formal language

A PRACTICAL INTRODUCTION

Adam Brooks Webber

Monmouth College

Franklin, Beedle & Associates, Inc.
8536 SW St. Helens Drive, Ste. D
Wilsonville, Oregon 97070
503/682-7668
www.fbeedle.com

Publisher	Jim Leisy (jimleisy@fbeedle.com)
Production Editor	Tom Sumner
Editor	Stephanie Welch
Order Processing	Jaron Ayres

Printed in the U.S.A.

Rights and Permissions
Franklin, Beedle & Associates, Incorporated
8536 SW St. Helens Drive, Ste. D
Wilsonville, Oregon 97070

Library of Congress Cataloging-in Publication Data

Webber, Adam Brooks.

Formal language : a practical introduction / Adam Brooks Webber.

p. cm.

ISBN-13: 978-1-59028-197-0

1. Formal languages--Textbooks. 2. Computational complexity--Textbooks. I. Title.

QA267.3.W423 2007

005.13'1--dc22

2007027145

CONTENTS

PREFACE

A Word to the Instructor

This book is an undergraduate college-level introduction to formal languages, automata, computability, and complexity. I emphasize *undergraduate*. There are other textbooks on this subject, including some that I admire very much, but most assume a level of mathematical maturity and interest that only a few of my undergraduate students actually have. Most of these students are bright people whose primary interest in computing is practical. They like programming and have some aptitude for it. They hope to find jobs in the computing industry. They think (sometimes incorrectly) that they are not going to do graduate work on the theoretical side of computer science. It is for students like these that I wrote this book.

In general, I have found that these students do not thrive on a diet of unmotivated abstractions. This book tries to lead them gently into the beautiful abstractions at the foundations of computer science, using concrete examples and exercises. The chapters on finite automata include applications and implementation techniques; the chapters on stack-based automata include a discussion of applications to compiler construction; the chapters on computability use simple Java-like code examples. Included among the exercises are many Java programming problems, which serve both to hold student interest and to enhance understanding.

This book does not emphasize proof technique. There's no doubt that the theory of formal languages and automata is an ideal place to introduce induction and other proof techniques for students who need to learn them. The book includes one chapter and a number of exercises covering inductive proofs, and additional chapters devoted to pumping-lemma proofs. However, the material is not central to the book, and the instructor can use as much or as little of it as desired. The book includes proofs for almost all the theorems, but usually stops short of presenting detailed inductions, and most of the exercises do not require students to generate novel proofs. My feeling is that too much stress on proof technique turns many students off. The concepts of formal languages and automata are among the most beautiful and enduring in all of computer science, and have many practical applications as well. The book tries to make these treasures accessible to all students, not just to those with a knack for the mathematically rigorous proof.

This book deviates from the usual syllabus for its subject in one other important respect: it does not provide a detailed treatment of PDAs and DPDAs. The reason for this is, again, my concern for maintaining my students' interest and enthusiasm. The general operation of PDAs is simple enough, but there are all those special cases to worry about: the end of the input string, the bottom of the stack, the ε-transitions, the definition of acceptance. These are minor technicalities, to be sure, but there are enough of them that I find my students become discouraged—and if they look to other books for guidance that only makes it worse, because no two agree on these details. The formalism generates too much heat and not enough light. In place of PDAs this book investigates *stack machines*, a simpler computational formalism that captures the CFLs, connects neatly with CFGs, and provides a good foundation for a discussion of parsing techniques.

With few other exceptions, the ideas, theorems, and proofs in this book are standard. They are presented without the detailed historical attributions that may be found in graduate-level texts. However, many of the chapters conclude with a Further Reading section, giving references into the literature for interested students.

I designed the book to be accessible to sophomores in a computer-science major—perhaps even to second-semester freshmen. The presentation does not rely on any college-level mathematical prerequisites. The only computer-science prerequisite is the ability to read simple code examples: a Java-like syntax is used, but the ability to read any C-family language will suffice.

In some academic departments, the material on complexity—Chapters 19, 20, and 21, and Appendices B and C—will be positioned in a separate course, together with related material on algorithms and complexity analysis. In my experience there is more than enough even in Chapters 1 through 18 to fill a semester-long course, so those wishing to cover the entire subject of the book may want to consider skipping or abbreviating, particularly Chapters 9, 11, 14, and 18.

In many CS and CE departments (such as at the University of Wisconsin—Milwaukee, where I began the development of this book) a course in this subject is a required part of the major. At other institutions (including Monmouth College, where I completed it) the course has gradually fallen into disuse; it has sometimes become an elective, rarely if ever offered. The reason for this decline, I believe, is the mismatch between the students and the treatment of the subject. Instructors do not enjoy the drudgery of having to frog-march undergraduates through a difficult textbook that they barely understand and quickly forget; students enjoy the experience even less. I hope that *Formal Language: A Practical Introduction* will help to rekindle the excitement of teaching and studying this subject.

Introduction

It sometimes seems that everything is a science these days: political science and beauty science, food science and waste science, creation science and mortuary science. Just for fun, try searching the Web for any well-formed phrase ending with the word *science*. Weed science, reincarnation science, sales science—it’s hard to find one that is *not* in use.

Because of this oddly mixed company, the name *computer science* triggers some skepticism. That’s unfortunate, because it is actually an excellent field of study for many undergraduates. Looking for a liberal-arts education? Computer science has deep connections throughout the web of human knowledge and is highly relevant to broad, ongoing changes in our culture. Looking for a rigorous scientific education? There are research-oriented departments of computer science at all the world's great universities, engaged in a dramatic scientific adventure as our young science begins to reveal its secrets. Looking for a vocational education? Computer science is a solid foundation for a variety of career paths in the computer industry.

The subject of this book, formal language, is at the heart of computer science, and it exemplifies the virtues just cited:

- Formal language is connected to many other branches of knowledge. It is where computer science, mathematics, linguistics, and philosophy meet. Understanding formal language helps you see this interconnectedness, so that you will never make the mistake of thinking of computer science as a separate intellectual kingdom.
- Formal language is a rigorous branch of mathematics, with many open questions at the frontiers. This book covers only the basics, but if you find the basics interesting there is much more to study. Some advanced topics are identified here, with pointers to further reading; others may be found in any graduate-level text.
- Formal language is very useful. This book stresses applications wherever they arise, and they arise frequently. Techniques that derive from the study of formal language are used in many different practical computer systems, especially in programming languages and compilers.

In addition, formal language has two special virtues, not shared with most of the other branches of computer science:

- Formal language is accessible. This book treats formal language in a way that does not assume the reader knows any advanced mathematics.
- Finally, formal language is stable—at least in the elementary parts. Almost everything in this book has been known for 30 years or more. That's a very long time in computer science. The computer you bought as a freshman may be obsolete by the time you're a senior, and the cutting-edge programming language you learned may be past its prime, but the things you learn about formal language will not lose their relevance.

No one who loves language can take much pleasure in the prospect of studying a subject called *formal language*. It sounds suspiciously abstract and reductionistic. It sounds as if all the transcendent beauty of language will be burned away, fired under a dry heat of definitions and theorems and proofs, until nothing is left but an ash of syntax. It sounds abstract—and it is, undeniably. Yet from this abstraction arise some of the most beautiful and enduring ideas in all of computer science.

This book has two major goals. The first is to help you understand and appreciate the beautiful and enduring ideas of formal language. These ideas are the birthright of all computer scientists, and they will profoundly change the way you think about computation. They are not only among the most beautiful, but also among the most useful tools in computer science. They are used to solve problems in a wide variety of practical applications, and they are especially useful for defining programming languages and for building language systems. The second purpose of this book is to help you develop a facility with these useful tools. Our code examples are in Java, but they are not particularly Java-centric and should be accessible to any programmer.

There is also a third major reason to study formal language, one that is *not* a primary focus of this book: to learn the techniques of mathematical proof. When you are learning about formal language, it can also be a good time to learn proof techniques, because the subject is full of theorems to practice on. But this book tries to make the beautiful and useful ideas for formal language accessible to students at all levels of mathematical interest and ability. To that end, although the book presents and discusses many simple proofs, it does not try to teach advanced proof techniques. Relatively few of the exercises pose challenging proof problems. Those planning graduate-level study of theoretical computer science would be well advised not to rely exclusively on this book for that kind of training.

Acknowledgments

Today the territory mapped in this book is part of the intellectual commons of our discipline, but it was not always so. I am indebted to the many researchers, living and dead, who first explored this territory. I am honored to be able to share their work with others. I am particularly indebted to those scholars who first shared it me, for their style, grace, and elegant precision: Scot Drysdale at Dartmouth College, and Dexter Kozen and Juris Hartmanis at Cornell University.

Thanks to Jim Leisy, my publisher at Franklin, Beedle & Associates. His feedback and support, both for this project and for my previous book, have been extremely valuable. Thanks also to Stephanie Welch and Tom Sumner, my editors, for their great patience and diligence with this project. Thanks to the anonymous reviewers who commented on early drafts of the book. The remaining defects are, of course, entirely my own.

I have been encouraged and supported in this work by two excellent department chairmen: K. Vairavan at the University of Wisconsin—Milwaukee, and Lyle Welch at Monmouth College. Monmouth has been a particularly supportive environment for a project like this, with its emphasis on the undergraduate experience. I'm proud to have been able to teach there, and I thank the Monmouth students who endured my well-meaning experimentation with early drafts of this book.

Finally, thanks to my family: my wife Kelly, my children Fern and Fox, my parents Howard and Helen Webber, and my wife's parents, Harold and Margaret Autrey. Their loving support has been unfailing. For their sake I hope that the next book I write will be more fun for them to read.

Web Site

The web site for this book is at http://www.webber-labs.com/fl.html. Materials there include a full set of slides for each chapter, and all the larger code examples from the text and exercises. There are also instructions for contacting the author to report defects, and for accessing additional instructors-only materials.

1
Fundamentals

Algebraists use the words **group**, **ring**, *and* **field** *in technical ways, while entomologists have precise definitions for common words like* **bug** *and* **fly**. *Although it can be slightly confusing to overload ordinary words like this, it's usually better than the alternative, which is to invent new words. So most specialized fields of study make the same choice, adding crisp, rigorous definitions for words whose common meaning is fuzzy and intuitive.*

The study of formal language is no exception. We use crisp, rigorous definitions for basic terms such as **alphabet**, **string**, *and* **language**.

1.1 Alphabets

Formal language begins with sets of symbols called alphabets:

An *alphabet* is any finite set of symbols.

A typical alphabet is the set Σ = {*a*, *b*}. It is the set of two symbols, *a* and *b*. There is no semantics; we do not ascribe any meaning to *a* or to *b*. They are just symbols, which could as well be 0 and 1, or 空 and 手.

Different applications of formal languages use different alphabets. If you wanted to work with decimal numbers, you might use the alphabet {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. If you wanted to work with the contents of computer memories, you might use the alphabet {0, 1}. If you wanted to work with text files on a computer, you might use one of the standard machine-text alphabets, like ASCII or Unicode.

It is a convention in the study of formal languages, which is followed in this book, to use the first few Latin letters, like *a* and *b*, in most example alphabets. This helps you resist the urge to ascribe meaning to the symbols. When you see a string like 1000, you naturally think of it as representing the decimal number one thousand, while the string *abbb* does not tempt you to make any irrelevant interpretations. Because the symbols in the alphabet are uninterpreted, our results apply to all alphabets.

Another convention followed in the book is to use the symbol Σ to stand for the alphabet currently in use. When it is necessary to manipulate more than one alphabet at once, subscripts are used: Σ_1, Σ_2, and so on.

The empty set {} is a legal alphabet, but not usually an interesting one. (Some people use the special symbol ∅ for the empty set, but this book just uses {}.)

1.2 Strings

The symbols from an alphabet are put together into strings:

A *string* is a finite sequence of zero or more symbols.

For example, *abbb* and 010 are strings. In formal languages, unlike most programming languages, strings are not written with quotation marks around them.

The length of a string is the number of symbols in it. To refer to the length of a string, bracket the string with vertical lines: |*abbb*| = 4.

When the symbols in a string are part of a particular alphabet Σ, we say that the string is "a string over Σ." In this usage, the word "over" means "built using symbols from." So, for example, the set of all strings of length 2 over the alphabet {*a*, *b*} is the set of all strings of length 2 that can be built using the symbols *a* and *b*: {*aa*, *bb*, *ab*, *ba*}.

The special symbol ε is used to represent the string of length zero: the string of no symbols. Note here that ε is not a symbol in any alphabet we will use; it simply stands for the empty string, much as you would write `""` in some programming languages. For example, the set of all strings of length 2 or less over the alphabet $\{a, b\}$ is $\{\varepsilon, a, b, aa, bb, ab, ba\}$. The length of ε is zero, of course: $|\varepsilon| = 0$. Be careful not to confuse the empty set, {}, with the empty string, ε, and note also that $\{\} \neq \{\varepsilon\}$; the set {} contains nothing, while the set {ε} contains one thing: the empty string.

When describing languages and proving things about them, it is sometimes necessary to use variables that stand for strings, such as $x = abbb$. This is a natural concept in programming languages; in Java one writes `String x = "abbb"`, and it is clear from the syntax that `x` is the name of a variable and `abbb` are the characters making up the string to which `x` refers. In the notation used for describing formal languages there is not so much syntax, so you have to rely more on context and on naming conventions. The convention followed in the book is to use the last few Latin letters, like *x*, *y*, and *z*, as string variables, not as symbols in alphabets.

For example, the following definition of concatenation uses string variables.

> The *concatenation* of two strings *x* and *y* is the string containing all the symbols of *x* in order, followed by all the symbols of *y* in order.

To refer to the concatenation of two strings, just write them right next to each other. For example, if $x = abc$ and $y = def$ then the concatenation of *x* and *y* is $xy = abcdef$. For any string *x*, we have $x\varepsilon = \varepsilon x = x$. So ε is the identity element for concatenation of strings, just as 0 is the identity element for addition of natural numbers and just as 1 is for multiplication of natural numbers.

Speaking of numbers, we'll denote the set of natural numbers, {0, 1, …}, as $\mathcal{N}$. Any natural number $n \in \mathcal{N}$ can be used like an exponent on a string, denoting the concatenation of that string with itself, *n* times. Thus for any string *x*,

$$x^0 = \varepsilon \text{ (that is, zero copies of } x\text{)}$$
$$x^1 = x$$
$$x^2 = xx$$
$$x^3 = xxx$$

and, in general,

$$\mathrm{x}^n = \underbrace{xx \cdots x}_{n \text{ times}}$$

When the alphabet does not contain the symbols (and), which is almost all the time, you can use parentheses to group symbols together for exponentiation. For example, $(ab)^7$ denotes the string containing seven concatenated copies of *ab*: $(ab)^7 = ababababababab$.

1.3 Languages

A *language* is a set of strings over some fixed alphabet.

A special notation is used to refer to the set of all strings over a given alphabet.

The *Kleene closure* of an alphabet Σ, written Σ^*, is the set of all strings over Σ.

For example, $\{a\}^*$ is the set of all strings of zero or more as: $\{\varepsilon, a, aa, aaa, ...\}$, and $\{a, b\}^*$ is the set of all strings of zero or more symbols, each of which is either a or b: $\{\varepsilon, a, b, aa, bb, ab, ba, aaa, ...\}$. This allows a more compact way of saying that x is a string over the alphabet Σ; we can just write $x \in \Sigma^*$. (Here the symbol $\in$ is used as in standard mathematical notation for sets and means "is an element of.")

Except for the special case of $\Sigma = \{\}$, the Kleene closure of any alphabet is an infinite set. Alphabets are finite sets of symbols, and strings are finite sequences of symbols, but a language is any set of strings—and most interesting languages are in fact infinite sets of strings.

Languages are often described using *set formers*. A set former is written like a set, but it uses the symbol |, read as "such that," to add extra constraints or conditions limiting the elements of the set. For example, $\{x \in \{a, b\}^* \mid |x| \leq 2\}$ is a set former that specifies the set of all strings x over the alphabet $\{a, b\}$, such that the length of x is less than or equal to 2. Thus,

$$\{x \in \{a, b\}^* \mid |x| \leq 2\} = \{\varepsilon, a, b, aa, bb, ab, ba\}$$

$$\{xy \mid x \in \{a, aa\} \text{ and } y \in \{b, bb\}\} = \{ab, abb, aab, aabb\}$$

$$\{x \in \{a, b\}^* \mid x \text{ contains one } a \text{ and two } b\text{s}\} = \{abb, bab, bba\}$$

$$\{a^n b^n \mid n \geq 1\} = \{ab, aabb, aaabbb, aaaabbbb, ...\}$$

That last example shows why set former notation is so useful: it allows you to describe infinite languages without trying to list the strings in them. Unless otherwise constrained, exponents in a set former are assumed to range over all of $\mathcal{N}$. For example,

$$\{(ab)^n\} = \{\varepsilon, ab, abab, ababab, abababab, ...\}$$

$$\{a^n b^n\} = \{\varepsilon, ab, aabb, aaabbb, aaaabbbb, ...\}$$

The set former is a very expressive tool for defining languages, but it lacks precision. In one example above we used an English-language constraint: "x contains one a and two bs." That's allowed—but it makes it easy to write set formers that are vague, ambiguous, or self-contradictory. For that matter, even if we stick to mathematical constraints like "$|x| \leq 2$" and "$n \geq 1$," we still have the problem of explaining exactly which mathematical constraints are permitted and exactly what they mean. We won't try to do that—we'll continue to use set formers throughout this book, but we'll use them in an informal way, without any formal definition of what they mean. Our quest in subsequent chapters will be to find other tools for defining languages, formal tools that are precise and unambiguous.

Exercises

EXERCISE 1

Restate each of the following languages by listing its contents. For example, if the language is shown as $\{x \in \{a, b\}^* \mid |x| \leq 2\}$, your answer should be $\{\varepsilon, a, b, aa, bb, ab, ba\}$.

a. $\{x \in \{a, b, c\}^* \mid |x| \leq 2\}$
b. $\{xy \mid x \in \{a, aa\}$ and $y \in \{aa, aaa\}\}$
c. $\{\}^*$
d. $\{a^n \mid n$ is less than 20 and divisible by 3$\}$
e. $\{a^n b^m \mid n < 2$ and $m < 3\}$

EXERCISE 2

List all strings of length 3 or less in each of the following languages:

a. $\{a\}^*$
b. $\{a, b\}^*$
c. $\{a^n b^n\}$
d. $\{xy \mid x \in \{a\}^*$ and $y \in \{b\}^*\}$
e. $\{a^n b^m \mid n > m\}$

EXERCISE 3

Many applications of formal language theory do associate meanings with the strings in a language. Restate each of the following languages by listing its contents:

a. $\{x \in \{0, 1\}^* \mid x$ is a binary representation, without unnecessary leading zeros, of a number less than 10$\}$
b. $\{x \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \mid x$ is a decimal representation, without unnecessary leading zeros, of a prime number less than 20$\}$
c. $\{x \in \{\mathrm{a, b, \ldots, z}\}^* \mid x$ is a two-letter word in English$\}$

EXERCISE 4

Restate each of the following languages using set former notation.

a. the language of all strings over the alphabet $\{a, b\}$ that begin with *a*
b. the language of all even-length strings over the alphabet $\{a, b, c\}$
c. the language of strings consisting of zero or more copies of the string *ba*
d. the language of strings consisting of any number of *a*s followed by the same number of *b*s, followed by the same number of *c*s

EXERCISE 5

This exercise illustrates one of the perils of set formers: their use can lead to logical contradictions. This example is related to Russell's Paradox, a famous contradiction in naive set theory discovered by Bertrand Russell in 1901, which led to an important body of work in mathematics.

A set former is itself a string, so one can define languages that contain set formers. For example, the set of all set formers that define the empty set would include such strings as "{}", "$\{x \mid |x| < 0\}$", and "$\{x \mid x$ is not equal to $x\}$". (We put quotation marks around set formers considered as strings.)

a. Give an example of a string x that is a set former that defines a language that includes x itself.
b. Give an example of a string y that is a set former that defines a language that does not include y itself.
c. Consider r = "{set formers $x \mid$ the language defined by x does not include x}". (Your set former y from Part b, above, is an example of a string in the language defined by r.) Show that assuming r is in the language defined by r leads to a contradiction. Show that assuming r is *not* in the language defined by r also leads to a contradiction.

2

Finite Automata

One way to define a language is to construct an automaton—a kind of abstract computer that takes a string as input and produces a yes-or-no answer. The language it defines is the set of all strings for which it says yes.

The simplest kind of automaton is the finite automaton. The more complicated automata we discuss in later chapters have some kind of unbounded memory to work with; in effect, they will be able to grow to whatever size necessary to handle the input string they are given. But in this chapter, we begin with finite automata, and they have no such power. A finite automaton has a finite memory that is fixed in advance. Whether the input string is long or short, complex or simple, the finite automaton must reach its decision using the same fixed and finite memory.

2.1 Man, Wolf, Goat, and Cabbage

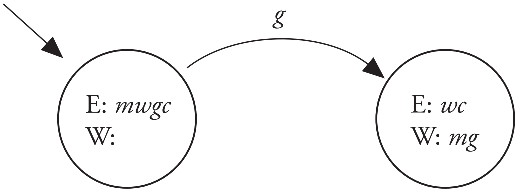
A man is traveling with a wolf, a goat, and a cabbage. He comes to the east bank of a river and wants to cross. A tiny rowboat is available, but it is so small that the man can take at most one of his possessions with him in the boat. If he leaves the goat and the cabbage alone together, the goat will eat the cabbage. If he leaves the wolf and the goat alone together, the wolf will eat the goat. How can the man get to the west bank without losing any of his possessions?

In this old riddle, there are really only four moves to choose from at each step:

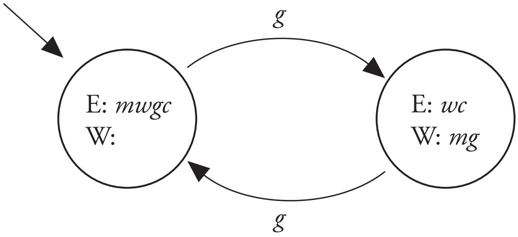
- Man crosses with wolf.
- Man crosses with goat.
- Man crosses with cabbage.
- Man crosses with nothing.

If we abbreviate these moves as *w* (wolf), *g* (goat), *c* (cabbage), and *n* (nothing), then we can represent a sequence of moves as a string over the alphabet {*w*, *g*, *c*, *n*}. A seven-move solution to the problem is represented by the string *gnwgcng*: first cross with the goat, then cross back with nothing, then cross with the wolf, then cross back with the goat (leaving the wolf on the west side), then cross with the cabbage (leaving the goat on the east side again), then cross back with nothing, and finally cross with the goat.

