in P.

Most of the rest of this chapter deals with the question of which languages, and which problems, are in P. We have seen a language that is, in Example 11.2. We can construct "artificial" languages that are definitely not (see Exercise 11.19). The satisfiability problem and the traveling salesman problem seem to be good candidates for real-life problems that are not in P. In the next section we will introduce the set *NP*, which contains every problem in P but also many other apparently difficult problems such as these two. Most people believe that the satisfiability problem is not in P and therefore that $P \neq NP$, but no one has proved this, and the $P \stackrel{?}{=} NP$ question remains very perplexing. See the review article by Lance Fortnow in the September 2009 issue of *Communications of the ACM* for a recent report on the status of this question.

11.2 THE SET *NP* AND POLYNOMIAL VERIFIABILITY

The satisfiability problem, which we discussed briefly in Section 11.1, seems like a hard problem because no one has found a decision algorithm that is significantly

better than testing possible assignments one at a time. Testing is not hard; for any reasonable way of describing a Boolean expression with n distinct variables, we can easily find out whether a given truth assignment satisfies the expression. The difficulty is that the number of possible assignments is exponential in n. Even if we could test each one in constant time, independent of n, the total time required would not be polynomial.

In Chapter 7, when we were interested only in whether a language was recursively enumerable and not with how long it might take to test an input string, we used nondeterminism in several examples to simplify the construction of a TM (see Examples 7.28 and 7.30). We can consider a nondeterministic TM here as well, which accepts *Satisfiable* by guessing an assignment and then testing (deterministically) whether it satisfies the expression. This approach doesn't obviously address the issue of complexity. The steps involved in guessing and testing an assignment can easily be carried out in polynomial time, as we will see in more detail below, but because the nondeterminism eliminates the time-consuming part of the brute-force approach, the approach doesn't seem to shed much light on how hard the problem is.

Nevertheless, at least it provides a simple way of formulating the question posed by problems like *Sat*: If we have a language L that can be accepted in polynomial time by a *nondeterministic* TM (see Definition 11.6), is there any reason to suspect that L is in P?

We start by defining the time complexity of an NTM, making the assumption as we did in Definition 11.1 that no input string can cause it to loop forever. The definition is almost the same as the earlier one, except that for each n we must now consider the maximum number of moves the machine makes, not only over all strings of length n but also over all possible sequences of moves it might make on a particular input string of length n.

Definition 11.6 The Time Complexity of a Nondeterministic Turing Machine

If T is an NTM with input alphabet Σ such that, for every $x \in \Sigma^*$, every possible sequence of moves of T on input x eventually halts, the time complexity $\tau_T : \mathcal{N} \rightarrow \mathcal{N}$ is defined by letting $\tau_T(n)$ be the maximum number of moves T can possibly make on any input string of length n before halting. As before, if we speak of an NTM as having a time complexity, we are assuming implicitly that no input string can cause it to loop forever.

Definition 11.7 The Set *NP*

NP is the set of languages L such that for some NTM T that cannot loop forever on any input, and some integer k, T accepts L and $\tau_T(n) = O(n^k)$. We say that a language in *NP* can be accepted in *nondeterministic polynomial time*.

As before, we can speak of a decision problem being in *NP* if, for some reasonable way of encoding instances, the corresponding language is.

Because an ordinary Turing machine can be considered an NTM, it is clear that $P \subseteq NP$. Before starting to discuss the opposite inclusion, we consider two examples, including a more detailed discussion of the satisfiability problem.

The Satisfiability Problem **EXAMPLE 11.8**

We let *Sat* denote the satisfiability problem. An instance is an expression involving occurrences of the Boolean variables $x_1, x_2, \ldots, x_v$, for some $v \geq 1$. We use the term *literal* to mean either an occurrence of a variable x_i or an occurrence of $\overline{x}_i$, which represents the negation $\neg x_i$ of x_i. The expression is assumed to be in *conjunctive normal form*, or CNF: It is the conjunction

$$C_1 \wedge C_2 \wedge \ldots C_c$$

of clauses, or conjuncts, C_i, each of which is a disjunction (formed with $\vee$'s) of literals. For an expression of this type, *Sat* asks whether the expression can be satisfied by some truth assignment to the variables.

For example,

$$(x_1 \vee x_3 \vee x_4) \wedge (\overline{x}_1 \vee x_3) \wedge (\overline{x}_1 \vee x_4 \vee \overline{x}_2) \wedge \overline{x}_3 \wedge (x_2 \vee \overline{x}_4)$$

is an expression with five conjuncts that is satisfied by the truth assignment

$$x_2 = x_4 = \mathit{true}, \qquad x_1 = x_3 = \mathit{false}$$

We will encode instances of *Sat* by omitting parentheses and $\vee$'s and using unary notation for subscripts. For example, the expression

$$(x_1 \vee \overline{x}_2) \wedge (x_2 \vee x_3 \vee \overline{x}_1) \wedge (\overline{x}_4 \vee x_2)$$

will be represented by the string

$$\wedge x1\overline{x}11 \wedge x11x111\overline{x}1 \wedge \overline{x}1111x11$$

Satisfiable will be the language over $\Sigma = \{\wedge, x, \overline{x}, 1\}$ containing all the encoded yes-instances.

If the size of a problem instance is taken to be the number k of literals (not necessarily distinct), and n is the length of the string encoding the instance, then $k \leq n$, and it is easy to see that n is bounded by a quadratic function of k. Therefore, if $\mathit{Satisfiable} \in NP$, it makes sense to say that the decision problem *Sat* is in *NP*.

The first step carried out by an NTM T accepting *Satisfiable* is to verify that the input string represents a valid CNF expression and that for some v, the variables are $x_1, x_2, \ldots, x_v$. If this is the case, T attempts to satisfy the expression, keeping track as it proceeds which conjuncts have been satisfied so far and which variables within the unsatisfied conjuncts have been assigned values. The iterative step consists of finding the first conjunct not yet satisfied; choosing a literal within that conjunct that has not been assigned a value (this is the only occurrence of nondeterminism); giving the variable in that literal the value that satisfies the conjunct; marking the conjunct as satisfied; and giving the same value to all

subsequent occurrences of that variable in unsatisfied conjuncts, marking any conjuncts that are satisfied as a result. The loop terminates in one of two ways. Either all conjuncts are eventually satisfied, or the literals in the first unsatisfied conjunct are all found to have been falsified. In the first case T accepts, and in the second it rejects. If the expression is satisfiable, some choice of moves causes T to guess a truth assignment that works, and otherwise no choice of moves leads to acceptance.

The TM T can be constructed so that, except for a few steps that take time proportional to n, all its actions are minor variations of the following operation: Begin with a string of 1's in the input, delimited at both ends by some symbol other than 1, and locate some or all of the other occurrences of this string that are similarly delimited. We leave it to you to convince yourself that a single operation of this type can be done in polynomial time, and that the number of such operations that must be performed is also no more than a polynomial. Our conclusion is that *Satisfiable*, and therefore *Sat*, are in *NP*.

EXAMPLE 11.9 Composites and Primes

A decision problem that is more familiar and almost obviously in *NP* is the problem to which we will give the name *Comp*: Given a natural number n, is it a composite number—that is, is there some smaller number other than 1 that divides it evenly? The way the problem is stated makes it easy to apply the guess-and-test strategy, just as in Example 11.8. An NTM can examine the input n, guess a positive integer p strictly between 1 and n, divide n by p, and determine whether the remainder is 0. Or, taking even greater advantage of nondeterminism, it could guess two posive integers p and q, both between 1 and n, and multiply them to determine whether $n = pq$ (see Example 7.28).

One subtlety related to any problem whose instances are numbers has to do with how we measure the size of an instance. It might seem that an obvious measure of the size of the instance n is n, which would be the length of an input string if we used unary notation. However, people don't do it this way: Using binary notation would allow an input string with approximately $\log n$ digits, and it would be a little unreasonable to say that an algorithm required polynomial time if that meant a polynomial function of n but an exponential function of the input length.

It is easy to see that in the case of *Comp*, an NTM can carry out the steps described above in polynomial time, even as a function of $\log n$. Therefore, *Comp* is indeed in *NP*.

Without nondeterminism, the problem appears intimidating at first. The most obvious brute-force approach, testing natural numbers one at a time as possible divisors, certainly does not produce a polynomial-time solution. We can improve on this, at least to the extent that only primes less than n need to be considered as possible divisors, but the improvement is not enough to make the time a polynomial in $\log n$. However, this problem does not seem to have quite the combinatorial complexity of the assignment problem, and there are additional mathematical tools available to be applied, if only because properties of primes and composites have been studied for centuries. In any case, the problem *Comp* is in a different category than *Sat*, because it has actually been proven to be in P (see the paper by Agrawal, Kayal, and Saxena).

This is slightly curious. The complementary problem to *Comp* is *Prime*: Given a natural number n, is n prime? If *Comp* is in P, then *Prime* is also (see Exercise 11.9), but it's

not even clear at first that *Prime* is in *NP*. Although the two problems ask essentially the same question, a nondeterministic approach is available for one version (Does there exist a number that is a factor of n?) and not obviously available for the other (Is p a nondivisor of n for *every* p?). In fact, nondeterminism can be used for the problem *Prime*, as suggested in the next paragraph, but the corresponding question is difficult in general: It is not known whether the complementary problem of every problem in *NP* is itself in *NP* (see Exercise 11.10).

Although the proof that *Comp* or *Prime* is in P is complicated, here is a way to approach the problem *Prime* nondeterministically. It involves a characterization of primes due to Fermat: An integer n greater than 3 is prime if and only if there is an integer x, with $1 < x < n - 1$, satisfying

$$x^{n-1} \equiv_n 1, \text{ and for every } m \text{ with } 1 < m < n - 1, \; x^m \not\equiv_n 1$$

The fact that the characterization begins "there is an integer x" is what suggests some hope. It's still not apparent that this approach will be useful, because deciding that the statement is true seems to require performing a test on every number between 1 and $n - 1$. However, this turns out not to be too serious an obstacle (Exercise 11.24).

Let us return to the satisfiability problem in order to obtain a slightly different characterization of *NP*, based on the guess-and-verify strategy we have mentioned in all our examples. In the discussion, we consider a Boolean expression e, and an assignment a of truth values to the variables in e; it may help to clarify things if we identify e and a with the strings that represent them.

The NTM we described that accepts *Satisfiable* takes the input string e and guesses an assignment a. This is the nondeterministic part of its operation. The remainder of the computation is deterministic and consists of checking whether a satisfies e. If the expression is satisfiable and the guesses in the first part are correct, the string a can be interpreted as a *certificate* for e, proving, or attesting to the fact, that e is satisfiable. By definition, every string e possessing a certificate is in *Satisfiable*. A certificate may be difficult to obtain in the first place, but the process of verifying that it is a certificate for e is straightforward.

We can isolate the second portion of the computation by formulating the definition of a *verifier* for *Satisfiable*: a deterministic algorithm that operates on a pair (e, a) (we take this to mean a Turing machine operating on the input string $e\$a$, for some symbol \$ not in the alphabet) and either accepts it or rejects it. We interpret acceptance to mean that a is a certificate for e and that e is therefore in *Satisfiable*; rejection is interpreted to mean, not that e fails to be in *Satisfiable*—there may be a string different from a that is a certificate for e—but only that a is not a certificate for e. An expression (string) e is an element of *Satisfiable* if and only if there is a string a such that the string $e\$a$ is accepted by the verifier.

The issue of polynomial time arises when we consider how many steps are required by a verifier to accept or reject the string $e\$a$. However, it is more appropriate to describe this number in terms of $|e|$, the size of the instance of the original problem, than in terms of $|e\$a|$, the length of the entire string.

Definition 11.10 A Verifier for a Language

If $L \subseteq \Sigma^*$, we say that a TM T is a *verifier* for L if T accepts a language $L_1 \subseteq \Sigma^*\{\$\}\Sigma^*$, T halts on every input, and

$$L = \{x \in \Sigma^* \mid \text{ for some } a \in \Sigma^*, x\$a \in L_1\}$$

A verifier T is a *polynomial-time verifier* if there is a polynomial p such that for every x and every a in Σ^*, the number of moves T makes on the input string $x\$a$ is no more than $p(|x|)$.

Theorem 11.11

For every language $L \in \Sigma^*$, $L \in NP$ if and only if L is polynomially verifiable—i.e., there is a polynomial-time verifier for L.

Proof

By definition, if $L \in NP$, there is an NTM T_1 accepting L in nondeterministic polynomial time. This means that the function τ_{T_1} is bounded by some polynomial q. We can obtain a verifier T for L by considering strings m representing sequences of moves of T_1. The verification algorithm takes an input $x\$m$, executes the moves of T_1 corresponding to m on the input x (or, if m represents a sequence of more than $q(|x|)$ moves, executes the first $q(|x|)$), and accepts if and only if this sequence of moves causes T_1 to accept. It is clear that the verifier operates in polynomial time as a function of $|x|$.

On the other hand, if T is a polynomial-time verifier for L, and p is the associated polynomial, then we can construct an NTM T_1 that operates as follows: On input x, T_1 places the symbol \$ after the input string, followed by a sequence of symbols, with length no more than $p(|x|)$, that it generates nondeterministically; it then executes the verifier T. T_1 accepts the language L, because the strings in L are precisely those that allow a sequence of moves leading to acceptance by the verifier. Furthermore, it is easy to see that the number of steps in the computation of T_1 is bounded by a polynomial function of $|x|$.

A verifier for the language corresponding to *Comp* accepts strings $x\$p$, where x represents a natural number n and p a divisor of n; a certificate for x is simply a number. In the case of the traveling salesman problem, mentioned in Section 11.1, a certificate for an instance of the problem is a permutation of the cities so that visiting them in that order is possible with the given constraint. We will see several more problems later in the chapter for which there is a straightforward guess-and-verify approach. According to Theorem 11.11, these examples are typical of all the languages in *NP*. The proof of the theorem shows why this is not surprising:

For every x in such a language, a sequence of moves by which the NTM accepts x constitutes a certificate for the string.

11.3 POLYNOMIAL-TIME REDUCTIONS AND *NP*-COMPLETENESS

Just as we can show that a problem is decidable by reducing it to another one that is, we can show that languages are in P by reducing them to others that are. (If we have languages we know are not in P, we can also use reductions to find others that are not.) In Chapter 9, reducing a language L_1 to another language L_2 required only that we find a computable function f such that deciding whether $x \in L_1$ is equivalent to deciding whether $f(x) \in L_2$. In this case, we said that deciding the first language is no harder than deciding the second, because we considered only two degrees of hardness, decidable and undecidable. To see whether $x \in L_1$, all we have to do is compute $f(x)$ and see whether it is in L_2; and f is computable. Because one decidable language can be harder (i.e., take longer) to decide than another, a reduction should now satisfy some additional conditions. Not surprisingly, an appropriate choice is to assume that the function be computable in polynomial time.

Definition 11.12 Polynomial-Time Reductions

If L_1 and L_2 are languages over respective alphabets Σ_1 and Σ_2, a polynomial-time reduction from L_1 to L_2 is a function $f : \Sigma_1^* \to \Sigma_2^*$ satisfying two conditions: First, for every $x \in \Sigma_1^*$, $x \in L_1$ if and only if $f(x) \in L_2$; and second, f can be computed in polynomial time—that is, there is a TM with polynomial time complexity that computes f. If there is a polynomial-time reduction from L_1 to L_2, we write $L_1 \leq_p L_2$ and say that L_1 is polynomial-time reducible to L_2.

If we interpret $\leq_p$ to mean "is no harder than," then the statements in the following theorem are not surprising.

Theorem 11.13

1. Polynomial-time reducibility is transitive: If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.
2. If $L_1 \leq_p L_2$ and $L_2 \in P$, then $L_1 \in P$.

Proof

1. Suppose the two reductions are $f : \Sigma_1^* \to \Sigma_2^*$ and $g : \Sigma_2^* \to \Sigma_3^*$. Then the composite function $g \circ f$ is a function from Σ_1^* to Σ_3^*, and for every $x \in \Sigma_1^*$, $x \in L_1$ if and only if $g \circ f(x) \in L_3$. To show $g \circ f$ is a

polynomial-time reduction from L_1 to L_3, it is therefore sufficient to show that it can be computed in polynomial time.
Suppose T_f and T_g compute f and g, respectively, in polynomial time; then the composite TM $T_f T_g$ computes $g \circ f$. From the fact that T_f computes f in polynomial time, we can see that $|f(x)|$ is itself bounded by a polynomial function of $|x|$. From this fact and the inequality

$$\tau_{T_f T_g}(|x|) \leq \tau_{T_f}(|x|) + \tau_{T_g}(|f(x)|)$$

it then follows that the composite TM also operates in polynomial time.

2. Suppose T_2 is a TM accepting L_2 in polynomial time, and T_f is a TM computing f in polynomial time, where f is the reduction from L_1 to L_2. The composite TM $T_f T_2$ accepts L_1, because $x \in L_1$ if and only if $f(x) \in L_2$. On an input string x, the number of steps in the computation of $T_f T_2$ is the number of steps required to compute $f(x)$ plus the number of steps required by T_2 on the input $f(x)$. Just as in statement 1, the length of the string $f(x)$ is bounded by a polynomial in $|x|$, and this implies that the number of steps in T_2's computation must also be polynomially bounded as a function of $|x|$.

Definition 11.12 and Theorem 11.13 extend to decision problems, provided that we require reasonable encodings of instances. As a result, constructing an appropriate polynomial-time reduction allows us to conclude that a problem is in P or NP. Just as in Chapter 9, however, it is also a way of showing that one problem is at least as hard as another. In our first example we consider a decision problem involving undirected graphs, and the reduction we construct will allow us to say that it is at least as complex as *Sat*.

EXAMPLE 11.14 A Reduction from *Sat* to the Complete Subgraph Problem

We begin with some terminology. An undirected graph is a pair $G = (V, E)$, where V is a finite nonempty set of vertices and E is a set of edges. An edge is an unordered pair (v_1, v_2) of vertices ("unordered" means that (v_1, v_2) and (v_2, v_1) are the same). If e is the edge (v_1, v_2), then v_1 and v_2 are the end points of e, are said to be joined by e, and are said to be adjacent. A *complete subgraph* of G is a subgraph (a graph whose vertex set and edge set are subsets of the respective sets of G) in which every two vertices are adjacent. It is clear that if a graph has a complete subgraph with k vertices, then for every $n < k$, it has a complete subgraph with n vertices. Figure 11.15 shows a graph G with seven vertices and ten edges. There is a complete subgraph with the vertices 3, 5, 6, and 7, and no subgraph of G with more than 4 vertices is complete.

The *complete subgraph problem* is the problem: Given a graph G and a positive integer k, does G have a complete subgraph with k vertices? We can represent the graph by a string of edges followed by the integer k, where we represent the vertices by integers 1, 2, ..., n and use unary notation for each vertex and for the number k. For example, the instance

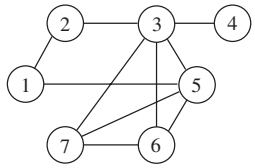


Figure 11.15 |
A graph with 7 vertices and 10 edges.

consisting of the graph in Figure 11.15 and the integer 4 might be represented by the string

$$(1, 11)(11, 111)(111, 1111)(111, 11111)\ldots(111111, 1111111)1111$$

A graph with n vertices has no more than $n(n-1)/2$ edges, and it is reasonable to take the number n as the size of the instance.

The complete subgraph problem is in *NP*, because an NTM can take a string encoding an instance (G, k), nondeterministically select k of the vertices, and examine the edges to see whether every pair among those k is adjacent. We will construct a reduction from the satisfiability problem *Sat* to the complete subgraph problem. It is easiest to discuss the reduction by considering the instances themselves, rather than the strings representing them.

Suppose x is the CNF expression

$$x = \bigwedge_{i=1}^{c} \bigvee_{j=1}^{d_i} a_{i,j}$$

where each $a_{i,j}$ is a literal. We must describe an instance $f(x) = (G_x, k_x)$ of the complete subgraph problem so that x is satisfiable if and only if G_x has a complete subgraph with k_x vertices. We construct $G_x = (V_x, E_x)$ so that vertices correspond to occurrences of literals in x:

$$V_x = \{(i, j) \mid 1 \leq i \leq c \text{ and } 1 \leq j \leq d_i\}$$

The edge set E_x is specified by saying that (i, j) is adjacent to (l, m) if and only if the corresponding literals are in different conjuncts of x and there is a truth assignment that makes them both true. In other words,

$$E_x = \{((i, j), (l, m)) \mid i \neq l \text{ and } a_{i,j} \neq \neg a_{l,m}\}$$

Finally, we define k_x to be c, the number of conjuncts in x.

If x is satisfiable, then there is a truth assignment Θ such that for every i there is a literal a_{i,j_i} that is given the value *true* by Θ. The vertices

$$(1, j_1), (2, j_2), \ldots, (k, j_k)$$

determine a complete subgraph of G_x, because we have chosen the edges of G_x so that every two of these vertices are adjacent. On the other hand, suppose there is a complete subgraph of G_x with k_x vertices. Because none of the corresponding literals is the negation of another, some truth assignment makes them all true. Because these literals must be

in distinct conjuncts, the assignment makes at least one literal in each conjunct true and therefore satisfies x.