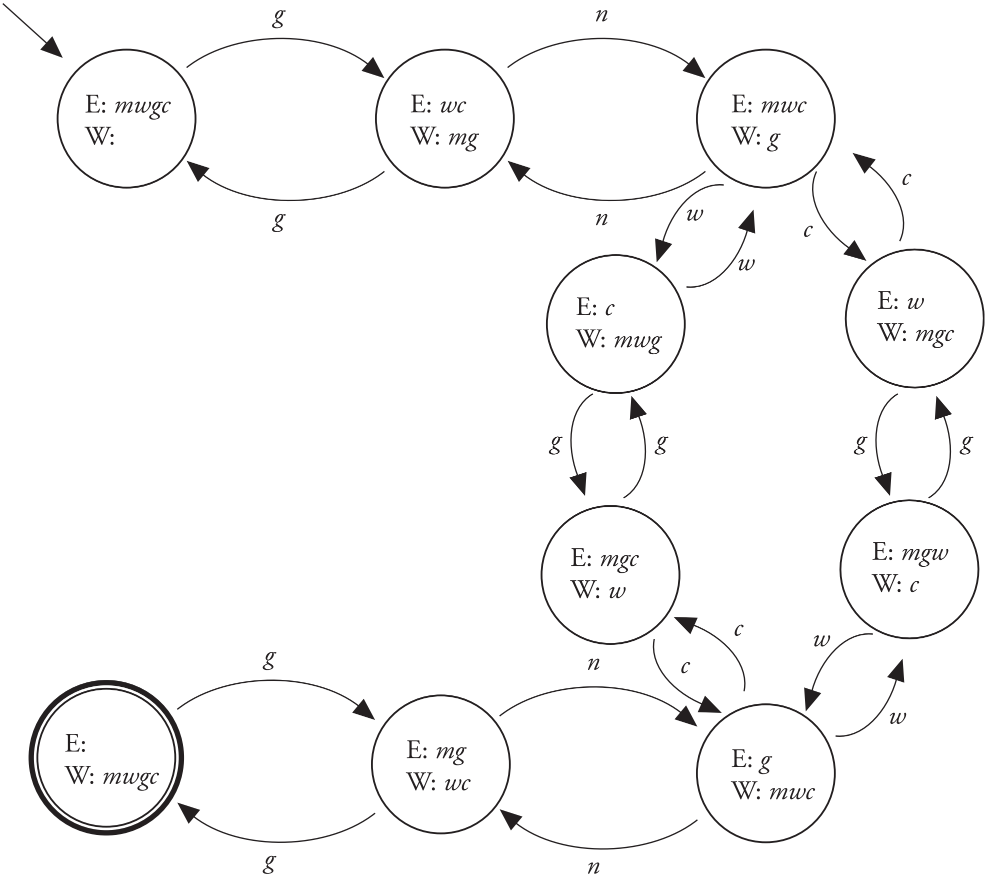
Each move is a *state transition*, taking the puzzle from one state to another state. This can be shown graphically. For instance, this illustration shows that if everything is on the east side of the river, and the *g* move is used, the new state has the wolf and cabbage on the east side and the man and goat on the west side (*m* represents the man):



What happens if the *g* move is used again? Then the man rows back to the east side with the goat, and the whole puzzle returns to the initial state. We include this transition as another arrow:



In this way, a state transition diagram can show all legal transitions and all reachable states, omitting those illegal states where something gets eaten:



An extra arrow indicates the start state, and a double circle indicates the state in which the diagram accepts that the solution is correct.

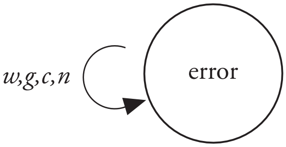
By starting in the start state and making one transition on each symbol in the string *gnwgcng*, you can check that this is indeed a minimum-length solution to the problem. The other minimum-length solution is *gncgwng*. There are longer solutions too, which involve repeated states. For example, the man can just row back and forth with the goat any number of times before completing the solution: *gggggggggncgwng.* In fact, there are infinitely many possible solutions, though only two minimum-length solutions. The language of strings representing legal solutions to the problem can be described with reference to the diagram above: $\{x \in \{w, g, c, n\}^* \mid$ starting in the start state and following the transitions of x ends up in the accepting state$\}$.

2.2 Not Getting Stuck

One problem with the previous diagram is that it gets stuck on many nonsolution strings. What happens, for example, if you follow the previous diagram with the string *gnwn*? The last move, the final *n* in that string, is illegal, since it leaves the goat and the wolf alone on the same side of the river. However, the diagram does not actually say what to do in that case; it just leaves no alternative. The string *gnwn* is not in the language of legal solutions, but the diagram shows this only by getting stuck when the final *n* is reached.

It is a useful property for such a diagram to be fully specified—to show exactly one transition from every state for every symbol in the alphabet, so that it never gets stuck. A fully specified diagram would show what to do on any string, all the way to the end. To fix the previous diagram so that it has this property, we need to give it one additional state: a state representing the fact that an error has been found in the solution.

In the man-wolf-goat-cabbage puzzle, errors are unrecoverable. Once eaten, the goat cannot be restored, so no string beginning with *gnwn* can be a solution to the puzzle, no matter what else follows. The error state needed for the diagram looks like this:



The transition arrow is labeled with all the symbols in the alphabet and simply returns to the same state. This shows that the state is a trap: once reached, it cannot be departed from. All the transitions that were unspecified in the original diagram can now be shown as explicit transitions to this error state. The resulting fully specified diagram is shown on the following page.

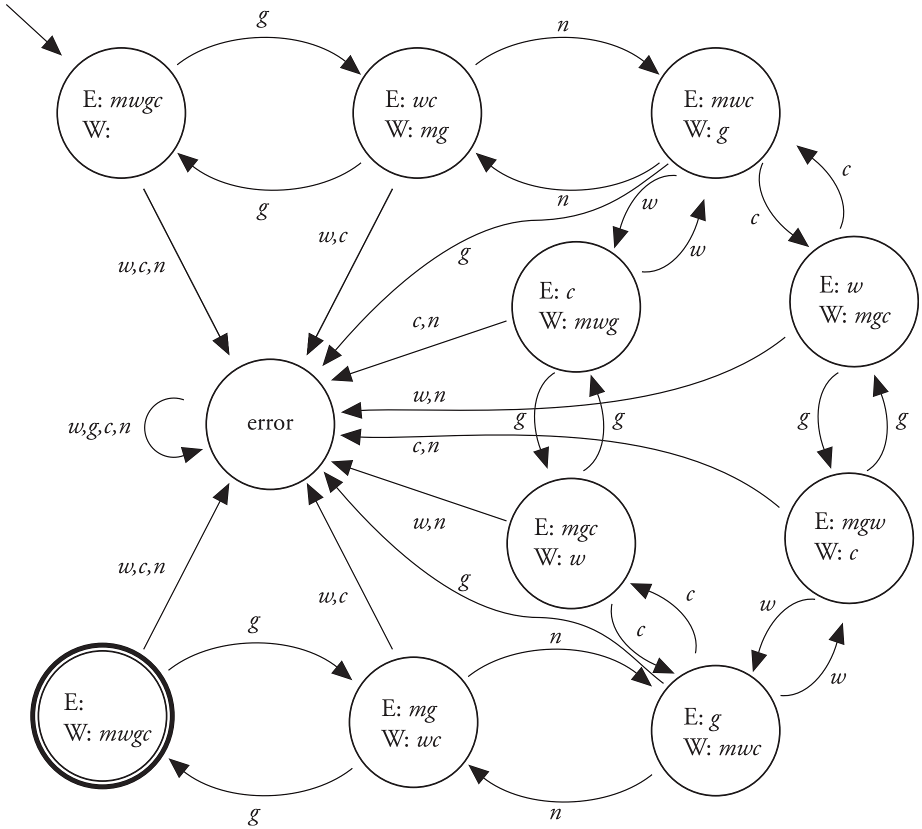
This more elaborate diagram can handle any string over the alphabet {*w*, *g*, *c*, *n*}. By starting in the start state and following the transitions on each character in the string, you end up in a final state. If that final state is the accepting state, the string is a solution; if it is any other state, the string is not a solution.

2.3 Deterministic Finite Automata

The man-wolf-goat-cabbage diagram is an example of a deterministic finite automaton (or DFA), a kind of abstract machine for defining languages.

Here is an informal definition of a DFA; the formal definition comes a little later.

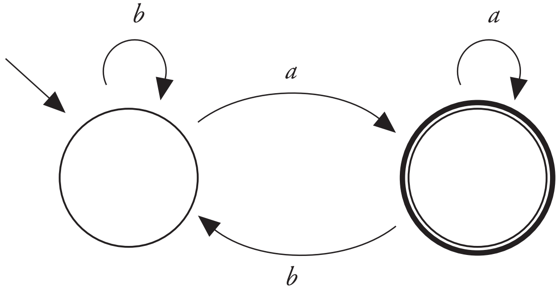
(Informal) A *deterministic finite automaton* or *DFA* is a diagram with a finite number of *states* represented by circles. An arrow points to one of the states, the unique *start state*; double circles mark any number of the states as *accepting*



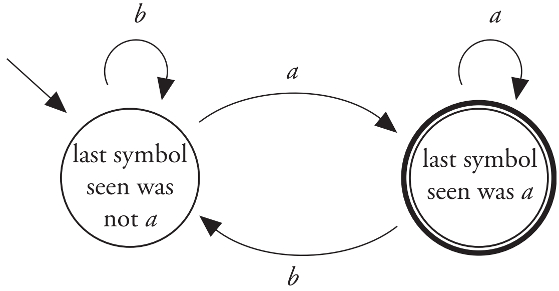
states. From every state, for every symbol in the alphabet Σ, there is exactly one arrow labeled with that symbol going to another state (or back to the same state).

DFAs define languages, just as our man-wolf-goat-cabbage automaton defined the language of solutions to the puzzle. Given any string over its alphabet, a DFA can read the string and follow its state-to-state transitions. At the end of the string it is either in an accepting state or not. If it is an accepting state, we say that the machine accepts the string; otherwise we say that it rejects the string. The language defined by a DFA M is just the set of strings in Σ^* accepted by M.

For example, here is a DFA for $\{xa \mid x \in \{a, b\}^*\}$—that is, the language of strings over the alphabet $\{a, b\}$ that end in a:

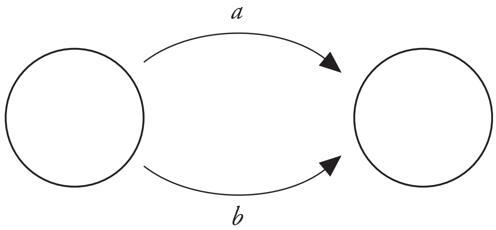


This diagram meets all the requirements for a DFA: it has a finite set of states with a single start state, and it has exactly one transition from every state on every symbol in the alphabet {*a, b*}. Unlike the man-wolf-goat-cabbage diagram, the states in this DFA are unlabeled. It does not hurt to label the states, if they have some meaning that you want to call attention to, or if you need to refer to the states by name for some reason. We could, for example, draw the DFA above like this:

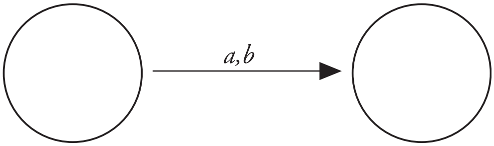


But the labels of states (unlike the labels on the arrows) have no impact on the behavior of the machine. They are rather like comments in a computer program.

There is one important convention you should know about before attempting the exercises at the end of the chapter. If a DFA has more than one arrow with the same source and destination states, like this:



then we usually draw it more compactly as a single arrow with a list of symbols as its label, like this:



These two forms mean exactly the same thing: they both show a state transition that the DFA can make on either *a* or *b*.

2.4 The 5-Tuple

A diagram of a DFA is reasonably easy for people to look at and understand. Mathematically, however, a diagram of a DFA is not very useful. To study DFAs with mathematical rigor—to prove things about them and the languages they accept—it is important to have a more abstract definition.

A DFA has five key parts, so we describe it mathematically as a 5-tuple.

A DFA M is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where
- Q is the finite set of states
- Σ is the alphabet (that is, a finite set of symbols)
- $\delta \in (Q \times \Sigma \to Q)$ is the transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accepting states

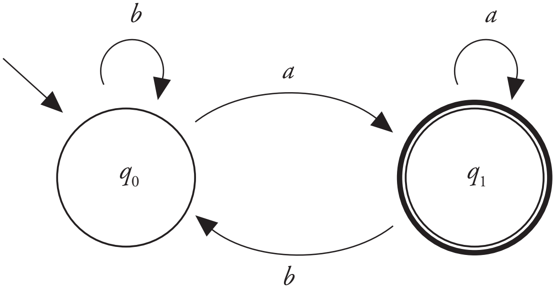
The first part of a DFA is the set Q of states; this corresponds to the set of states drawn as circles in a DFA diagram. The different states in Q are usually referred to as q_i for different values of i; note that the definition implies that there must be at least one state in Q, namely the start state $q_{0.}$

The alphabet Σ is the second part of a DFA. When you draw a diagram for a DFA, the alphabet is implicit; you can tell what it must be by looking at the labels on the arrows. For the formal definition of a DFA, however, the alphabet is explicit.

The third part of a DFA is the transition function δ. The definition says $\delta \in (Q \times \Sigma \to Q)$. This is the mathematical way of giving the type of the function, and it says that δ takes two inputs, a state from Q and a symbol from Σ, and produces a single output, another state in Q. If the DFA is in state q_i reading symbol a, then $\delta(q_i, a)$ is the state to go to next. Thus the transition function encodes all the information about the arrows in a DFA diagram.

The fourth part of the DFA is the start state q_0. There must be exactly one start state. The fifth and final part of the DFA is F, a subset of Q identifying the accepting states, those that are double-circled in the DFA diagram. There is nothing in the definition that prevents F from being {}—in that case there would be no accepting states, so the machine would reject all strings and the language would be {}. There is also nothing that prevents F from being equal to Q—in that case all states would be accepting states, so the machine would accept all strings and the language would be Σ^*.

Consider for example this DFA for the language $\{xa \mid x \in \{a, b\}^*\}$:



Formally, the DFA shown is $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $F = \{q_1\}$, and the transition function δ is

$$\delta(q_0, a) = q_1$$
$$\delta(q_0, b) = q_0$$
$$\delta(q_1, a) = q_1$$
$$\delta(q_1, b) = q_0$$

A DFA is a 5-tuple: it is a mathematical structure with five parts given in order. The names given to those five parts in the definition above—Q , Σ, and so forth—are conventional, but the machine M could just as well have been defined as $M = (\{q_0, q_1\}, \{a, b\}, \delta, q_0, \{q_1\})$, without naming all the parts. The important thing is to specify the five parts in the required order.

2.5 The Language Accepted by a DFA

A DFA's transition function δ says how the DFA makes a single move: one state-to-state transition, reading one symbol from an input string. Intuitively, we understand that the DFA makes a sequence of these moves, one for each symbol in the string, until the end of the string is reached. To capture this intuition about making a sequence of moves, an extended transition function δ^* can be defined.

Where the δ function takes a single symbol as its second parameter, the δ^* function takes a string of zero or more symbols. So for any string $x \in \Sigma^*$ and any state $q_i \in Q$, $\delta^*(q_i, x)$ should be the state the DFA ends up in, starting from q_i and making the sequence of δ-transitions on the symbols in x. There is a simple recursive definition for this δ^*:

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, xa) = \delta(\delta^*(q, x), a)$$

The first line in the definition is the base case. It just says that when the input is the empty string, no transition is made: the final state is the same as the initial state. The second line in the definition is the recursive case. It says that for a string *xa*, where *x* is any string and *a* is the final symbol in the string, you first make all the transitions for *x*, and then one final δ-transition using the final symbol *a*.

The definition above constructs an extended transition function δ* from any basic transition function δ. Using these extended transition functions, it is possible to say exactly what it means for a DFA to accept a string:

A string $x \in \Sigma^*$ is accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ if and only if $\delta^*(q_0, x) \in F$.

That is, a string *x* is accepted if and only if the DFA, started in its start state and taken through the sequence of transitions on the symbols in *x*, ends up in one of its accepting states. We can also define *L*(*M*), the language accepted by a DFA *M*:

For any DFA $M = (Q, \Sigma, \delta, q_0, F)$, $L(M)$ denotes the language accepted by *M*, which is $L(M) = \{x \in \Sigma^* \mid \delta^*(q_0, x) \in F\}$.

A language that can be recognized by a DFA is called a *regular* language.

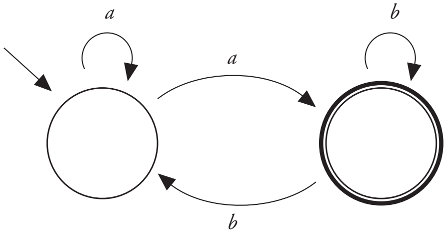
A regular language is one that is *L*(*M*) for some DFA *M*.

A direct way to prove that a language is regular is to give a DFA that accepts it. To prove that a language is *not* regular is generally much harder, and this is a topic that will be addressed later in this book.

Exercises

EXERCISE 1

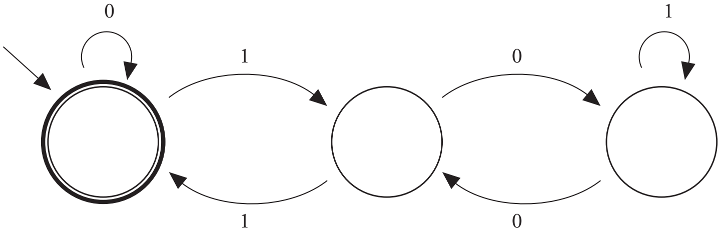
For each of the following strings, say whether it is in the language accepted by this DFA:



a. *b*
b. ε
c. *ab*
d. *abba*
e. *ababaaab*

EXERCISE 2

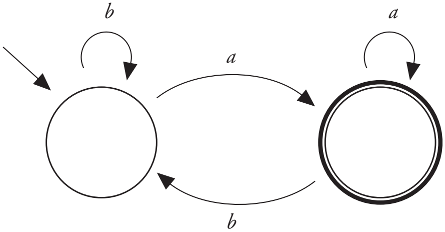
For each of the following strings, say whether it is in the language accepted by this DFA:



a. 0
b. 11
c. 110
d. 1001
e. 101

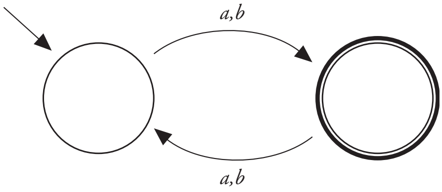
EXERCISE 3

Say what language is accepted by each of the following DFAs. Do not just describe the language in English; write your answer as a set, using set former notation if necessary. For example, if the DFA shown is

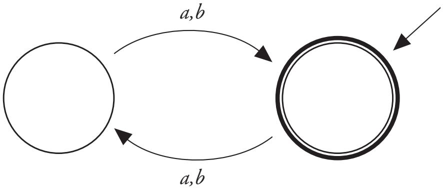


then your answer might be $\{xa \mid x \in \{a, b\}^*\}$.

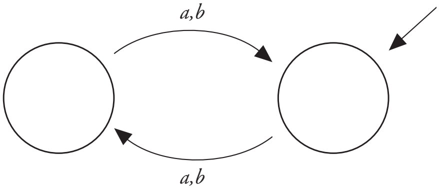
a.



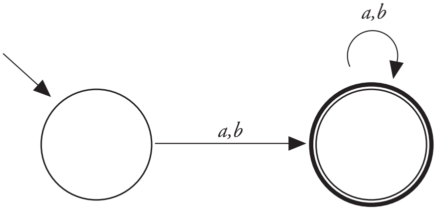
b.



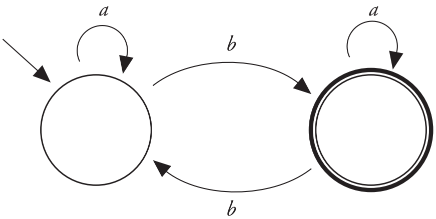
c.



d.



e.



EXERCISE 4

For each of the following languages, draw a DFA accepting that language. The alphabet for the DFA in all cases should be $\{a, b\}$.

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

b. $\{a\}^*$

c. $\{x \in \{a, b\}^* \mid \text{the number of } a\text{s in } x \text{ is odd}\}$

d. $\{ax \mid x \in \{a, b\}^*\}$

e. $\{(ab)^n\}$

EXERCISE 5

Draw the DFA diagram for each of the following DFAs.

a. $(\{q_0, q_1\}, \{a, b\}, \delta, q_0, \{q_1\})$, where the transition function δ is

$$\delta(q_0, a) = q_1$$
$$\delta(q_0, b) = q_1$$
$$\delta(q_1, a) = q_0$$
$$\delta(q_1, b) = q_0$$

b. $(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_0\})$, where the transition function δ is

$$\delta(q_0, 0) = q_0$$
$$\delta(q_0, 1) = q_1$$
$$\delta(q_1, 0) = q_2$$
$$\delta(q_1, 1) = q_0$$
$$\delta(q_2, 0) = q_1$$
$$\delta(q_2, 1) = q_2$$

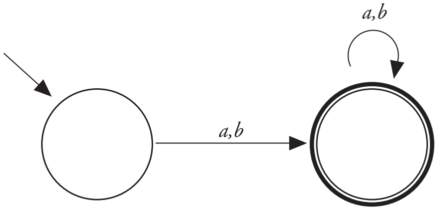
c. $(\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_2\})$, where the transition function δ is

$$\delta(q_0, a) = q_1$$
$$\delta(q_0, b) = q_0$$
$$\delta(q_1, a) = q_1$$
$$\delta(q_1, b) = q_2$$
$$\delta(q_2, a) = q_2$$
$$\delta(q_2, b) = q_2$$

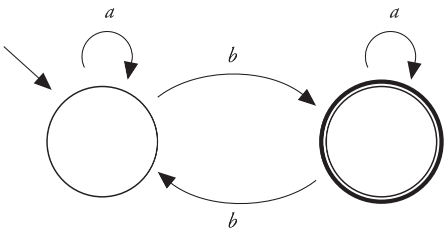
EXERCISE 6

State each of the following DFAs formally, as a 5-tuple.

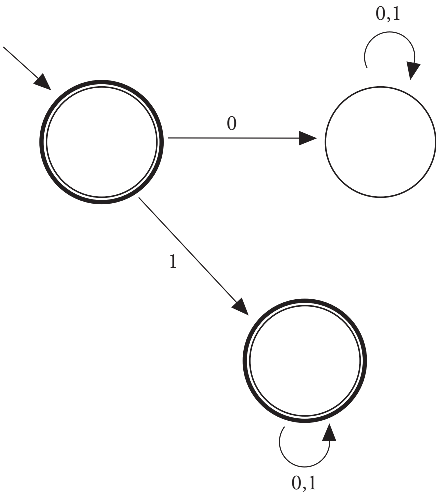
a.



b.



c.

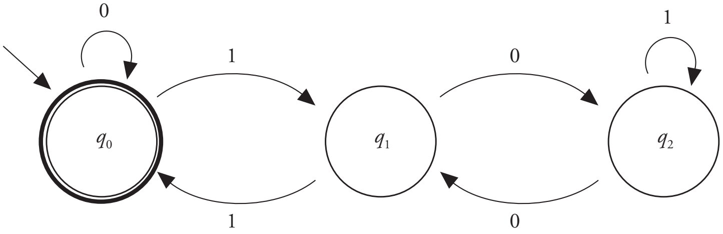


EXERCISE 7

For each machine M in the previous exercise, state what the language $L(M)$ is, using set formers.

EXERCISE 8

Evaluate each of these expressions for the following DFA:



a. $\delta(q_2, 0)$
b. $\delta^*(q_0, 010)$
c. $\delta(\delta^*(q_1, 010), 1)$
d. $\delta^*(q_2, \varepsilon)$
e. $\delta^*(q_2, 1101)$
f. $\delta^*(q_2, 110111)$

EXERCISE 9

Draw the diagram for a DFA for each of the following languages.

a. $\{x \in \{a, b\}^* \mid x \text{ contains at least 3 } a\text{s}\}$
b. $\{x \in \{a, b\}^* \mid x \text{ contains at least 3 consecutive } a\text{s}\}$

c. $\{x \in \{a, b\}^* \mid \text{the substring } bab \text{ occurs somewhere in } x\}$

d. $\{x \in \{a, b\}^* \mid x \text{ has no two consecutive } a\text{s}\}$

e. $\{axb \mid x \in \{a, b\}^*\}$

EXERCISE 10

(This exercise requires some knowledge of fundamental graph algorithms.) The following questions ask for algorithms that operate on DFAs. You should assume that the input is a DFA M represented as a 5-tuple.

a. Give an algorithm that determines whether $L(M) = \{\}$.

b. Give an algorithm that finds a minimum-length string in $L(M)$. Assume that $L(M) \neq \{\}$.

c. Give an algorithm that determines whether $L(M)$ is finite.

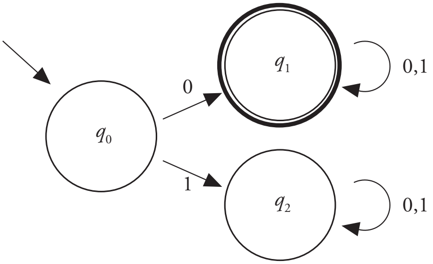
3

Closure Properties for Regular Languages

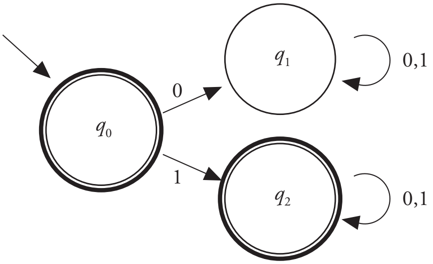
Once we have defined some languages formally, we can consider combinations and modifications of those languages: unions, intersections, complements, and so on. Such combinations and modifications raise important questions. For example, is the intersection of two regular languages also regular—capable of being recognized directly by some DFA?

3.1 Closed Under Complement

Consider the language $L = \{0x \mid x \in \{0, 1\}^*\}$. We can show that this is a regular language by constructing a DFA to recognize it:



L is the language of strings over the alphabet $\{0, 1\}$ that start with a 0. The complement of L, written as $\overline{L}$, is the language of strings over the alphabet $\{0, 1\}$ that do *not* start with a 0. We can show that this is a regular language too, by constructing a DFA to recognize it:



Comparing the two DFAs, you can see that the machine for $\overline{L}$ is the same as the machine for L, with one difference: the accepting states and the nonaccepting states have been exchanged. Whenever the first machine accepts, the second machine rejects; whenever the first machine rejects, the second machine accepts.

This trick for constructing a complemented DFA can be generalized to handle any complemented language. First, here is a formal definition of the complement of a language:

If L is a language over some alphabet Σ, the *complement* of L is $\overline{L} = \{x \in \Sigma^* \mid x \notin L\}$.

Notice that the complement of a language is always taken with respect to the underlying alphabet. This guarantees that the complement of a complement is the original language: $\overline{\overline{L}} = L$.

Now we can easily prove a useful property of the regular languages:

Theorem 3.1: If L is any regular language, $\overline{L}$ is also a regular language.

Proof : Let L be any regular language. By definition there must be some DFA $M = (Q, \Sigma, \delta, q_0, F)$ with $L(M) = L$. Define a new DFA $M' = (Q, \Sigma, \delta, q_0, Q - F)$ using the same set of states, alphabet, transition function, and start state as M, but using $Q–F$ as its set of accepting states. Now for any $x \in \Sigma^*$, M' accepts x if and only if M rejects x. So $L(M') = \overline{L}$. Since M' is a DFA that accepts $\overline{L}$, it follows that $\overline{L}$ is regular.

This theorem shows that if you take any regular language and complement it, you still get a regular language. In other words, you cannot leave the class of regular languages by using the complement operation. We say that the regular languages are *closed under complement.* One useful thing about such *closure properties* is that they give you shortcuts for proving that a language belongs to a particular class. If you need to know whether L is a regular language and you already know that $\overline{L}$ is, you can immediately conclude that L is (without actually constructing a DFA for it) simply by invoking the closed-under-complement property of the regular languages.

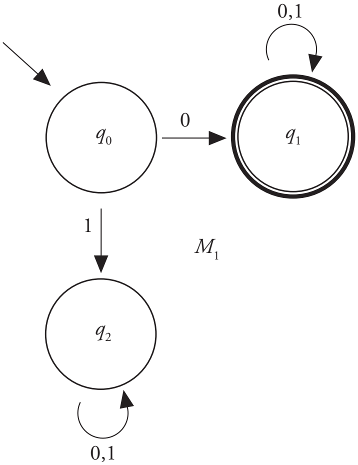
3.2 Closed Under Intersection

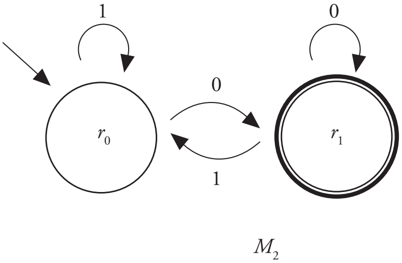
We can define the intersection of two languages in a straightforward way:

If L_1 and L_2 are languages, the *intersection* of L_1 and L_2 is $L_1 \cap L_2 = \{x \mid x \in L_1$ and $x \in L_2\}$.

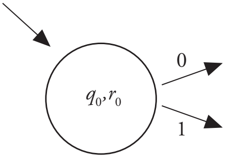
This definition works fine no matter what the underlying alphabets are like. If L_1 and L_2 are over the same alphabet Σ, which is the most common case, then $L_1 \cap L_2$ is a language over the same alphabet—it may not actually use all the symbols of Σ, but it certainly does not use more. Similarly, if L_1 and L_2 are languages over different alphabets Σ_1 and Σ_2, then $L_1 \cap L_2$ is a language over $\Sigma_1 \cap \Sigma_2$. We will generally assume that L_1 and L_2 are over the same alphabet.

The regular languages are closed under intersection. As in the previous section, we can prove this by construction: given any two DFAs, there is a mechanical way to combine them into a third DFA that accepts the intersection of the two original languages. For example, consider the languages $L_1 = \{0x \mid x \in \{0, 1\}^*\}$ and $L_2 = \{x0 \mid x \in \{0, 1\}^*\}$: the language of strings that start with a 0 and the language of strings that end with a 0. Here are DFAs M_1 and M_2 for the two languages:

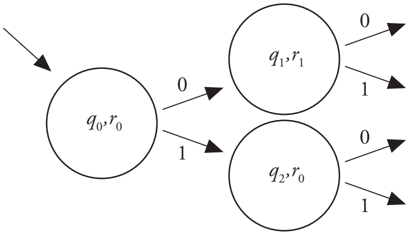




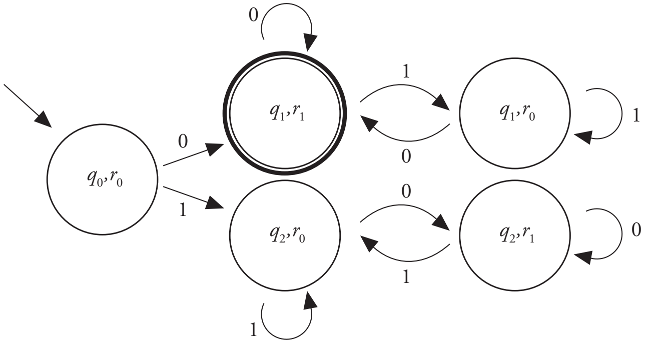
We will define a DFA M_3 that keeps track of where M_1 is *and* where M_2 is after each symbol. Initially, M_1 is in state q_0 and M_2 is in state r_0; our new machine will reflect this by starting in a state named (q_0, r_0):



Now when M_1 is in state q_0 and reads a 0, it goes to state q_1, and when M_2 is in state r_0 and reads a 0, it goes to state r_1, so when M_3 is in state (q_0, r_0) and reads a 0, it goes to a new state named (q_1, r_1). That way it keeps track of what both M_1 and M_2 are doing. We can define the transition on 1 similarly, adding this to the construction of M_3:



The process of constructing M_3 is not infinite, since there are only six possible states of the form (q_i, r_j). In fact, the construction is already about to repeat some states: for example, from (q_1, r_1) on a 0 it will return to (q_1, r_1). The fully constructed DFA M_3 looks like this:



Notice that we did not need a state for the pair (q_0, r_1). That is because this combination of states is unreachable: it is not possible for M_1 to be in the state q_0 while M_2 is in the state r_1. The constructed machine accepts in the state (q_1, r_1), which is exactly where both M_1 and M_2 accept. Since the new machine in effect simulates both the original machines and accepts only when both accept, we can conclude that the language accepted is $L_1 \cap L_{2.}$

The construction uses an operation on sets that you may not have seen before. Given any two sets, you can form the set of all pairs of elements—pairs in which the first is from the first set and the second is from the second set. In mathematics this is called the *Cartesian product* of two sets:

If S_1 and S_2 are sets, the Cartesian product of S_1 and S_2 is
$S_1 \times S_2 = \{(e_1, e_2) \mid e_1 \in S_1 \text{ and } e_2 \in S_2\}$.

To express the general form of the construction combining two DFAs, we use the Cartesian product of the two sets of states. For this reason, the construction used in the following proof is called the *product construction.*

Theorem 3.2: If L_1 and L_2 are any regular languages, $L_1 \cap L_2$ is also a regular language.

Proof: Let L_1 and L_2 be any two regular languages. Since they are regular, there must be some DFAs $M_1 = (Q, \Sigma, \delta_1, q_0, F_1)$ with $L(M_1) = L_1$ and $M_2 = (R, \Sigma, \delta_2, r_0, F_2)$ with $L(M_2) = L_{2.}$ Construct a new DFA $M_3 = (Q \times R, \Sigma, \delta, (q_0, r_0), F_1 \times F_2)$, where δ is defined so that for all $q \in Q$, $r \in R$, and $a \in \Sigma$, we have $\delta((q, r), a) = (\delta_1(q, a), \delta_2(r, a))$. Since this DFA simulates both M_1 and M_2 and accepts if and only if both accept, we can conclude that $L(M_3) = L_1 \cap L_2$. It follows that $L_1 \cap L_2$ is a regular language.

Notice that the machine constructed for the proof includes all the states in $Q \times R$, which may include some unreachable states. In the case of our earlier example, this is a six-state DFA instead of the five-state DFA. The extra state is not reachable by any path from the start state, so it has no effect on the language accepted.

3.3 Closed Under Union

The union of two languages is defined in the usual way:

If L_1 and L_2 are languages, the *union* of L_1 and L_2 is
$L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2, \text{ or both}\}$.

As with intersection, we will generally assume that L_1 and L_2 are over the same alphabet, but the definition works fine even if they are not.

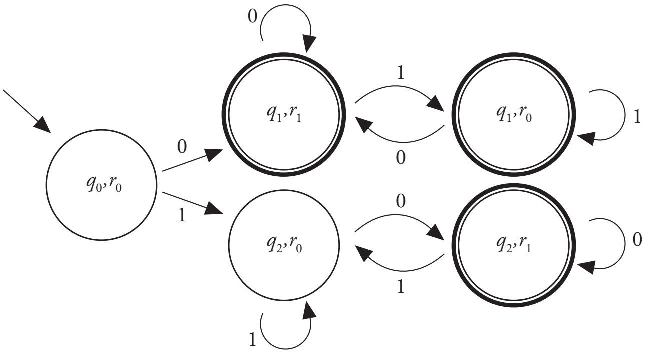
The regular languages are closed under union. There is a simple proof of this using DeMorgan's laws. There is also a direct proof using the subset construction, but choosing a different set of accepting states.

Theorem 3.3: If L_1 and L_2 are any regular languages, $L_1 \cup L_2$ is also a regular language.

Proof 1: Using DeMorgan's laws, $L_1 \cup L_2 = \overline{\overline{L_1} \cap \overline{L_2}}$. This defines union in terms of intersection and complement, for which the regular languages have already been shown to be closed.

Proof 2: Use the product construction as in the proof of closure under intersection, but for the accepting states use $(F_1 \times R) \cup (Q \times F_2)$. Now the constructed DFA simulates both M_1 and M_2 and accepts if and only if either or both accept. We conclude that $L(M_3) = L_1 \cup L_2$ and that $L_1 \cup L_2$ is a regular language.

The product construction for union is almost the same as the product construction for intersection. The only difference is in the choice of accepting states. For intersection we make the new DFA accept when *both* the original DFAs accept, while for union we make the new DFA accept when *either or both* of the original DFAs accept. For example, in Section 3.2 we produced a DFA for the intersection of the two languages $\{0x \mid x \in \{0, 1\}^*\}$ and $\{x0 \mid x \in \{0, 1\}^*\}$. For the union of those same two languages, we could use the same set of states and the same transition function:



The only difference is that this DFA accepts in more states. It accepts if either or both of the original DFAs accept.

3.4 DFA Proofs Using Induction

Rigorous proofs about DFAs often require a special technique: mathematical induction. It is a technique that is useful in many areas of mathematics, including the foundations of computer science. Proofs that use induction are related in an interesting way to programs that use recursion.

Although it is not a primary aim of this book to teach proof techniques, this is too good an opportunity to miss. Mathematical induction and DFAs are a perfect match; you can learn a lot about DFAs by doing inductive proofs on them, and you can learn a lot about proof technique by proving things about DFAs.

If you read the proof of Theorem 3.2 closely, you may have noticed a place where the proof made an important step rather informally. The theorem stated that the regular languages are closed for intersection, and the proof used the product construction. Here is that proof again:

> **Proof:** Let L_1 and L_2 be any two regular languages. Since they are regular, there must be some DFAs $M_1 = (Q, \Sigma, \delta_1, q_0, F_1)$ with $L(M_1) = L_1$ and $M_2 = (R, \Sigma, \delta_2, r_0, F_2)$ with $L(M_2) = L_{2.}$ Construct a new DFA $M_3 = (Q \times R, \Sigma, \delta, (q_0, r_0), F_1 \times F_2)$, where δ is defined so that for all $q \in Q$, $r \in R$, and $a \in \Sigma$, we have $\delta((q, r), a) = (\delta_1(q, a), \delta_2(r, a))$. Since this DFA simulates both M_1 and M_2 and accepts if and only if both accept, we can conclude that $L(M_3) = L_1 \cap L_2$. It follows that $L_1 \cap L_2$ is a regular language.

What exactly does it mean that the new DFA "simulates both M_1 and M_2"? Formally speaking, it means this:

> **Lemma 3.1:** In the product construction, for all $x \in \Sigma^*$, $\delta^*((q_0, r_0), x) = (\delta_1^*(q_0, x), \delta_2^*(r_0, x))$.

That is, for any input string x, the new machine finds exactly the pair of final states that the two original machines reach. Our proof of Theorem 3.2 assumed Lemma 3.1 was true without proving it.

Does Lemma 3.1 require any proof? By construction, we know that for all $q \in Q$, $r \in R$, and $a \in \Sigma$, we have $\delta((q, r), a) = (\delta_1(q, a), \delta_2(r, a))$. This is something like Lemma 3.1, but only for a single symbol a, not for a whole string x. Whether the leap from the δ property to the corresponding δ^* property is obvious enough to be passed over without comment is largely a matter of taste. Whenever you prove something, you have to exercise judgment about how much detail to give. Not enough detail makes a proof unconvincing, while too much makes it tiresome (and just as unconvincing). But even if you think Lemma 3.1 is obvious enough not to require any proof, let's prove it anyway.

For any fixed length of x, it is easy to prove Lemma 3.1. For example, when $|x| = 0$

$$\begin{aligned}
&\delta^*((q_0, r_0), x) \\
&\quad = \delta^*((q_0, r_0), \varepsilon) && \text{(since } |x| = 0\text{)} \\
&\quad = (q_0, r_0) && \text{(by the definition of } \delta^*\text{)} \\
&\quad = (\delta_1^*(q_0, \varepsilon), \delta_2^*(r_0, \varepsilon)) && \text{(by the definitions of } \delta_1^* \text{ and } \delta_2^*\text{)} \\
&\quad = (\delta_1^*(q_0, x), \delta_2^*(r_0, x)) && \text{(since } |x| = 0\text{)}
\end{aligned}$$

And when $|x| = 1$

$$\begin{aligned}
&\delta^*((q_0, r_0), x) \\
&\quad = \delta^*((q_0, r_0), ya) && \text{(for some symbol } a \text{ and string } y\text{)} \\
&\quad = \delta(\delta^*((q_0, r_0), y), a) && \text{(by the definition of } \delta^*\text{)} \\
&\quad = \delta((\delta_1^*(q_0, y), \delta_2^*(r_0, y)), a) && \text{(using Lemma 3.1 for } |y| = 0\text{)} \\
&\quad = (\delta_1(\delta_1^*(q_0, y), a), \delta_2(\delta_2^*(r_0, y), a)) && \text{(by the construction of } \delta\text{)} \\
&\quad = (\delta_1^*(q_0, ya), \delta_2^*(r_0, ya)) && \text{(by the definitions of } \delta_1^* \text{ and } \delta_2^*\text{)} \\
&\quad = (\delta_1^*(q_0, x), \delta_2^*(r_0, x)) && \text{(since } x = ya\text{)}
\end{aligned}$$

Notice that we used the fact that we already proved Lemma 3.1 for strings of length 0. When $|x| = 2$

$$\begin{aligned}
&\delta^*((q_0, r_0), x) \\
&\quad = \delta^*((q_0, r_0), ya) && \text{(for some symbol } a \text{ and string } y\text{)} \\
&\quad = \delta(\delta^*((q_0, r_0), y), a) && \text{(by the definition of } \delta^*\text{)} \\
&\quad = \delta((\delta_1^*(q_0, y), \delta_2^*(r_0, y)), a) && \text{(using Lemma 3.1 for } |y| = 1\text{)} \\
&\quad = (\delta_1(\delta_1^*(q_0, y), a), \delta_2(\delta_2^*(r_0, y), a)) && \text{(by the construction of } \delta\text{)} \\
&\quad = (\delta_1^*(q_0, ya), \delta_2^*(r_0, ya)) && \text{(by the definitions of } \delta_1^* \text{ and } \delta_2^*\text{)} \\
&\quad = (\delta_1^*(q_0, x), \delta_2^*(r_0, x)) && \text{(since } x = ya\text{)}
\end{aligned}$$

Here, we used the fact that we already proved Lemma 3.1 for strings of length 1. As you can see, the proof for $|x| = 2$ is almost the same as the proof for $|x| = 1$. In general, you can easily continue with proofs for $|x| = 3, 4, 5, 6$, and so on, each one using the fact that the lemma was already proved for shorter strings. Unfortunately, this is a proof process that never ends. What we need is a *finite* proof that Lemma 3.1 holds for all the infinitely many different lengths of x.

To prove Lemma 3.1 for all string lengths we use an inductive proof.

Proof: By induction on $|x|$.

Base case: When $|x| = 0$, we have

$$\begin{aligned}
&\delta^*((q_0, r_0), x) \\
&\quad = \delta^*((q_0, r_0), \varepsilon) && \text{(since } |x| = 0\text{)} \\
&\quad = (q_0, r_0) && \text{(by the definition of } \delta^*\text{)}
\end{aligned}$$

$$\begin{aligned}
&= (\delta_1^*(q_0, \varepsilon), \delta_2^*(r_0, \varepsilon)) && \text{(by the definitions of } \delta_1^* \text{ and } \delta_2^*\text{)}\\
&= (\delta_1^*(q_0, x), \delta_2^*(r_0, x)) && \text{(since } |x| = 0\text{)}
\end{aligned}$$

Inductive case: When $|x| > 0$, we have

$$\begin{aligned}
&\delta^*((q_0, r_0), x)\\
&\quad= \delta^*((q_0, r_0), ya) && \text{(for some symbol } a \text{ and string } y\text{)}\\
&\quad= \delta(\delta^*((q_0, r_0), y), a) && \text{(by the definition of } \delta^*\text{)}\\
&\quad= \delta((\delta_1^*(q_0, y), \delta_2^*(r_0, y)), a) && \text{(by the inductive hypothesis, since } |y| < |x|\text{)}\\
&\quad= (\delta_1(\delta_1^*(q_0, y), a), \delta_2(\delta_2^*(r_0, y), a)) && \text{(by the construction of } \delta\text{)}\\
&\quad= (\delta_1^*(q_0, ya), \delta_2^*(r_0, ya)) && \text{(by the definitions of } \delta_1^* \text{ and } \delta_2^*\text{)}\\
&\quad= (\delta_1^*(q_0, x), \delta_2^*(r_0, x)) && \text{(since } x = ya\text{)}
\end{aligned}$$

This proof demonstrates that Lemma 3.1 holds when $|x| = 0$ and then shows that whenever it holds for some length $|x|$, it also holds for $|x| + 1$. It follows that the lemma holds for all string lengths.

Every inductive proof has one or more *base cases* and one or more *inductive cases* using the *inductive hypothesis.* In the proof of Lemma 3.1, the base case is exactly the same as the previous proof for the special case $|x| = 0$. The inductive case then shows how to prove the lemma for any $|x| > 0$. It is a generalized form of the previous proofs for the special cases $|x| = 1$ and $|x| = 2$. Instead of referring to the specific length of the shorter string y for which the lemma has already been proved, it refers to the inductive hypothesis: the assumption that the lemma is already known to hold for any string y with $|y| < |x|$.

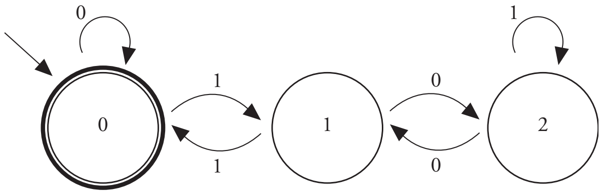
Proof with induction is a lot like programming with recursion. The proof above can be thought of as a program for making a proof for x of any size:

```
void proveit(int n) {
  if (n==0) {
    base case: prove for empty string
  }
  else {
    proveit(n-1);
    prove for strings of length n, assuming n-1 case has been proved
  }
}
```

Not all inductive proofs follow this pattern exactly, of course. There are as many different ways to prove something with induction as there are to program something with recursion. But DFA-related proofs often do follow this pattern. Many such proofs use induction on the length of the string, with the empty string as the base case.