In order to construct the string representing (G_x, k_x) from the string representing x, a number of steps are necessary: associating the occurrences of literals in x with integers from 1 through n; for each one, finding every other one occurring in a different conjunct that is not the negation of that one; and constructing the string representing that edge. Each of these steps can evidently be carried out in a number of steps that is no more than a polynomial in $|x|$, and the overall time is still no more than polynomial.

The reduction in Example 11.14 allows us to compare the complexity of the two decision problems, but not to say any more about the actual complexity of either one. If a polynomial-time algorithm were to be found for the complete subgraph problem, it would follow that there is also one for *Sat* as well; if it could be shown that the complete subgraph problem had no polynomial algorithm, this reduction would still leave the question for *Sat* unresolved. One approach to answering some of these questions might be to try to identify a *hardest* problem in *NP*, or perhaps several problems of similar difficulty that seem to be among the hardest, and then to try to say something about how hard the problems actually are. At least the first of these two steps is potentially feasible. According to Theorem 11.13, it makes sense to say a problem is a hardest problem in *NP* if every other problem in *NP* is reducible to it.

Definition 11.16 *NP*-Hard and *NP*-Complete Languages

A language L is *NP*-hard if $L_1 \leq_p L$ for every $L \in NP$. L is *NP*-complete if $L \in NP$ and L is *NP*-hard.

Theorem 11.17

1. If L and L_1 are languages such that L is *NP*-hard and $L \leq_p L_1$, then L_1 is also *NP*-hard.
2. If L is any *NP*-complete language, then $L \in P$ if and only if $P = NP$.

Proof

Both parts of the theorem follow immediately from Theorem 11.13.

Definition 11.16 and Theorem 11.17 can be extended to decision problems; an *NP*-complete problem is one for which the corresponding language is *NP*-complete. Theorem 11.17 provides a way of obtaining more *NP*-complete problems, provided that we can find one to start with.

Remarkably, we can. It was shown in the early 1970s, independently by Cook and Levin, that *Sat* is *NP*-complete (see Theorem 11.18). Example 11.14 provides another example, and it turns out that there are many more, including problems that involve graphs, networks, sets and partitions, scheduling, number theory, logic, and other areas.

Theorem 11.17 indicates both ways in which the idea of *NP*-completeness is important. On the one hand, if someone were ever to find an *NP*-complete problem that could be solved by a polynomial-time algorithm, then the $P \stackrel{?}{=} NP$ question would be resolved; *NP* would disappear as a separate entity, and researchers could concentrate on finding polynomial-time algorithms for all the problems now known to be in *NP*. On the other hand, as long as the question remains open (or if someone actually succeeds in proving that $P \neq NP$), the difficulty of a problem can be convincingly demonstrated by showing that it is *NP*-complete.

11.4 | THE COOK-LEVIN THEOREM

An *NP*-complete problem is a problem in *NP* to which every other problem in *NP* can be reduced. Theorem 11.18, proved independently by Stephen Cook and Leonid Levin in 1971 and 1972, says that the satisfiability problem has this property. Before we state and sketch the proof of the result, we consider another problem that we can reduce to *Sat*—not because the conclusion is interesting (the problem has only a finite set of instances, which means that we don't need to reduce it to anything in order to solve it), but because the approach illustrates in a much simpler setting the general principle that is used in the proof of the theorem and can be used to reduce other problems to *Sat*.

In both this simple example and the proof of the theorem itself, it is convenient to use a fact about logical propositions (see Exercise 11.13): For every proposition p involving Boolean variables $a_1, a_2, \ldots, a_t$ and their negations, p is logically equivalent to another proposition q involving these variables that is in conjunctive normal form.

The problem is this: Given an undirected graph $G = (V, E)$ with three vertices, is G connected? (A graph with three vertices is connected if and only if at least two of the three possible edges are actually present.) Reducing the problem to *Sat* means constructing a function f from the set of 3-vertex graphs to the set of Boolean expressions such that for every graph G, G is connected if and only if $f(G)$ is satisfiable. For a finite problem like this, polynomial-time computability is automatic.

We represent the vertices of G using the numbers 1, 2, and 3. We consider three Boolean variables, labeled $x_{1,2}$, $x_{1,3}$, and $x_{2,3}$, and we think of $x_{i,j}$ as being associated with the statement "there is an edge from i to j." With this interpretation of the three variables, a formula "asserting" that the graph is connected is

$$S: \ (x_{1,2} \wedge x_{1,3}) \vee (x_{1,2} \wedge x_{2,3}) \vee (x_{1,3} \wedge x_{2,3})$$

(S is not in conjunctive normal form—this is one of the places we use the fact referred to above.) The statement we get from our interpretation of the variables $x_{i,j}$ has nothing to do so far with the specific graph G but is simply a general statement of what it means for a three-vertex graph to be connected. The question for G is whether the statement can actually be true when the constraints are added, which enumerate the edges that are missing from this particular instance of the problem. For the graph with only the edge from 1 to 3, for example, the statement

says "The edges from 1 to 2 and from 2 to 3 are missing, and at least two edges are present." The corresponding expression is

$$\overline{x}_{1,2} \wedge \overline{x}_{2,3} \wedge S$$

where as usual the bars over the x mean negation. This expression is not satisfied, no matter what values are assigned to the three variables.

Our function f is obtained exactly this way. The expression $f(G)$ is the conjunction containing all terms $\overline{x}_{i,j}$ corresponding to the edges not present, as well as the CNF version of the expression S constructed above. In this simple case, for every possible G, G is connected if and only if $f(G)$ can be satisfied by some truth assignment to the three Boolean variables.

This example, in addition to being simple, is perhaps atypical, because it is clear that the more edges there are in the graph, the more likely it is to be connected. As a result, asking whether the expression $f(G)$ is satisfiable boils down to asking whether it is *true* when all the edges not explicitly listed as missing are actually present. In general, we don't expect that trying any particular single assignment will be enough to test for satisfiability.

Let us be a little more explicit as to why the connectedness of G is equivalent to the satisfiability of the expression $f(G)$. We have said that "we think of $x_{i,j}$ as being associated with the statement 'there is an edge from i to j.' " How we *think* of the variable $x_{i,j}$ isn't really the issue. Rather, we have constructed the expression S so that its logical structure mirrors exactly the structure of the statement "G is connected" in this simple case; our interpretation of $x_{i,j}$ simply makes this logical equivalence a little more evident. The additional portion of the expression $f(G)$ also mirrors exactly the constraints on the graph, arising from certain edges not being present. Therefore, the question of whether we can specify the edges in a way consistent with these constraints so that G is connected is exactly the same as the question of whether we can assign truth values to the variables $x_{i,j}$ so that the entire expression $f(G)$ is satisfied.

Theorem 11.18 The Cook-Levin Theorem
The language *Satisfiable* (or the decision problem *Sat*) is *NP*-complete.

Proof
We know from Example 11.8 that *Satisfiable* is in *NP*, and now we must show it is *NP*-hard, which means that for every language $L \in NP$, there is a polynomial-time reduction from L to *Satisfiable*.

Suppose L is in *NP*. Then L must be $L(T)$ for some nondeterministic Turing machine $T = (Q, \Sigma, \Gamma, q_0, \delta)$ with polynomial time complexity. Let p be a polynomial such that $\tau_T(n) \leq p(n)$ for every n. We make additional assumptions about T and p, both with no loss of generality: We assume that T never rejects by trying to move its tape head left from square 0, and we assume that $p(n) \geq n$ for every n. We denote by *CNF* the set of CNF expressions, with no restrictions on the number of variables

in the expression or on the names of the variables. To complete the proof, it will be sufficient to construct a function

$$g : \Sigma^* \to CNF$$

satisfying these two conditions.

1. For every $x \in \Sigma^*$, x is accepted by T if and only if $g(x)$ is satisfiable.
2. The corresponding function $f : \Sigma^* \to \{\wedge, x, \overline{x}, 1\}^*$ that reduces $L(T)$ to *Satisfiable*, obtained by relabeling variables in CNF expressions and encoding the expressions appropriately, is computable in polynomial time.

The construction of g is the more complicated part of the proof, because the expression $g(x)$ will be a complicated expression. Fortunately, it is relatively easy to verify that f is polynomial-time computable.

Two configurations C and C' of T will be called *consecutive* in either of two cases: Either $C \vdash_T C'$, or $C = C'$ and T cannot move from configuration C. Then condition 1 above can be restated as follows: For every $x \in \Sigma^*$, $g(x)$ is satisfiable if and only if there is a sequence of consecutive configurations

$$C_0, C_1, \ldots, C_{p(|x|)}$$

such that C_0 is the initial configuration corresponding to input x, and $C_{p(|x|)}$ is an accepting configuration.

We begin by enumerating the states and tape symbols of T. Let

$$Q = \{q_0, q_1, \ldots, q_{t-2}\},\ q_{t-1} = h_r,\ \text{and } q_t = h_a$$
$$\sigma_0 = \Delta \text{ and } \Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_s\}$$
$$\Gamma = \{\sigma_0, \ldots, \sigma_s, \sigma_{s+1}, \ldots, \sigma_{s'}\}$$

Next, let us fix a string

$$x = \sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_n} \in \Sigma^*$$

and let $N = p(|x|) = p(n)$. The statement that T accepts x involves only the first $N + 1$ configurations of T, and therefore only the tape squares 0, 1, ..., N of T (because in N moves, the tape head can't move past square N). With this in mind, we can describe the Boolean variables we will use in the expression $g(x)$ (not surprisingly, many more than were necessary in our connected-graph example above) and, for each one, say how we will interpret it. The interpretations all have to do with details of a configuration of T.

$Q_{i,j}$: after i moves, T is in state q_j

$H_{i,k}$: after i moves, the tape head is on square k

$S_{i,k,l}$: after i moves, the symbol in square k is σ_l

Here the numbers i and k vary from 0 to N, j varies from 0 to t, and l varies from 0 to s'. There are enough variables here to describe (although

of course it's the interpretations, not the variables, that describe) every detail of every one of the first $N + 1$ configurations of T.

Here is the way we construct $g(x)$, and it may help to keep in mind the connected-graph illustration before the theorem:

$$g(x) = g_1(x) \wedge S(x)$$

The subexpression $g_1(x)$ corresponds to the statement that the initial configuration of T corresponds to the input string x. Although parts of the remaining portion $S(x)$ involve the number N, $g_1(x)$ is the only part of $g(x)$ that actually involves the symbols of x. The expression $S(x)$ describes all the relationships among the variables that must hold, in order for the corresponding statement to say that the details of the configurations are consistent with the operation of a Turing machine, and that the final configuration is an accepting one.

It is easy to understand the formula $g_1(x)$:

$$g_1(x) = Q_{0,0} \wedge H_{0,0} \wedge S_{0,0,0} \wedge \bigwedge_{k=1}^{n} S_{0,k,i_k} \wedge \bigwedge_{k=n+1}^{N} S_{0,k,0}$$

The corresponding statement says that after 0 moves, the state is q_0, the tape head position is 0 and there is a blank in that position, the squares 1–n contain the symbols of x, and the remaining positions up to N are blank.

The expression $S(x)$ can be described as the conjunction of six subexpressions

$$S(x) = \bigwedge_{i=2}^{7} g_i(x)$$

We will describe only two of these in detail: $g_2(x)$, which is extremely simple, and $g_3(x)$, which is the most complicated of the six.

The statement corresponding to $g_2(x)$ says that after N steps, T is in the accepting state; the actual expression is

$$g_2(x) = Q_{N,t}$$

The statement corresponding to $g_3(x)$ involves the transition function of T; it says that whenever T is in a configuration from which it can move, the state and current symbol one step later are those that result from one of the legal moves of T. Suppose we denote by CM (for "can move") the set of pairs (j, l) for which $\delta(q_j, \sigma_l) \neq \emptyset$. We want our statement to say the following: For every i from 0 to $N - 1$, for every tape square k from 0 to N, and for every pair $(j, l) \in CM$, if at time i the tape head is at square k and the current state-symbol combination is (j, l), there is a move in $\delta(q_j, \sigma_l)$ such that at time $i + 1$, the state, the symbol at square k, and the new tape head position are precisely those specified by that

move. If there is only one move in the set $\delta(q_j, \sigma_l)$, say $(q_{j'}, \sigma_{l'}, D)$, the appropriate expression for that choice of i, j, k, and l can be written

$$(Q_{i,j} \wedge H_{i,k} \wedge S_{i,k,l}) \rightarrow (Q_{i+1,j'} \wedge H_{i+1,k'} \wedge S_{i+1,k,l'})$$

where

$$k' = \begin{cases} k+1 & \text{if } D = \text{R} \\ k & \text{if } D = \text{S} \\ k-1 & \text{if } D = \text{L} \end{cases}$$

Because, in general, $\delta(q_j, \sigma_l)$ may have several elements, we need

$$(Q_{i,j} \wedge H_{i,k} \wedge S_{i,k,l}) \rightarrow \bigvee_m (Q_{i+1,j_m} \wedge H_{i+1,k_m} \wedge S_{i+1,k,l_m})$$

where m ranges over all moves in $\delta(q_j, \sigma_l)$ and, for a given m, (j_m, l_m, k_m) is the corresponding triple (j', l', k') for that move. Thus the complete expression we want, except that it is not in conjunctive normal form, looks like this:

$$\bigwedge_{i,j,k,l} ((Q_{i,j} \wedge H_{i,k} \wedge S_{i,k,l}) \rightarrow \bigvee_m (Q_{i+1,j_m} \wedge H_{i+1,k_m} \wedge S_{i+1,k,l_m}))$$

where i ranges from 0 to $N-1$, k ranges from 0 to N, and (j, l) varies over all pairs in CM. Each of the conjuncts in this expression can be converted to CNF, and the result is

$$g_3(x) = \bigwedge_{i,j,k,l} F_{i,j,k,l}$$

where $F_{i,j,k,l}$ is in conjunctive normal form and the number of literals in the expression is bounded independently of n.

The interpretations of the remaining pieces of $g(x)$, the subexpressions $g_4(x)$-$g_7(x)$, are as follows:

4. For every i, if T cannot move at time i, then the configuration at time $i+1$ is unchanged.
5. T is in exactly one state at each step.
6. Each square of the tape contains exactly one symbol at each step.
7. The only changes in the tape are those that result from legal moves of T. In other words, for each i and each k, if the tape head is not on square k at time i, then the symbol in that square is unchanged at time $i+1$.

It's possible to convince yourself that the statement corresponding to the expression $S(x)$ is equivalent to the statement that T changes state, tape symbol, and tape head position at each step in accordance with the rules of a Turing machine and ends up at time N in the accepting state. Therefore, the statement corresponding to $g(x) = g_1(x) \wedge S(x)$ says that x is accepted by T. If x is indeed accepted by T, then there is some sequence of moves T can make on input x that leads to the accepting state within

N moves; there are assignments to the Boolean variables representing details of the configurations of T that are consistent with that sequence of moves; and therefore, there are assignments that satisfy the expression $g(x)$ (perhaps several such assignments, as a result of the nondeterminism of T). On the other hand, if $g(x)$ is satisfiable, then an assignment that satisfies it corresponds to a particular sequence of moves of T by which x is accepted.

We will say a little about the number of steps required to compute $g(x)$ and $f(x)$. A Turing machine T_g whose output is $g(x)$ uses a number of ingredients in its computation, among them the input string x, the integers $n = |x|$ and $N = p(n)$, the integers t, s, and s', and the transition table for T, consisting of a set of 5-tuples. T_g starts by entering all this information at the beginning of its tape, in a "reference" section. After this reference section on the tape is space for all the active integer variables i, j, k, and so on, used in the expression. The computation of $p(n)$ can be done in polynomial time, and the reference section can be set up in polynomial time.

The essential principle that guarantees that $g(x)$ can be computed in polynomial time is this: The total number of literals in the expression $g(x)$ is bounded by a polynomial function of $|x|$; computing and writing each one may require many individual operations, but the number of such operations is bounded by a polynomial, and carrying out each one (for example, searching a portion of the reference section of the tape to compare two quantities) can be done in polynomial time. Therefore, T_g takes no more than polynomial time.

In order to obtain $f(x)$ from $g(x)$, it is sufficient to relabel the variables so that for some ν they are $x_1, \ldots, x_\nu$. It is not hard to see that this can be done within time bounded by a polynomial function of $|g(x)|$, which is bounded in turn by a polynomial function of $|x|$.

11.5 SOME OTHER *NP*-COMPLETE PROBLEMS

We know from the Cook-Levin theorem that the satisfiability problem is *NP*-complete, and we know from Example 11.14 that it can be reduced to the complete subgraph problem. The conclusion, as a result of Theorem 11.17, is stated below.

Theorem 11.19
The complete subgraph problem is *NP*-complete.

Later in this section we will consider two other decision problems involving undirected graphs. Our next example is one of several possible variations

on the satisfiability problem; see Papadimitriou, *Computational Complexity*, for a discussion of some of the others. The problem *3-Sat* is the same as *Sat*, except that every conjunct in the CNF expression is assumed to be the disjunction of three or fewer literals. We denote by *3-Satisfiable* the corresponding language, with yes-instances encoded the same way. There is an obvious sense in which *3-Sat* is no harder than *Sat*. On the other hand, it is not significantly easier.

Theorem 11.20
3-Sat is *NP*-complete.

Proof
3-Satisfiable is clearly in *NP*, because *Satisfiable* is. We will prove that *3-Satisfiable* is *NP*-hard by constructing a polynomial-time reduction of *Sat* to it.

Let

$$x = \bigwedge_{i=1}^{n} A_i = A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

where each A_i is a disjunction of literals. We define $f(x)$ by the formula

$$f(x) = \bigwedge_{i=1}^{n} B_i$$

where B_i is simply A_i whenever A_i has three or fewer literals, and if $A_i = \bigvee_{j=1}^{k} a_j$ with $k > 3$, B_i is defined by

$$B_i = (a_1 \vee a_2 \vee \alpha_1) \wedge (a_3 \vee \overline{\alpha}_1 \vee \alpha_2) \wedge (a_4 \vee \overline{\alpha}_2 \vee \alpha_3) \wedge \ldots$$
$$\wedge (a_{k-3} \vee \overline{\alpha}_{k-5} \vee \alpha_{k-4}) \wedge (a_{k-2} \vee \overline{\alpha}_{k-4} \vee \alpha_{k-3}) \wedge (a_{k-1} \vee a_k \vee \overline{\alpha}_{k-3})$$

The assumption here is that the new variables $\alpha_1, \alpha_2, \ldots, \alpha_{k-3}$ do not appear either in the expression x or in any of the other B_j's. In the simplest case, $k = 4$, there are two clauses:

$$B_i = (a_1 \vee a_2 \vee \alpha_1) \wedge (a_3 \vee a_4 \vee \overline{\alpha}_1)$$

The formula for B_i is complicated, but here are the features that are crucial:

1. Every conjunct after the first one involves a literal of the form $\overline{\alpha}_r$.
2. Every conjunct before the last one involves a literal of the form α_s.
3. For every m with $2 < m < k - 1$, all the conjuncts before the one containing a_m contain an α_r for some r with $1 \leq r \leq m - 2$; and all the conjuncts after this one contain an $\overline{\alpha}_r$ for some r with $m - 1 \leq r \leq k - 3$.

Suppose now that Θ is a truth assignment satisfying x, which means that it satisfies each of the A_i's. We want to show that for every i,

with this assignment Θ to the a_j's, some choice of values for the additional variables $\alpha_1, \ldots, \alpha_{k-3}$ makes B_i true. This is true if Θ makes a_1 or a_2 true, because according to statement 1, assigning the value *false* to every α_r works. It is true if Θ makes either a_{k-1} or a_k true, because according to statement 2, assigning the value *true* to every α_r works in this case. Finally, if Θ makes a_m true for some m with $2 < m < k-1$, then according to statement 3, letting α_r be *true* for $1 \leq r \leq m-2$ and *false* for the remaining r's is enough to satisfy the formula B_i.

Conversely, if Θ does not satisfy x, then Θ fails to satisfy some A_i and therefore makes all the a_j's in A_i false. It is not hard to see that with that assignment Θ, no assignment of values to the extra variables can make B_i true. For the first conjunct of B_i to be true, α_1 must be true; for the second to be true, α_2 must be true; ...; for the next-to-last to be true, α_{k-3} must be true; and now the last conjunct is forced to be false. Therefore, if x is unsatisfiable, so is $f(x)$.

Now that we have the function f, we can easily obtain a function $f_1 : \Sigma^* \to \Sigma^*$, where $\Sigma = \{x, \overline{x}, \wedge, 1\}$, such that for every $x \in \Sigma^*$, $x \in$ *Satisfiable* if and only if $f_1(x) \in$ *3-Satisfiable*. To complete the proof it is enough to show that f_1 is computable in polynomial time, and this follows almost immediately from the fact that the length of $f_1(x)$ is bounded by a polynomial function of $|x|$.