3.5 A Mystery DFA

Consider this DFA:



What language does it accept? By experimenting with it, you can see that it rejects the strings 1, 10, 100, 101, 111, and 1000, while it accepts 0, 11, 110, and 1001. Do you see a pattern there? Can you give an intuitive characterization of the language, before reading further?

First, here is a lemma that summarizes the transition function δ:

Lemma 3.2.1: For all states $i \in Q$ and symbols $c \in \Sigma$,
$\delta(i, c) = (2i + c) \bmod 3$.

Proof: By enumeration. Notice that we have named the states with the numbers 0, 1 and 2, which lets us do direct arithmetic on them. We have

$$\begin{aligned}
\delta(0, 0) &= 0 = (2 \times 0 + 0) \bmod 3\\
\delta(0, 1) &= 1 = (2 \times 0 + 1) \bmod 3\\
\delta(1, 0) &= 2 = (2 \times 1 + 0) \bmod 3\\
\delta(1, 1) &= 0 = (2 \times 1 + 1) \bmod 3\\
\delta(2, 0) &= 1 = (2 \times 2 + 0) \bmod 3\\
\delta(2, 1) &= 2 = (2 \times 2 + 1) \bmod 3
\end{aligned}$$

Now for any string x over the alphabet $\{0, 1\}$, define val(x) as the natural number for which x is a binary representation. (For completeness, define val(ε) = 0.) So, for example, val(11) = 3, val(111) = 7, and val(000) = val(0) = val(ε) = 0. Using this, we can describe the language accepted by the mystery DFA: $L(M) = \{x \mid \text{val}(x) \bmod 3 = 0\}$. In other words, it is the language of strings that are binary representations of numbers that are divisible by three.

Let's prove by induction that $L(M) = \{x \mid \text{val}(x) \bmod 3 = 0\}$. This illustrates a common pitfall for inductive proofs and a technique for avoiding it. If you try to prove directly the hypothesis $L(M) = \{x \mid \text{val}(x) \bmod 3 = 0\}$, you get into trouble:

Lemma 3.2.2 (weak): $L(M) = \{x \mid \text{val}(x) \bmod 3 = 0\}$.

Proof: By induction on $|x|$.

Base case: When $|x| = 0$, we have

$$\begin{aligned}
&\delta^*(0, x)\\
&\quad = \delta^*(0, \varepsilon) && \text{(since } |x| = 0\text{)}\\
&\quad = 0 && \text{(by definition of } \delta^*\text{)}
\end{aligned}$$

So in this case $x \in L(M)$ and $\mathrm{val}(x) \bmod 3 = 0$.

Inductive case: When $|x| > 0$, we have

$$\begin{aligned}
&\delta^*(0, x)\\
&\quad = \delta^*(0, yc) && \text{(for some symbol } c \text{ and string } y\text{)}\\
&\quad = \delta(\delta^*(0, y), c) && \text{(by definition of } \delta^*\text{)}\\
&\quad = ???
\end{aligned}$$

Here the proof would falter, because the inductive hypothesis we're using is not strong enough to make progress with. It tells us that $\delta^*(0, y) = 0$ if and only if $\mathrm{val}(x) \bmod 3 = 0$, but it does not tell us what $\delta^*(0, y)$ is when $\mathrm{val}(x) \bmod 3 \neq 0$. Without knowing that, we can't make progress from here.

To make a successful proof of Lemma 3.2.2, we actually need to prove something even stronger. We will prove that $\delta^*(0, x) = \mathrm{val}(x) \bmod 3$. This implies the weak version of Lemma 3.2.2, because the state of 0 is the only accepting state. But it is stronger than that, because it tells you exactly what state the DFA will end up in after reading any string in $\{0,1\}^*$. Using that trick, let's try the proof again:

Lemma 3.2.2 (strong): $\delta^*(0, x) = \mathrm{val}(x) \bmod 3$.

Proof: By induction on $|x|$.

Base case: When $|x| = 0$, we have

$$\begin{aligned}
&\delta^*(0, x)\\
&\quad = \delta^*(0, \varepsilon) && \text{(since } |x| = 0\text{)}\\
&\quad = 0 && \text{(by definition of } \delta^*\text{)}\\
&\quad = \mathrm{val}(x) \bmod 3 && \text{(since } \mathrm{val}(x) \bmod 3 = \mathrm{val}(\varepsilon) \bmod 3 = 0\text{)}
\end{aligned}$$

Inductive case: When $|x| > 0$, we have

$$\begin{aligned}
&\delta^*(0, x)\\
&\quad = \delta^*(0, yc) && \text{(for some symbol } c \text{ and string } y\text{)}\\
&\quad = \delta(\delta^*(0, y), c) && \text{(by definition of } \delta^*\text{)}\\
&\quad = \delta(\mathrm{val}(y) \bmod 3, c) && \text{(using the inductive hypothesis)}\\
&\quad = (2(\mathrm{val}(y) \bmod 3) + c) \bmod 3 && \text{(by Lemma 3.2.1)}\\
&\quad = 2(\mathrm{val}(y) + c) \bmod 3 && \text{(using modulo arithmetic)}\\
&\quad = \mathrm{val}(yc) \bmod 3 && \text{(using binary arithmetic, } \mathrm{val}(yc) = 2(\mathrm{val}(y)) + c\text{)}\\
&\quad = \mathrm{val}(x) \bmod 3 && \text{(since } x = yc\text{)}
\end{aligned}$$

We conclude by induction that $\delta^*(0, x) = \text{val}(x) \bmod 3$. Since the only accepting state is 0, the language of strings accepted by the DFA is $L(M) = \{x \mid \text{val}(x) \bmod 3 = 0\}$.

This technique is something you will often need to use with inductive proofs; to make the induction go through, you will often find that you need to prove something stronger, something more detailed, than your original hypothesis. This is a little counterintuitive at first. It seems like it should be more difficult to prove something stronger and easier to prove something weaker. But with induction, you get to use the thing you are trying to prove as the inductive hypothesis, so proving something stronger can give you more to work with.

Exercises

EXERCISE 1

Draw DFAs for the following languages:

a. $\{0x0 \mid x \in \{0, 1\}^*\}$
b. the complement of the language above
c. $\{xabay \mid x \in \{a,b\}^* \text{ and } y \in \{a,b\}^*\}$
d. the complement of the language above
e. $\{waxayaz \mid w, x, y, \text{ and } z \text{ are all in } \{a, b\}^*\}$
f. the complement of the language above

EXERCISE 2

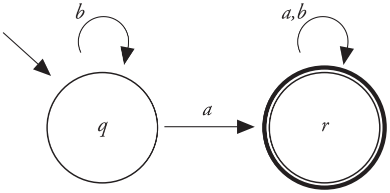
The DFA constructed for the example in Section 3.2 accepts the language of strings over the alphabet {0, 1} that start and end with a 0. It is not, however, the smallest DFA to do so. Draw a DFA for the same language, using as few states as possible.

EXERCISE 3

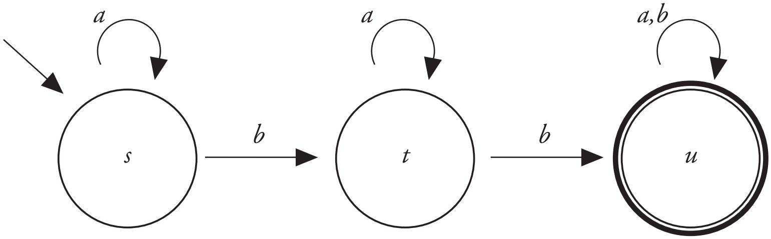
The DFA constructed for the example in Section 3.2 has five states, but the full DFA given by the product construction has six. Draw the full six-state DFA.

EXERCISE 4

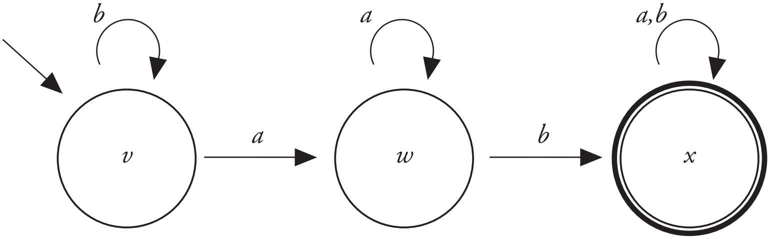
Consider the following three DFAs and the languages they define:



$L_1 = \{xay \mid x \in \{a, b\}^* \text{ and } y \in \{a, b\}^*\}$



$L_2 = \{xbybz \mid x, y, \text{ and } z \text{ are all in } \{a, b\}^*\}$



$L_3 = \{xaby \mid x \in \{a, b\}^* \text{ and } y \in \{a, b\}^*\}$

Apply the product construction to create DFAs for each of the following languages. Label each state with the pair of states to which it corresponds. You do not need to show unreachable states.

a. $L_1 \cup L_2$
b. $\overline{L} \cap L_2$
c. $L_1 \cup L_3$
d. $L_2 \cap L_3$

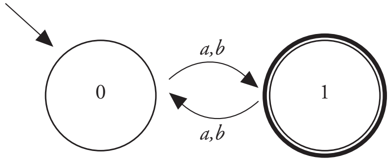
EXERCISE 5

If L_1 and L_2 are languages, the *difference* of L_1 and L_2 is $L_1 - L_2 = \{x \mid x \in L_1 \text{ and } x \notin L_2\}$.

a. Using the product construction to combine two DFAs $M_1 = (Q, \Sigma, \delta_1, q_0, F_1)$ and $M_2 = (R, \Sigma, \delta_2, r_0, F_2)$, what should the set of accepting states be so that the resulting machine accepts $L(M_1) - L(M_2)$?
b. Prove that the regular sets are closed for set difference, *without* using the product construction.

EXERCISE 6

Consider this DFA M:



Prove by induction that $L(M) = \{x \in \{a, b\}^* \mid |x| \bmod 2 = 1\}$.

EXERCISE 7

Prove formally that for any DFA $M = (Q, \Sigma, \delta, q_0, F)$, any strings $x \in \Sigma^*$ and $y \in \Sigma^*$ and any state $q \in Q$, $\delta^*(q, xy) = \delta^*(\delta^*(q, x), y)$. Use induction on $|y|$.

EXERCISE 8

Draw a DFA that accepts the language of strings that are binary representations of numbers that are divisible by 2.

EXERCISE 9

Draw a DFA that accepts the language of strings that are binary representations of numbers that are divisible by 4.

EXERCISE 10

Draw a DFA that accepts the language of strings that are binary representations of numbers that are divisible by 5.

EXERCISE 11

Prove that your DFA from Exercise 10 accepts the language it is supposed to accept. Use induction on the length of the input string.

EXERCISE 12

For any $n \in \mathcal{N}$, $n \neq 0$, define the DFA $M_n = (\{0, 1, \ldots, n - 1\}, \{0, 1\}, \delta, 0, \{0\})$, where $\delta(i, c) = (2i + c) \bmod n$. Prove that $L(M_n) = \{x \mid \text{val}(x) \bmod n = 0\}$.

4

Deterministic Finite-Automata Applications

We have seen how DFAs can be used to define formal languages. In addition to this formal use, DFAs have practical applications. DFA-based pieces of code lie at the heart of many commonly used computer programs.

4.1 An Overview of DFA Applications

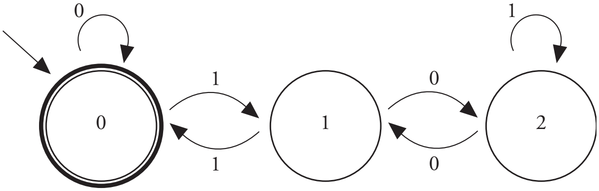
Deterministic finite automata have many practical applications:

- Almost all compilers and other language-processing systems use DFA-like code to divide an input program into tokens like identifiers, constants, and keywords and to remove comments and white space.
- For many applications that accept typed commands, the command language is quite complex, almost like a little programming language. Such applications use DFA-like code to process the input command.
- Text processors often use DFA-like code to search a text file for strings that match a given pattern. This includes most versions of Unix tools like awk, egrep, and Procmail, along with a number of platform-independent systems such as MySQL.
- Speech-processing and other signal-processing systems often use a DFA-like technique to transform an incoming signal.
- Controllers use DFA-like techniques to track the current state of a wide variety of finite-state systems, from industrial processes to video games. They can be implemented in hardware or in software.

Sometimes, the DFA-like code in such applications is generated automatically by special tools such as lex, which we discuss more later. In this chapter we concentrate on examples of DFA-like code constructed by hand.

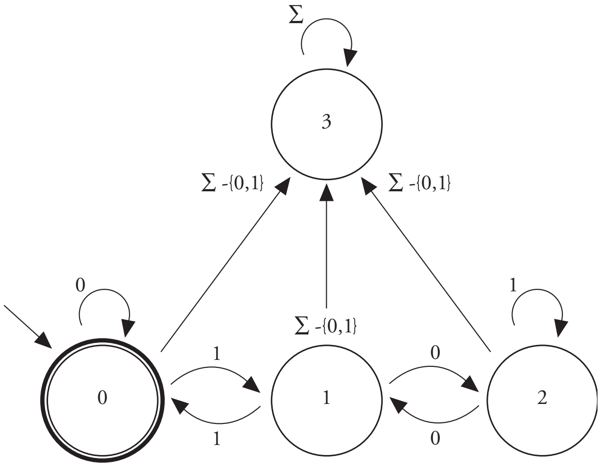
4.2 A DFA-based Text Filter in Java

The last chapter introduced a DFA that accepts strings that are binary representations of numbers that are divisible by three. Here is that DFA again:



Now let's see how this DFA can be implemented in the Java programming language.

The first thing to deal with is the input alphabet. The DFA above uses the alphabet {0, 1}, which is the alphabet of interest for this problem. But the program will work with a typed input string, so we do not have the luxury of restricting the alphabet in this way. The program should accept `"011"` (a representation for the number 3) and reject `"101"` (a representation for the number 5), but it must also properly reject strings like `"01i"` and `"fred"`. The alphabet for the Java implementation must be the whole set of characters that can occur in a Java string—that is, the whole set of values making up the Java `char` type. The DFA we actually implement will have four states, like this:



An object of the Mod3 class represents such a DFA. A Mod3 object has a current state, which is encoded using the integers 0 through 3. The class definition begins like this:

```
/**
 * A deterministic finite-state automaton that
 * recognizes strings that are binary representations
 * of natural numbers that are divisible
 * by 3. Leading zeros are permitted, and the
 * empty string is taken as a representation for 0
 * (along with "0", "00", and so on).
 */
public class Mod3 {

  /*
   * Constants q0 through q3 represent states, and
   * a private int holds the current state code.
   */
  private static final int q0 = 0;
  private static final int q1 = 1;
  private static final int q2 = 2;
  private static final int q3 = 3;

  private int state;
```

The int variables q0, q1, q2, and q3 are private (visible only in this class), static (shared by all objects of this class), and final (not permitted to change after initialization).

In effect, they are named constants. The `int` field named `state` will hold the current state of each `Mod3` object.

The next part of this class definition is the transition function, a method named `delta`.

```
/**
 * The transition function.
 * @param s state code (an int)
 * @param c char to make a transition on
 * @return the next state code
 */
static private int delta(int s, char c) {
  switch (s) {
    case q0: switch (c) {
          case '0': return q0;
          case '1': return q1;
          default: return q3;
        }
    case q1: switch (c) {
          case '0': return q2;
          case '1': return q0;
          default: return q3;
        }
    case q2: switch (c) {
          case '0': return q1;
          case '1': return q2;
          default: return q3;
        }
    default: return q3;
  }
}
```

The `delta` method is declared `private` (cannot be called from outside the class) and `static` (is not given an object of the class to work on). It is thus a true function, having no side effects and returning a value that depends only on its parameters. It computes the transition function shown in the previous DFA diagram.

Next, the class defines methods that modify the `state` field of a `Mod3` object. The first resets it to the start state; the second applies the transitions required for a given input string.

```
  /**
   * Reset the current state to the start state.
   */
  public void reset() {
    state = q0;
  }

  /**
   * Make one transition on each char in the given
   * string.
   * @param in the String to use
   */
  public void process(String in) {
    for (int i = 0; i < in.length(); i++) {
      char c = in.charAt(i);
      state = delta(state, c);
    }
  }
```

The `process` method makes one transition on each character in the input string. Note that `process` handles the empty string correctly—by doing nothing.

The only other thing required is a method to test whether, after processing an input string, the DFA has ended in an accepting state:

```
  /**
   * Test whether the DFA accepted the string.
   * @return true if the final state was accepting
   */
  public boolean accepted() {
    return state==q0;
  }
}
```

That is the end of the class definition. To test whether a string `s` is in the language defined by this DFA, we would write something like

```
    Mod3 m = new Mod3();
    m.reset();
    m.process(s);
    if (m.accepted()) ...
```

To demonstrate the Mod3 class we will use it in a Java application. This is a simple filter program. It reads lines of text from the standard input, filters out those that are not binary representations of numbers that are divisible by three, and echoes the others to the standard output.

```
import java.io.*;

/**
 * A Java application to demonstrate the Mod3 class by
 * using it to filter the standard input stream. Those
 * lines that are accepted by Mod3 are echoed to the
 * standard output.
 */
public class Mod3Filter {
  public static void main(String[] args)
        throws IOException {

    Mod3 m = new Mod3(); // The DFA
    BufferedReader in =  // Standard input
      new BufferedReader(new InputStreamReader(System.in));

    // Read and echo lines until EOF

    String s = in.readLine();
    while (s!=null) {
      m.reset();
      m.process(s);
      if (m.accepted()) System.out.println(s);
      s = in.readLine();
    }
  }
}
```

To test this program, we can create a file named numbers containing the numbers zero through ten in binary:

```
0
1
10
11
100
101
110
111
1000
1001
1010
```

After compiling `Mod3Filter` (and `Mod3`), we can use it on the file `numbers` to filter out all the numbers not divisible by 3. On a Unix system, the command `java Mod3Filter < numbers` produces this output:

```
0
11
110
1001
```

4.3 Table-driven Alternatives

The `Mod3` class implements the transition function in the obvious way, as a Java method `delta`. The `delta` method branches on the current state, and then within each alternative branches again on the current character.

An alternative implementation is to encode the transition function as a table, a two-dimensional array `delta`, indexed using the current state and symbol. Instead of calling the function `delta(s,c)`, passing it the state `s` and character `c` as parameters, we perform an array reference `delta[s,c]`. Array references are generally faster than function calls, so this should make the DFA run faster. For example, the `process` method could look something like this:

```
static void process(String in) {
  for (int i = 0; i < in.length(); i++) {
    char c = in.charAt(i);
    state = delta[state, c];
  }
}
```

Of course, the array delta must first be initialized with the appropriate transitions, so that delta[q0,'0'] is q0, delta[q0,'1'] is q1, and so on. To avoid the possibility of the reference delta[state,c] being out of bounds, delta will have to be initialized with a very large array. The program uses only 4 values for state, but there are 65,536 possible values for c! That is because Java uses Unicode, a 16-bit character encoding, for the char type. Depending on the source of the input string, we may be able to restrict this considerably—we may know, for example, that the characters are 7-bit ASCII (with values 0 through 127). Even so, we will have to initialize the array so that delta[state,c] is q3 for every value of c other than '0' and '1'.

Instead of using a very large array, we could use a small array but handle the exception that occurs when the array reference is out of bounds. In Java this is the ArrayIndexOutOfBoundsException; the process method could catch this exception and use the state q3 as the next state whenever it occurs. The definition of the delta array and the process method would then be the following:

```
/*
 * The transition function represented as an array.
 * The next state from current state s and character c
 * is at delta[s,c-'0'].
 */
static private int[][] delta =
   {{q0,q1},{q2,q0},{q1,q2},{q3,q3}};

/**
 * Make one transition on each char in the given
 * string.
 * @param in the String to use
 */
public void process(String in) {
  for (int i = 0; i < in.length(); i++) {
    char c = in.charAt(i);
    try {
      state = delta[state][c-'0'];
    }
    catch (ArrayIndexOutOfBoundsException ex) {
      state = q3;
    }
  }
}
```

This is a reasonable way to solve the problem by hand. Automatically generated systems usually use the full table with an element for every possible input character. One reason for this is that when the full array is used, `process` need contain no reference to individual states or characters. This way, any DFA can be implemented using the same `process` code, just by substituting a different transition table.

Incidentally, the transition table is usually stored in a more compressed form than we have shown. Our implementation used a full 32-bit `int` for each entry in the table, which is quite wasteful. It would be relatively easy to implement this using one byte per entry, and it would be possible to use even less, since we really need only two bits to represent our four possible states. The degree of compression chosen is, as always, a trade-off: heavily compressed representations take less space, but using them slows down each table access and thus slows down each DFA transition.

Exercises

EXERCISE 1

Reimplement `Mod3` using the full transition table. Assume that the characters in the input string are in the range 0 through 127.

EXERCISE 2

Using the DFA-based approach, write a Java class `Mod4Filter` that reads lines of text from the standard input, filters out those that are not binary representations of numbers that are divisible by four, and echoes the others to the standard output.

EXERCISE 3

Using the DFA-based approach, write a Java class `ManWolf` that takes a string from the command line and reports whether or not it represents a solution to the man-wolf-goat-cabbage problem of Chapter 2. For example, it should have this behavior:

```
> java ManWolf gncgwng
That is a solution.
> java ManWolf ggggggggg
That is not a solution.
```

5

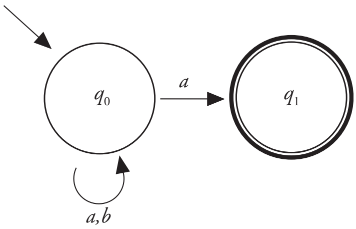
Nondeterministic Finite Automata

A DFA has exactly one transition from every state on every symbol in the alphabet. By relaxing this requirement we get a related but more flexible kind of automaton: the **nondeterministic finite automaton** *(NFA).*

NFAs are a bit harder to think about than DFAs because they do not appear to define simple computational processes. They may seem at first to be unnatural, like puzzles invented by professors for the torment of students. But have patience! NFAs and other kinds of nondeterministic automata arise naturally in many ways, as you will see later in this book, and they too have a variety of practical applications.

5.1 Relaxing a Requirement

Consider the following state machine:



This is not a DFA, since it violates the rule that there must be exactly one transition from every state on every symbol in the alphabet. There is no transition from the q_1 state on either *a* or *b*, and there is more than one transition from the q_0 state on an *a*. This is an example of a nondeterministic finite automaton (NFA).

How can you tell whether such a machine accepts a given string? Consider the string *aa*. Following the arrows, there are three possible sequences of moves:

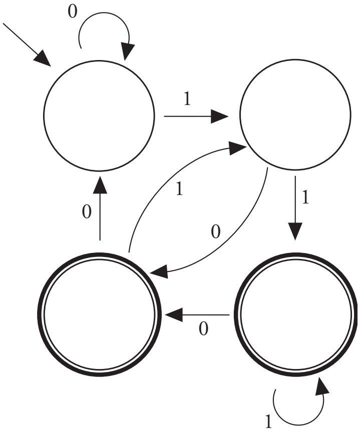
1. from q_0 back to q_0 on the first *a*, and back to q_0 again on the second;
2. from q_0 back to q_0 on the first *a*, then to q_1 on the second; and
3. from q_0 to q_1 on the first *a*, then getting stuck with no legal move on the second.