In addition to the complete subgraph problem, many other important combinatorial problems can be formulated in terms of graphs. Here is a little more terminology. A *vertex cover* for a graph G is a set C of vertices such that every edge of G has an endpoint in C. For a positive integer k, we may think of the integers $1, 2, \ldots, k$ as distinct "colors," and use them to color the vertices of a graph. A *k-coloring* of G is an assignment to each vertex of one of the k colors so that no two adjacent vertices are colored the same. In the graph G shown in Figure 11.15, the set $\{1, 3, 5, 7\}$ is a vertex cover for G, and it is easy to see that there is no vertex cover with fewer than four vertices. Because there is a complete subgraph with four vertices, G cannot be k-colored for any $k < 4$. Although the absence of a complete subgraph with $k+1$ vertices does not automatically imply that the graph has a k-coloring, you can easily check that in this case there is a 4-coloring of G.

The *vertex cover problem* is this: Given a graph G and an integer k, is there a vertex cover for G with k vertices? The *k-colorability problem* is the problem: Given G and k, is there a k-coloring of G? Both problems are in *NP*. In the second case, for example, colors between 1 and k can be assigned nondeterministically to all the vertices, and then the edges can be examined to determine whether the two endpoints of each one are colored differently.

Theorem 11.21
The vertex cover problem is *NP*-complete.

Proof
We show that the problem is *NP*-hard by reducing the complete subgraph problem to it. That is, we show that an instance $I = (G, k)$ of the complete subgraph problem can be transformed in polynomial time to an instance $f(I) = (G_1, k_1)$ of the vertex cover problem in such a way that G has a complete subgraph with k vertices if and only if G_1 has a vertex cover with k_1 vertices.

If $G = (V, E)$, let G_1 be the *complement* of G: $G_1 = (V, E_1)$, where

$$E_1 = \{(i, j) \mid i, j \in V \text{ and } (i, j) \notin E\}$$

and let $k_1 = |V| - k$.

On the one hand, if vertices $v_1, v_2, \ldots, v_k$ determine a complete subgraph of G, then because no edge of G_1 joins two of the v_i's, every edge in G_1 must have as an endpoint an element of $V - \{v_1, \ldots, v_k\}$, so that this set of k_1 vertices is a vertex cover for G_1.

Conversely, suppose $U = \{u_1, \ldots, u_{|V|-k}\}$ is a vertex cover for G_1. Then the set $V - U$ has k vertices; we will show that it determines a complete subgraph of G. By the definition of complement, every two vertices are joined either by an edge in G or by one in G_1. However, two vertices in $V - U$ cannot be joined by an edge in G_1, because by definition of vertex cover, an edge in G_1 must contain a vertex in U. Therefore, two vertices in $V - U$ must be joined by an edge in G.

Because the transformation from G to G_1 can be carried out in polynomial time, the proof is complete.

Theorem 11.22
The k-colorability problem is *NP*-complete.

Proof
We show that *3-Sat* is polynomial-time reducible to the k-colorability problem. This means we must show how to construct, for each CNF expression x in which all conjuncts have three or fewer literals, a graph G_x and an integer k_x such that x is satisfiable if and only if there is a k_x-coloring of G_x.

The construction is very simple for the x's with fewer than four distinct variables and more complicated for the others. In the first case, we let I_1 be a fixed yes-instance of the k-colorability problem and I_0 a fixed no-instance. We can simply *answer* the instance x, because there

are no more than eight possible truth assignments to the variables, and let the assigned value be I_1 if x is a yes-instance and I_0 otherwise.

Otherwise, let $x = A_1 \wedge A_2 \wedge \ldots \wedge A_c$ be an instance of *3-Sat* with the n variables $x_1, x_2, \ldots, x_n$, where $n \geq 4$. We construct the instance (G_x, k_x) by letting $k_x = n + 1$ and defining the graph $G_x = (V_x, E_x)$ as follows. There are $3n + c$ vertices:

$$V_x = \{x_1, x_2, \ldots, x_n, \quad \overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n, \quad y_1, y_2, \ldots, y_n, \quad A_1, A_2, \ldots, A_c\}$$

There are six types of edges:

$$E_x = \{(x_i, \overline{x}_i) \mid 1 \leq i \leq n\} \cup \{(y_i, y_j) \mid i \neq j\} \cup \{(y_i, x_j) \mid i \neq j\}$$
$$\cup \{(y_i, \overline{x}_j) \mid i \neq j\} \cup \{(x_i, A_j) \mid x_i \notin A_j\} \cup \{(\overline{x}_i, A_j) \mid \overline{x}_i \notin A_j\}$$

(The notations $x_i \notin A_j$ and $\overline{x}_i \notin A_j$ both mean that the literal does not occur in the conjunct A_j.)

Suppose, on the one hand, that this graph is $(n + 1)$-colorable. No two of the y_i's can be colored the same, because every two are adjacent; therefore, n distinct colors are required for those vertices. Let us assume that the y_i's are colored with the colors $1, 2, \ldots, n$. For any fixed i, consider the vertices x_i and $\overline{x}_i$. Neither can be colored the same as y_j for any $j \neq i$, because they are both adjacent to every such y_j; and they cannot be colored the same, because they are adjacent to each other; the only possibility is that color i is used for one and color $n + 1$ for the other.

We have now accounted for the first $3n$ vertices, and we consider the remaining c. For each j with $1 \leq j \leq c$, there must be some variable x_i such that neither x_i nor $\overline{x}_i$ appears in A_j, because A_j has no more than three literals and there are at least four distinct variables. Since A_j is adjacent to both x_i and $\overline{x}_i$, and one of them is colored $n + 1$, A_j cannot be. The vertex A_j must be colored with color i for some i with $1 \leq i \leq n$. One of the two vertices x_i and $\overline{x}_i$ is colored i; the one that is must appear in the conjunct A_j, because otherwise it would be adjacent to the vertex A_j. Let z_j be a literal (either x_i or $\overline{x}_i$) appearing in A_j that is colored the same as A_j. Then no z_j is the negation of any other z_l, because if it were, one of the two would be colored $n + 1$. This means that there is a way of assigning truth values to the variables that makes every z_j true. The effect of this assignment is to satisfy each A_j, and it follows that x is satisfiable.

This argument is essentially reversible. We begin by coloring each y_i with color i. If x is satisfied by the assignment Θ, then each conjunct A_j contains a literal z_j that is made true by Θ. If z_j is either x_i or $\overline{x}_i$, we color it with color i and its negation with color $n + 1$. Once we have done this, if x_l is a literal remaining uncolored, so is $\overline{x}_l$; we use color l to color whichever one is made true by Θ, and $n + 1$ to color the other. To see that this is permissible, suppose for example that x_l is colored with l. If $x_l \notin$

A_j, then A_j cannot be colored l because of x_l (because $x_l \notin A_j$), and A_j cannot be colored l because of $\overline{x}_l$, because $\overline{x}_l$ is not true for the assignment Θ. We conclude that if x is satisfiable, then G_x is $(n+1)$-colorable.

As in the previous theorem, it is easy to see that this construction can be carried out in polynomial time, and it follows that *3-Sat* is polynomial-time reducible to the k-colorability problem.

We now have five problems that are *NP*-complete, and there are thousands of other problems that are known to be. The more that are discovered, the more potential ways there are to reduce *NP*-complete problems to other problems, and the number of problems known to be *NP*-complete is growing constantly. The book by Garey and Johnson, even though it was published some time ago, remains a very good reference for a general discussion of the topic and contains a varied list of *NP*-complete problems, grouped according to category (graphs, sets, and so on).

NP-completeness is still a somewhat mysterious property. Some decision problems are in P, while others that seem similar turn out to be *NP*-complete (Exercises 11.20 and 11.26). In the absence of either an answer to the $P \stackrel{?}{=} NP$ question or a definitive way of characterizing the problems that are tractable, people generally take a pragmatic approach. Many real-life decision problems require some kind of solution. If a polynomial-time algorithm does not present itself, maybe the problem can be shown to be *NP*-complete. In this case, it is probably not worthwhile spending a lot more time looking for a polynomial-time solution: Finding one would almost certainly be difficult, because if one were found, many other problems would have polynomial-time solutions, and many people have already spent a lot of time looking unsuccessfully for them. The next-best thing might be to look for an algorithm that produces an approximate solution to the problem, or one that provides a solution for a restricted set of instances. Both approaches represent active areas of research.

EXERCISES

11.1. Suppose $f, g : \mathcal{N} \to \mathcal{N}$. Show that if $f + g = O(\max(f, g))$.

11.2. In Example 11.2, find a formula for $\tau_T(n)$ when n is odd.

11.3. Find the time complexity function for each of these TMs:

a. The TM in Example 7.10 that computes the reverse function from $\{a, b\}^*$ to $\{a, b\}^*$.

b. The *Copy* TM shown in Figure 7.19.

11.4. Show that for any decidable decision problem, there is a way to encode instances of the problem so that the corresponding language can be accepted by a TM with linear time complexity.

11.5. Suppose $L_1, L_2 \subseteq \Sigma^*$ can be accepted by TMs with time complexity τ_1 and τ_2, respectively. Find appropriate functions g and h such that

$L_1 \cup L_2$ and $L_1 \cap L_2$ can be accepted by TMs with time complexity in $O(g)$ and $O(h)$, respectively.

11.6. Describe in at least some detail a two-tape TM accepting the language of Example 11.2 and having linear time complexity.

11.7. Show that if L can be accepted by a multitape TM with time complexity f, then L can be accepted by a one-tape machine with time complexity $O(f^2)$.

11.8. The nondeterministic Turing machine we described that accepts *Satisfiable* repeats the following operation or minor variations of it: starting with a string of 1's in the input string, delimited at both ends by a symbol other than 1, and locating some or all of the other occurrences of this string that are similarly delimited. How long does an operation of this type take on a one-tape TM? Use your answer to argue that the TM accepting *Satisfiable* has time complexity $O(n^3)$.

11.9. a. Show that if $L \in P$, then $L' \in P$. (L' is the complement of L.)

b. Explain carefully why the fact that $L \in NP$ does not obviously imply that $L' \in NP$.

11.10. a. Let L_1 and L_2 be languages over Σ_1 and Σ_2, respectively. Show that if $L_1 \leq_p L_2$, then $L' \leq_p L_2'$.

b. Show that if there is an NP-complete language L whose complement is in NP, then the complement of every language in NP is in NP.

11.11. Show that if $L_1, L_2 \subseteq \Sigma^*$, $L_1 \in P$, and L_2 is neither $\emptyset$ nor Σ^*, then $L_1 \leq_p L_2$.

11.12. a. If every instance of problem P_1 is an instance of problem P_2, and if P_2 is hard, then P_1 is hard. True or false?

b. Show that *3-Sat* $\leq_p$ *Sat*.

c. Generalize the result in part (b) in some appropriate way.

11.13. Show that every Boolean expression F involving the variables a_1, a_2, $\ldots$, a_t is logically equivalent to an expression of the form

$$\bigwedge_{i=1}^{k} \bigvee_{j=1}^{l_i} b_{i,j}$$

where each $b_{i,j}$ is either an a_r or an $\overline{a}_r$ (i.e., an expression in conjunctive normal form), as follows:

a. Show that if for every assignment Θ of truth values to the variables $a_1, \ldots, a_t$, and every j with $1 \leq j \leq t$, we define $\alpha_{\Theta,j}$ by the formula

$$\alpha_{\Theta,j} = \begin{cases} a_j & \text{if the assignment } \Theta \text{ makes } a_j \text{ true} \\ \overline{a}_j & \text{otherwise} \end{cases}$$

then the two expressions

$$\bigvee_{\Theta \in S} \bigwedge_{j=1}^{t} \alpha_{\Theta,j} \text{ and } \neg F$$

are logically equivalent, where S is the set of assignments that satisfy $\neg F$. This shows that $\neg F$ is equivalent to an expression in *disjunctive normal form* (a disjunction of subexpressions, each of which is a conjunction of literals).

b. It follows that $\neg F$ is equivalent to an expression in disjunctive normal form. By generalizing the De Morgan laws (see Section 1.1), show that F is equivalent to a CNF expression.

11.14. In each case below, find an equivalent expression that is in conjunctive normal form.

a. $a \rightarrow (b \wedge (c \rightarrow (d \vee e)))$

b. $\vee_{i=1}^{n}(a_i \wedge b_i)$

11.15. Show that if $k \geq 4$, the k-satisfiability problem is NP-complete.

11.16. Show that both of the following decision problems are in P.

a. *DNF-Sat*: Given a Boolean expression in disjunctive normal form (the disjunction of clauses, each of which is a conjunction of literals), is it satisfiable?

b. *CNF-Tautology*: Given a Boolean expression in CNF, is it a tautology (i.e., satisfied by every possible truth assignment)?

11.17. Show that the general satisfiability problem, Given an arbitrary Boolean expression, not necessarily in conjunctive normal form, involving the variables $x_1, x_2, \ldots, x_v$, is it satisfiable? is NP-complete.

11.18. Explain why it is appropriate to insist on binary notation when encoding instances of the primality problem, but not necessary to do this when encoding subscripts in instances of the satisfiability problem.

11.19. Consider the language L of all strings $e(T)e(x)1^n$, where T is a nondeterministic Turing machine, $n \geq 1$, and T accepts x by some sequence of no more than n moves. Show that the language L is NP-complete.

11.20. Show that the 2-colorability problem (Given a graph, is there a 2-coloring of the vertices?) is in P.

11.21. Consider the following algorithm to solve the vertex cover problem. First, we generate all subsets of the vertices containing exactly k vertices. There are $O(n^k)$ such subsets. Then we check whether any of the resulting subgraphs is complete. Why is this not a polynomial-time algorithm (and thus a proof that $P = NP$)?

11.22. Let f be a function in PF, the set of functions from Σ^* to Σ^* computable in polynomial time. Let A (a language in Σ^*) be in P. Show

that $f^{-1}(A)$ is in P, where by definition, $f^{-1}(A) = \{z \in \Sigma^* \mid f(z) \in A\}$.

11.23. In an undirected graph G, with vertex set V and edge set E, an *independent* set of vertices is a set $V_1 \subseteq V$ such that no two elements of V' are joined by an edge in E. Let *IS* be the decision problem: Given a graph G and an integer k, is there an independent set of vertices with at least k elements? Denote by *VC* and *CSG* the vertex cover problem and the complete subgraph problem, respectively. Construct a polynomial-time reduction from each of the three problems *IS*, *VC*, and *CSG* to each of the others. (Part of this problem has already been done, in the proof of Theorem 11.21. In the remaining parts, you are not to use the *NP*-completeness of any of these problems.)

11.24. Show that if the positive integer n is not a prime, then for every x with $0 < x < n - 1$ that satisfies $x^{n-1} \equiv_n 1$, there is a prime factor p of $n - 1$ satisfying

$$x^{(n-1)/p} \equiv_n 1$$

This makes it a little easier to understand how it might be possible to test a number n for primeness in nondeterministic polynomial time: In the result of Fermat mentioned in Example 11.9, we don't have to test that $x^m \not\equiv_n 1$ for *every* m satisfying $1 < m < n - 1$, but only for numbers m of the form $(n - 1)/p$, where p is a prime. Suggestion: Because of Fermat's result, if $x^{n-1} \equiv_n 1$ and n is not prime, there is an integer m with $1 < m < n - 1$ such that $x^m \equiv_n 1$; show that the smallest such m is a divisor of $n - 1$.

11.25. For languages $L_1, L_2 \subseteq \{a, b\}^*$, let

$$L_1 \oplus L_2 = L_1\{a\} \cup L_2\{b\}$$

a. Show that $L_1 \leq_p L_1 \oplus L_2$ and $L_2 \leq_p L_1 \oplus L_2$.

b. Show that for any languages L, L_1, and L_2 over $\{a, b\}$, with $L \neq \{a, b\}^*$, if $L_1 \leq_p L$ and $L_2 \leq_p L$, then $L_1 \oplus L_2 \leq_p L$.

11.26. †Show that the 2-Satisfiability problem is in P.

11.27. †Show that both P and NP are closed under the operations of union, intersection, concatenation, and Kleene *.

11.28. †Show that the following decision problem is *NP*-complete: Given a graph G in which every vertex has even degree, and an integer k, does G have a vertex cover with k vertices? (The degree of a vertex is the number of edges containing it.) Hint: given an arbitrary graph G, find a way to modify it by adding three vertices so that all the vertices of the new graph have even degree.

11.29. Give an alternate proof of the *NP*-completeness of the vertex cover problem, by directly reducing the satisfiability problem to it. Below is a suggested way to proceed; show that it works.

Starting with a CNF expression $x = \bigwedge_{i=1}^{m} A_i$, where $A_i = \bigvee_{j=1}^{n_i} a_{i,j}$ and x involves the n variables $v_1, \ldots, v_n$, construct an instance (G, k) of the vertex cover problem as follows: first, there are vertices v_l^t and v_l^f for each variable v_l, and these two are connected by an edge; next, for each conjunct A_i there is a complete subgraph G_i with n_i vertices, one for each literal in A_i; finally, for a variable v_l and a conjunct A_i, there is an edge from the vertex in G_i corresponding to v_l, either to v_l^t (if v_l appears in A_i) or to v_l^f (if v_l doesn't appear in A_i). Finally, let k be the integer $n + (n_1 - 1) + \ldots + (n_m - 1)$.

11.30. Show that the following decision problem is *NP*-complete. Given a finite set A, a collection C of subsets of A, and an integer k, is there a subset A_1 of A having k or fewer elements such that $A_1 \cap S \neq \emptyset$ for each S in the collection C?

11.31. †The *exact cover* problem is the following: given finite subsets $S_1, \ldots, S_k$ of a set, with $\bigcup_{i=1}^{k} S_i = A$, is there a subset J of $\{1, 2, \ldots, k\}$ such that for any two distinct elements i and j of J, $S_i \cap S_j = \emptyset$ and $\bigcup_{i \in J} S_i = A$?

Show that this problem is *NP*-complete by constructing a reduction from the k-colorability problem.

11.32. †The *sum-of-subsets* problem is the following: Given a sequence $a_1, a_2, \ldots, a_n$ of integers, and an integer M, is there a subset J of $\{1, 2, \ldots, n\}$ such that $\sum_{i \in J} a_i = M$?

Show that this problem is *NP*-complete by constructing a reduction from the exact cover problem.

11.33. †The *partition* problem is the following: Given a sequence $a_1, a_2, \ldots, a_n$ of integers, is there a subset J of $\{1, 2, \ldots, n\}$ such that $\sum_{i \in J} a_i = \sum_{i \notin J} a_i$?

Show that this problem is *NP*-complete by constructing a reduction from the sum-of-subsets problem.

11.34. †The *0–1 knapsack* problem is the following: Given two sequences $w_1, \ldots, w_n$ and $p_1, \ldots, p_n$ of nonnegative numbers, and two numbers W and P, is there a subset J of $\{1, 2, \ldots, n\}$ such that $\sum_{i \in J} w_i \leq W$ and $\sum_{i \in J} p_i \geq P$? (The significance of the name is that w_i and p_i are viewed as the *weight* and the *profit* of the ith item, respectively; we have a "knapsack" that can hold no more than W pounds, and the problem is asking whether it is possible to choose items to put into the knapsack subject to that constraint, so that the total profit is at least P.)

Show that the 0–1 knapsack problem is *NP*-complete by constructing a reduction from the partition problem.

This page intentionally left blank

SOLUTIONS TO SELECTED EXERCISES

CHAPTER 1

1.3(b). $(m_1 \leq m_2) \wedge ((m_1 < m_2) \vee (d_1 < d_2))$. The second clause in this statement (the portion after the $\wedge$) could also be written $((m_1 = m_2) \rightarrow (d_1 < d_2))$.

1.8(d). Two correct expressions are

$$(A \cap B) \cup (A \cap C) \cup (B \cap C) \cap (A \cap B \cap C)'$$
$$\text{and } (A \cap B \cap C') \cup (A \cap B' \cap C) \cup (A' \cap B \cap C).$$

1.12(b). One algorithm is described by the following pseudocode, which processes the symbols left-to-right, starting with the leftmost bracket. The answer is the final value of e. This algorithm, however, is correct only if we can assume that there are no duplicate elements in the set. The problem of counting the distinct elements without this assumption is more complicated, because duplicate elements such as $\{\emptyset, \{\emptyset\}\}$ and $\{\{\emptyset\}, \emptyset\}$ might not be written the same way.

```
read ch, the next nonblank character;
e = 0; (element count)
u = 1; (current number of unmatched left brackets)
while there are nonblank characters remaining
{ read ch, the next nonblank character;
  if ch is a left bracket
    u = u + 1;
  else if ch is a right bracket
    u = u - 1;
  if u = 1 and ch is either Ø or a right bracket
    e = e + 1;
}
```

1.15(a). If either A or B is a subset of the other, then the two sets are equal. If not, then $2^A \cup 2^B \subseteq 2^{A \cup B}$, and if $a \in A - B$ and $b \in B - A$, the subset $\{a, b\}$ is an element of $2^{A \cup B}$ that is not in $2^A \cup 2^B$.