Only the second of these is an accepting sequence of moves. But the convention is that if there is any way for an NFA to accept a string, that string is taken to be part of the language defined by the NFA. The string *aa* is in the language of the NFA above because there is a sequence of legal moves that reaches the end of *aa* in an accepting state. The fact that there are also sequences of legal moves that do not succeed is immaterial.

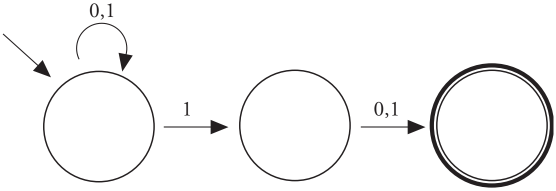
The language defined by the NFA above is $\{xa \mid x \in \{a, b\}^*\}$—that is, the language of strings over the alphabet $\{a, b\}$ that end in *a*. The machine can accept any such string by staying in the q_0 state until it reaches the *a* at the end, then making the transition to q_1 as its final move.

A DFA can be thought of as a special kind of NFA—an NFA that happens to specify exactly one transition from every state on every symbol. In this sense every DFA is an NFA, but not every NFA is a DFA. So in some ways the name *nondeterministic finite automaton* is misleading. An NFA might more properly be called a *not-necessarily-deterministic finite automaton*.

The extra flexibility afforded by NFAs can make them much easier to construct. For example, consider the language $\{x \in \{0, 1\}^* \mid \text{the next-to-last symbol in } x \text{ is } 1\}$. The smallest DFA for the language is this one:



The connection between the language and this DFA is not obvious. (With some effort you should be able to convince yourself that the four states correspond to the four possibilities for the last two symbols seen—00, 01, 10, or 11—and that the two accepting states are the states corresponding to 10 and 11, where the next-to-last symbol is a 1.) An NFA for the language can be much simpler:

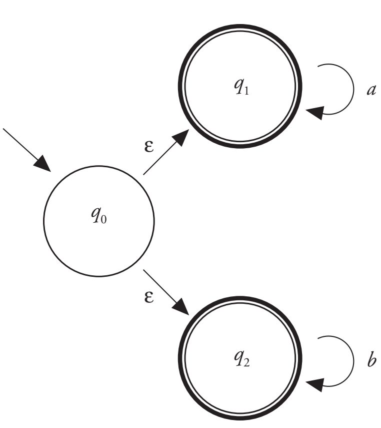


This NFA has fewer states and far fewer transitions. Moreover, it is much easier to understand. It can clearly accept a string if and only if the next-to-last symbol is a 1.

5.2 Spontaneous Transitions

An NFA has one additional trick up its sleeve: it is allowed to make a state transition spontaneously, without consuming an input symbol.

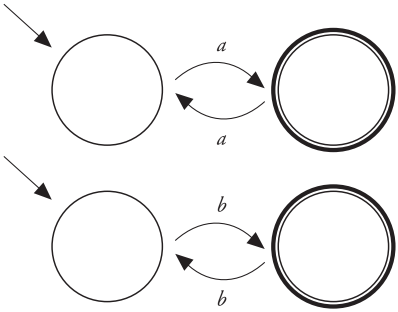
In NFA diagrams the option of making a non-input-consuming transition is shown as an arrow labeled with ε. (Recall that ε is not normally used as a symbol in an alphabet, but stands for the empty string. That makes sense here since no symbols are consumed when an ε-transition is made.) For example, this NFA accepts any string of all *a*s and also accepts any string of all *b*s:



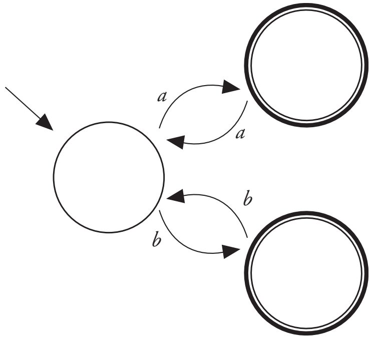
To accept the string *aa*, this NFA would first make an ε-transition to q_1, then make a transition back to q_1 on the first *a*, and finally another transition back to q_1 on the second *a*.

Notice that although q_0 is not an accepting state, this NFA does accept the empty string. Because of the ε-transitions, it has three possible sequences of moves when the input is the empty string: it can make no moves, staying in q_0; it can move to q_1 and end there; or it can move to q_2 and end there. Since two of these are sequences that end in accepting states, the empty string is in the language defined by this NFA. In general, any state that has an ε-transition to an accepting state is, in effect, an accepting state too.

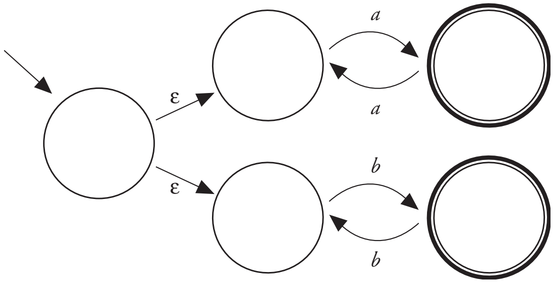
ε-transitions are especially useful for combining smaller automata into larger ones. The example above is an NFA that accepts $\{a^n\} \cup \{b^n\}$, and it can be thought of as the combination of two smaller NFAs, one for the language $\{a^n\}$ and one for the language $\{b^n\}$. For another example, consider these two NFAs:



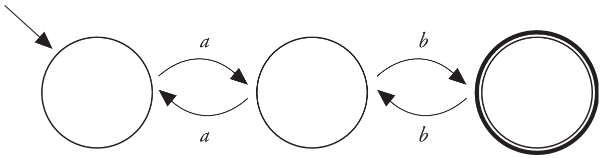
The first accepts $L_1 = \{a^n \mid n \text{ is odd}\}$, and the second accepts $L_2 = \{b^n \mid n \text{ is odd}\}$. Suppose you wanted an NFA for the union of the two languages, $L_1 \cup L_2$. You might think of combining the two smaller NFAs by unifying their start states, like this:



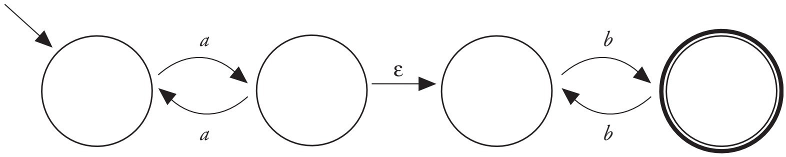
But that does not work—it accepts strings like *aab*, which is not in either of the original languages. Since NFAs may *return* to their start states, you cannot form an NFA for the union of two languages just by combining the start states of their two NFAs. But you can safely combine the two NFAs using ε-transitions, like this:



Similarly, suppose you wanted an NFA for the concatenation of the two languages, $\{xy \mid x \in L_1 \text{ and } y \in L_2\}$. You might think of combining the two smaller NFAs by using the accepting state of the first as the start state of the second, like this:



But again that does not work—it accepts strings like *abbaab*, which is not in the desired language. As before, the problem can be solved using ε-transitions:



Using such techniques, large NFAs can be composed by combining smaller NFAs. This property of NFAs is called *compositionality*, and it is one of the advantages NFAs have over DFAs in some applications. We will see further examples of this in later chapters.

5.3 Nondeterminism

NFAs and DFAs both define languages, but only DFAs really define direct computational procedures for testing language membership. It is easy to check whether a DFA accepts a given input string: you just start in the start state, make one transition for each symbol in the string, and check at the end to see whether the final state is accepting. You can carry this computational procedure out for yourself or write a program to do it. But what about the same test for an NFA—how can you test whether an NFA accepts a given input string? It is no longer so easy, since there may be more than one legal sequence of steps for an input, and you have to determine whether at least one of them ends in an accepting state. This seems to require searching through all legal sequences of steps for the given input string, but how? In what order? The NFA does not say. It does not fully define a computational procedure for testing language membership.

This is the essence of the *nondeterminism* of NFAs:

1. For a given input there can be more than one legal sequence of steps.
2. The input is in the language if at least one of the legal sequences says so.

The first part of nondeterminism is something you may be familiar with from ordinary programming. For example, a program using a random-number generator can make more than one legal sequence of steps for a given input. But randomness is not the same thing as nondeterminism. That second part of nondeterminism is the key, and it is what makes nondeterministic automata seem rather alien to most programmers.

Because of their nondeterminism, NFAs are harder to implement than DFAs. In spite of this higher level of abstraction, NFAs have many practical applications, as we will see in later chapters. We will see algorithmic techniques for deciding whether an NFA accepts a given string, and we will see that, in many applications, the compactness and compositionality of NFAs outweigh the difficulties of nondeterminism.

5.4 The 5-Tuple for an NFA

To prove things about DFAs, we found that diagrams were not very useful. We developed a formalism, the 5-tuple, to represent their essential structure. Now we'll do the same thing for NFAs.

First, we need a preliminary definition. Given any set S, you can form a new set by taking the set of all the subsets of S, including the empty set and S itself. In mathematics this is called the *powerset* of the original set:

If S is a set, the powerset of S is $P(S) = \{R \mid R \subseteq S\}$.

The primary difference between a DFA and an NFA is that a DFA must have exactly one transition from every state on every symbol, while an NFA can have any number, zero or more. To reflect this change in the mathematical description of an NFA, we make a subtle but important change: the output of the transition function, which is a *single state* for the DFA, is a *set of states* for an NFA.

An NFA M is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

Q is the finite set of states

Σ is the alphabet (that is, a finite set of symbols)

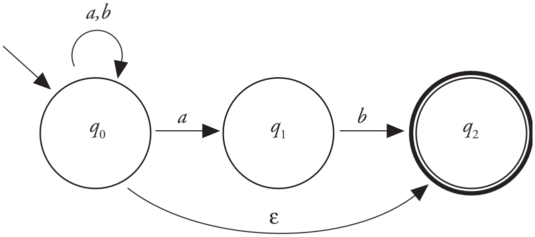
$\delta \in (Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q))$ is the transition function

$q_0 \in Q$ is the start state

$F \subseteq Q$ is the set of accepting states

The definition says $\delta \in (Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q))$; this is the mathematical way of giving the type of the function, and it says that δ takes two inputs, a state from Q and a symbol from $\Sigma \cup \{\varepsilon\}$, and produces a single output, which is some subset of the set Q. If the NFA is in the q_i state, then $\delta(q_i, a)$ is the set of possible next states after reading the a symbol and $\delta(q_i, \varepsilon)$ is the set of possible next states that can be reached on an ε-transition, without consuming an input symbol. Thus the transition function encodes all the information about the arrows in an NFA diagram. All the other parts of the NFA—the set of states, the alphabet, the start state, and the set of accepting states—are the same as for a DFA.

Consider for example this NFA:



Formally, the NFA shown is $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $F = \{q_2\}$, and the transition function δ is

$\delta(q_0, a) = \{q_0, q_1\}$
$\delta(q_0, b) = \{q_0\}$
$\delta(q_0, \varepsilon) = \{q_2\}$
$\delta(q_1, a) = \{\}$
$\delta(q_1, b) = \{q_2\}$
$\delta(q_1, \varepsilon) = \{\}$
$\delta(q_2, a) = \{\}$

$\delta(q_2, b) = \{\}$
$\delta(q_2, \varepsilon) = \{\}$

When the diagram has no arrow from a given state on a given symbol, as for $\delta(q_1, a)$, the transition function reflects this by producing {}, the empty set, as the set of possible next states. When the diagram has a single arrow from a given state on a given symbol, as for $\delta(q_0, b)$, the transition function produces a singleton set such as $\{q_0\}$, showing that there is exactly one possible next state. When the diagram allows more than one transition from a given state on a given symbol, as for $\delta(q_0, a)$, the transition function produces a set containing more than one possible next states.

The example above demonstrates an odd technicality about NFA alphabets. We assumed for the example that $\Sigma = \{a, b\}$, but all we really know from looking at the diagram is that Σ contains a and b. It could contain other symbols as well. A DFA diagram tells you unequivocally what the alphabet must be, because it gives a transition from every state on every symbol in the alphabet. An NFA need not give transitions on every symbol, so the alphabet is not fully determined by the diagram. Usually this is an inconsequential technicality; the language defined by the machine above is $\{a, b\}^*$, even if we take it that $\Sigma = \{a, b, c\}$. But sometimes you do need to know the exact alphabet (as when taking the complement of the language), and at such times an NFA diagram by itself is inadequate.

5.5 The Language Accepted by an NFA

As an NFA operates, it reads its input string and changes its state. At any point in the NFA's operation, its future depends on two things: the current state, and that part of the input string that is still to be read. Taken together, these two pieces of information are called an *instantaneous description* (*ID*) for the NFA. The following definitions are all made with respect to some fixed NFA $M = (Q, \Sigma, \delta, q_0, F)$:

An instantaneous description for an NFA is a pair (q, x) where
$q \in Q$ is the current state
$x \in \Sigma^*$ is the unread part of the input

The δ function for the NFA determines a relation $\mapsto$ on IDs; we write $I \mapsto J$ if I is an ID and J is an ID that could follow from I after one move of the NFA.

$\mapsto$ is a relation on IDs. For any string $x \in \Sigma^*$ and any $\omega \in \Sigma$ or $\omega = \varepsilon$,
$(q, \omega x) \mapsto (r, x)$ if and only if $r \in \delta(q, \omega)$.

The definition permits $\omega \in \Sigma$ or $\omega = \varepsilon$. Thus the move can be either a move on one symbol of input or an ε-transition. Next we define an extended relation $\mapsto^*$ for sequences of zero or more steps.

$\mapsto^*$ is a relation on IDs, with $I \mapsto^* J$ if and only if there is a sequence of zero or more $\mapsto$ relations that starts with I and ends with J.

Notice here that $\mapsto^*$ is reflexive; for any ID I, $I \mapsto^* I$ by a sequence of zero moves. Using the $\mapsto^*$ relation, we can define the δ* function for M:

$\delta^*(q, x) = \{r \mid (q, x) \mapsto^* (r, \varepsilon)\}$

Thus, for any string $x \in \Sigma^*$ and any state $q \in Q$, $\delta^*(q, x)$ is the set of possible states for the NFA to end up in, starting from q and making some sequence of legal δ-transitions on the symbols in x. Using this extended transition function, it is possible to express the idea that an NFA accepts a string if it has at least one path to an accepting state on that string:

A string $x \in \Sigma^*$ is accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ if and only if $\delta^*(q_0, x)$ contains at least one element of F.

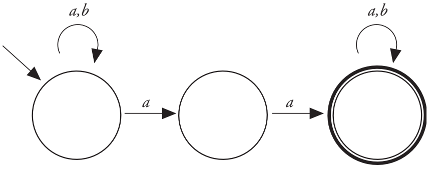
A string x is accepted if and only if the NFA, started in its start state and taken through some sequence of transitions on the symbols in x, has at least one accepting state as a possible final state. We can also define $L(M)$, the language accepted by an NFA M:

For any NFA $M = (Q, \Sigma, \delta, q_0, F)$, $L(M)$ denotes the language accepted by M, which is $L(M) = \{x \in \Sigma^* \mid \delta^*(q_0, x)$ contains at least one element of $F\}$.

Exercises

EXERCISE 1

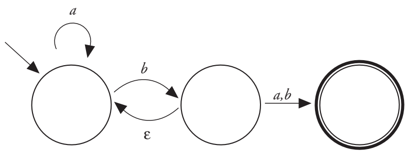
For each of the following strings, say whether it is in the language accepted by this NFA:



a. *aa*
b. *baabb*
c. *aabaab*
d. *ababa*
e. *ababaab*

EXERCISE 2

For each of the following strings, say whether it is in the language accepted by this NFA:

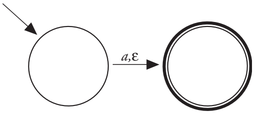


a. *bb*
b. *bba*
c. *bbabb*
d. *aabaab*
e. *ababa*
f. *abbaab*

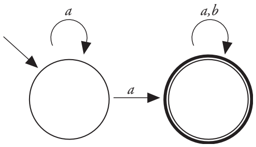
EXERCISE 3

Draw a DFA that accepts the same language as each of the following NFAs. Assume that the alphabet in each case is {*a, b*}; even though not all the NFAs explicitly mention both symbols, your DFAs must.

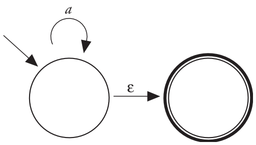
a.



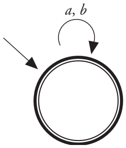
b.



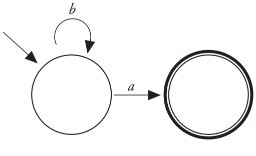
c.



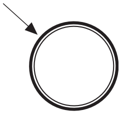
d.



e.



f.



EXERCISE 4

Draw the diagram for an NFA for each of the following languages. Use as few states and as few transitions as possible. Don't just give a DFA, unless you are convinced it is necessary.

a. $\{x \in \{a, b\}^* \mid x$ contains at least 3 *a*s$\}$

b. $\{x \in \{a, b\}^* \mid x$ starts with at least 3 consecutive *a*s$\}$

c. $\{x \in \{a, b\}^* \mid x$ ends with at least 3 consecutive *a*s$\}$

d. $\{x \in \{a, b\}^* \mid x$ contains at least 3 consecutive *a*s$\}$

e. $\{x \in \{a, b\}^* \mid x$ has no two consecutive *a*s$\}$

f. $\{axb \mid x \in \{a, b\}^*\}$

g. $\{x \in \{0, 1\}^* \mid x$ ends in either 0001 or 1000$\}$

h. $\{x \in \{0, 1\}^* \mid x$ either starts with 000 or ends with 000, or both$\}$

EXERCISE 5

Draw an NFA for each of the following languages. *Hint:* Try combining smaller NFAs using ε-transitions.

a. $\{a^n \mid n$ is even$\} \cup \{b^n \mid n$ is odd$\}$

b. $\{(ab)^n\} \cup \{(aba)^n\}$

c. $\{x \in \{a, b\}^* \mid x$ has at least 3 consecutive *a*s at the start or end (or both)$\}$

d. $\{x \in \{a, b\}^* \mid$ the number of *a*s in x is odd$\} \cup \{b^n \mid n$ is odd$\} \cup \{(aba)^n\}$

e. $\{a^n \mid n$ is divisible by at least one of the numbers 2, 3, or 5$\}$

f. $\{xy \in \{a, b\}^* \mid$ the number of *a*s in x is odd and the number of *b*s in y is even$\}$

g. $\{xy \mid x \in \{(ab)^n\}$ and $y \in \{(aba)^m\}\}$

EXERCISE 6

a. Draw a DFA for the set of strings over the alphabet $\{a\}$ whose length is divisible by 3 or 5 (or both).

b. Draw an NFA for the same language, using at most nine states.

EXERCISE 7

Draw the diagram for each of the following NFAs.

a. $(\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_0\})$, where the transition function δ is

$\delta(q_0, 0) = \{\}$
$\delta(q_0, 1) = \{q_1\}$
$\delta(q_0, \varepsilon) = \{\}$
$\delta(q_1, 0) = \{q_1\}$
$\delta(q_1, 1) = \{q_0\}$
$\delta(q_1, \varepsilon) = \{\}$

b. $(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$, where the transition function δ is

$\delta(q_0, 0) = \{q_0, q_1\}$
$\delta(q_0, 1) = \{q_0\}$
$\delta(q_0, \varepsilon) = \{\}$
$\delta(q_1, 0) = \{\}$
$\delta(q_1, 1) = \{q_2\}$
$\delta(q_1, \varepsilon) = \{q_0\}$
$\delta(q_2, 0) = \{\}$
$\delta(q_2, 1) = \{\}$
$\delta(q_2, \varepsilon) = \{\}$

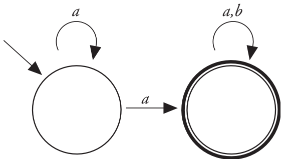
c. $(\{q_0, q_1, q_2\}, \{a, b, c\}, \delta, q_0, \{q_0\})$, where the transition function δ is

$\delta(q_0, a) = \{q_1\}$
$\delta(q_0, b) = \{\}$
$\delta(q_0, c) = \{\}$
$\delta(q_0, \varepsilon) = \{\}$
$\delta(q_1, a) = \{\}$
$\delta(q_1, b) = \{q_2\}$
$\delta(q_1, c) = \{\}$
$\delta(q_1, \varepsilon) = \{\}$
$\delta(q_2, a) = \{\}$
$\delta(q_2, b) = \{\}$
$\delta(q_2, c) = \{q_0\}$
$\delta(q_2, \varepsilon) = \{\}$

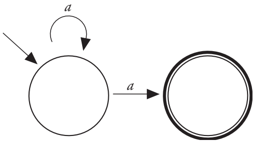
EXERCISE 8

State each of the following NFAs formally, as a 5-tuple. The alphabet in each case should be $\{a, b\}$.

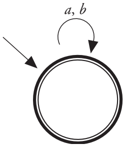
a.



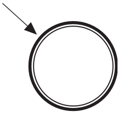
b.



c.



d.

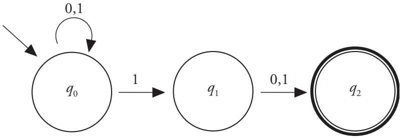


EXERCISE 9

For each machine M in the previous exercise, state what the language $L(M)$ is, using sets and/or set formers.

EXERCISE 10

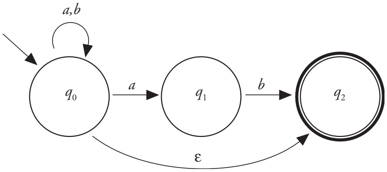
Evaluate each of these expressions for the following NFA:



a. $\delta^*(q_0, 010)$
b. $\delta(q_2, 0)$
c. $\delta^*(q_2, \varepsilon)$
d. $\delta^*(q_0, 1101)$
e. $\delta^*(q_0, 1011)$

EXERCISE 11

Evaluate each of these expressions for the following NFA:



a. $\delta(q_0, a)$
b. $\delta(q_0, \varepsilon)$
c. $\delta^*(q_0, a)$
d. $\delta^*(q_0, ab)$
e. $\delta(q_1, \varepsilon)$
f. $\delta^*(q_1, \varepsilon)$

EXERCISE 12

Construct NFAs with the following properties:

a. an NFA N with $L(N) = \{a\}$, that has exactly one accepting sequence of moves
b. an NFA N with $L(N) = \{a\}$, that has exactly two accepting sequence of moves
c. an NFA N with $L(N) = \{a\}$, that has infinitely many accepting sequences of moves

EXERCISE 13

Prove formally that for any NFA $M = (Q, \Sigma, \delta, q_0, F)$, any strings $x \in \Sigma^*$ and $y \in \Sigma^*$, and any state $q \in Q$,

$$\delta^*(q, xy) = \bigcup_{q' \in \delta^*(q, x)} \delta^*(q', y).$$

Hint: induction is not helpful here. Use the definition of δ^* for NFAs to express the sets in terms of sequences of moves using the $\mapsto^*$ relation.