1.21(a). No, because for every subset $A \in T$, $f(A) = f(\mathcal{N} - A)$, so that f is not one-to-one.

1.24. We consider the case when the relation is to be reflexive, not symmetric, and transitive. If R is a reflexive relation on the set, it must contain the ordered pairs $(1, 1)$, $(2, 2)$, and $(3, 3)$. If it is reflexive and not symmetric, it must contain at least one additional pair; suppose it contains $(1, 2)$. We can verify that the relation with only these four pairs is transitive. If x, y, and z are elements of $\{1, 2, 3\}$ such that (x, y) and (y, z) are in R, then at least one of the statements $x = y$ and $y = z$ must be true, and it follows that (x, z) must be in R. The conclusion is that $R = \{(1, 1), (2, 2), (3, 3), (1, 2)\}$ is a reflexive, nonsymmetric, transitive relation on $\{1, 2, 3\}$ containing as few ordered pairs as possible.

1.34. Suppose that $xy = yx$. In order to find a string α such that $x = \alpha^p$ and $y = \alpha^q$ for some p and q, which implies that $|\alpha|$ is a divisor of both $|x|$ and $|y|$, we let d be the greatest common divisor of $|x|$ and $|y|$. Then we can write $x = x_1x_2 \dots x_p$ and $y = y_1y_2 \dots y_q$, where all the x_i's and y_j's are strings of length d and the numbers p and q have no common factors.

Now we would like to show that the x_i's and y_j's are actually all the same string α. The first step is to observe that $x^q y^p = y^p x^q$, because the fact that $xy = yx$ allows us to rearrange the pieces of $x^q y^p$ so that the string doesn't change but all the occurrences of y have been moved to the beginning.

The string $x^q y^p = y^p x^q$ has length $2pqd$. The prefix of $x^q y^p$ of length pqd is x^q, and the prefix of $y^p x^q$ of the same length is y^p. Therefore, $x^q = y^p$.

In the string $x^q = (x_1x_2 \dots x_p)^q$, the substring x_1 occurs at positions $1, pd + 1, 2pd + 1, \dots, (q - 1)pd + 1$. In the string $y^p = (y_1y_2 \dots y_q)^p$, the strings that occur at these positions (which must therefore be equal) are $y_{r_0}, y_{r_1}, \dots, y_{r_{q-1}}$, where for each i, $r_i \equiv ip + 1 \bmod q$. The numbers $r_0, r_1, \dots, r_{q-1}$ must be distinct, because the only way for r_i and r_j to be the same if $0 \le i < j \le q - 1$ is for ip and jp to be congruent mod q; this means that $(j - i)p \equiv 0 \bmod q$, or that $(j - i)p$ is divisible by q, and this is impossible if p and q have no common factors. The strings y_{r_0}, $\dots$, $y_{r_{q-1}}$ are all equal, and because we now know that these subscripts are distinct, we may conclude that all the y_j's are equal, say to the string α. Virtually the same argument shows that all the x_i's are equal, and we have the conclusion we want.

1.35. No. Suppose $L = L_1L_2$ and neither L_1 nor L_2 is $\{\Lambda\}$. Because every even-length string of a's is in L, there are arbitrarily long strings of a's that must be in either L_1 or L_2. Similarly, there are arbitrarily long strings of b's that are in L_1 or L_2. It is not possible for L_1 to have a nonnull string of a's and L_2 to have a nonnull string of b's, because the concatenation could not be in L; similarly, there can't be a string of b's in L_1 and a string of a's in L_2. The only possibilities, therefore, are for all the strings of a's and all the strings of b's to be in L_1 or for all these strings to be in

L_2. Suppose, however, that they are all in L_1. Let y be a nonnull string in L_2. Then y contains both a's and b's. L_1 contains a string x of a's with $|x| \geq |y|$. But then xy, which is in L_1L_2, has b's in the second half and not in the first, so that it can't be in L. A similar argument shows that L_2 can't contain all the strings of a's and the strings of b's. This argument shows that our original assumption, that $L = L_1L_2$ and neither L_1 nor L_2 is $\{\Lambda\}$, cannot be true.

1.40(c). To make the solution easier to describe, take the set A to be $\{1, 2, \ldots, n\}$. If R is symmetric, specifying R is accomplished by specifying the pairs (i, j) in R for which $i \leq j$, and two different such choices lead to two different symmetric relations. Therefore, the number of symmetric relations on A is the number of subsets of the set $\{(i, j) \mid 1 \leq i \leq j \leq n\}$. Because this set has $n(n + 1)/2$ elements, the number of subsets is $2^{n(n+1)/2}$.

1.44(f). We might try to either find a formula for L or find a simple property that characterizes the elements of L. First we observe that every element of L starts with a. If this doesn't seem clear, you can construct a proof by structural induction.

It may be helpful, in order to find a formula for L, to write $L = \{a\}L_1$ and to think about what the language L_1 is. The statement $a \in L$ tells us that $\Lambda \in L_1$, and the recursive part of the definition of L tells us that if $y \in L_1$ (i.e., if $ay \in L$), then yb and yba are both in L_1. We can recognize these statements about L_1 as a recursive definition of the language $\{b, ba\}^*$, which means that $L = \{a\}\{b, ba\}^*$.

Strings in L_1 cannot start with a and cannot contain two a's in a row. (Again, this is a straightforward structural induction proof.) Furthermore, every string that doesn't start with a and doesn't contain two consecutive a's is an element of L_1. This can also be proved by mathematical induction on the length of the string. Therefore, L_1 is the set of all strings in $\{a, b\}^*$ that do not begin with a and do not contain two consecutive a's, and it follows that L is the set of all strings in $\{a, b\}^*$ that begin with a and do not contain two consecutive a's.

1.56. You can construct a proof using mathematical induction that for every $n \in \mathcal{N}$, every subset A of $\mathcal{N}$ that contains n has a smallest element. This will be sufficient, because it will imply that every nonempty subset of $\mathcal{N}$ has a smallest element. (Saying that A is nonempty means that there is some n such that A contains n.)

1.60. We show using mathematical induction that for every $n \in \mathcal{N}$, if $y \in L_0$ and $|y| = n$, then $y \in L$. It will be appropriate to use strong induction.

The basis step is to show that if $y \in L_0$ and $|y| = 0$, then $y \in L$. This is true because if $|y| = 0$, then $y = \Lambda$, and $\Lambda \in L$ because of the first statement in the definition of L.

The induction hypothesis is that $k \in \mathcal{N}$ and that for every $y \in L_0$ with $|y| \leq k$, $y \in L$.

In the induction step, we will show that for every $y \in L_0$ with $|y| = k + 1$, $y \in L$. Suppose y is a string in L_0 with $|y| = k + 1$. This means that $y = a^i b^j$ for some i and j such that $i \geq j$ and $i + j = k + 1$. We must show using the definition of L that $y \in L$. Because $k \geq 0$, y cannot be Λ, and so in order to use the definition to show that $y \in L$, we must show that $y = ax$ for some $x \in L$ or that $y = axb$ for some $x \in L$. The statements $y = a^i b^j$, $i \geq j$, and $i + j > 0$ imply that $i > 0$, which means that $y = ax$ for some string x; if j is also positive, then y is also axb for some string x. The question in either case is whether $x \in L$. According to our induction hypothesis, strings in L_0 that are shorter than y are in L; so the real question in either of our two cases is whether $x \in L_0$.

If $j > 0$, then $y = axb$, where $x = a^{i-1}b^{j-1}$, and $i - 1 \geq j - 1$ because $i \geq j$. Therefore, in this case $x \in L_0$. In the other case, when $j = 0$, then $y = a^i = ax$, where $x = a^{i-1} = a^{i-1}b^0$, and this string is also in L_0. In either case, the induction hypothesis tells us that $x \in L_0$, and the definition of L then tells us that $y \in L$.

Strong induction is necessary in the first case, where $y = axb$ and $|x| = k + 1 - 2 = k - 1$, because we need to be able to say that a string in L_0 of length less than k is in L.

1.67. First we use structural induction to show that every $x \in L$ has equal numbers of a's and b's. The basis step is to show that $n_a(\Lambda) = n_b(\Lambda)$, and this is true because both numbers are 0. The induction hypothesis is that x and y are two strings in L and that both x and y have equal numbers of a's and b's. In the induction step, we must show that both $axby$ and $bxay$ also do. In both cases, the number of a's is $n_a(x) + n_a(y) + 1$ and the number of b's is $n_b(x) + n_b(y) + 1$. The induction hypothesis tells us that $n_a(x) = n_b(x)$ and similarly for y, so that the conclusion we want follows.

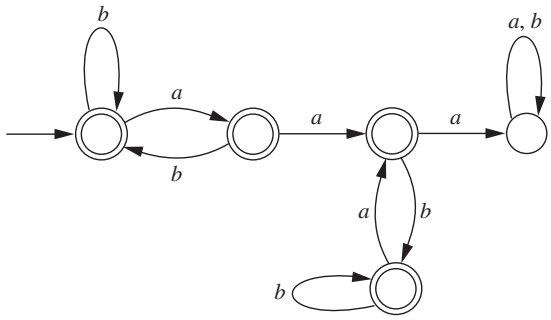
The converse is proved by strong induction on $|x|$. The basis step is to show that $\Lambda \in L$, and this is true by definition of L. The induction hypothesis is that $k \geq 0$ and that every string x with $|x| \leq k$ having equal numbers of a's and b's is in L. The statement to be shown in the induction step is that if x has equal numbers of a's and b's and $|x| = k + 1$, then $x \in L$.

If x is either aby or bay, for some y with $|y| = k - 1$, then y has equal numbers of a's and b's, and is in L by the induction hypothesis. Therefore, x, which is either $a\Lambda by$ or $b\Lambda ay$, is in L by definition of L. The remaining case is when x starts with either aa or bb; we complete the proof in the first case. For a string $z \in \{a, b\}^*$, let $d(z) = n_a(z) - n_b(z)$, and consider $d(z)$ for the prefixes z of x. Because $d(aa) = 2$ and $d(x) =$

0, and adding a single symbol to a prefix changes the d value by 1, there must exist a prefix z longer than aa for which $d(z) = 1$. Let z_1 be the *longest* such prefix. Then x must be z_1bz_2 for some string z_2—otherwise, because $d(z_1a) = 2$, there would be a prefix longer than z_1 for which $d = 1$. We now know that $x = aay_1bz_2$, and $d(ay_1) = d(aay_1b) = 0$. Therefore, $d(z_2) = 0$ as well. By the induction hypothesis, both ay_1 and z_2 are in L. Therefore, $x = a(ay_1)bz_2$ is also in L, according to the definition of L.

CHAPTER 2

2.1(h).

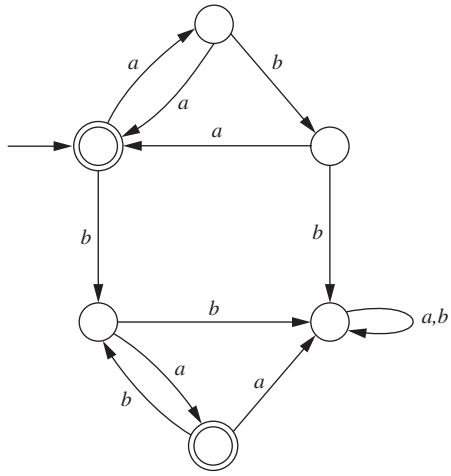


2.5. See the proof of the formula $(xy)^r = y^r x^r$ in Example 1.27, and use the same approach. Interpret the statement to be "For every string y, $P(y)$ is true," where $P(y)$ is "For every state q and every string x, $\delta^*(q, xy) = \delta^*(\delta^*(q, x), y)$." Use structural induction on y, based on the recursive definition of Σ^* in Example 1.17.

2.9(a). Suppose L has this property for some finite set S and some n. If L_0 is the set of elements of L with length less than n, then $L = L_0 \cup \Sigma^* S$. The finite language L_0 can be accepted by an FA. For each $y \in S$, the language of strings over Σ that end with y can be accepted by an FA, and $\Sigma^* S$ is the union of a finite number of languages of this form. Therefore, L is the union of a finite number of languages that can be accepted by an FA.

2.9(b). Suppose L is finite. Let n be an integer larger than the length of the longest string in L, and let S be the empty set. The statement "For every x with $|x| \geq n$, $x \in L$ if and only if x ends with an element of S" is true, because both parts of the if-and-only-if statement are false.

2.12(h).



2.21(b). Consider any two strings a^m and a^n, where $m < n$. These can be distinguished by the string ba^{m+2}, because $a^m ba^{m+2} \in L$ and $a^n ba^{m+2} \notin L$.

2.22(b). Suppose for the sake of contradiction that L can be accepted by a finite automaton with n states. Let $x = a^n ba^{n+2}$. Then $x \in L$ and $|x| \geq n$, and according to the pumping lemma, $x = uvw$ for some strings u, v, and w such that $|uv| \leq n$, $|v| > 0$, and $uv^i w \in L$ for every $i \geq 0$.

Because the first n symbols of x are a's, the string v must be a^j for some $j > 0$, and $uv^2w = a^{n+j}ba^{n+2}$. This is a contradiction, because $uv^2w \in L$ but $n + 2 \neq n + j + 1$.

2.23(b). For the language L in Exercise 2.21(b), statement I is not sufficient to produce a contradiction, because the strings $u = a$, $v = a$, and $w = \Lambda$ satisfy the statement.

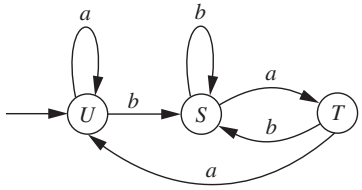
2.27(d). Let $M = (Q, \Sigma, q_0, A, \delta)$ and suppose that $\delta^*(q_0, x) = q$. Then x is a prefix of an element of L if and only if the set of states reachable from q (see Example 2.34) contains an element of A.

2.29(a). If a statement of this type is false, it is often possible to find a very simple counterexample. In (a), for example, take L_1 to be a language over Σ that cannot be accepted by an FA and L_2 to be Σ^*.

2.30(b). Notice that this definition of arithmetic progression does not require the integer p to be positive; if it did, every arithmetic progression would be an infinite set, and the result would not be true. Suppose A is accepted by the FA $M = (Q, \{a\}, q_0, F, \delta)$. Because Q is finite, there are integers m and p such that $p > 0$ and $\delta^*(q_0, a^m) = \delta^*(q_0, a^{m+p})$. It follows that for every $n \geq m$, $\delta^*(q_0, a^n) = \delta^*(q_0, a^{n+p})$. In particular, for every n satisfying $m \leq n < m + p$, if $a^n \in A$, then $a^{n+ip} \in A$ for every $i \geq 0$, and if $a^n \notin A$, then $a^{n+ip} \notin A$ for every $i \geq 0$. If $n_1, n_2, \ldots, n_r$ are the values

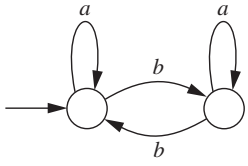
of n between m and $m + p - 1$ for which $a^n \in A$, and $P_1, \ldots, P_r$ are the corresponding arithmetic progressions (that is, $P_j = \{n_j + ip \mid i \geq 0\}$), then S is the union of the sets $P_1, \ldots, P_r$ and the finite set of all k with $0 \leq k < m$ for which $a^k \in A$. Therefore, S is the union of a finite number of arithmetic progressions.

2.38a. According to Theorem 2.36, the languages having this property can be accepted by a 3-state FA in which the states are these three sets: the set S of strings ending with b; the set T of strings ending with ba; and the set U of strings that don't end with either b or ba. The state U will be the initial state, because the set U contains Λ, and the transitions are shown in this diagram.



It might seem as though each of the eight ways of designating the accepting states from among these three produces an FA accepting one of the languages. However, if L is one of these eight languages and L can actually be accepted by an FA with fewer than three states, then I_L will have fewer than three equivalence classes. This is the case for the languages $\emptyset$ (accepted by the FA in which none of the three states is accepting), $\{a, b\}^*$ (the complement of $\emptyset$), S (accepted by the FA in which S is the only accepting state), and $T \cup U$ (the complement of S). The remaining four ways of choosing accepting states give us the languages T, $S \cup U$, $S \cup T$, and U, and these are the answer to the question.

2.42(a). If I_L has two equivalence classes, then both states of the resulting FA must be reachable from the initial state, and there must be exactly one accepting state. There are three ways of drawing transitions from the initial state, because the other state must be reachable, and there are four ways of drawing transitions from the other state. Each of these twelve combinations gives a different set A of strings corresponding to the initial state, and each of these twelve is a possible answer to (a). For example, the set $(\{a\} \cup \{b\}\{a\}^*\{b\})^*$ corresponds to the transitions shown below.



2.42(b). For each of these twelve sets A, there are two possible languages L for which I_L has two equivalence classes and $A = [\Lambda]$: A and the complement of A.

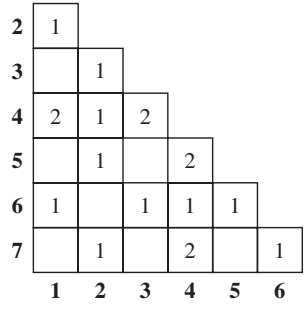
2.48. We sketch the proof that two strings x and y are equivalent if and only if they have the same number of a's and the same number of b's. It is easy to see that two strings like this are equivalent, or L-indistinguishable. Suppose that x has i a's and $i + p$ b's, and y has j a's and $j + q$ b's, and either $i \neq j$ or $i + p \neq j + q$. One case in which this happens is when $p \neq q$; we consider this case, and the case in which $p = q$ and $i \neq j$ is similar.

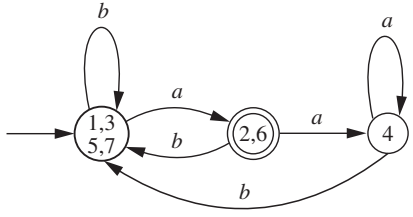
We want to show that for some z, $xz \in L$ and $yz \notin L$. We can make xz an element of L by picking z to contain N a's and $N - p$ b's, for any number N that is at least p, so that xz has $i + N$ a's and $i + p + (N - p) = i + N$ b's. The string yz will have $j + N$ a's and $j + (q - p) + N$ b's, and all we need to show in this case is that an N can be chosen such that the second number is not a multiple of the first. The ratio of the two numbers is

$$\frac{j + N + (q - p)}{j + N} = 1 + \frac{q - p}{j + N}$$

The second term in the second expression might be positive or negative, but in order to guarantee that it is not an integer, it is sufficient to pick N such that $j + N$ is larger than $|q - p|$.

2.55(f). See the figure below. As in Example 2.41, the square corresponding to a pair (i, j) contains either nothing or a positive integer. In the first case, the pair is not marked during the execution of Algorithm 2.40, and in the second case the integer represents the pass of the algorithm in which the pair is marked.





2.60. Let L be the language $\{a^n \mid n > 0 \text{ and } n \text{ is not prime}\}$. Every sufficiently large integer is the sum of two positive nonprimes, because an even integer 8 or greater is the sum of two even integers 4 or greater, and an odd number n is $1 + (n - 1)$. Therefore, L^2 contains all sufficiently long strings of a's, which means that L^2 can be accepted by an FA. However, L cannot.

2.61. If $L \subseteq \{a\}^*$, then the subset $Lengths(L^*) = \{|x| \mid x \in L^*\}$ is closed under addition. It is possible to show that every subset of $\mathcal{N}$ closed under addition is the union of a finite number of arithmetic progressions, so that the result follows from Exercise 2.30(a).

CHAPTER 3

3.5(b). The set $\{\emptyset\} \cup \{\{x\} \mid x \in \Sigma^*\}$ of all zero- or one-element languages over Σ.

3.7(h). $(b + abb)^*$.

3.7(k). Saying that a string doesn't contain bba means that if bb appears, then a cannot appear anywhere later in the string. Therefore, a string not containing bba consists of a prefix not containing bb, possibly followed by a string of b's. By including in the string of b's *all* the trailing b's, we may require that the prefix doesn't end with b. Because $(a + ba)^*$ corresponds to strings not containing bb and not ending with b, one answer is $(a + ba)^*b^*$.

3.13(b). Suppose r satisfies $r = c + rd$ and Λ does not correspond to d, and let x be a string corresponding to r. To show that x corresponds to cd^*, assume for the sake of contradiction that x does not correspond to cd^j for any j. If we can show that for every $n \geq 0$, x corresponds to the regular expression rd^n, we will have a contradiction (because strings corresponding to d must be nonnull), and we may then conclude that $r = cd^*$.

The proof is by induction. We know that x corresponds to $r = rd^0$. Suppose that $k \geq 0$ and that x corresponds to rd^k. The assumption is that $rd^k = (c + rd)d^k = cd^k + rd^{k+1}$. If x does not correspond to cd^k, it must correspond to rd^{k+1}.