EXERCISE 14

You're given an integer $i > 0$. Give the 5-tuple for an NFA for the language $\{0^i x \mid x \in \{0, 1\}^*\}$.

EXERCISE 15

You're given a DFA $M = (Q, \Sigma, \delta, q_0, F)$. Show how to construct the 5-tuple for a new NFA N with $L(N) = L(M) \cup \{\varepsilon\}$.

EXERCISE 16

Prove Theorem 3.3 (closure of regular languages under union) again, this time using NFAs in the construction.

EXERCISE 17

Show how to take any given NFA $M = (Q, \Sigma, \delta, q_0, F)$ and construct another NFA $N = (Q', \Sigma, \delta', q_0', F')$, so that $|F'| = 1$ and $L(M) = L(N)$. Be sure to give a convincing proof that the two languages are equal.

EXERCISE 18

Suppose you're given an NFA M that has a single accepting state. Show how to construct the 5-tuple for a new NFA N with $L(N) = \{xy \mid x \in L(M) \text{ and } y \in L(M)\}$. Be sure to give a convincing proof that $L(N)$ is the language specified.

6

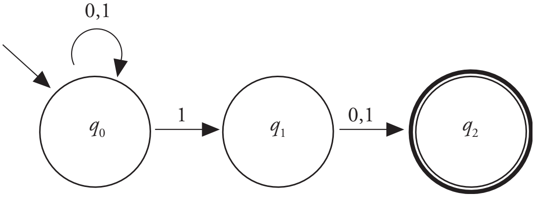
NFA Applications

The problem with implementing NFAs is that, being nondeterministic, they do not really define computational procedures for testing language membership. To implement an NFA we must give a computational procedure that can look at a string and decide whether the NFA has at least one sequence of legal transitions on that string leading to an accepting state. This seems to require searching through all legal sequences for the given input string—but how?

One approach is to implement a direct backtracking search. Another is to convert the NFA into a DFA and implement that instead. This conversion is both useful and theoretically interesting. The fact that it is always possible shows that in spite of their extra flexibility, NFAs have exactly the same power as DFAs; they can define exactly the regular languages.

6.1 An NFA Implemented with a Backtracking Search

Here is an NFA that accepts the language of strings over the alphabet {0, 1} that have a 1 as the next-to-last symbol.



Now let's see how this NFA can be implemented in the Java language. As in Chapter 4, we will write the NFA as a Java class. As in the table-based implementation in Section 4.3, we will use a table to represent the transition function δ. Where δ(*q*, *a*) is the set of zero or more possible next states from the state *q* reading the symbol *a*, in our Java array `delta[q][a-'0']` will be an array of zero or more possible next states. Thus the `delta` array is a three-dimensional array of `int` values.

```
/**
 * A nondeterministic finite-state automaton that
 * recognizes strings of 0s and 1s with 1 as the
 * next-to-last character.
 */
public class NFA1 {

  /*
   * The transition function represented as an array.
   * The entry at delta[s,c-'0'] is an array of 0 or
   * more ints, one for each possible move from
   * the state s on the character c.
   */
  private static int[][][] delta =
     {{{0},{0,1}}, // delta[q0,0], delta[q0,1]
      {{2},{2}},   // delta[q1,0], delta[q1,1]
      {{},{}}};    // delta[q2,0], delta[q2,1]
```

To determine whether the NFA accepts a string, we will do the obvious thing: just search all possible paths through the machine on that string. If we find one that ends in an accepting state, we accept; otherwise we reject. This search is easy to implement using recursion:

```
/**
 * Test whether there is some path for the NFA to
 * reach an accepting state from the given state,
 * reading the given string at the given character
 * position.
 * @param s the current state
 * @param in the input string
 * @param pos index of the next char in the string
 * @return true if the NFA accepts on some path
 */
private static boolean accepts
    (int s, String in, int pos) {
  if (pos==in.length()) { // if no more to read
    return (s==2); // accept if final state is q2
  }

  char c = in.charAt(pos++); // get char and advance
  int[] nextStates;
  try {
    nextStates = delta[s][c-'0'];
  }
  catch (ArrayIndexOutOfBoundsException ex) {
    return false; // no transition, just reject
  }

  // At this point, nextStates is an array of 0 or
  // more next states.  Try each move recursively;
  // if it leads to an accepting state return true.

  for (int i=0; i < nextStates.length; i++) {
    if (accepts(nextStates[i], in, pos)) return true;
  }

  return false; // all moves fail, return false
}
```

The code uses some of the same tricks we saw in Chapter 4. In particular, it uses exception handling to catch out-of-bounds array references, so that we do not need the array to have entries for `char` values other than `'0'` and `'1'`. The new trick here is the recursive search. A call to `accepts(s,in,pos)` tests whether an accepting state is reachable, starting from the state `s`, reading the string `in` from the position `pos`. In the base case, `pos` is equal to `in.length`, so the entire string has already been read; in that case the NFA accepts only if `s` is the accepting state, state 2. In the recursive case, for each possible next move, the code makes a recursive call to test whether an accepting state is reachable from there.

Finally, we need a `public` method that clients can call to test whether a string is in the language accepted by the NFA. We will use the name `accepts` for this method too, since Java permits this kind of overloading. `accepts(in)` just starts up the recursive search by making an initial call to `accepts(0,in,0)`—starting from the state 0 (the start state) and the character position 0 (the first character in the string).

```
  /**
   * Test whether the NFA accepts the string.
   * @param in the String to test
   * @return true if the NFA accepts on some path
   */
  public static boolean accepts(String in) {
    return accepts(0, in, 0); // start in q0 at char 0
  }
}
```

That is the end of the class definition. To test whether a string `s` is in the language defined by this NFA, we would write something like

```
    if (NFA1.accepts(s)) ...
```

This implementation is not particularly object-oriented, since the whole thing uses static methods and fields and no objects of the `NFA1` class are created. All the information manipulated in the recursive search is carried in the parameters, so no objects were needed.

6.2 An NFA Implemented with a Bit-Mapped Parallel Search

The implementation just given uses a backtracking search. It tries one sequence of moves; if that fails it backs up and tries another, and so on until it finds a successful sequence or runs out of sequences to try. In effect, it tries sequences one at a time. Another approach to NFA implementation is to search all paths simultaneously.

This is not as difficult as it sounds. We will implement this like the table-driven DFA of Chapter 4, except that instead of keeping track of the single state the machine is in after each character, we will keep track of the set of possible states after each character.

```
/**
 * A nondeterministic finite-state automaton that
 * recognizes strings with 0 as the next-to-last
 * character.
 */
public class NFA2 {

  /*
   * The current set of states, encoded bitwise:
   * state i is represented by the bit 1<<i.
   */
  private int stateSet;
```

We use a bit-coded integer to represent a set of states. The << operator in Java shifts an integer to the left, so `1<<0` is 1 shifted left 0 positions (thus `1<<0` is 1), `1<<1` is 1 shifted left 1 position (thus `1<<1` is 2), `1<<2` is 1 shifted left 2 positions (thus `1<<2` is 4), and so on. In general, an NFA implemented this way will use the number `1<<i` to represent the `i` state. For this NFA, there are only the three states `1<<0`, `1<<1`, and `1<<2`. Since we are using one bit for each state, we can form any set of states just by combining the state bits with logical OR. For example, the set $\{q_0, q_2\}$ can be represented as `1<<0|1<<2`.

The initial state set is $\{q_0\}$. This method initializes the current state set to that value:

```
  /**
   * Reset the current state set to {the start state}.
   */
  public void reset() {
    stateSet = 1<<0; // {q0}
  }
```

We will use the table-driven approach for the transition function, just as we did for DFAs. In this case, however, each entry in the table is not just a single state, but a bit-coded set of states:

```
  /*
   * The transition function represented as an array.
   * The set of next states from a given state s and
   * character c is at delta[s,c-'0'].
   */
  static private int[][] delta =
  {{1<<0, 1<<0|1<<1}, // delta[q0,0] = {q0}
                      // delta[q0,1] = {q0,q1}
   {1<<2, 1<<2}, // delta[q1,0] = {q2}
                 // delta[q1,1] = {q2}
   {0, 0}}; // delta[q2,0] = {}
            // delta[q2,1] = {}
```

The `process` method looks at characters in the input string and keeps track of the set of possible states after each one. This is the only part that is significantly more complicated than the DFA case, because for each character in the input string we have to execute a little loop. To compute the next set of states given a symbol *c*, we need to take the union of the sets δ(*q*, *c*) for each state *q* in the current set of states. The `process` method does that in its inner loop; its outer loop repeats the computation once for each symbol in the input string.

```
/**
 * Make one transition from state-set to state-set on
 * each char in the given string.
 * @param in the String to use
 */
public void process(String in) {
  for (int i = 0; i < in.length(); i++) {
    char c = in.charAt(i);
    int nextSS = 0; // next state set, initially empty
    for (int s = 0; s <= 2; s++) { // for each state s
      if ((stateSet&(1<<s)) != 0) { // if maybe in s
        try {
          nextSS |= delta[s][c-'0'];
        }
        catch (ArrayIndexOutOfBoundsException ex) {
          // in effect, nextSS |= 0
        }
      }
    }
    stateSet = nextSS; // new state set after c
  }
}
```

All that remains is a method to test whether the NFA should accept. It accepts if the final set of possible states includes the accepting state q_2.

```
  /**
   * Test whether the NFA accepted the string.
   * @return true if the final set includes
   *         an accepting state.
   */
  public boolean accepted() {
    return (stateSet&(1<<2))!=0; // true if q2 in set
  }
}
```

That is the end of the class definition. To test whether a string `s` is in the language defined by this NFA, we would write something like

```
NFA2 m = new NFA2();
m.reset();
m.process(s);
if (m.accepted()) ...
```

The interface to this is largely the same as for the DFAs we developed in Chapter 4.

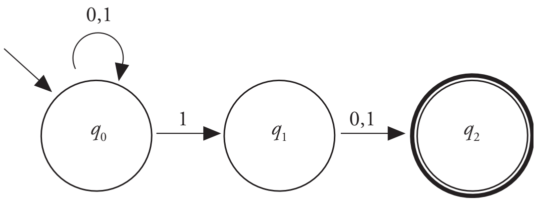
Because the Java `int` type is a 32-bit word, our `NFA2` code generalizes simply for any NFA of up to 32 states. For NFAs of up to 64 states, the 64-bit `long` type could be used easily. To implement an NFA with more than 64 states in Java would require some additional programming. One could, for example, use an array of $\lceil n/32 \rceil$ `int` variables to represent the set of n states. The `process` loop would be considerably more complicated, and slower, for such large NFAs.

As the `NFA2` implementation runs, the `stateSet` field keeps track of the set of states that the NFA could be in. This basic idea—keeping track of the set of possible states after each symbol is read—is the basis for a construction that lets us convert any NFA into a DFA.

6.3 The Subset Construction

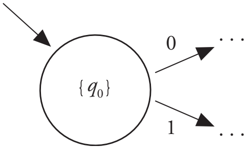
For any NFA, there is a DFA that recognizes the same language. Thus, while NFAs can be simpler to write than DFAs and can use fewer states and transitions, they are not any more powerful than DFAs. The power of nondeterminism does not make it possible for finite-state machines to define any additional languages. This can be proved by showing how to take any NFA and construct a DFA that accepts the same language.

Before giving the general method of constructing a DFA from any NFA, let's look at an example. We begin with our NFA for $\{x \in \{0, 1\}^* \mid \text{the next-to-last symbol in } x \text{ is } 1\}$:

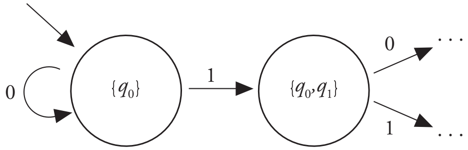


We will construct a DFA for the same language. The new DFA will in fact simulate the NFA, by keeping track of the states the NFA might be in after each symbol of the input string. Being (potentially) nondeterministic, the NFA can have more than one legal sequence of steps for an input, and so can be in more than one possible state after each symbol of the input string. So each state of the DFA must correspond to some subset of the set of states from the NFA. That is why this is called the *subset construction*.

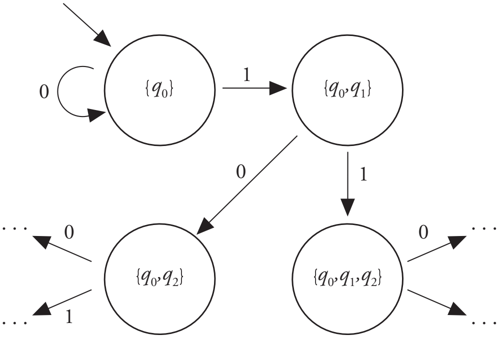
Initially, of course, the NFA is in its start state. We will create a start state for the DFA and label it with the set of states $\{q_0\}$:



Next, suppose the NFA is in q_0 and reads a 0. What is the set of possible states after that? It is $\delta(q_0, 0) = \{q_0\}$ again, so the transition on 0 in our DFA should return to the state labeled $\{q_0\}$. If the NFA is in q_0 and reads a 1, its set of possible next states is $\delta(q_0, 1) = \{q_0, q_1\}$, so we will add a new state labeled $\{q_0, q_1\}$ to the DFA under construction. So far we have

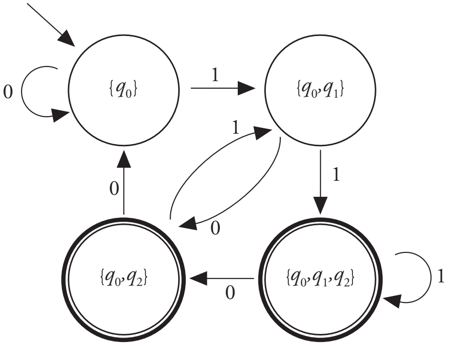


Next, suppose the NFA is in one of the states $\{q_0, q_1\}$ and reads a 0. What is the set of possible states after that? We have $\delta(q_0, 0) = \{q_0\}$ and $\delta(q_1, 0) = \{q_2\}$, so the next set of possible states is $\delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_2\}$. Similarly, if the NFA is in one of the states $\{q_0, q_1\}$ and reads a 1, its set of possible next states is $\delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0, q_1, q_2\}$. We add these two new states to the DFA under construction, producing



Each step in the construction follows this same pattern. Every state of the DFA is labeled with a set of states from the NFA. The transition on a symbol *a* from a state labeled *R* in the DFA is determined by taking the union, over all $r \in R$, of the NFA's $\delta(r, a)$. (This is exactly what the inner loop of the `process` method in our `NFA2` implementation computes.) Because there are only finitely many different subsets of the set of states in the NFA; this construction eventually stops with a finite DFA. In our example, in fact, we have already found all the reachable subsets of states; the remaining transitions all return to sets already added to the DFA.

The only remaining problem is to decide which of the states of the DFA should be accepting states. Since the NFA will accept whenever its set of possible final states contains at least one accepting state, that is what the DFA should do too. All the states labeled with subsets containing at least one accepting state of the NFA should be accepting states of the DFA. (This is exactly what the `accepted` method of our `NFA2` implementation does.) The final result is this:



If you now discard the state labels, you can see that this is the same as the DFA given in Section 5.1 for the original language, $\{x \in \{0, 1\}^* \mid \text{the next-to-last symbol in } x \text{ is } 1\}$.

The construction just illustrated is the subset construction. The following proof expresses it formally:

Lemma 6.1: If $L = L(N)$ for some NFA N, then L is a regular language.

Proof: Suppose $L = L(N)$ for some NFA $N = (Q_N, \Sigma, \delta_N, q_N, F_N)$. Construct a new DFA $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$, where

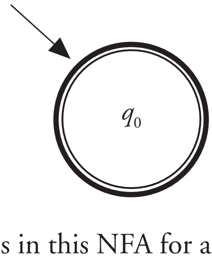
$$\begin{aligned} Q_D &= P(Q_N) \\ \delta_D(R, a) &= \bigcup_{r \in R} \delta^*_N(r, a), \text{ for all } R \in Q_D \text{ and } a \in \Sigma \\ q_D &= \delta^*_N(q_N, \varepsilon) \\ F_D &= \{\mathrm{R} \in Q_D \mid \mathrm{R} \cap F_N \neq \{\}\} \end{aligned}$$

Now by construction we have $\delta^*_D(q_D, x) = q_D = \delta^*_N(q_N, x)$ for all $x \in \Sigma^*$. Thus D simulates N's behavior on each input string x, accepting if and only if N accepts. We conclude that $L = L(N) = L(D)$, and it follows that L is a regular language.

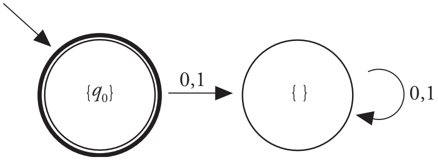
The jump from the definition of δ_D to the conclusion that $\delta^*_D(q_D, x) = q_D = \delta^*_N(q_N, x)$ can be made more rigorous using a routine inductive proof of the kind demonstrated in Section 3.4. Notice that in the base case, $\delta^*_D(q_D, \varepsilon) = q_D = \delta^*_N(q_N, \varepsilon)$. In the construction, the start state for the new DFA is $q_D = \delta^*_N(q_N, \varepsilon)$. Often this start state will be simply the set $q_D = \{q_N\}$, as it was in our earlier example of the construction. But the start state of the DFA will contain additional states from the NFA if the NFA has ε-transitions from its start state.

The machine constructed for the proof includes all the states in $P(Q_N)$, which may include some unreachable states. (We saw something similar in Section 3.2 with the product construction.) In the case of our earlier example, this will be an eight-state DFA instead of the four-state DFA. The extra states are not reachable by any path from the start state, so they have no effect on the language accepted.

The construction always includes a DFA state for {}, the empty set of NFA states, because the empty set is a subset of every set. The DFA will be in this state on any string for which the NFA has no legal sequence of moves—any string x for which $\delta^*_N(q_N, x) = \{\}$. For example, consider this NFA, which accepts only the empty string:



There is no legal sequence of moves in this NFA for any string other than the empty string. The set of states $Q_N = \{q_0\}$, and the powerset of that is $Q_D = P(Q_N) = \{\{\}, \{q_0\}\}$. The DFA produced by the construction will therefore be this two-state machine:



No matter what the original NFA is, the construction always gives

$$\delta_D(\{\}, a) = \bigcup_{r \in \{\}} \delta^*_N(r, a) = \{\}$$

so the {} state in the DFA produced by the subset construction always has transitions back to itself for all symbols in the alphabet. In this way, the subset construction automatically provides {} as a nonaccepting trap state. This trap state will be reachable from the start state if and only if the original NFA can get stuck—that is, if and only if there is some input string x for which $\delta^*_N(q_{N,} x) = \{\}$.

We have seen two different approaches for implementing NFAs. The subset construction gives us a third: we can first convert the NFA into a DFA using the subset construction, then implement that using the techniques of Chapter 4.

6.4 NFAs Are Exactly as Powerful as DFAs

The subset construction shows that NFAs are no more powerful than DFAs. It is also true, more obviously, that NFAs are no *less* powerful than DFAs.

Lemma 6.2: If L is any regular language, there is some NFA N for which $L(N) = L$.

Proof: Let L be any regular language. By definition there must be some DFA $D = (Q, \Sigma, \delta_D, q_0, F)$ with $L(D) = L$. This DFA immediately gives an equivalent NFA $N = (Q, \Sigma, \delta_N, q_0, F)$, where we define $\delta_N(q, a) = \{\delta_D(q, a)\}$ for all $q \in Q$ and $a \in \Sigma$, and $\delta_N(q, \varepsilon) = \{\}$ for all $q \in Q$. Clearly, we have $\delta^*_N(q, x) = \{\delta^*_D(q, x)\}$ for all $q \in Q$ and $x \in \Sigma^*$, and so N accepts x if and only if D accepts x. It follows that $L(N) = L(D) = L$.

Although in diagrams it is clear that every DFA is an NFA, it is not quite true with formal 5-tuples, since the δ functions of DFAs and NFAs have different types. However, this difference is trivial, as the proof above illustrates; wherever a DFA has $\delta_D(q, a) = r$, an equivalent NFA should have $\delta_N(q, a) = \{r\}$. The jump in the proof from $\delta_N(q, a) = \{\delta_D(q, a)\}$ to $\delta^*_N(q, x) = \{\delta^*_D(q, x)\}$ can be made more rigorous using a routine inductive proof of the kind demonstrated in Section 3.4.

From the previous two lemmas it follows that the set of languages that can be defined using NFAs is exactly the set of regular languages.

Theorem 6.1: A language L is $L(N)$ for some NFA N if and only if L is a regular language.

Proof: It follows immediately from Lemmas 6.1 and 6.2.

We conclude that allowing nondeterminism in finite automata can make them more compact and easier to construct, but in the sense shown in this theorem, it neither weakens nor strengthens them.

6.5 DFA or NFA?

We have seen a number of techniques for recognizing regular languages using DFA or NFA ideas. No one technique is best in all circumstances.

The big advantage of DFA techniques is their speed. They run quickly and predictably, taking time that is simply proportional to the length of the string. One disadvantage of DFA techniques is that they are difficult to extend for nonregular constructs; NFAs are better for such applications, as discussed below. Another potential disadvantage is that the size of the DFA code (including the size of the transition array, if used) is proportional to the number of states in the DFA. This is usually not a problem, but there are languages for which the smallest DFA is exponentially larger than the smallest NFA. The subset construction gives some intuition about how that can happen: for an NFA with n states, the subset construction yields a DFA with 2^n states and can still have exponentially many states left over, even after the unreachable ones are eliminated. Compilers are an example of an application for which these trade-offs favor DFAs. The first phase of most compilers is a *scanner* that breaks the source file into tokens like keywords, identifiers, and constants. These token languages are purely regular, the DFAs tend to be of modest size, and speed is critical, so DFA techniques work well here.

NFA-based techniques using a backtracking search are widely used for other applications. They tend to be slower, but they have their own advantages. For one thing, they do not suffer from the problem of having exponential-size worst cases. In practice, though, this is not as important as their major advantage, which is that they are easier to extend for nonregular-language constructs. Many programs that use regular expressions need all the power of regular languages and then some. This includes text-processing tools like emacs, vi, and sed, languages like Perl and PHP, and language libraries like those for Java and the .NET languages. In a later chapter we discuss more about regular expressions and their applications. For now, suffice it to say that backtracking, NFA-based implementations are more easily extended to handle the nonregular constructs required by these applications.

There are also many variations and many possible optimizations. Some systems use a DFA implementation for purely regular tasks and an NFA only for the nonregular tasks.

Some systems use hybrid techniques that combine the DFA and NFA approaches, like the bit-mapped parallel search, augmented with optimizations like the caching of frequently used state sets and transitions.