3.14. Make a sequence of passes through the expression. In each pass, replace any regular expression of the form $\emptyset + r$ or $r + \emptyset$ by r, any regular expression of the form $\emptyset r$ or $r\emptyset$ by $\emptyset$, and any occurrence of $\emptyset^*$ by Λ. Stop after any pass in which no changes are made. If $\emptyset$ remains in the expression, then the expression must actually be $\emptyset$, in which case the corresponding language is empty.

3.17(b). $\{aaa\}^*$ cannot be described this way. To simplify the discussion slightly, we assume the alphabet is $\{a\}$; every subset of $\{a\}^*$ describable by a generalized regular expression not involving * has a property that this language lacks.

For $L \subseteq \{a\}^*$, we let $e(L)$ be the set of all even integers k such that $k = |x|$ for some $x \in L$, and we let $e'(L)$ be the set of nonnegative even integers not in $e(L)$. Similarly, a positive odd number k is in the set $o(L)$ if $k = |x|$ for some $x \in L$, and $k \in o'(L)$ otherwise. We will say that L is *finitary* if either $e(L)$ or $e'(L)$ is finite and either $o(L)$ or $o'(L)$ is finite. In other words, L is finitary if, in the set of all nonnegative even integers, the subset of those that are lengths of

strings in L either is finite or has finite complement, and similarly for the set of all positive odd integers.

It can be verified that if L_1 and L_2 are both finitary, then all the languages $L_1 \cup L_2$, $L_1 \cap L_2$, L_1L_2, and L_1' are also finitary. We will check this in one case. Suppose, for example, that L_1 and L_2 are nonempty and that $e(L_1)$ is finite, $o(L_1)$ is finite, $e(L_2)$ is finite, and $o'(L_2)$ is finite. This means that L_1 is actually finite, and L_2 has only a finite number of even-length strings but contains all the odd-length strings except for a finite number.

Then $L_1 \cup L_2$ contains only a finite number of even-length strings, and it contains all but a finite number of the odd-length strings, so that $e(L_1 \cup L_2)$ and $o'(L_1 \cup L_2)$ are finite. Because $L_1 \cap L_2$ is finite, both $e(L_1 \cap L_2)$ and $o(L_1 \cap L_2)$ are finite.

Consider the number $e(L_1L_2)$. If L_1 has no strings of odd length, then the even-length strings in L_1L_2 are all obtained by concatenating even-length strings in L_1 with even-length strings in L_2, which means that $e(L_1L_2)$ is finite; if there is an odd-length string in L_1, however, then because L_2 contains almost all the odd-length strings, L_1L_2 contains almost all the even-length strings—i.e., $e'(L_1L_2)$ is finite. Similarly either $o(L_1L_2)$ is finite (if L_1 contains no even-length strings), or $o'(L_1L_2)$ is finite (if L_1 has a string of even length).

The number $e'(L_1')$ is finite, because every nonnegative even integer that is not the length of an element of L_1' must be the length of an element of L_1, and $o'(L_1')$ is also finite. Similarly, $e'(L_2')$ and $o(L_2')$ are finite.

The language $L = \{aaa\}^*$ is not finitary, because all four of the sets $e(L)$, $e'(L)$, $o(L)$, and $o'(L)$ are infinite. Therefore, L cannot be obtained from the finitary languages $\emptyset$ and $\{a\}$ using only the operations of union, intersection, concatenation, and complement.

3.28(a). We must show two things: first, that for every $s \in S$, $s \in \Lambda(T)$, and second, that for every $t \in \Lambda(T)$, $\Lambda(\{t\}) \subseteq \Lambda(T)$. The first is true because $S \subseteq T$ and (by definition) $T \subseteq \Lambda(T)$. The second is part of the definition of $\Lambda(T)$.

3.28(f). We always have $S \subseteq \Lambda(S)$ and therefore $\Lambda(S)' \subseteq S' \subseteq \Lambda(S')$. If the first and third of these three sets are equal, then the equality of the first two implies that $S = \Lambda(S)$, and the equality of the second and third implies that $S' = \Lambda(S')$. The converse is also clearly true: If $S = \Lambda(S)$ and $S' = \Lambda(S')$, then $\Lambda(S)' = \Lambda(S')$. Thus, the condition $\Lambda(S') = \Lambda(S)'$ is true precisely when there is neither a Λ-transition from an element of S to an element of S' nor a Λ-transition from an element of S' to an element of S. (A more concise way to say this is to say that S and S' are both Λ-closed—see Exercise 3.29.)

3.30. The proof is by structural induction on y and will be very enjoyable for those who like mathematical notation. The basis step is for $y = \Lambda$. For

every $q \in Q$ and every $x \in \Sigma^*$,

$$\begin{aligned}\cup\{\delta^*(p, \Lambda) \mid p \in \delta^*(q, x)\} &= \cup\{\Lambda(\{p\}) \mid p \in \delta^*(q, x)\} \\ &= \Lambda(\cup\{\{p\} \mid p \in \delta^*(q, x)\}) \\ &= \Lambda(\delta^*(q, x)) = \delta^*(q, x) = \delta^*(q, x\Lambda)\end{aligned}$$

In this formula, we are using parts of Exercise 3.28. The first equality uses the definition of δ^*. The second uses part (d) of Exercise 3.28. The third is simply the definition of union. The fourth uses part (b) of Exercise 3.28 and the fact that according to the definition, $\delta^*(q, x)$ is $\Lambda(S)$ for some set S.

Now assume that for the string y,

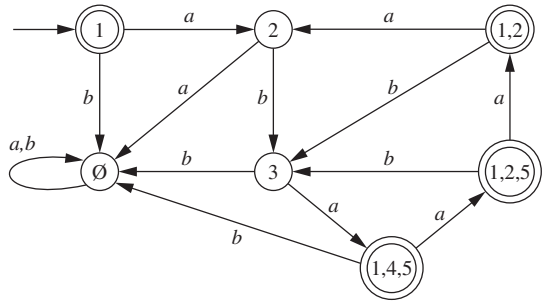
$$\delta^*(q, xy) = \cup\{\delta^*(p, y) \mid p \in \delta^*(q, x)\}$$

for every state q and every string $x \in \Sigma^*$. Then for every $\sigma \in \Sigma$,

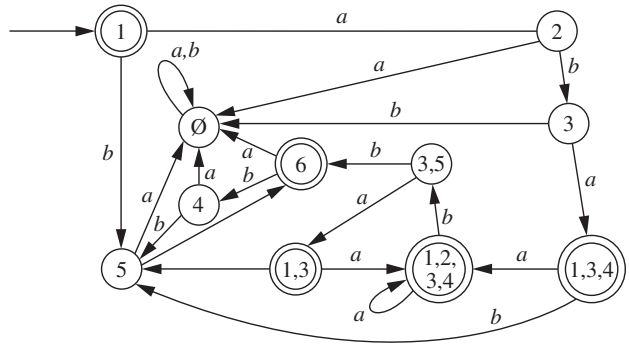
$$\begin{aligned}\delta^*(q, x(y\sigma)) &= \delta^*(q, (xy)\sigma) \\ &= \Lambda(\cup\{\delta(p, \sigma) | p \in \delta^*(q, xy)\}) \\ &= \Lambda(\{s | \exists p(p \in \delta^*(q, xy) \wedge s \in \delta(p, \sigma))\}) \\ &= \Lambda(\{s | \exists p(p \in \cup\{\delta^*(r, y) | r \in \delta^*(q, x)\} \wedge s \in \delta(p, \sigma))\}) \\ &= \Lambda(\{s | \exists r(r \in \delta^*(q, x) \wedge \exists p(p \in \delta^*(r, y) \wedge s \in \delta(p, \sigma)))\}) \\ &= \Lambda(\{s | \exists r(r \in \delta^*(q, x) \wedge s \in \cup\{\delta(p, \sigma) | p \in \delta^*(r, y)\})\}) \\ &= \Lambda(\cup\{\cup\{\delta(p, \sigma) | p \in \delta^*(r, y)\} | r \in \delta^*(q, x)\}) \\ &= \cup\{\Lambda(\cup\{\delta(p, \sigma) | p \in \delta^*(r, y)\}) | r \in \delta^*(q, x)\} \\ &= \cup\{\delta^*(r, y\sigma) | r \in \delta^*(q, x)\}\end{aligned}$$

The second equality uses the definition of δ^*. The third is just a restatement of the definition of union. The fourth uses the induction hypothesis. The fifth, sixth, and seventh are also just restatements involving the definitions of various unions. The eighth uses part (c) of Exercise 3.28 (actually, the generalization of part (c) from the union of two sets to an arbitrary union). The ninth is the definition of δ^*.

3.38(d).



3.40(b).



3.51(b). Simplified tables with the regular expressions $r(p, q, k)$ for $0 \le k \le 2$ are shown in Table 1 below.

The language corresponds to the regular expression $r(1, 2, 2) + r(1, 3, 2)r(3, 3, 2)^*r(3, 2, 2)$, or

$$(a + b)a^* + (a + b)a^*b((a + ba + bb)a^*b)^*(a + ba + bb)a^*$$

3.54. The reduction step in which p is eliminated is to redefine $e(q, r)$ for every pair (q, r) where neither q nor r is p. The new value of $e(q, r)$ is

$$e(q, r) + e(q, p)e(p, p)^*e(p, r)$$

where the $e(q, r)$ term in this expression represents the previous value. The four-part figure on the next page illustrates this process for the FA shown in Figure 3.40b. The first picture shows the NFA obtained by adding the states q_0 and q_f; the subsequent pictures describe the machines obtained by eliminating the states 1, 2, and 3, in that order. The regular expression $e(q_0, q_f)$ in the last picture describes the language. It looks the same as the regular expression obtained in the solution to Exercise 3.51(b),

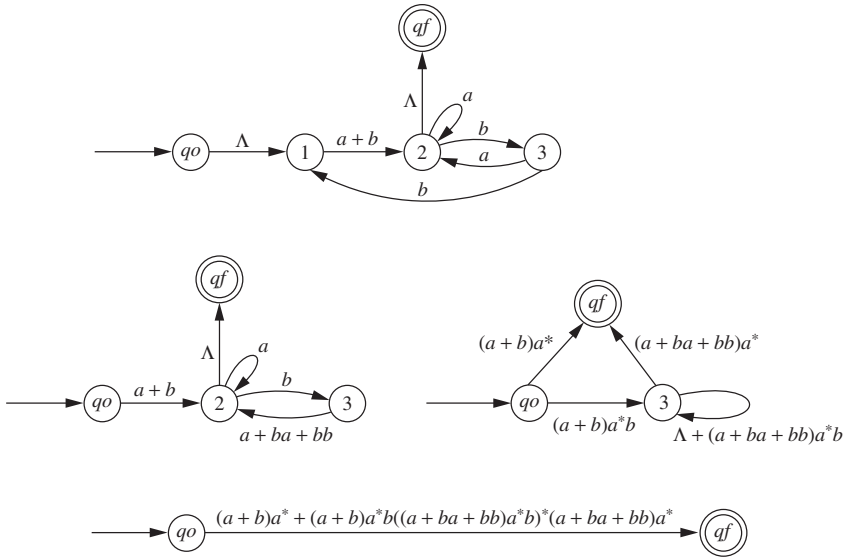
Table 1

p	*r(p,1,0)*	*r(p,2,0)*	*r(p,3,0)*
1	Λ	$a + b$	$\emptyset$
2	$\emptyset$	$\Lambda + a$	b
3	b	a	Λ

p	*r(p,1,1)*	*r(p,2,1)*	*r(p,3,1)*
1	Λ	$a + b$	$\emptyset$
2	$\emptyset$	$\Lambda + a$	b
3	b	$a + ba + bb$	Λ

p	*r(p,1,2)*	*r(p,2,2)*	*r(p,3,2)*
1	Λ	$(a + b)a^*$	$(a + b)a^*b$
2	$\emptyset$	a^*	a^*b
3	b	$(a + ba + bb)a^*$	$\Lambda + (a + ba + bb)a^*b$

although the two algorithms do not always produce regular expressions that look the same.



CHAPTER 4

4.1(d). The set of all strings in $\{a, b\}^*$ not containing the substring bb.

4.6. If these two productions are the only ones with variables on the right side, then no variable other than S can ever be involved in a derivation from S, and the only other S-productions are $S \to x_1 \mid x_2 \mid \ldots \mid x_n$, where each x_i is an element of $\{a, b\}^*$. In this case, no string of the form ayb that is not one of the x_i's can be derived, which implies that not all nonpalindromes can be generated.

4.13(b). $S \to aaSb \mid aaSbbb \mid aSb \mid \Lambda$. It is straightforward to check that every string $a^i b^j$ obtained from this CFG satisfies the inequalities $i/2 \leq j \leq 3i/2$. We show the converse by strong induction on i. If $i \leq 2$, the strings $a^i b^j$ satisfying the inequalities are Λ, ab, aab, $aabb$, and $aabbb$, and all these strings can be derived from S. Suppose that $k \geq 2$ and that for every $i \leq k$ and every j satisfying $i/2 \leq j \leq 3i/2$, the string $a^i b^j$ can be derived. We must show that for every j satisfying $(k+1)/2 \leq j \leq 3(k+1)/2$, the string $a^{k+1} b^j$ can be obtained. We do this by considering several cases.

If k is odd and $j = (k+1)/2$, then $a^{k+1} b^j = (aa)^j b^j$, which can be obtained by using the first production j times.

If k is even and $j = k/2 + 1$, then $a^{k+1} b^j = (aa)^{k/2} abb^{k/2}$, and the string can be obtained by using the first production $k/2$ times and the third production once.

If k is odd and $j = (k+3)/2$, then $a^{k+1}b^j = (aa)^{(k-1)/2}a^2b^2b^{(k-1)/2}$, and the string can be obtained by using the first production $(k-1)/2$ times and the third production twice.

If k is even and $j = k/2+2$, then $a^{k+1}b^j = (aa)^{k/2-1}a^3b^3b^{k/2-1}$, and the string can be obtained by using the first production $k/2-1$ times and the third production 3 times.

We have taken care of the cases in which $(k+1)/2 \le j \le (k+4)/2$. In all the remaining cases, $(k+5)/2 \le j \le 3(k+1)/2$ (and k is either even or odd). It is not difficult to check that $(k-1)/2 \le j-3 \le 3(k-1)/2$. It follows from the induction hypothesis that the string $a^{k-1}b^{j-3}$ can be obtained from the CFG. Therefore, the string $a^{k+1}b^j$ can be obtained by using the same derivation except that one more step is added in which the second production is used.

4.17. Call this context-free grammar G, and let L be the language of strings with $n_a(x) = 2n_b(x)$. We sketch the proof by strong induction that $L \subseteq L(G)$. The string Λ is in $L(G)$. Suppose $k \ge 0$ and every string in L of length k or less is in $L(G)$, and let $x \in L$, with $|x| = k+1$.

For a string z, let $d(z) = n_a(z) - 2n_b(z)$. One of these six statements must be true: (1) x begins with aab; (2) x begins with ab; (3) x begins with aaa and ends with b; (4) x begins with aaa and ends with a; (5) x begins with ba; (6) x begins with bb. We prove the induction step in cases (2) and (4).

If $x = aby$, then $d(ab) = -1$ and $d(x) = d(aby) = 0$. Consider the first prefix of x for which $d \ge 0$. Adding the last symbol of this prefix causes d to increase, which means that the prefix ends with a. Therefore, $x = abwaz$, where $d(w) = d(z) = 0$. The induction hypothesis implies that w and z are both in $L(G)$; we can then derive x by starting the derivation with

$$S \Rightarrow aSbSaS \Rightarrow abSaS$$

and continuing to derive w from the first S and z from the second.

In case (4), $x = aaaya$. Because $d(aaa) > 0$, we consider the shortest nonnull prefix of x for which $d \le 0$. This prefix must end with b, so that $x = aaawbza$. The possible values of $d(aaaw)$ are 1 and 2. If it is 1, then $d(aaw) = 0$, so that $x = a(aaw)bza$. In this case, aaw and z are in $L(G)$ because of the induction hypothesis; we can derive x by starting the derivation with

$$S \Rightarrow aSbSaS \Rightarrow aSbSa$$

and continuing so as to derive aaw from the first S and z from the second. If the value is 2, then $d(aw) = 0$, and $x = aa(aw)b(za)$, so that $d(za)$ must be 0. The induction hypothesis tells us that aw and za are in $L(G)$, and we can then obtain x by a derivation that starts

$$S \Rightarrow aSaSbS \Rightarrow aaSbS$$

4.32(a). In a derivation tree for the string with i occurrences of a, where $i > 1$, the root node has two children, each of which is an S. From those two S's, i occurrences of a must be obtained. The possibilities are that 1 of them comes from the left of the two and $i - 1$ from the right, or two come from the left and $i - 2$ from the right, or $\ldots$, or $i - 1$ come from the left and 1 comes from the right. The resulting formula is

$$n_i = \sum_{j=1}^{i-1} n_j n_{i-j}$$

4.40(b). Using the criterion in part (a), we prove by strong induction that every string x of parentheses in which no prefix has more right parentheses than left can be derived from the grammar. The string Λ can be derived. Suppose that $k \geq 0$ and that for every string x of length $\leq k$ in which every prefix has at least as many left parentheses as right, x can be derived.

Suppose that $|x| = k + 1$ and every prefix of x has at least as many left parentheses as right. Then $x = (y$ for some y. If every prefix of y has at least as many left parentheses as right, then y can be derived from the grammar, by the induction hypothesis; therefore, $x = (y$ can, because we can start the derivation $S \Rightarrow (S$ and continue by deriving y from S.

If y has a prefix with an excess of right parentheses, let y_1 be the smallest such prefix. Then y_1 must be $y_2)$, for some string y_2. The string y_2 has equal numbers of left and right parentheses, and every prefix of y_2 has at least as many left as right. We can write $x = (y_2)z$ for some string z. Because every prefix of x has at least as many left as right, and because (y_2) has equal numbers, every prefix of z must have at least as many left as right. Therefore, by the induction hypothesis, $S \Rightarrow^* y_2$ and $S \Rightarrow^* z$. It follows that $S \Rightarrow^* (y_2)z$, since we can start the derivation $S \Rightarrow (S)S$ and continue by deriving y_2 from the first S and z from the second.

4.45. Consider the CFG in part (b). We prove by strong induction that for every $n \geq 0$, and every x such that $S \Rightarrow^* x$ and $|x| = n$, x has only one leftmost derivation. This is true for $n = 0$. Suppose that $k \geq 0$ and that the statement is true for every $n \leq k$, and now suppose that $S \Rightarrow^* x$ and $|x| = k + 1$. The first step of a derivation of x is $S \Rightarrow S(S)$. It follows that $x = (y)z$, where $S \Rightarrow^* y$, $S \Rightarrow^* z$, $|y| \leq k$, and $|z| \leq k$. Therefore, by the induction hypothesis, both y and z have only one leftmost derivation. In order to conclude that x does, we must eliminate the possibility that x can be written in two different ways as $x = (y)z$, where $S \Rightarrow^* y$ and $S \Rightarrow^* z$. The reason this is impossible is that if $x = (y)z$ and y and z are both balanced strings of parentheses, then the right parenthesis shown is the unique mate of the left one. This is shown in Theorem 4.25 for a different grammar, and the argument for this CFG is essentially the same.

4.56(b). In order to show that $L(G_1) \subseteq L(G)$, one can prove that if $T \Rightarrow^*_{G_1} x$ or $U \Rightarrow^*_{G_1} x$, then $S \Rightarrow^*_G x$. In the first case, the statement is true because both productions $S \rightarrow aSbS \mid c$ are present in G; the second can be proved using strong induction on $|x|$.

In order to prove that $L(G) \subseteq L(G_1)$, we can adapt the results of Exercise 4.40. We can see from the productions in G that every string generated from S must end with c, and therefore that every b in such a string must be immediately preceded by c; moreover, every string generated from S has exactly one more c than b. These observations allow us to say that strings in $L(G)$ match the regular expression $a^*c(ba^*c)^*$, and Exercise 4.40 suggests that $L(G)$ is the set of strings matching this expression in which every prefix has at least as many a's as b's. Although we will not prove this characterization of $L(G)$, we will use it in showing that $L(G) \subseteq L(G_1)$. The proof uses strong induction to prove the following three statements simultaneously: (i) for every $x \in L(G)$ having a derivation in G that involves only the productions $S \rightarrow aSbS \mid c$ (which means that $x \in L(G)$ and x has equal numbers of a's and b's), $S_1 \Rightarrow T \Rightarrow^*_{G_1} x$; (ii) for every $x \in L(G)$ having a derivation in G that begins with the production $S \rightarrow aS$, $S_1 \Rightarrow U \Rightarrow^*_{G_1} x$; and (iii) for every x in $L(G)$, if x has more a's than b's and every derivation of x in G begins with the production $S \rightarrow aSbS$, then $S_1 \Rightarrow U \Rightarrow^*_{G_1} aTbU \Rightarrow^*_{G_1} x$.

The basis step is straightforward. Suppose all three statements are true for strings of length $\leq k$, and now suppose $x \in L(G)$ and $|x| = k + 1$. If x has a derivation involving only $S \rightarrow aSbS$ and $S \rightarrow c$, then either $x = c$, or $x = aybz$ for strings y and z in $L(G)$ that also have derivations involving only these productions. Because G_1 contains the productions $T \rightarrow aTbT \mid c$, it follows from the induction hypothesis that $S_1 \Rightarrow T \Rightarrow^*_{G_1} x$. If x has a derivation beginning with $S \rightarrow aS$, then $x = ay$ for some $y \in L(G)$, the induction hypothesis implies that $y \in L(G_1)$, and $x \in L(G_1)$ because G_1 contains the productions $S_1 \rightarrow U$ and $U \rightarrow aS_1$. Finally, suppose that $x \in L(G)$, x has more a's than b's, and every derivation of x in G begins with the production $S \rightarrow aSbS$. Then the string x matches the regular expression $a^*c(ba^*c)^*$, every prefix has at least as many a's as b's, and there is at least one b. Let $x = x_1y_1$, where x_1 is of the form a^*cba^*c. If x started with aa, then x_1 would be ax_2 for some $x_2 \in L(G)$, and x would have a derivation beginning $S \Rightarrow aS$; therefore, $x = acbz$ for some string z. The string z also matches the regular expression $a^*c(ba^*c)^*$, every prefix of z has at least as many a's as b's, and z has more a's than b's. The induction hypothesis implies that $U \Rightarrow^*_{G_1} z$, and we may conclude that $S_1 \Rightarrow U \Rightarrow aTbU \Rightarrow^*_{G_1} x$.

4.56(c). It can be shown by strong induction on n that for every x with $|x| \geq 1$, if x can be derived from one of the three variables in n steps, then x has only one leftmost derivation from that variable. We present the induction step in the third case, and show that if x can be derived from U in $k + 1$ steps, then x has only one leftmost derivation from U. If x has a prefix

of the form ayb, where y has equal numbers of a's and b's, then the first step in a derivation of x from U must be $U \Rightarrow aTbU$ (because if there were a derivation starting $U \Rightarrow aS_1$, some string derivable from S_1 would have more b's than a's), and if no prefix of x has this form, then the first step in a derivation of x from U must be $U \Rightarrow aS_1$.

If the first step in the derivation of x from U is $U \Rightarrow aS_1$, the induction hypothesis tells us that there is only one way to continue a leftmost derivation, which means that x has only one leftmost derivation from U. If the first step is $U \Rightarrow aTbU$, then $x = aybz$, where y and z can be derived from T and U, respectively, in k or fewer steps. By the induction hypothesis, y has only one leftmost derivation from T and z has only one from U. Furthermore, there is only one choice for the string y, because the prefix ayb of x must be the shortest prefix having equal numbers of a's and b's (otherwise y would have a prefix with an excess of b's). Therefore, x has only one leftmost derivation from U.

CHAPTER 5

5.8(b). See Table 2.

Table 2

Move No.	State	Input	Stack Symbol	Move(s)
1	q_0	a	Z_0	(q_1, aZ_0)
2	q_0	b	Z_0	(q_0, bZ_0)
3	q_0	a	b	(q_1, Λ)
4	q_0	b	b	(q_0, bb)
5	q_1	a	Z_0	(q_2, aaZ_0)
6	q_1	b	Z_0	(q_1, bZ_0)
7	q_1	a	a	(q_2, aaa)
8	q_1	a	b	(q_2, a)
9	q_1	b	a	(q_1, Λ)
10	q_1	b	b	(q_1, bb)
11	q_2	a	Z_0	$(q_2, aZ_0), (q_2, aaZ_0)$
12	q_2	a	a	$(q_2, aa), (q_2, aaa)$
13	q_2	a	b	$(q_2, \Lambda), (q_2, a)$
14	q_2	b	a	(q_2, Λ)
15	q_2	b	b	(q_2, bb)
16	q_2	Λ	Z_0	(q_3, Z_0)
	(all other combinations)			none

The initial state is q_0 and the accepting state q_3. The first a read will be used to cancel one b, the second will be used to cancel two b's, and subsequent a's can be used either way. (Note that every string in the language has at least two a's.)

The PDA stays in q_0 as long as no a's have been read. If and when the first a is read, it will be used to cancel a single b, either by removing b

from the stack, or, if a is the first input symbol, saving it on the stack for a subsequent b to cancel. The state q_1 means that a single a has been read; if and when a second a is read, it will be used to cancel two b's (either by popping two b's from the stack, or by popping one from the stack and pushing an a to be canceled later, or by pushing two a's), and the PDA will go to state q_2. In q_2, each subsequent a will cancel either one or two b's, and the machine can enter q_3 if the stack is empty except for Z_0.

5.13. An NFA can be constructed to accept L, having as states pairs (q, α), where q is a state in M and α is a string of k or fewer stack symbols, representing the current contents of M's stack. Such a pair provides a complete description of M's current status, and for every such pair and every input (either Λ or an input symbol) it is possible using M's definition to specify the pairs that might result. Furthermore, there are only finitely many such pairs. The initial state of the NFA is (q_0, Z_0), where q_0 is the initial state of M, and the pairs (q, α) that are designated as accepting states are those for which q is an accepting state of M.

5.15. Yes. We can carry out the construction in the solution to Exercise 5.13, with one addition. There is one additional state s to which any transition goes that would otherwise result in a pair (q, α) with $|\alpha| > k$. All transitions from s return to s. This guarantees that the NFA functions correctly, because for every input string x in L, there is a sequence of moves corresponding to x that never causes the NFA to enter s.

5.18(c). In the PDA shown in Table 3, q_0 is the initial state and the only accepting state. Let x be the current string. Because we wish to compare $n_a(x)$ to $2n_b(x)$, we will essentially treat every b we read as if it were two b's. This means that if we read a b when the top stack symbol is either b or Z_0, we push two b's onto the stack and go to q_1. If the top stack symbol is a when we read a b, we pop it off and go to a temporary state; at that point, without reading an input symbol, we pop a second a off if there is one, and push a b on otherwise, and go to q_1 in either case. Finally, top stack symbol

Table 3

Move No.	State	Input	Stack Symbol	Move(s)
1	q_0	a	Z_0	(q_1, aZ_0)
2	q_0	b	Z_0	(q_1, bbZ_0)
3	q_1	a	a	(q_1, aa)
4	q_1	a	b	(q_1, Λ)
5	q_1	b	b	(q_1, bb)
6	q_1	b	a	(q_t, Λ)
7	q_t	Λ	a	(q_1, Λ)
8	q_t	Λ	Z_0	(q_1, bZ_0)
9	q_1	Λ	Z_0	(q_0, Z_0)
	(all other combinations)			none

Z_0 in either q_0 or q_1 means that currently the number of a's is twice the number of b's, so that if we are in q_1 we make a Λ-transition to q_0.

5.19. To do this for L_1L_2, the basic idea is the same as for FAs: The composite machine acts like M_1 until it reaches an accepting state, then takes a Λ-transition to the initial state of M_2. However, we have to be more careful because of the stack. For example, it's possible for M_1 to accept a string and to have an empty stack when it's done processing that string, but if our composite machine ever has an empty stack, it can't move. The solution is to fix it so that the composite machine can never empty its stack. This can be done by inserting a new stack symbol Z under the initial stack symbol of M_1 and then returning to the initial state of M_1.

From that point on, the moves are the same as those of M_1, unless Z appears on top of the stack or unless the state is an accepting state of M_1. If Z is on top of the stack and the state is not an accepting state in M_1, the machine crashes. If it reaches an accepting state of M_1, then regardless of what's on top of the stack, it takes a Λ-transition to the initial state of M_2 and pushes the initial stack symbol of M_2 onto the stack. From then on, it acts like M_2, except that it never removes the initial stack symbol of M_2 from the stack. (If M_2 ever removed its initial stack symbol, that would be the last move it made, so our new machine can achieve the same result as M_2 without ever removing this symbol.)

5.20. Let $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ be a DPDA accepting L. We construct a DPDA $M_1 = (Q_1, \Sigma, \Gamma, q_0', Z_0, A, \delta_1)$ accepting the new language as follows. $Q_1 = Q \cup Q_1$, where Q_1 is a set containing another copy q' of every $q \in Q$. (In particular, the initial state of M_1 is the primed version of q_0.) For each $q' \in Q'$, and each $a \in \Sigma \cup \{\Lambda\}$ and $X \in \Gamma$, $\delta_1(q', a, X) = \delta(q, a, X)'$. In other words, for inputs that are Λ or ordinary alphabet symbols, the new PDA behaves the same way with the states q' that M does with the original states. In addition, if $q \in A$, then for every stack symbol X, $\delta_1(q', \#, X) = \{(q, X)\}$. Finally, δ_1 agrees with δ for elements of Q and inputs other than #.

What this means is that M_1 acts the same way as M up to the point where the input # is encountered, except that the new states allow it to remember that it has not yet seen #. Once this symbol is seen, if the current state is q' for some $q \in A$ (i.e., if M would have accepted the current string), then the machine switches over to the original states for the rest of the processing. It enters an accepting state subsequently if and only if both the substring preceding the # and the entire string except for the # would be accepted by M.

5.25(b). For $x \in \{a, b\}^*$, let $d(x) = 2n_b(x) - n_a(x)$. We use the three states q_0, q_+, and q_- to indicate that the quantity $d(x)$ is currently 0, positive, or negative, respectively, and the number of $*$'s on the stack indicates the current absolute value of $d(x)$. q_0 is the accepting state. The counter automaton is shown in Table 4.

Table 4

Move No.	State	Input	Stack Symbol	Move(s)
1	q_0	a	Z_0	$(q_-, *Z_0)$
2	q_0	b	Z_0	$(q_+, **Z_0)$
3	q_-	a	$*$	$(q_-, **)$
4	q_-	b	$*$	(q_t, Λ)
5	q_t	Λ	Z_0	$(q_+, *Z_0)$
6	q_t	Λ	$*$	(q_-, Λ)
7	q_-	Λ	Z_0	(q_0, Z_0)
8	q_+	b	$*$	$(q_+, ***)$
9	q_+	a	$*$	(q_+, Λ)
10	q_+	Λ	Z_0	(q_0, Z_0)
	(all other combinations)			none

Line 4 contains the move in which $d(x) < 0$ and we read the symbol b. If there are at least two $*$'s on the stack, we pop two off; if there is only one, we go to q_+ and push one $*$ onto the stack. The state q_t is a temporary state used in this procedure.

From both q_- and q_+, we can make a Λ-transition to q_0 when the stack is empty except for Z_0.

5.34(a). See Table 5.

Table 5

Stack (reversed)	Unread Input	Derivation Step
Z_0	[]][[]]	
Z_0 [	]][[]]	
Z_0 [S	]][[]]	$\Rightarrow$ []][[]]
Z_0 [S]	[[]]	
Z_0 [S][	[]]	
Z_0 [S][[	]]	
Z_0 [S][[S	]]	$\Rightarrow$ [S][[]]
Z_0 [S][[S]	]	
Z_0 [S][[S]S	]	$\Rightarrow$ [S][[S]]
Z_0 [S][S	]	$\Rightarrow$ [S][[S]S]
Z_0 [S][S]		
Z_0 [S][S]S		$\Rightarrow$ [S][S]
Z_0 [S]S		$\Rightarrow$ [S][S]S
Z_0 S		$\Rightarrow$ [S]S
		S

5.41(c).

$$S \to S_1\$ \quad S_1 \to abX \quad X \to TX \mid \Lambda \quad T \to aU \quad U \to Tbb \mid b$$

CHAPTER 6

6.2(e). Suppose L is a CFL, and let n be the integer in the pumping lemma. Let $u = a^n b^n c^n$. Then $u = vwxyz$, where the usual three conditions on v, w, x, y, and z hold.

If wy contains at least one a, then since $|wxy| \leq n$, wy can contain no c's. Therefore, vw^0xy^0z contains fewer than n a's but exactly n c's, and so it is impossible for this string to be in L. If wy contains no a's, then vw^2xy^2z contains either more than n b's or more than n c's, but exactly n a's. In this case also, the string cannot be in L. Therefore, we have a contradiction.

6.9(a). Call this language L. Then L is the union of the two languages $\{a^i b^j c^k | i \geq j\}$ and $\{a^i b^j c^k \mid i \geq k\}$, each of which is a CFL. The complement of L in $\{a, b, c\}^*$ is $L' = (\{a\}^*\{b\}^*\{c\}^*)' \cup \{a^i b^j c^k \mid i < j \text{ and } i < k\}$. From the formula $L' \cap \{a\}^*\{b\}^*\{c\}^* = \{a^i b^j c^k \mid i < j \text{ and } i < k\}$ and Theorem 6.13, it follows that if L is a CFL, so is $\{a^i b^j c^k \mid i < j \text{ and } i < k\}$. However, we can show using the pumping lemma that this is not the case. Suppose $\{a^i b^j c^k \mid i < j \text{ and } i < k\}$ is a CFL and let n be the integer in the pumping lemma. Let $u = a^n b^{n+1} c^{n+1}$. Then $u = vwxyz$, where the usual conditions hold. If wy contains a's, it can contain no c's, and we obtain a contradiction by considering vw^2xy^2z. If wy contains no a's, we obtain a contradiction by considering vxz.

6.11(a). Suppose $L = \{a^i b^{i+k} a^k \mid k \neq i\}$ is a CFL. Let n be the integer in Ogden's lemma, let $u = a^n b^{2n+n!} a^{n+n!}$, and designate the first n positions of u as the distinguished positions. Then $u = vwxyz$, where wy contains at least one from the first group of a's and $vw^i xy^i z \in L$ for every $i \geq 0$. If either w or y contained both a's and b's, then clearly vw^2xy^2z would not be in L. Also, if neither w nor y contained any b's, then vw^2xy^2z would not preserve the balance between a's and b's required for membership in L. Therefore, w contains only a's from the first group, and y contains only b's. If the lengths of w and y were different, vw^2xy^2z could not be in L. If the lengths are equal, say p, then let $i = 1 + n!/p$. Then

$$vw^i xy^i z = a^{n+(i-1)p} b^{2n+n!+(i-1)p} a^{n+n!} = a^{n+n!} b^{2n+2n!} a^{n+n!}$$

which is not an element of L. This contradiction implies that L is not a CFL.

6.16(b). Yes. Given a PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ accepting L, we can construct another PDA $M_1 = (Q_1, \Sigma, \Gamma, q_1, Z_0, A_1, \delta_1)$ accepting the set of prefixes of elements of L as follows. Q_1 can be taken to be $Q \times \{0, 1\}$—in other words, a set containing two copies $(q, 0)$ and $(q, 1)$ of each element of Q. The initial state q_1 is $(q_0, 0)$, and the set A_1 is $A \times \{1\}$. For each combination $(q, 0)$ (where $q \in Q$), $a \in \Sigma$, and $X \in \Gamma$, the set $\delta_1((q, 0), a, X)$ is simply $\{((p, 0), \alpha) \mid (p, \alpha) \in \delta(q, a, X)\}$; however, $\delta_1((q, 0), \Lambda, X)$ contains not only the elements $((p, 0), \alpha)$ for which

$(p, \alpha) \in \delta(q, \Lambda, X)$, but one additional element, $((q, 1), X)$. For each $q \in Q$, $a \in \Sigma$, and $X \in \Gamma$, $\delta_1((q, 1), a, X) = \emptyset$. Finally, for each $q \in Q$ and each $X \in \Gamma$, $\delta_1((q, 1), \Lambda, X)$ is the union of the sets $\{((p, 1), \alpha) \mid (p, \alpha) \in \delta(q, a, X)\}$ over all $a \in \Sigma \cup \{\Lambda\}$.

The idea here is that M_1 starts out in state $(q_0, 0)$ and acts on the states $(q, 0)$ exactly the same way M acts on the states q, until such time as M_1 takes a Λ-transition from its current state $(q, 0)$ to $(q, 1)$. From this point on, M_1 can make the same moves on the states $(q, 1)$ that M can make on the states q, changing the stack in the same way, but without reading any input. If $xy \in L$, then for any sequence of moves M makes corresponding to xy, ending in the state r, one possible sequence of moves M_1 can make on xy is to copy the moves of M while processing x, reaching some state $(q_x, 0)$, then make a Λ-transition to $(q_x, 1)$, then continue simulating the sequence on the states $(p, 1)$ but making only Λ-transitions, ending at the state $(r, 1)$ having processed the string x. Therefore, x is accepted by M_1. Conversely, if x is accepted by M_1, there is a sequence of moves corresponding to x that ends at a state $(p, 1)$, where $p \in A$. In this case, the moves of the sequence that involve states $(q, 0)$, ending at $(q_x, 0)$, correspond to moves M can make processing x, ending at the state q_x; and the remaining Λ-transitions that end at $(p, 1)$ correspond to transitions that M can make using the symbols of some string y. Therefore, if M_1 accepts x, then there is a string y such that M accepts xy.

6.16(f). No. Suppose L is a CFG, and let n be the integer in the pumping lemma. Let $u = a^p b^p c^p$, where p is a prime larger than n. Then $u = vwxyz$, where $|wy| > 0$, $|wxy| \leq n$, and $vw^i xy^i z \in L$ for every $i \geq 0$. These conditions imply that wy contains no more than two distinct symbols. Consider the string $vw^0xy^0z = vxz$, and let $j = n_a(vxz)$, $k = n_b(vxz)$, and $m = n_c(vxz)$. At least one of these is less than p, and all three are positive. Because p is prime, it is not possible for all three to have a nontrivial common factor. Therefore, L is not a CFG.

6.20(a). The languages L_1 and L_2 in Example 6.12 are both DCFLs. The language $(L_1 \cup L_2)' \cap \{a\}^*\{b\}^*\{c\}^*$ is $\{a^i b^j c^k \mid i > j > k\}$, which can be shown by the pumping lemma not to be a CFL. It follows from Theorem 6.13 that $(L_1 \cup L_2)'$ is not a CFL, and it follows from the last paragraph of Section 6.2 that $L_1 \cup L_2$ is not a DCFL.

6.20(c). Let L_1 and L_2 be as in part (a). Let $L_3 = \{d\}L_1 \cup L_2$. Then L_3 is a DCFL, because in order to accept it, the presence or absence of an initial d is all that needs to be checked before executing the appropriate DPDA to accept either L_1 or L_2. The language $\{d\}^*$ is also a DCFL. However, $\{d\}^*L_3$ is not a DCFL. The reason is that $\{d\}^*L_3 \cap \{d\}\{a\}^*\{b\}^*\{c\}^* = \{d\}L_1 \cup \{d\}L_2$. If $\{d\}^*L_3$ were a DCFL, then this language would be also (see Exercise 6.8), and it follows easily that $L_1 \cup L_2$ would be as well.

6.21(a). Let $L_1 = \{x\#y \mid x \in \mathit{pal} \text{ and } xy \in \mathit{pal}\}$. Then if *pal* were a DCFL, L_1 would be, and therefore, by Exercise 6.8, $L_1 \cap \{a\}^*\{b\}^*\{a\}^*\{\#\}\{b\}^*\{a\}^*$ would be also. However, this intersection is $\{a^i b^j a^i \# b^j a^i \mid i, j \geq 0\}$, and it's easy to show using the pumping lemma that this language is not even a CFL.

6.23. To simplify things slightly, let $D = \{i \mid a^i \in L\}$. Then it will be sufficient to show that D is a finite union of (not necessarily infinite) arithmetic progressions of the form $\{m + ip \mid i \geq 0\}$. (See Exercise 2.30.)

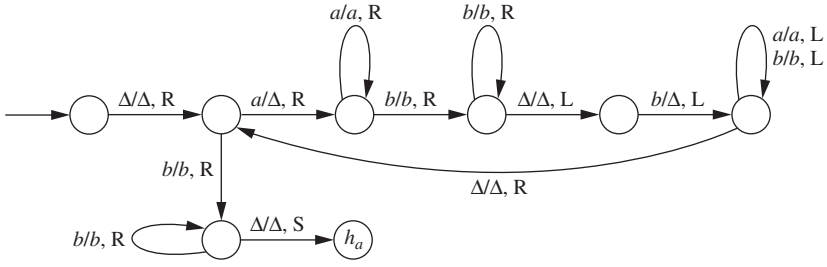
Let n be the integer in the pumping lemma applied to L. Then for every $m \in D$ with $m \geq n$, there is an integer p_m with $0 < p_m \leq n$ for which $m + ip_m \in D$ for every $i \geq 0$. There are only a finite number of distinct p_m's; let p be the least common multiple of all of them. Then for every $m \in D$ with $m \geq n$, $m + ip \in D$ for every $i \geq 0$.

For each j with $0 \leq j < p$, the set $S_j = \{r \in D \mid r \geq n \text{ and } r \equiv j \bmod p\}$ is either empty or an infinite arithmetic progression. (If it's not empty, it's the arithmetic progression $\{m_j + ip \mid i \geq 0\}$, where m_j is the smallest element of S_j.) Furthermore, $\{r \in D \mid r \geq n\}$ is the union of the sets S_j. Therefore, because the set $\{r \in D \mid r \leq n\}$ is the union of one-element arithmetic progressions, D is a finite union of arithmetic progressions.