6.6 Further Reading

Automata are implemented in an amazing variety of applications. For a look at the state of the art in this active research area, see the proceedings of the annual Conference on Implementation and Application of Automata. As of this writing, the most recent one was

Antipolis, Sophia. "Implementation and Application of Automata, 10th International Conference (CIAA 2005)." Published as *Lecture Notes in Computer Science*, vol. 3845. Paris: Springer-Verlag, 2005.

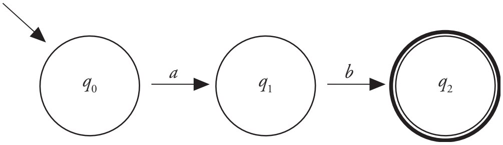
Exercises

EXERCISE 1

The DFA constructed for the example in Section 6.3 has four states, but the full DFA given by the formal subset construction has eight. Draw the full eight-state DFA, including the unreachable states.

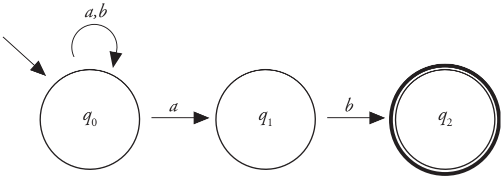
EXERCISE 2

Convert this NFA into a DFA using the subset construction. Draw the DFA and label each state with the corresponding subset of states from the NFA. You do not have to show unreachable states, if any.



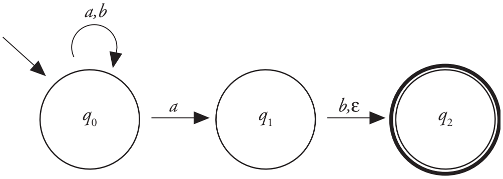
EXERCISE 3

Convert this NFA into a DFA using the subset construction. Draw the DFA and label each state with the corresponding subset of states from the NFA. You do not have to show unreachable states, if any.



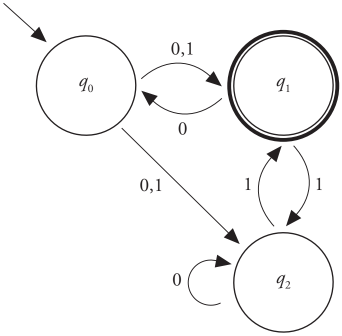
EXERCISE 4

Convert this NFA into a DFA using the subset construction. Draw the DFA and label each state with the corresponding subset of states from the NFA. You do not have to show unreachable states, if any.



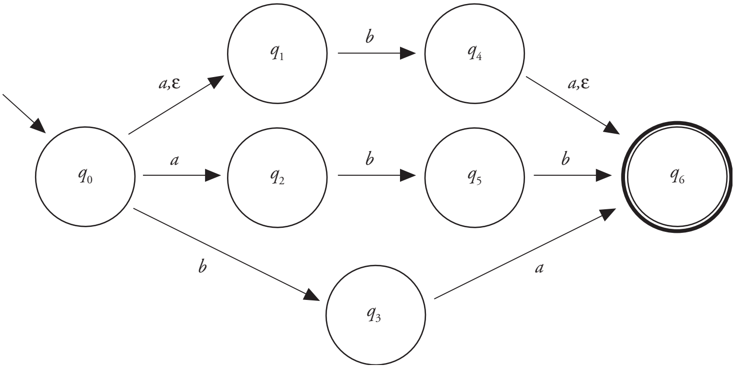
EXERCISE 5

Convert this NFA into a DFA using the subset construction. Draw the DFA and label each state with the corresponding subset of states from the NFA. You do not have to show unreachable states, if any.



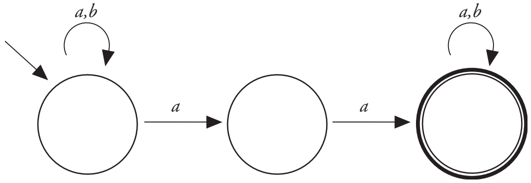
EXERCISE 6

Convert this NFA into a DFA using the subset construction. Draw the DFA and label each state with the corresponding subset of states from the NFA. You do not have to show unreachable states, if any.



EXERCISE 7

Write a Java class that implements this NFA:



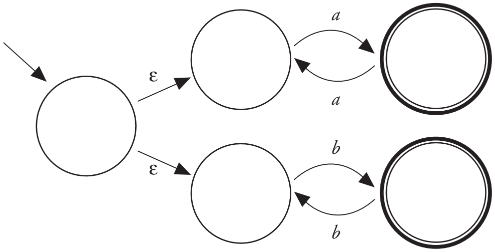
Use a table-driven, backtracking-search approach as in Section 6.1.

EXERCISE 8

Repeat the previous exercise, but use a table-driven, parallel-search approach as in Section 6.2.

EXERCISE 9

Write a Java class that implements this NFA:



Use a table-driven, backtracking-search approach as in Section 6.1. You will need to consider how best to modify this approach to incorporate ε-transitions.

EXERCISE 10

Repeat the previous exercise, but use a table-driven, parallel-search approach as in Section 6.2. Again, you will need to consider how best to modify the approach to incorporate ε-transitions.

EXERCISE 11

Rewrite the proof of Lemma 6.2, but include a rigorous inductive proof to show that $\delta^*_N(q, x) = \{\delta^*_D(q, x)\}$.

EXERCISE 12

Rewrite the proof of Lemma 6.1, but include a rigorous inductive proof to show that $\delta^*_D(q_D, x) = \delta^*_N(q_N, x)$.

7

Regular Expressions

Most programmers and other power-users of computer systems have used tools that match text patterns. You may have used a Web search engine with a pattern like `travel cancun OR acapulco`*, trying to find information about either of two travel destinations. You may have used a search tool with a pattern like* `gr[ea]y`*, trying to find files with occurrences of either* `grey` *or* `gray`*. Any such pattern for text works as a definition of a formal language. Some strings match the pattern and others do not; the language defined by the pattern is the set of strings that match. In this chapter we study a particular kind of formal text pattern called a* **regular expression**. *Regular expressions have many applications. They also provide an unexpected affirmation of the importance of some of the theoretical concepts from previous chapters.*

This is a common occurrence in mathematics. The first time a young student sees the mathematical constant π*, it looks like just one more school artifact: one more arbitrary symbol whose definition to memorize for the next test. Later, if he or she persists, this perception changes. In many branches of mathematics and with many practical applications,* π *keeps on turning up. "There it is again!" says the student, thus joining the ranks of mathematicians for whom mathematics seems less like an artifact invented and more like a natural phenomenon discovered.*

So it is with regular languages. We have seen that DFAs and NFAs have equal definitional power. It turns out that regular expressions also have exactly that same definitional power: they can be used to define all the regular languages and only the regular languages. There it is again!

7.1 Regular Expressions, Formally Defined

Let's define two more basic ways of combining languages. The concatenation of two languages is the set of strings that can be formed by concatenating an element of the first language with an element of the second language:

If L_1 and L_2 are languages, the concatenation of L_1 and L_2 is $L_1L_2 = \{xy \mid x \in L_1$ and $y \in L_2\}$.

The Kleene closure of a language is the set of strings that can be formed by concatenating any number of strings, each of which is an element of that language:

If L is a language, the Kleene closure of L is $L^* = \{x_1x_2 \ldots x_n \mid n \geq 0$, with all $x_i \in L\}$.

Note that this generalizes the definition we used in Chapter 1 for the Kleene closure of an alphabet.

There is a common mistake to guard against as you read the Kleene closure definition. It does *not* say that L^* is the set of strings that are concatenations of zero or more copies of some string in L. That would be $\{x^n \mid n \geq 0$, with $x \in L\}$; compare that with the actual definition above. The actual definition says that L^* is the set of strings that are concatenations of zero or more substrings, each of which is in the language L. Each of those zero or more substrings may be a different element of L. For example, the language $\{ab, cd\}^*$ is the language of strings that are concatenations of zero of more things, each of which is either *ab* or *cd*: $\{\varepsilon$, *ab*, *cd*, *abab*, *abcd*, *cdab*, *cdcd*, …$\}$. Note that because the definition allows *zero or more*, L^* always includes ε.

Now we can define regular expressions.

A regular expression is a string r that denotes a language $L(r)$ over some alphabet Σ. The six kinds of regular expressions and the languages they denote are as follows. First, there are three kinds of *atomic* regular expressions:

1. Any symbol $a \in \Sigma$ is a regular expression with $L(a) = \{a\}$.
2. The special symbol ε is a regular expression with $L(\varepsilon) = \{\varepsilon\}$.
3. The special symbol ∅ is a regular expression with $L(\emptyset) = \{\}$.

There are also three kinds of *compound* regular expressions, which are built from smaller regular expressions, here called r, r_1, and r_2:

4. $(r_1 + r_2)$ is a regular expression with $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
5. (r_1r_2) is a regular expression with $L(r_1r_2) = L(r_1)L(r_2)$
6. $(r)^*$ is a regular expression with $L((r)^*) = (L(r))^*$

The parentheses in compound regular expressions may be omitted, in which case * has highest precedence and + has lowest precedence.

Regular expressions make special use of the symbols ε, ∅, +, *, (, and), so for simplicity we will assume that these special symbols are not included in Σ. Regular expressions can look just like ordinary strings, so you will occasionally have to rely on the context to decide whether a string such as *abc* is just a string or a regular expression denoting the language {*abc*}.

Unfortunately, the term *regular expression* is heavily overloaded. It is used for the text patterns of many different tools like awk, sed, and grep, many languages like Perl, Python, Ruby, and PHP, and many language libraries like those for Java and the .NET languages. Each of these applications means something slightly different by *regular expression*, and we will see more about these applications later.

7.2 Examples of Regular Expressions

Using just the six basic kinds of regular expressions, you can define quite a wide variety of languages. For starters, any simple string denotes the language containing just that string:

Regular expression: ab
Language denoted: {ab}

Notice that the formal definition permits this because *a* and *b*, by themselves, are atomic regular expressions of type 1; they are combined into a compound regular expression of type 5; and the parenthesis in (*ab*) are omitted since they are unnecessary.

Any finite language can be defined, just by listing all the strings in the language, separated by the + operator:

Regular expression: *ab* + *c*
Language denoted: {*ab, c*}

Again we were able to omit all the parentheses from the fully parenthesized form ((*ab*) + *c*). The inner pair is unnecessary because + has lower precedence than concatenation. Thus, *ab* + *c* is equivalent to (*ab*) + *c*, not to *a*(*b* + *c*), which would denote a different language: {*ab*, *ac*}.

The only way to define an infinite language using regular expressions is with the Kleene star:

Regular expression: *ba**
Language denoted: $\{ba^n\}$ = {*b*, *ba, baa, baaa*, ...}

The Kleene star has higher precedence than concatenation. *ba** is equivalent to *b*(*a**), not to (*ba*)*, which would denote the language $\{(ba)^n\}$.

L((*r*) *) is the language of strings that are concatenations of zero or more substrings, each of which is in the language *L*(*r*). Because this is *zero or more* and not *one or more*, we always have $\varepsilon \in L((r)^*)$. Each of the concatenated substrings can be a different element of $L(r)$. For example:

Regular expression: $(a + b)^*$
Language denoted: $\{a, b\}^*$

In that example the parentheses were necessary, since * has higher precedence than +. The regular expression $a + b^*$ would denote the language $\{a\} \cup \{b^n\}$.

It is important to understand that $(a + b)^*$ is not the same as $(a^* + b^*)$. Any confusion about this point will cause big problems as we progress. In $L((a + b)^*)$ we concatenate zero or more things together, each of which may be either an a or a b. That process can construct every string over the alphabet $\{a, b\}$. In $L(a^* + b^*)$ we take the union of two sets: the set of all strings over the alphabet $\{a\}$, and the set of all strings over the alphabet $\{b\}$. That result contains all strings of only *a*s and all strings of only *b*s, but does not contain any strings that contain both *a*s and *b*s.

Occasionally the special symbol ε appears in a regular expression, when the empty string is to be included in the language:

Regular expression: $ab + \varepsilon$
Language denoted: $\{ab, \varepsilon\}$

The special symbol ∅ appears very rarely. Without it there is no other way to denote the empty set, but that is all it is good for. It is not useful in compound regular expressions, since for any regular expression r we have $L((r)\varnothing) = L(\varnothing(r)) = \{\}$, $L(r + \varnothing) = L(\varnothing + r) = L(r)$, and $L(\varnothing^*) = \{\varepsilon\}$.

The subexpressions in a compound expression may be compound expressions themselves. This way, compound expressions of arbitrary complexity may be developed:

Regular expression: $(a + b)(c + d)$
Language denoted: $\{ac, ad, bc, bd\}$

Regular expression: $(abc)^*$
Language denoted: $\{(abc)^n\} = \{\varepsilon, abc, abcabc, abcabcabc, \ldots\}$

Regular expression: a^*b^*
Language denoted: $\{a^nb^m\} = \{\varepsilon, a, b, aa, ab, bb, aaa, aab, \ldots\}$

Regular expression: $(a + b)^*aa(a + b)^*$
Language denoted: $\{x \in \{a, b\}^* \mid x \text{ contains at least two consecutive } a\text{s}\}$

Regular expression: $(a + b)^*a(a + b)^*a(a + b)^*$
Language denoted: $\{x \in \{a, b\}^* \mid x \text{ contains at least two } a\text{s}\}$

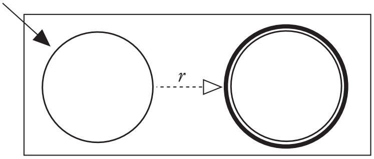
Regular expression: $(a^*b^*)^*$
Language denoted: $\{a, b\}^*$

In this last example, the regular expression $(a^*b^*)^*$ turns out to denote the same language as the simpler $(a + b)^*$. To see why, consider that $L(a^*b^*)$ contains both a and b. Those two symbols alone are enough for the Kleene star to build all of $\{a, b\}^*$. In general, whenever $\Sigma \subseteq L(r)$, then $L((r)^*) = \Sigma^*$.

7.3 For Every Regular Expression, a Regular Language

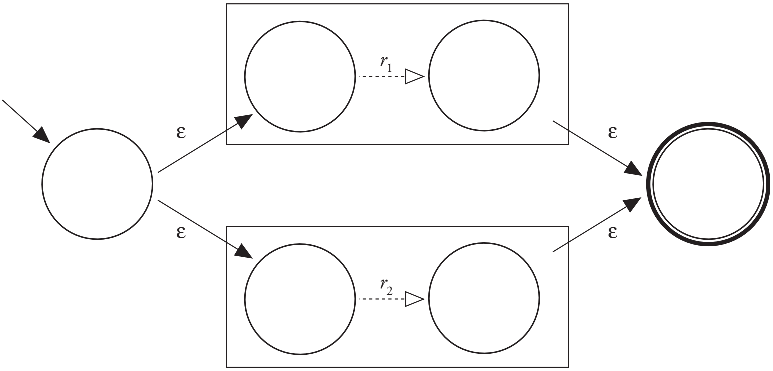
We will now demonstrate that any language defined by a regular expression can be defined by both of the other mechanisms already studied in this book: DFAs and NFAs. To prove this, we will show how to take any regular expression and construct an equivalent NFA. We choose NFAs for this, rather than DFAs, because of a property observed in Section 5.2: large NFAs can be built up as combinations of smaller ones by the judicious use of ε-transitions.

To make the constructed NFAs easy to combine with each other, we will make sure they all have exactly one accepting state, not the same as the start state. For any regular expression r we will show how to construct an NFA N with $L(N) = L(r)$ that can be pictured like this:



The constructed machine will have a start state, a single accepting state, and a collection of other states and transitions that ensure that there is a path from the start state to the accepting state if and only if the input string is in $L(r)$.

Because all our NFAs will have this form, they can be combined with each other very neatly. For example, if you have NFAs for $L(r_1)$ and $L(r_2)$, you can easily construct an NFA for $L(r_1 + r_2)$:

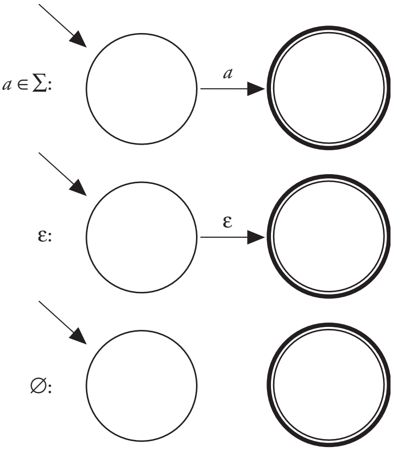


In this new machine there is a new start state with ε-transitions to the two original start states and a new accepting state with ε-transitions from the two original accepting states (which are no longer accepting states in the new machine). Clearly the new machine has a path from the start state to the accepting state if and only if the input string is in $L(r_1 + r_2)$. And it has the special form—a single accepting state, not the same as the start state—which ensures that it can be used as a building block in even larger machines.

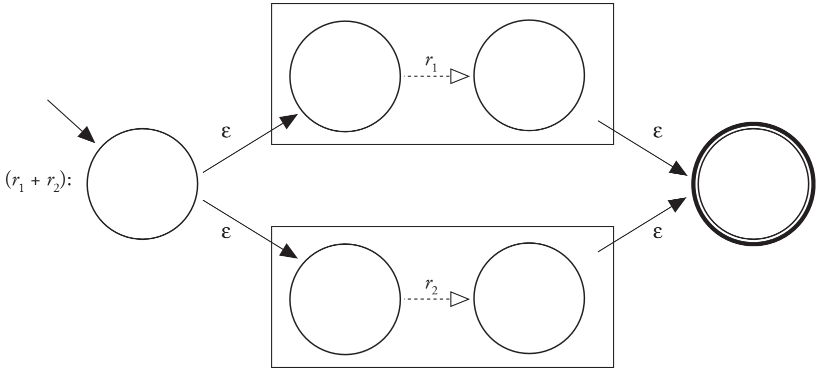
The previous example shows how to do the construction for one kind of regular expression—$r_1 + r_2$. The basic idea of the following proof sketch is to show that the same construction can be done for all six kinds of regular expressions.

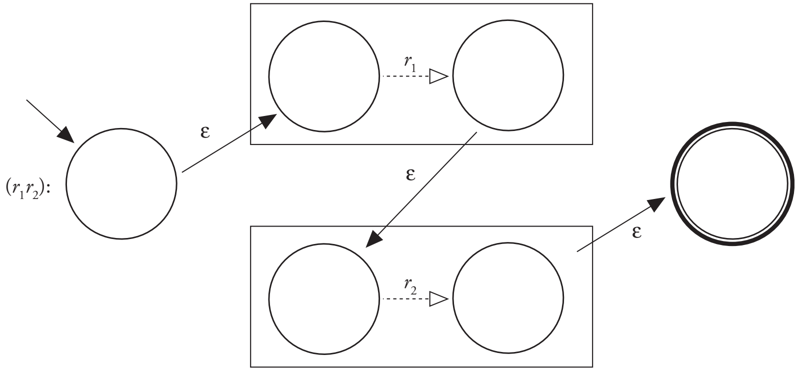
Lemma 7.1: If r is any regular expression, there is some NFA N that has a single accepting state, not the same as the start state, with $L(N) = L(r)$.

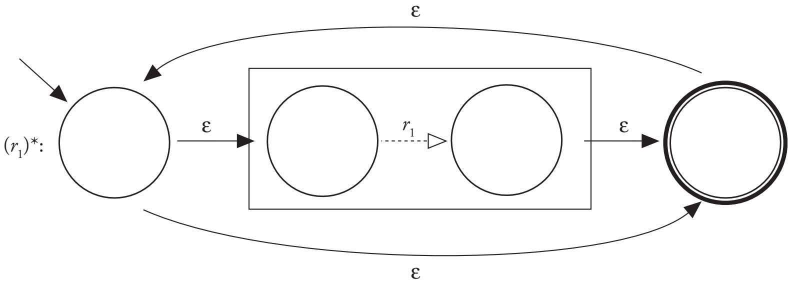
Proof sketch: For any of the three kinds of atomic regular expressions, an NFA of the desired kind can be constructed as follows:



For any of the three kinds of compound regular expressions, given appropriate NFAs for regular subexpressions r_1 and r_2, an NFA of the desired kind can be constructed as follows:







Thus, for any regular expression *r*, we can construct an equivalent NFA with a single accepting state, not the same as the start state.

7.4 Regular Expressions and Structural Induction

The proof sketch for Lemma 7.1 leaves out a number of details. A more rigorous proof would give the 5-tuple form for each of the six constructions illustrated and show that each machine actually accepts the language it is supposed to accept. More significantly, a rigorous proof would be organized as a *structural induction*.

As we have observed, there are as many different ways to prove something with induction as there are to program something with recursion. Our proofs in Chapter 3 concerning DFAs used induction on a natural number—the length of the input string. That is the style of induction that works most naturally for DFAs. Structural induction performs induction on a recursively defined structure and is the style of induction that works most naturally for regular expressions. The base cases are the atomic regular expressions, and the inductive cases are the compound forms. The inductive hypothesis is the assumption that the proof has been done for structurally simpler cases; for a compound regular expression r, the inductive hypothesis is the assumption that the proof has been done for r's subexpressions.

Using structural induction, a more formal proof of Lemma 7.1 would be organized like this:

Proof: By induction on the structure of r.

Base cases: When r is an atomic expression, it has one of these three forms:

1. Any symbol $a \in \Sigma$ is a regular expression with $L(a) = \{a\}$.
2. The special symbol ε is a regular expression with $L(\varepsilon) = \{\varepsilon\}$.
3. The special symbol $\varnothing$ is a regular expression with $L(\varnothing) = \{\}$.

(For each atomic form you would give the NFA, as in the previous proof sketch.)

Inductive cases: When r is a compound expression, it has one of these three forms:

4. $(r_1 + r_2)$ is a regular expression with $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
5. (r_1r_2) is a regular expression with $L(r_1r_2) = \{xy \mid x \in L(r_1) \text{ and } y \in L(r_2)\}$
6. $(r_1)^*$ is a regular expression with
$$L((r_1)^*) = \{x_1x_2 \ldots x_n \mid n \geq 0, \text{ with all } x_i \in L(r_1)\}$$

By the inductive hypothesis, there is an NFA that has a single accepting state, not the same as the start state, for r_1 and $r_{2.}$

(For each compound form you would give the ε-NFA, as in the previous proof sketch, and show that it accepts the desired language.)

A proof using this style of induction demonstrates that Lemma 7.1 holds for any atomic regular expression and then shows that whenever it holds for some expressions r_1 and r_2, it also holds for $(r_1 + r_2)$, (r_1r_2), and $(r_1)^*$. It follows that the lemma holds for all regular expressions.

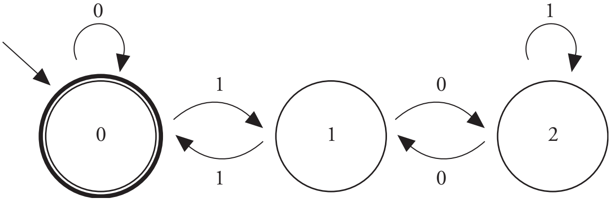
7.5 For Every Regular Language, a Regular Expression

There is a way to take any NFA and construct an equivalent regular expression.