CHAPTER 7

7.3. A configuration is determined by specifying the state, the tape contents, and the tape head position. There are $s + 2$ possibilities for the state, including h_a and h_r; $(t + 1)^{n+1}$ possibilities for the tape contents, since there are $n + 1$ squares, each of which can have an element of Γ or a blank; and $n + 1$ possibilities for the tape head position. The total number of configurations is therefore $(s + 2)(t + 1)^{n+1}(n + 1)$.

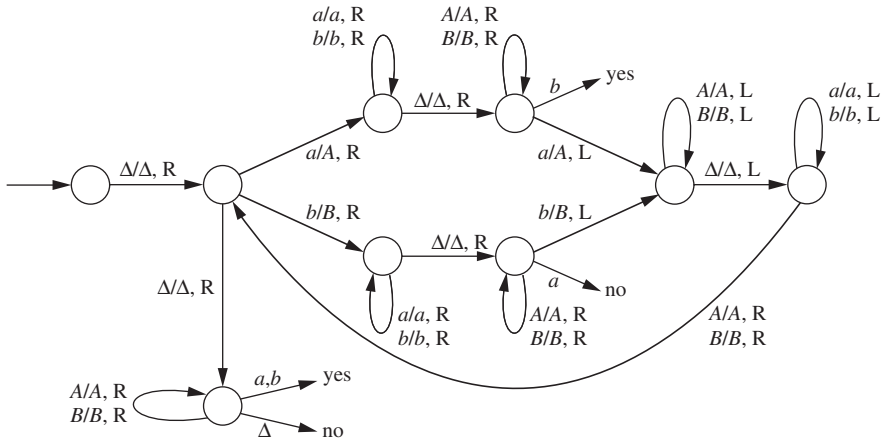
7.4(b).



7.10. A Λ-transition in an NFA or a PDA makes it possible for the device to make a move without processing an input symbol. Turing machines have such moves already, because they are not restricted to a single left-to-right pass through the input; they can make moves in which the tape head remains stationary or moves to the left.

7.13. Suppose there is such a TM T_0, and consider a TM T_1 that halts in the configuration $h_a\Delta 1$. Let n be the number of the highest-numbered square read by T_0 in the composite machine T_1T_0. Now let T_2 be another TM that halts in the configuration $h_a\Delta 1\Delta^n 1$. Because the first n tape squares are identical in the final configurations of T_1 and T_2, T_0 will not read the rightmost 1 left by T_2, and so T_2T_0 will not halt with the tape head positioned on the rightmost nonblank symbol left by T_2.

7.17(f). The figure does not show the transitions for the last phase of the computation. In two places, "yes" stands for the moves that erase the tape and halt in the accepting state with output 1; "no" stands for exactly the same moves except that the output in that case is 0.



7.24. Here is a verbal description of one solution. Start by inserting an additional blank at the beginning of the tape, to obtain $\Delta\underline{\Delta}x$, where x is the input string. Make a sequence of passes. The first moves the first symbol of x one square to the left and the last symbol one square to the right; the second moves the second symbol of x one square to the left and the next-to-last symbol one square to the right; etc. If x has odd length, reject. The tape now looks like $\Delta x_1\Delta\Delta x_2$, where x_1 and x_2 are the first and second halves of x. Now begin comparing symbols of x_1 and x_2, but replacing symbols that have been compared by blanks, rather than uppercase letters as in Example 7.5. In order for this to work, we check before each pass to see whether the symbols that are about to be compared are the last symbols in the respective halves; this avoids any attempt to make another pass if there are no more nonblank symbols, which would cause the TM to reject by attempting to move the tape head off the tape.

7.33(a). If the tape head does not move past square n, the possible number of nonhalting configurations is $s(t+1)^{n+1}(n+1)$, where s and t are the sizes of Q and Γ, respectively (see the solution to Exercise 7.3). Within

that many moves, the TM will have either halted or repeated a nonhalting configuration, so that if it has not halted it is in an infinite loop.

7.33(b). Let s be the size of Q. Suppose you watch the TM for s moves after it has reached the first square after the input string, and that each move is to the right. Then the TM has encountered the combination (q, Δ) twice for some state q, and it is in an infinite loop.

7.35. Yes. The assumption on T implies that T can make no more than k moves in which it does not move its tape head to the right. (Every such move increases the difference between the number of moves made and the current tape head position, and the assumption doesn't allow the difference to exceed k.) This observation allows us to build another TM T_1 that mimics T but always moves its tape head to the right, by allowing T_1 enough states so that it can remember the contents of the $k/2$ squares preceding the current one. By examining the transition table for T, we can predict the contents of the current tape square and the state to which T_1 goes when it finally moves its tape head to the right. Therefore, there is a TM that accepts the same language as T and always moves its tape head to the right. This implies that $L(T)$ is regular (see Exercise 7.34(b)).

7.42. A Post machine simulates a move to the left on the tape by remembering the last two symbols that have been removed from the front of the queue. Suppose for example that the initial contents of the queue are $abaab$, with the a at the front. The machine starts by placing a marker X at the rear, to produce $abaabX$. It proceeds to remove symbols from the front, remembering the last two, and placing the symbols at the rear but lagging behind by one. After one such move, the contents are $baabX$ and the machine remembers that it has removed an a; after two steps the contents are $aabXa$ and the machine remembers that the last symbol was b. When it removes X, it places X at the rear *immediately*, followed by the b it remembers removing before X, to produce $abaaXb$. At this point it begins moving symbols from the front to the rear and continues until it has removed the X, not replacing it. The contents of the queue are now $babaa$, and the move to the left has been effectively simulated.

CHAPTER 8

8.14. Suppose that L is infinite and recursive. Then L is the range of some computable increasing function from Σ^* to Σ^*. Because the function is increasing, it represents a bijection f from Σ^* to L, and we may consider its inverse $f^{-1} : L \to \Sigma^*$, which is also a bijection and also increasing. Let $x_0, x_1, \ldots$ be an enumeration of Σ^* in canonical order. The function f^{-1} is also computable, because we can evaluate $f^{-1}(y)$ by computing $f(x_0), f(x_1), \ldots$ until we find the string x_i such that $f(x_i) = y$.

A subset A of Σ^* is recursive if and only if $f(A) = \{f(x) \mid x \in A\}$ is recursive. The reason is that determining whether a string x is an element

of A is equivalent to determining whether $f(x)$ is an element of $f(A)$, and because f and f^{-1} are computable, if we can do one of these, we can do the other. For the same reason, A is recursively enumerable if and only if $f(A)$ is.

Now we can answer the question in the exercise. We can produce a subset S of L that is not recursively enumerable by starting with a subset A of Σ^* that is not and letting S be $f(A)$. The same technique works to find an S that is recursively enumerable but not recursive.

8.20(b).

$$S \to ABCS \mid ABBCS \mid AABBCS \mid BCS \mid BBCS \mid CS \mid BC$$
$$AB \to BA \; BA \to AB \; AC \to CA \; CA \to AC \; BC \to CB \; CB \to BC$$
$$A \to a \qquad B \to b \qquad C \to c$$

Checking that every string generated by this grammar satisfies the desired inequalities is a straightforward argument. The other direction is more challenging. Let us denote by n_A, n_B, and n_C the numbers of A's, B's, and C's, respectively, in a string obtained by the first seven productions. Let $x = n_A$, $y = n_B - n_A - 1$, and $z = 2n_C - n_B - 1$, and for each $i \leq 6$ let n_i be the number of times the ith production is used in the derivation of the string. We may write the equations

$$\begin{array}{lllllllllllll} x & = & n_1 & + & n_2 & + & 2n_3 & & & & & & \\ y & = & & & n_2 & & & + & n_4 & + & 2n_5 & & \\ z & = & n_1 & & & & & + & n_4 & & & + & 2n_6 \end{array}$$

It is sufficient to show that for every choice of x, y, and z whose sum is even, the equations are satisfied by some choice of nonnegative integer values for all the x_i's. This can be verified by considering cases. When x, y, and z are all even, for example, there is a solution in which n_1 is the minimum of x and z, and $n_2 = n_4 = 0$.

8.32. We will show that each production $\alpha \to \beta$, where $|\beta| \geq |\alpha|$, can be replaced by a set of productions of the desired form, so that the new grammar generates the same language.

Suppose $\alpha = \gamma_1 A_1 \gamma_2 A_2 \gamma_3 \ldots \gamma_n A_n \gamma_{n+1}$, where the A_i's are variables and each γ_i is a string of terminals. The first step is to introduce new variables $X_1, \ldots, X_n$ and productions of the desired type that allow the string $\gamma_1 X_1 \gamma_2 X_2 \gamma_3 \ldots \gamma_n X_n \gamma_{n+1}$ to be generated. These new variables can appear only beginning with the string α and will be used only to obtain the string β; the effect of this first step is to guarantee that none of the productions we are adding will cause strings not in the language to be generated. The first production is

$$\gamma_1 A_1 \gamma_2 A_2 \gamma_3 \ldots \gamma_n A_n \gamma_{n+1} \to \gamma_1 X_1 \gamma_2 A_2 \gamma_3 \ldots \gamma_n A_n \gamma_{n+1}$$

The next allows A_2 to be replaced by X_2, and so forth; the last allows A_n to be replaced by X_n.

The second step is to introduce additional variables, as well as productions that allow us to generate a string $Y_1Y_2 \dots Y_k$, where each Y_i is a variable and $k = |\alpha|$. For example, if $\gamma_1 = abc$, we can start with the productions

$$cX_1 \to Y_3X_1 \qquad bY_3 \to Y_2Y_3 \qquad aY_2 \to Y_1Y_2$$

The variable Y_4 will be the same as X_1. Similarly, if $\gamma_2 = defg$, we would use $X_1d \to X_1Y_5$, $Y_5e \to Y_5Y_6$, and so forth. All these productions are of the right form and they allow us to obtain the string $Y_1 \dots Y_k$. The Y_i's, like the X_i's, are reserved for this purpose and are used nowhere else in the grammar.

The third step is to add still more productions that produce the string β. Let

$$\beta = Z_1Z_2 \dots Z_kZ_{k+1} \dots Z_m$$

where $m \geq k$ and each Z_i is either a variable or a terminal. The productions we need are

$$Y_1Y_2 \to Z_1Y_2, \quad Y_2Y_3 \to Z_2Y_3, \quad \dots, \quad Y_{k-1}Y_k \to Z_{k-1}Y_k,$$
$$\text{and } Z_{k-1}Y_k \to Z_{k-1}Z_kZ_{k+1} \dots Z_m$$

All the productions we have added are of the proper form; they permit β to be derived from α; each one except the last one has a new variable on the right side; and the left side of the last one can have been obtained only starting from the string α. Therefore, only the strings derivable in the original grammar can be derived in the new one.

8.34(a). Let S_1 and S_2 be the start symbols of G_1 and G_2, respectively. Let G have new start symbol S as well as all the variables in G_1 and G_2 and three new variables L, M, and R. G will have the productions in G_1 and those in G_2 and the new productions

$$S \to LS_1MS_2R \qquad L\sigma \to \sigma L \qquad LM \to L \qquad LR \to \Lambda$$

(for every $\sigma \in \Sigma$). The idea is that the variables L and M prevent any interaction between the productions of G_1 and those of G_2. As long as M remains in place, no production can be applied whose left side involves variables in G_1 and terminal symbols produced by a production in G_2; and at the point when M is eliminated by the production $LM \to L$, there can be no variables in G_1 remaining, because until L reaches M it can move only by passing terminal symbols. Thereafter, as long as there are variables remaining in the string, they are separated by L from any terminal symbols that once preceded M. Therefore, no production can be applied whose left side involves both terminals produced by G_1 and variables in G_2.

8.41(c). Consider the sets P_1, the set of nonempty subsets of $\mathcal{N}$; P_2, the set of nonempty subsets of $\mathcal{N} - \{0\}$; and P_3, the set of two-subset partitions of $\mathcal{N}$.

P_1 is uncountable because $2^{\mathcal{N}}$ is. There is a bijection from P_1 to P_2, because there is a bijection from $\mathcal{N}$ to $\mathcal{N} - \{0\}$, and so P_2 is uncountable. Finally, we construct a bijection from P_2 to P_3, which will imply that P_3 is uncountable. For a nonempty subset A of $\mathcal{N} - \{0\}$, let $f(A)$ be the partition consisting of A and $\mathcal{N} - A$. The function f is onto, because for every two-set partition of $\mathcal{N}$, one of the two sets is a nonempty set not containing 0. It is also one-to-one, because if the two sets of a partition are A and $\mathcal{N} - A$, one of them fails to contain 0, so that the partition cannot be both $f(A)$ and $f(\mathcal{N} - A)$.

The set P_3 is a subset of the set of all finite partitions of $\mathcal{N}$, which is therefore also uncountable.

8.41(g). For a subset $S = \{n_0, n_1, \dots\}$ of $\mathcal{N}$, where the n_i's are listed in increasing order, we may consider the function $f : \mathcal{N} \to \mathcal{N}$ that takes the value 0 at every number i satisfying $0 \le i < n_0$, the value 1 at every i with $n_0 \le i < n_1$, and so forth. This correspondence defines a bijection from $2^{\mathcal{N}}$ to a subset of the set of all nondecreasing functions from $\mathcal{N}$ to $\mathcal{N}$, and it follows that the set of such functions is uncountable.

8.46(b). Suppose $f_0, f_1, \dots$ is a list of bijections from $\mathcal{N}$ to $\mathcal{N}$. We will describe a method for constructing a bijection $f : \mathcal{N} \to \mathcal{N}$ that is different from f_n for every n.

The idea of the construction is that for every n, two conditions will be true. First, $f(2n) \neq f_n(2n)$, which will guarantee that $f \neq f_n$; second, the set $\{f(2n), f(2n+1)\}$ will be $\{2n, 2n+1\}$, and the fact that this holds for every n will guarantee that f is a bijection from $\mathcal{N}$ to $\mathcal{N}$.

The second condition will be true because we will choose $f(2n)$ to be one of the two numbers $2n$ and $2n+1$, and $f(2n+1)$ to be the other one. The first will be true because at least one of these two numbers is different from $f_n(2n)$, and that's the one we choose for $f(2n)$. (If both numbers are different from $f_n(2n)$, we can choose either one for $f(2n)$.)

8.48(a). If we define $f_0 : A_0 \to B$ by $f_0(x) = f(x)$, then f_0 is one-to-one because f is. For $x \in A_0$, $f_0(x) \in B_1$, because the number of ancestors of $f_0(x)$ is one more than the number of ancestors of x. Finally, for every $y \in B_1$, $y = f(x)$ for some $x \in A_0$. Therefore, f_0 is a bijection from A_0 to B_1. Similarly, g_0 is a bijection from B_0 to A_1. It is not difficult to see that the function f_∞ from A_∞ to B_∞ defined by $f_\infty(x) = f(x)$ is a bijection.

We define $F : A \to B$ by letting $F(x) = f(x)$ if $x \in A_0$ or $x \in A_\infty$ and $F(x) = g^{-1}(x)$ if $x \in A_1(x)$. It follows that F is a bijection from A to B.

8.48(b). Let $f : A \to B$ and $g : B \to C$ be the two bijections. Then $g \circ f$ is a bijection from A to a subset of C. If there were a bijection $h : A \to C$, then h^{-1} would be a bijection from C to A; this would mean that $h^{-1} \circ g$ is a bijection from B to a subset of A. We already have a bijection from A to a subset of B, and the Schröder-Bernstein theorem would imply that there was a bijection from A to B; but we are assuming this is not the case.

CHAPTER 9

9.10. One answer to (a) is $C = A \cup B$ and $D = A \cap B$. For the reduction in (b), given Turing machines T_1 and T_2, we can construct T_1' and T_2' accepting $L(T_1) \cup L(T_2)$ and $L(T_1) \cap L(T_2)$, respectively. Then $L(T_1) = L(T_2)$ if and only if $L(T_1') \subseteq L(T_2')$.

9.12(l). This problem is undecidable, because we can reduce *Accepts*-Λ to it. Given a TM T, an instance of *Accepts*-Λ, we construct T_1 as follows. T_1 has all of T's states as well as one additional state q; it has all of T's tape symbols and an additional symbol \$. The transitions of T_1 are the same as those of T, with these additions and modifications. For every accepting move $\delta(p, a) = (h_a, b, D)$ of T, T_1 has instead the move $\delta_{T_1}(p, a) = (q_0, \$, \mathrm{S})$, where q_0 is the initial state of T. If the nonhalting states of T are enumerated $q_0, q_1, \ldots, q_n$, then T_1 has the initial transitions $\delta_{T_1}(q_i, \$) = (q_{i+1}, \$, \mathrm{S})$ for $0 \leq i < n$, $\delta_{T_1}(q_n, \$) = (q, \$, \mathrm{S})$, and $\delta_{T_1}(q, \$) = (h_a, \$, \mathrm{S})$.

To summarize: For any move that would cause T to accept, T_1 instead moves to q_0 and places \$ on the tape; thereafter, the \$ causes T_1 to cycle through its nonhalting states, q being the last, before accepting. If T accepts Λ, then some move causes T to accept, and therefore, T_1 enters all its nonhalting states when started with a blank tape. On the other hand, if T doesn't accept Λ, then starting with a blank tape, T_1 will never enter the state q and does not enter all its nonhalting states.

9.14(a). Suppose $x, y \in \{0, 1\}^*$, and let T be an instance of $P_{\{x\}}$. Then we construct T_1 so that it begins by looking at its input; if the input is x, T_1 replaces it by y; if the input is y, T_1 replaces it by x; and for any input other than x or y, it makes no changes. From this point on, T_1 simply executes T. It follows that T accepts x and no other strings if and only if T_1 accepts y and no other strings, so that the construction is a reduction from $P_{\{x\}}$ to $P_{\{y\}}$.

9.14(b). If T is an instance of $P_{\{x\}}$, we might try constructing T_1 such that it replaced either input y or input z by x before executing T. We would then be able to say that T accepts x if and only if T_1 accepts both y and z. However, we could not be sure that T accepts no strings other than x if and only if T_1 accepts no strings other than y or z.

We describe how T_1 might be constructed in the case when x precedes y and y precedes z in canonical order. Other cases can be handled similarly. T_1 begins by looking at its input and making these preliminary steps: if the input is either y or z, T_1 replaces it by x; If the input is x, T_1 replaces it by y; and if the input is any string w that follows z in canonical order, T_1 replaces it by its immediate predecessor in canonical order. After these preliminary steps, T_1 executes T on the string that is now on its tape.

If T accepts x and nothing else, then T_1 accepts both y and z. T_1 does not accept x, because when T_1 receives input x it simulates T on

input y, and y is not one of the strings T accepts; T_1 also does not accept any string other than x, y, and z. On the other hand, if T_1 accepts y and z and no other strings, then T accepts x. It doesn't accept any other string, because for every $w \neq x$, some input other than y or z would cause T_1 to process w by running T on it.

Suppose T_1 accepts y and z and nothing else. Because T_1 processes both y and z by executing T on the string x, T must accept x. T does not accept y, because T_1 processes input x by executing T on the string y, and T_1 does not accept input x. T does not accept z, because T_1 processes the input string that is the successor of z in canonical order by executing T on z, and T_1 does not accept this input string. And finally, T does not accept any string other than x, y, or z, because for every such string, T_1 executes T on it, as a result of receiving an input string—maybe the same one, maybe a different one—that is also a string other than x, y, or z, and T_1 does not accept that input string.

It follows that this construction is a reduction of $P_{\{x\}}$ to $P_{\{y,z\}}$.

9.20. The proof of Theorem 9.9 shows that the problem *Accepts* can be reduced to the problem *AcceptsEverything*; showing the corresponding result for the languages of yes-instances is straightforward. Furthermore, a function that defines a reduction from *Acc* to *AE* also defines a reduction from *Acc′* to *AE′*.

It is easy to see that *Acc* is recursively enumerable. If *Acc′* were also recursively enumerable, then *Acc* would be recursive, which would mean that *Accepts* would be decidable. Therefore, *Acc′* is not recursively enumerable, and it follows from Exercise 9.3 that *AE′* is not either.

For part (d), let T_0 be a TM that immediately accepts every input. To complete the definition of f, we must say only what $f(x)$ is for a string x not of the form $e(T)e(z)$. Because such a string is an element of *Acc′*, we want $f(x)$ to be an element of *AE*, and we define $f(x)$ to be $e(T_0)$ in this case.

Now if $x = e(T)e(z)$, the string $f(x)$ is $e(S_{T,z})$. In the case where T accepts z, say in n moves, the computation performed by $S_{T,z}$ on an input string of length at least n causes it to enter an infinite loop, because by the time it finishes simulating n moves of T on input z it will discover that T accepts z. Therefore, in this case, the TM $S_{T,z}$ does not accept every string, and $f(x) \notin AE$. In the other case, where T does not accept z, $S_{T,z}$ does accept every input, because no matter how long its input is, simulating that number of moves of T on input z will not cause T to accept. It follows that f defines a reduction from *Acc′* to *AE*.

Finally, if *AE* were recursively enumerable, part (d) and Exercise 9.3 would imply that *Acc′* would be recursively enumerable, but we know it is not.

9.23. We will show that there are TMs T_1 and T_2 accepting the same language such that T_1 accepts the string $e(T_1)$ and T_2 does not accept the string $e(T_2)$.

We recall that for a TM T having k moves, the string $e(T)$ that encodes T is the concatenation of k shorter strings, each of which encodes one of the moves of T. We know that there is an algorithm for determining whether a string is of the form $e(T)$; and it is easy to tell for a string $e(T)$ how many moves are encoded in the string $e(T)$. Let L be the language of all strings $e(T)$ that describe a Turing machine T and have an even number of encoded moves. Then there is a TM T_1 that accepts L. We can construct another TM T_2 that accepts the same language and is identical to T_1 except that it has one additional state, no moves to that state, and one move from that state. Then one of the two TMs has an even number of moves and the other an odd number; therefore, one accepts its own encoding and the other doesn't accept its own.

9.29. Suppose (α_1, β_1), (α_2, β_2), . . . , (α_n, β_n) is an instance of *PCP* in which the α_i's and β_i's are all strings over the alphabet $\{a\}$, and let $d_i = |\alpha_i| - |\beta_i|$. If $d_i = 0$ for some i, the instance is a yes-instance. If $d_i > 0$ for every i, or if $d_i < 0$ for every i, then the instance is a no-instance. And finally, if $d_i = p > 0$ and $d_j = q < 0$ for some i and j, then it is a yes-instance, because $\alpha_i^q \alpha_j^p = \beta_i^q \beta_j^p$.

9.33. The problem is decidable, and the following is a decision algorithm. First test x for membership in $L(G)$. If $x \notin L(G)$, then $L(G) \neq \{x\}$. If $x \in L(G)$, then $L(G) = \{x\}$ if and only if $L(G) \cap (\Sigma^* - \{x\}) = \emptyset$. We can test this last condition as follows. Construct a PDA M accepting $L(G)$, using the techniques described in Section 5.3. Using the algorithm described in the proof of Theorem 6.13, construct a PDA M_1 accepting $L(M) \cap (\Sigma^* - \{x\})$, which is the intersection of $L(M)$ and a regular language. Construct a CFG G_1 generating the language $L(M_1)$, using the algorithms described in the proofs of Theorems 5.28 and 5.29. Finally, use the algorithm in Section 6.3 to determine whether $L(G_1) = \emptyset$.

CHAPTER 10

10.2. Suppose b is computable, and suppose T_b is a Turing machine that computes it. Without loss of generality we can assume that T_b has tape alphabet $\{0, 1\}$. Let $T = T_b T_1$, where T_1 is a TM, also having tape alphabet $\{0, 1\}$, that moves its tape head to the first square to the right of its starting position in which there is either a 0 or a blank, writes a 1 there, and halts. Let m be the number of states of T. By definition of b, no TM with m states and tape alphabet $\{0, 1\}$ can end up with more than $b(m)$ 1's on the tape if it halts on input 1^m. However, T is a TM of this type that halts with output $1^{b(m)+1}$. This is a contradiction.

10.6. For every n, we can construct a TM T_n that halts in configuration $h_a \Delta 1^n$ when it starts with a blank tape. T_n requires no tape symbols other than 1, and we can construct it so that for some number k independent of n, T_n has $k + n$ states. Now suppose f is a computable function, and let T_f

be a TM with m states and tape alphabet $\{0, 1\}$ that computes f. Let T'_n be the composite TM $T_n T_f$, so that starting with a blank tape, T'_n halts with output $1^{f(n)}$. The number of states of T'_n is $k' + n$, for some constant k', and it follows from the definition of bb that $f(n) \leq bb(k' + n)$.

Suppose for the sake of contradiction that bb is computable. Then so is the function c defined by $c(n) = bb(2n)$. Therefore, $bb(2n) \leq bb(k' + n)$ for every n. But this is impossible, because for sufficiently large n, there are TMs with $2n$ states and tape alphabet $\{0, 1\}$ which, when started with a blank tape, can write more 1's before halting than any TM having only $n + k'$ states and tape alphabet $\{0, 1\}$.

10.15(c). We let $f_0, \ldots, f_4$ be the primitive recursive derivation of *Add* described in Example 10.5. Let f_5 be the constant (or function of zero variables) 1. Let $f_6 = p_2^2$. Let f_7 be the function obtained from *Add* (i.e., f_4), f_6, and f_6 using composition. Then $g(n) = 2^n$ is obtained from f_5 and f_7 by primitive recursion, because of the formulas

$$g(0) = 1$$
$$g(k+1) = 2^k + 2^k = Add(p_2^2(k, g(k)), p_2^2(k, g(k))) = f_5(f_7(k, g(k)))$$

10.22. The result follows from the formula

$$HighestPrime(k) = max\{y \leq k \mid Exponent(y, k) > 0\}$$

together with the next exercise.

10.23(a). $m^P(X, 0) = 0$, and

$$m^P(X, k+1) = \begin{cases} k+1 & \text{if } P(X, k+1) \\ m^P(X, k) & \text{if } \neg P(X, k+1) \end{cases}$$

10.23(b).

$$\begin{aligned} m^P(X, k) &= min\ \{y \leq k \mid \text{ for every } z \text{ with } y < z \leq k, \neg P(X, z)\} \\ &= min\ \{y \leq k \mid \text{ for every } z \leq k, z \leq y \vee \neg P(X, z)\} \end{aligned}$$

10.26(b). The last digit in the decimal representation of a natural number n is $Mod(n, 10)$; it is therefore sufficient to show that the function g is primitive recursive, where $g(0) = 1$, $g(1) = 14$, $g(2) = 141$, and so forth. $g(i)$ is the largest integer less than or equal to the real number $10^i\sqrt{2}$. Equivalently, $g(i)$ is the largest integer whose square does not exceed $(10^i\sqrt{2})^2 = 2 * 10^{2i}$. This means that $g(i) + 1$ is the smallest integer whose square is greater than $2 * 10^{2i}$. Therefore, if we apply the minimization operator to the primitive recursive predicate P defined by $P(x, y) = (y^2 > 2 * 10^{2x})$, $g(x)$ is obtained from subtracting 1 from the result. The only remaining problem is to show how a *bounded* minimization can be used in this last step. We can do this by specifying a value of y for which y^2 is certain to be greater than $2 * 10^{2x}$. $y = 2 * 10^x$ is such a value, and so

$$f(x) = m_P(x, 2 * 10^x) \dot{-} 1$$

10.32. If there were a solution to this problem, then there would be a solution to the problem: Given a TM computing some partial function, does it accept every input? We know that the (apparently) more general problem: Given an *arbitrary* TM, does it accept every input? is undecidable. It follows from this, however, that the more restricted problem is undecidable, because for every TM T there is another TM T' that computes a partial function and accepts exactly the same strings as T. (T' can be constructed as follows: First modify T such that instead of writing Δ it writes a different symbol, say Δ', and it makes the same moves on Δ' that it makes on Δ; then let $T' = TT_1$, where T_1 erases the tape and halts with the tape head on square 0.) T' accepts input x if and only if T does, and T' computes the partial function whose only value is Λ. Therefore, the given problem is undecidable.

10.33. If we let $g(x, y) = |f(y) - x|$, then $f^{-1}(x) = M_g(x) = \mu y[g(x, y) = 0]$.

CHAPTER 11

11.6. One approach is to have the TM copy the input onto a second tape and return the tape heads to the beginning; then to have the first tape head move two squares to the right for every square moved by the second (and to reject if the input length is odd), so that the second tape head is now at the beginning of the second half of the string; to move the first tape head back to the beginning; and then to compare the first half of tape 1 to the second half of tape 2, one symbol at a time.

11.7. Suppose T is a k-tape TM with time complexity $f(n)$. We present an argument that is correct provided $f(n) \geq n$; this will be sufficient as long as T actually reads its input string.

We also simplify the argument slightly by assuming that $k = 2$, though it is easy to adapt it to the general case. If T_1 is the one-tape TM that simulates T in the way described in Chapter 7, the first portion of T_1's operation is to create the "two-track" configuration in which the input string is on the first track. This can be done by starting at the left end of the tape, inserting Δ between each pair of consecutive input symbols, and then inserting the # marker at the end. Inserting the first Δ and returning the tape head takes approximately $2n$ moves, and each subsequent insertion takes less time than the previous one. The time to prepare the tape is therefore $O(n^2)$, which by our assumption on f is $O(f(n)^2)$.

At any subsequent stage of the simulation of T, the total number of tape squares to the left of the # marker is no more than $2f(n)$, because in $f(n)$ moves T cannot move past square $f(n)$ on any of its tapes.

Simulating a single move of T requires three iterations of the following: move the tape head to the current position on one of the tracks, make no more than some fixed number of moves, and move the tape head back to the

beginning. Each of these three iterations can be done in time proportional to the number of squares to the left of the #, which means time proportional to $f(n)$. Therefore, the total simulation, in which there are no more than $f(n)$ moves, can be accomplished in time $O(f(n)^2)$. Finally, eliminating the second track of the tape and preparing the final output can also be done in time $O(f(n)^2)$, because the length of the output string is no more than $f(n)$.

11.19. The language L is in NP; a nondeterministic procedure to accept L is to take the input string, and provided it is of the form $e(T)e(x)1^n$, choose a sequence of n moves of T on input x, accepting the input if the sequence causes T to accept x.

To show that L is NP-hard, let L_1 be a language in NP, and let T be an NTM accepting L_1 with nondeterministic time complexity bounded by a polynomial p. Consider the function $f : \Sigma^* \to \{0, 1\}^*$ defined by $f(x) = e(T)e(x)1^{p(|x|)}$; for every string x, $x \in L_1$ if and only if $f(x) \in L$, and it is easy to check that f is polynomial-time computable.

11.20. Consider the following procedure for coloring the vertices in each "connected component" of the graph. Choose one and color it white. Color all the vertices adjacent to it black. For each of those, color all the vertices white that are adjacent to that one and have not yet been colored. Continue this process until there are no colored vertices with uncolored adjacent vertices, and repeat this procedure for each component. This can be carried out in polynomial time, and it can be determined in polynomial time whether the resulting assignment of colors is in fact a 2-coloring of the graph. Furthermore, the graph can be 2-colored if and only if this procedure can be carried out so as to produce a 2-coloring.

11.24. If $x^{n-1} \equiv_n 1$ and n is not prime, then according to the result of Fermat mentioned in Example 11.9, $x^m \equiv_n 1$ for some $m < n - 1$. The smallest such m must be a divisor of $n - 1$. The reason is that when we divide $n - 1$ by m, we get a quotient q and a remainder r, so that

$$n - 1 = q * m + r \text{ and } 0 \leq r < m$$

This means that $x^{n-1} = x^{qm+r} = (x^m)^q * x^r$, and because x^{n-1} and x^m are both congruent to 1 mod n, we must have $(x^m)^q \equiv_n 1$ and therefore $x^r \equiv_n 1$. It follows that r must be 0, because $r < m$ and by definition m is the smallest *positive* integer with $x^m \equiv_n 1$. Therefore, $n - 1$ is divisible by m.

Every proper divisor m of $n - 1$ is of the form $(n - 1)/j$ for some $j > 1$ that is a product of (one or more) prime factors of $n - 1$. Therefore, some multiple of m, say $a * m$, is $(n - 1)/p$ for a single prime p. Because $x^{a*m} = (x^m)^a \equiv_n 1$, we may conclude that $x^{(n-1)/p} \equiv_n 1$.

11.27. The least obvious of the three statements is that P is closed under the Kleene * operation. Suppose $L \in P$. We describe in general terms an algorithm for accepting L^*, and it is not hard to see that it can be executed in polynomial time on a TM. (There are more efficient algorithms, whose runtimes would be lower-degree polynomials.)

For a string $x = a_1a_2 \ldots a_n$ of length n, and integers i and j with $1 \leq i \leq j \leq n+1$, denote by $x[i, j]$ the substring $a_i a_{i+1} \ldots a_{j-1}$ of length $j - i$. There are $(n+1)(n+2)/2$ ways of choosing i and j, although the substrings $x[i, j]$ are not all distinct. In the algorithm, we keep a table in which every $x[i, j]$ is marked with 1 if it has been found to be in L^* and 0 otherwise. We initialize the table by marking $x[i, j]$ with 1 if it is in L and 0 if not. Because $L \in P$, this initialization can be done in polynomial time. Now we begin a sequence of iterations, in each of which we examine every pair $x[i, j]$, $x[j, k]$ with $0 \leq i \leq j \leq k \leq n+1$; if $x[i, j]$ and $x[j, k]$ are both marked with 1, then their concatenation $x[i, k]$ is marked with 1. This continues until either we have performed n iterations or we have executed an iteration in which the table did not change. The string x is in L^* if and only if $x = x[1, n+1]$ is marked with 1 at the termination of the algorithm.

11.31. Let $G = (V, E)$ be a graph and k an integer. We construct an instance $(S_1, S_2, \ldots, S_m)$ of the exact cover problem in terms of G and k.

The elements belonging to the subsets $S_1, \ldots, S_m$ will be of two types: elements of V and pairs of the form (e, i) with $e \in E$ and $1 \leq i \leq k$. Specifically, for each $v \in V$ and each i with $1 \leq i \leq k$, let

$$S_{v,i} = \{v\} \cup \{(e, i) \mid v \text{ is an end point of } e\}$$

In addition, for each pair (e, i), let $T_{e,i} = \{(e, i)\}$. Then the union of all the sets $S_{v,i}$ and $T_{e,i}$ contains all the vertices of the graph, as well as all the possible pairs (e, i).

Suppose there is a k-coloring of G in which the colors are $1, 2, \ldots, k$, and that each vertex v is colored with the color i_v. Then we can construct an exact cover for our collection of sets as follows. We include every set S_{v,i_v}. The union of these contains all the vertices. We also include all the $T_{e,i}$'s for which the pair (e, i) has not already been included in some S_{v,i_v}. These sets form a cover—that is, their union contains all the elements in the original union. To see that it is an exact cover, it is sufficient to check that no two sets S_{v,i_v} and S_{w,i_w} can intersect if $v \neq w$. This is true if v and w are not adjacent; if they are, it follows from the fact that the colors i_v and i_w are different.

Suppose on the other hand that there is an exact cover for the collection of $S_{v,i}$'s and $T_{e,i}$'s. Then for every vertex v, v can appear in only one set in the exact cover, and so there can be only one i, say i_v, for which $S_{v,i}$ is included. Now we can check that if we color each vertex with the color i_v, we get a k-coloring of G. If not, then some edge e joins two vertices v and w with $i_v = i_w = i$. This means, however, that both S_{v,i_v} and S_{w,i_w} contain the pair (e, i), and this is impossible if the cover is exact.

This page intentionally left blank

SELECTED BIBLIOGRAPHY

Agrawal M, Kayal N, Saxena N: PRIMES Is in P, *Annals of Mathematics* 160(2): 781–793, 2004.

Bar-Hillel Y, Perles M, Shamir E: On Formal Properties of Simple Phrase Structure Grammars, *Zeitschrift für Phonetik Sprachwissenschaft und Kommunikations-forschung* 14: 143–172, 1961.

Bovet DP, Crescenzi P: *Introduction to the Theory of Complexity*. Englewood Cliffs, NJ: Prentice Hall, 1994.

Brown D, Levine JR, Mason T: *lex & yacc*, 2nd ed. Sebastopol, CA: O'Reilly Media, Inc., 1995.

Cantor G: Contributions to the Founding of the Theory of Transfinite Numbers. Mineola, NY: Dover, 1955.

Chomsky N: Three Models for the Description of Language, *IRE Transactions on Information Theory* 2: 113–124, 1956.

Chomsky N: On Certain Formal Properties of Grammars, *Information and Control* 2: 137–167, 1959.

Chomsky N: *Context-Free Grammars and Pushdown Storage*, Quarterly Progress Report No. 65, Cambridge, MA: Massachusetts Institute of Technology Research Laboratory of Electronics, 1962, pp. 187–194.

Church A: An Unsolvable Problem of Elementary Number Theory, *American Journal of Mathematics* 58: 345–363, 1936.

Cobham A: The Intrinsic Computational Difficulty of Functions, *Proceedings of the 1964 Congress for Logic, Mathematics, and Philosophy of Science*, New York: North Holland, 1964, pp. 24–30.

Cook SA: The Complexity of Theorem Proving Procedures, *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, New York: Association for Computing Machinery, 1971, pp. 151–158.

Davis MD: *Computability and Unsolvability*. New York: McGraw-Hill, 1958.

Davis, MD: *The Undecidable*. Hewlett, NY: Raven Press, 1965.

Davis, MD, Sigal R, Weyuker EJ: *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*, 2nd ed. New York: Academic Press, 1994.

Dowling WF: There Are No Safe Virus Tests, *American Mathematical Monthly* 96: 835–836, 1989.

Earley J: An Efficient Context-Free Parsing Algorithm, *Communications of the ACM* 13(2): 94–102, 1970.

Floyd RW, Beigel R: *The Language of Machines: An Introduction to Computability and Formal Languages*. New York: Freeman, 1994.

Fortnow L: The Status of the P versus NP Problem, *Communications of the ACM* 52(9): 78–86, 2009.

Garey MR, Johnson DS: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman, 1979.

Hartmanis J, Stearns RE: On the Computational Complexity of Algorithms, *Transactions of the American Mathematical Society* 117: 285–306, 1965.

Hopcroft JE., Motwani R, Ullman J: *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Reading, MA: Addison-Wesley, 1979.

Immerman N: Nondeterministic Space Is Closed Under Complementation, *SIAM Journal of Computing* 17: 935–938, 1988.

Karp RM: Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*. New York: Plenum Press, 1972, pp. 85–104.

Kleene SC: *Introduction to Metamathematics*. New York: Van Nostrand, 1952.

Kleene SC: Representation of Events in Nerve Nets and Finite Automata. In Shannon CE, McCarthy J

(eds.), *Automata Studies*. Princeton, NJ: Princeton University Press, 1956, pp. 3–42.

Knuth, DE: On the Translation of Languages from Left to Right, *Information and Control* 8: 607–639, 1965.

Levin L: Universal Search Problems (in Russian), *Problemy Peredachi Informatsii* 9(3): 265–266, 1973.

Lewis HR., Papadimitriou C: *Elements of the Theory of Computation*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1998.

Lewis PM II, Stearns RE: Syntax-Directed Transduction, *Journal of the ACM* 15: 465–488, 1968.

McCulloch WS, Pitts W: A Logical Calculus of the Ideas Immanent in Nervous Activity, *Bulletin of Mathematical Biophysics* 5: 115–133, 1943.

Moore EF: Gedanken Experiments on Sequential Machines. In Shannon CE, and McCarthy J (eds.), *Automata Studies*. Princeton, NJ: Princeton University Press, 1956, pp. 129–153.

Myhill J: *Finite Automata and the Representation of Events*, WADD TR-57-624, Wright Patterson Air Force Base, OH, 1957, pp. 112–137.

Nerode A: Linear Automaton Transformations, *Proceedings of the American Mathematical Society* 9: 541–544, 1958.

Oettinger AG: Automatic Syntactic Analysis and the Pushdown Store, *Proceedings of the Symposia in Applied Mathematics* 12, Providence, RI: American Mathematical Society 9, 1961 pp. 104–109.

Ogden O: A Helpful Result for Proving Inherent Ambiguity, *Mathematical Systems Theory* 2: 191–194, 1968.

Papadimitriou CH: *Computational Complexity*. Reading, MA: Addison-Wesley, 1994.

Post EL: A Variant of a Recursively Unsolvable Problem, *Bulletin of the American Mathematical Society* 52: 246–268, 1946.

Rabin MO: Real-Time Computation, *Israel Journal of Mathematics* 1: 203–211, 1963.

Rabin MO, Scott D: Finite Automata and Their Decision Problems, *IBM Journal of Research and Development* 3: 115–125, 1959.

Rosenkrantz DJ, Stearns RE: Properties of Deterministic Top-Down Grammars, *Information and Control* 17: 226–256, 1970.

Salomaa A: *Jewels of Formal Language Theory*. Rockville, MD: Computer Science Press, 1981.

Schützenberger MP: On Context-Free Languages and Pushdown Automata, *Information and Control* 6: 246–264, 1963.

Sipser M: *Introduction to the Theory of Computation*, 2nd ed. Florence, KY: Course Technology, 2005.

Szelepcsény R: The Method of Forcing for Nondeterministic Automata, *Bulletin of the EATCS* 33: 96–100, 1987.

Turing AM: On Computable Numbers with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* 2: 230–265, 1936.

van Leeuwen J (ed.): *Handbook of Theoretical Computer Science (Volume A, Algorithms and Complexity)*. Amsterdam: MIT Press/Elsevier, 1990.

Younger DH: Recognition and Parsing of Context-Free Languages in Time n^3, *Information and Control* 10(2): 189–208, 1967.

INDEX OF NOTATION

INDEX

D

E

N

O

P

T

U