> **Lemma 7.2:** If N is any NFA, there is some regular expression r with $L(N) = L(r)$.

The construction that proves this lemma is rather tricky, so it is relegated to Appendix A. For now, a short example will suffice.

Consider again this NFA (which is also a DFA) from Chapter 3:



We proved that this machine accepts the language of strings that are binary representations of numbers that are divisible by three. Now we will construct a regular expression for this same language. It looks like a hard problem; the trick is to break it into easy pieces.

When you see an NFA, you normally think only of the language of strings that take it from the start state to end in any accepting state. But let's consider some other languages that are relevant to this NFA. For example, what is the language of strings that take the machine from state 2 back to state 2, any number of times, *without passing through states 0 or 1*? Any string of zero or more 1s would do it, and that is an easy language to give a regular expression for:

$$1^*$$

Next, a bigger piece: what is the language of strings that take the machine from 1 back to 1, any number of times, *without passing through state 0*? The machine has no transition that allows it to go from 1 directly back to 1. So each single trip from 1 back to 1 must follow these steps:

1. Go from 1 to 2.
2. Go from 2 back to 2, any number of times, without passing through 0 or 1.
3. Go from 2 to 1.

The first step requires a 0 in the input string. The second step is a piece we already have a regular expression for: 1*. The third step requires a 0 in the input string. So the whole language, the language of strings that make the machine do those three steps repeated any number of times, is

(01*0)*

Next, a bigger piece: what is the language of strings that take the machine from 0 back to 0, any number of times? There are two ways to make a single trip from 0 back to 0. The machine can make the direct transition from state 0 to state 0 on an input symbol 0 or it can follow these steps:

1. Go from 0 to 1.
2. Go from 1 back to 1, any number of times, without passing through 0.
3. Go from 1 to 0.

The first step requires a 1 in the input string. The second step is a piece we already have a regular expression for: (01*0)*. The third step requires a 1 in the input string. So the whole language, the language of strings that make the machine go from state 0 back to state 0 any number of times, is

(0 + 1(01*0)*1)*

This also defines the whole language of strings accepted by the NFA—the language of strings that are binary representations of numbers that are divisible by three.

The proof of Lemma 7.2 is a construction that builds a regular expression for the language accepted by any NFA. It works something like the example just shown, defining the language in terms of smaller languages that correspond to restricted paths through the NFA. Appendix A provides the full construction. But before we get lost in those details, let's put all these results together:

Theorem 7.1 (Kleene's Theorem): A language is regular if and only if it is *L*(*r*) for some regular expression *r*.

Proof: Follows immediately from Lemmas 7.1 and 7.2.

DFAs, NFAs, and regular expressions all have equal power for defining languages.

Exercises

EXERCISE 1

Give a regular expression for each of the following languages.

a. $\{abc\}$
b. $\{abc, xyz\}$
c. $\{a, b, c\}^*$
d. $\{ax \mid x \in \{a, b\}^*\}$
e. $\{axb \mid x \in \{a, b\}^*\}$
f. $\{(ab)^n\}$
g. $\{x \in \{a, b\}^* \mid x$ contains at least three consecutive as$\}$
h. $\{x \in \{a, b\}^* \mid$ the substring bab occurs somewhere in $x\}$
i. $\{x \in \{a, b\}^* \mid x$ starts with at least three consecutive as$\}$
j. $\{x \in \{a, b\}^* \mid x$ ends with at least three consecutive as$\}$
k. $\{x \in \{a, b\}^* \mid x$ contains at least three as$\}$
l. $\{x \in \{0, 1\}^* \mid x$ ends in either 0001 or 1000$\}$
m. $\{x \in \{0, 1\}^* \mid x$ either starts with 000 or ends with 000, or both$\}$
n. $\{a^n \mid n$ is even$\} \cup \{b^n \mid n$ is odd$\}$
o. $\{(ab)^n\} \cup \{(aba)^n\}$
p. $\{x \in \{a\}^* \mid$ the number of as in x is odd$\}$
q. $\{x \in \{a, b\}^* \mid$ the number of as in x is odd$\}$
r. $\{x \in \{a, b\}^* \mid$ the number of as in x is odd$\} \cup \{b^n \mid n$ is odd$\} \cup \{(aba)^n\}$
s. $\{a^n \mid n$ is divisible by at least one of the numbers 2, 3, or 5$\}$
t. $\{x \in \{a, b\}^* \mid x$ has no two consecutive as$\}$
u. $\{xy \in \{a, b\}^* \mid$ the number of as in x is odd and the number of bs in y is even$\}$

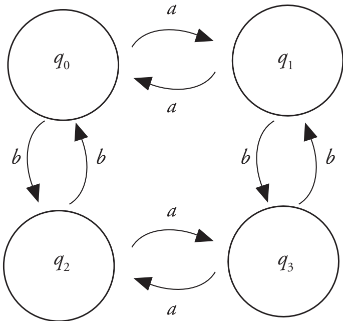
EXERCISE 2

For each of these regular expressions, give two NFAs: the exact one constructed by the proof of Lemma 7.1, and the smallest one you can think of.

a. $\varnothing$
b. ε
c. a
d. $0 + 1$
e. $(00)^*$
f. ab^*

EXERCISE 3

For the following DFA, give a regular expression for each of the languages indicated. When the question refers to a machine "passing through" a given state, that means entering and then exiting the state. Merely starting in a state or ending in it does not count as "passing through."



a. the language of strings that make the machine, if started in q_0, end in q_0, *without passing through* q_0, q_1, q_2, or q_3

b. the language of strings that make the machine, if started in q_0, end in q_2, *without passing through* q_0, q_1, q_2, or q_3

c. the language of strings that make the machine, if started in q_2, end in q_0, *without passing through* q_0, q_1, q_2, or q_3

d. the language of strings that make the machine, if started in q_2, end in q_2, *without passing through* q_0 or q_1

e. the language of strings that make the machine, if started in q_2, end in q_0, *without passing through* q_0 or q_1

f. the language of strings that make the machine, if started in q_0, end in q_2, *without passing through* q_0 or q_1

g. the language of strings that make the machine, if started in q_1, end in q_1, *without passing through* q_0 or q_1

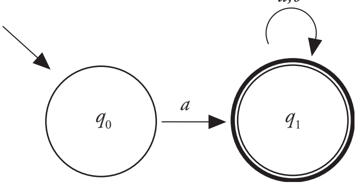
h. the language of strings that make the machine, if started in q_1, end in q_2, *without passing through* q_0 or q_1

i. the language of strings that make the machine, if started in q_0, end in q_0, *without passing through* q_0 or q_1

j. the language of strings that make the machine, if started in q_1, end in q_1, *without passing through* q_0

EXERCISE 4

(This exercise refers to material from Appendix A.) Let *M* be this NFA:

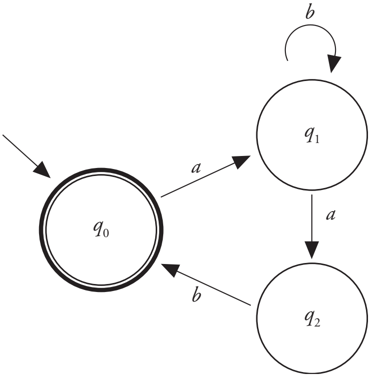


Give the simplest regular expression you can think of for each of the following internal languages of M.

a. $L(M, 0, 0, 2)$
b. $L(M, 0, 0, 1)$
c. $L(M, 0, 0, 0)$
d. $L(M, 0, 1, 2)$
e. $L(M, 0, 1, 1)$
f. $L(M, 0, 1, 0)$
g. $L(M, 1, 1, 2)$
h. $L(M, 1, 1, 1)$
i. $L(M, 1, 1, 0)$
j. $L(M, 1, 0, 2)$
j. $L(M, 1, 0, 1)$
l. $L(M, 1, 0, 0)$

EXERCISE 5

(This exercise refers to material from Appendix A.) Let M be this NFA:

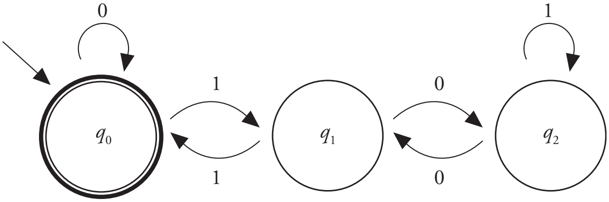


Give the simplest regular expression you can think of for each of the following internal languages of M. (Use the d2r construction if necessary.)

a. $L(M,2,2,3)$
b. $L(M,2,2,2)$
c. $L(M,1,1,3)$
d. $L(M,1,1,2)$
e. $L(M,1,1,1)$
f. $L(M,0,0,3)$
g. $L(M,0,0,2)$
h. $L(M,0,0,1)$
i. $L(M,0,0,0)$

EXERCISE 6

(This exercise refers to material from Appendix A.) Let *M* be this DFA:



Give the simplest regular expression you can think of for each of the following internal languages of *M*. (Use the d2r construction if necessary.)

a. $L(M, 2, 2, 3)$
b. $L(M, 2, 2, 2)$
c. $L(M, 1, 1, 3)$
d. $L(M, 1, 1, 2)$
e. $L(M, 1, 1, 1)$
f. $L(M, 0, 0, 3)$
g. $L(M, 0, 0, 2)$
h. $L(M, 0, 0, 1)$
i. $L(M, 0, 0, 0)$

EXERCISE 7

(This exercise refers to material from Appendix A.) The following code is used to verify the regular expression computed by d2r for the example in Section A.3:

```
public static void main(String args[]) {
    NFA N = new NFA();
    State q0 = N.getStartState();
    State q1 = N.addState();
    N.addX(q0,'a',q0);
    N.addX(q0,'b',q1);
    N.addX(q1,'b',q1);
    N.addX(q1,'a',q0);
    N.makeAccepting(q1);
    System.out.println("final:" + N.d2r(0,1,0));
}
```

This code makes use of two classes, `State` and `NFA`. The `NFA` class represents an NFA. When an `NFA` object is constructed, it initially contains only one state, the start state, with no accepting states and no transitions. Additional states are added dynamically by calling the `addState` method, which returns an object of the `State` class. Each transition on a symbol is added dynamically by calling the `addX` method, giving the source state, the symbol, and the target state. States are labeled as accepting by calling the `makeAccepting` method. Finally, the d2r construction is performed by calling the `d2r` method, passing it the integers *i*, *j*, and *k*. (States are considered to be numbered in the order they were added to the `NFA`, with the start state at number 0.) The `d2r` method returns a `String`. The code shown above creates an `NFA` object, initializes it as the example NFA, and prints out the regular expression for the language it accepts.

Implement the `State` and `NFA` classes according to this specification, along with any auxiliary classes you need. You need not implement ε-transitions in the NFA, and you may represent ε in regular expressions using the symbol `e`. Test your classes with the test code shown above, and check that your `d2r` returns the correct regular expression. (Recall that, for readability, some of the parentheses in the regular expression in Section A.3 were omitted. Your result may include more parentheses than shown there.)

EXERCISE 8

For any string x, define x^R to be that same string reversed. The intuition is plain enough, but to support proofs we'll need a formal definition:

$\varepsilon^R = \varepsilon$
$(ax)^R = x^R a$, for any symbol a and string x

We'll use the same notation to denote the set formed by reversing every string in a language:

$A^R = \{x^R \mid x \in A\}$

Using these definitions, prove the following properties of reversal:

a. Prove that for any strings x and y, $(xy)^R = y^R x^R$. *Hint:* Use induction on $|x|$.
b. Prove that for any string x, $(x^R)^R = x$. *Hint:* Use induction on $|x|$ and the result from Part a.
c. Prove that for any languages A and B, $(A \cup B)^R = A^R \cup B^R$.
d. Prove that for any languages A and B, $(AB)^R = B^R A^R$. You'll need the result from Part a.
e. Prove that for any language A, $(A^*)^R = (A^R)^*$. You'll need the result from Part a.
f. Prove that the regular languages are closed for reversal. *Hint:* Using structural induction, show that for every regular expression r, there is a regular expression r' with $L(r') = (L(r))^R$.

8

Regular Expression Applications

We have seen some of the implementation techniques related to DFAs and NFAs. These important techniques are like tricks of the programmer's trade, normally hidden from the end user. Not so with regular expressions; they are often visible to the end user and are part of the user interface of a variety of useful software tools.

8.1 The `egrep` Tool

The Unix tool egrep searches a file for lines that contain a substring matching a specified pattern and echoes all such lines to the standard output. For example, if a file named `names` contains these lines:

```
fred
barney
wilma
betty
```

then this command after the `%` prompt searches the file for lines containing an `a`:

```
% egrep 'a' names
barney
wilma
%
```

In that example, egrep searched for lines containing a simple constant substring, but the language of patterns understood by egrep can do much more. Various dialects of the patterns understood by egrep are also used by many other tools. Unfortunately, these patterns are often simply called *regular expressions*, though both in syntax and meaning they are a bit different from the regular expressions we studied in the last chapter. To keep the two ideas separate, this book refers to the patterns used by egrep and other tools using their common nickname: *regexps*. Some of the special characters used in egrep's regexp dialect are

- `*` This symbol is like our Kleene star. For any regexp *x*, *x*`*` matches strings that are concatenations of zero or more strings from the language specified by *x*.
- `|` This symbol is like our +. For any regexps *x* and *y*, *x*`|`*y* matches strings that match either *x* or *y* (or both).
- `()` These symbols are used for grouping.
- `^` When this special symbol is at the start of the regexp, it allows the regexp to match only at the start of the line.
- `$` When this special symbol is at the end of the regexp, it allows the regexp to match only at the end of the line.
- `.` This symbol matches any symbol (except the end-of-line marker).

For example, the regexp `a.*y` matches strings consisting of an `a`, followed by zero or more other characters, followed by a `y`. We can search the `names` file for any line containing such a string:

```
% egrep 'a.*y' names
barney
%
```

The regexp `.(..)*` matches strings consisting of an odd number of characters. Searching the file `names` with this pattern produces this result:

```
% egrep '.(..)*' names
fred
barney
wilma
betty
%
```

Why did the search match all the lines, even those with an even number of characters? It did this because egrep searches the file for all lines that *contain a substring* matching the specified pattern—and any line with one or more characters contains a substring with an odd number of characters. To match only odd-length lines, we must use the special symbols `^` and `$`:

```
% egrep '^.(..)*$' names
wilma
betty
%
```

One final example: Chapter 4 demonstrated a DFA-based, Java implementation of a filter program that echoes those lines of an input file that are binary representations of numbers that are divisible by three. In Section 7.5 a regular expression for this language was derived: (0 + 1(01*0)*1)*. Using the corresponding regexp, egrep can do the same thing our Java filter program did. If the file `numbers` contains the numbers zero through ten in binary:

```
0
1
10
11
100
101
110
111
```

```
1000
1001
1010
```

then this egrep command selects those that are divisible by three:

```
% egrep '^(0|1(01*0)*1)*$' numbers
0
11
110
1001
%
```

8.2 Nonregular Regexps

In some ways regexps have evolved beyond the formal regular expressions from which they developed. Many regexp dialects can define more than just the regular languages.

For example, special parentheses in some regexp dialects can do more than just group subexpressions—they can capture the text that was matched with that subexpression so that it can be referred to later. The grep utility (but not most versions of egrep) uses `\(` and `\)` as capturing parentheses and uses `\`*n* as a regexp that matches the same text captured by the *n*th previous capturing left parenthesis. Thus the regexp `^\(.*\)\1$` matches any line that consists of a string followed immediately by a repetition of that same string. If the file named `test` contains these lines:

```
abaaba
ababa
abbbabbb
abbaabb
```

then this grep command selects those lines that consist of repeated strings:

```
% grep '^\(.*\)\1$' test
abaaba
abbbabbb
%
```

The formal language corresponding to that example is $\{xx \mid x \in \Sigma^*\}$. This is not something that can be defined with plain regular expressions. A useful intuition about regular expressions is that, like DFAs, they can do only what you could implement on a computer using a fixed, finite amount of memory. Capturing parentheses clearly go beyond that limit, since they must capture a string whose size is unbounded. To implement `\(.*\)\1`, you would have to first store the string matched by `.*` somewhere, then test for a match with `\1` by comparing against that stored string. Since the stored string can be arbitrarily large, this cannot be done using a fixed, finite memory. That is, of course, just an informal argument. We will prove formally in later chapters that $\{xx \mid x \in \Sigma^*\}$ is not a regular language and no mere regular expression can define it.

8.3 Implementing Regexps

There are many other tools that use regexp dialects: text-processing tools like emacs, vi, and sed; compiler construction tools like lex; and programming languages like Perl, Ruby, Python, PHP, Java, and the .NET languages. How do these tools implement regexp matching? At a high level, the answer is simple: they convert the regexp into an NFA or DFA, then simulate that automaton on each string to be tested. This roughly follows the constructions already given in this book. Some systems convert the regexp into an NFA, and then use that NFA to test each string for a match. As we've seen, an NFA is more expensive than a DFA to simulate, but it is easier to extend for nonregular constructs like capturing parentheses. Other systems convert the regexp into an NFA and then convert that to a DFA. As we've seen, a DFA can be implemented more efficiently, but it is harder to extend beyond the regular languages and may also be much larger than the corresponding NFA. Of course, many optimizations and implementation tricks are possible; this book has described the constructions without giving much consideration to efficiency.

The usual operation of a DFA makes a sequence of moves reading the entire input string. Acceptance of the string is judged only at the end, after the whole string has been read. Regexp matching tools generally do something a little different: they search a string for a *substring* that matches the given regexp. This can still be done with a DFA, but the implementation details are slightly different. Imagine starting a DFA at the first character of the string. If it reaches a nonaccepting trap state or hits the end of the string without ever reaching an accepting state, that indicates that no substring starting at the first position matches. In that case the DFA must run again starting from the second position, then the third, and so on, checking each position in the string for a matching substring.

If the DFA ever enters an accepting state—even if the end of the string has not been reached—that indicates that a matching substring has been found. Since we started at the left end of the string and worked forward, it is a leftmost match. There is still a problem, however, since there may be more than one match starting at the same place. For example, the string `abb` contains three substrings that match the regexp `ab*`—`a`, `ab`, and `abb`—and they all start at the same place. Should the implementation report as soon as that first accepting state is reached (after the `a`), or should it continue to operate and enter accepting states again (after the first `b` and after the second `b`)? Different tools treat this situation differently, but many tools (like lex) are required to find the *longest* of the leftmost matches first, which would be the string abb. In that case, the DFA-based implementation must continue processing the string, always remembering the last accepting state that was entered and the position in the string that went with it. Then, as soon as the DFA enters a nonaccepting trap state or encounters the end of the string, it can report the longest leftmost match that was found.

Similar accommodations must be made by NFA-based automata. In particular, when an implementation using backtracking finds a match, it cannot necessarily stop there. If the longest match is required, it must remember the match and continue, exploring all paths through the NFA to make sure that the longest match is found.

8.4 Regular Expressions in Java

There is a predefined package in Java, java.util.regex, that lets you work with regular expressions. Here's a simple example of how it works:

```
import java.io.*;
import java.util.regex.*;

/**
 * A Java application to demonstrate the Java package
 * java.util.regex. We take one command-line argument,
 * which is treated as a regexp and compiled into a
 * Pattern. We then use that Pattern to filter the
 * standard input, echoing to standard output only
 * those lines that match the Pattern.
 */
class RegexFilter {
  public static void main(String[] args)
        throws IOException {

    Pattern p = Pattern.compile(args[0]); // the regexp
    BufferedReader in =  // standard input
      new BufferedReader(new InputStreamReader(System.in));

    // read and echo lines until EOF

    String s = in.readLine();
    while (s!=null) {
      Matcher m = p.matcher(s);
```

```
          if (m.matches()) System.out.println(s);
          s = in.readLine();
        }
      }
    }
```

This application is structured like Mod3Filter from Chapter 4. But now, instead of using a fixed DFA to test each line, it takes a parameter from the command line, treats it as a regexp, and uses it to construct a Pattern object. A Pattern is something like a representation of an NFA: it is a compiled version of the regexp, ready to be given an input string to test. In the main loop, the application tests each input line by creating a Matcher, an object that represents the NFA of the Pattern along with a particular input string and current state. A Matcher object can do many things, such as finding individual matches within a string and reporting their locations. But here, we just use it to test the entire string to see whether the string matches the pattern. Using this application, we could do our divisible-by-three filtering with this command:

```
% java RegexFilter '^(0|1(01*0)*1)*$' < numbers
```

The regex package also provides a single, static method that combines all those steps, so you don't have to keep track of separate Pattern and Matcher objects. To test whether a String s is in the language defined by a regexp String r, you can just evaluate Pattern.matches(r,s). This is easier for the programmer and makes sense if you are going to use the regexp only once. But if (as in our example above) you need to use the same regexp repeatedly, the first technique is more efficient; it compiles the regexp into a Pattern that can be used repeatedly.

Scripting languages—Perl, Python, PHP, Tcl, Ruby, JavaScript, VBScript, and so on—often have extra support for programming with regular expressions. In Perl, for example, operations involving regular expressions don't look like method calls; regular expressions are used so often that the language provides a special-purpose syntax that is much more compact. The Perl expression $x =~ m/a*b/ evaluates to true if and only if the string variable $x matches the regexp a*b.

8.5 The lex Tool

Tools like egrep convert a regexp into an automaton, simulate the automaton, and then discard it. That makes sense for applications that are typically run with a different and fairly short regexp each time. But other applications need to use the same regexp every time they run. Compilers, for example, often use a collection of regexps to define the tokens—keywords, constants, punctuation, and so on—in the programming language to be compiled.

These regexps do not change from one compilation to the next, and it would not make sense to perform the conversion from regexp to automaton every time the compiler is run. Instead, the automaton can be a fixed, preconstructed part of the compiler. Tools like lex help with this; they convert regexps into high-level-language code that can be used as a fixed part of any application.

The lex tool converts a collection of regexps into C code for a DFA. The input file to lex has three sections: a definition section, a rules section, and a section of user subroutines. Lex processes this input file and produces an output file containing DFA-based C code, which can be a stand-alone application or can be used as part of a larger program. The three sections are separated by lines consisting of two percent signs, so a lex input file looks like this:

definition section
`%%`
rules section
`%%`
user subroutines

The definition section can include a variety of preliminary definitions, but for the simple examples in this book this section is empty. The user subroutines section can include C code, which is copied verbatim to the end of the lex output file. This can be used to define C functions used from inside the rules section, but for the simple examples in this book this section too is empty. Thus, in our examples, the lex input file has only the rules section, like this:

`%%`
rules section
`%%`

The rules section is a list of regexps. Each regexp is followed by some C code, which is to be executed whenever a match is found for the regexp. For example, this lex program prints the line `Found one.` for each occurrence of the string `abc` in the input file and ignores all other characters:

```
%%
abc    {fprintf(yyout, "Found one.\n");}
.|\n   {}
%%
```

The lex program above contains two regexps. The first is abc; whenever that is found, the C code {fprintf(yyout, "Found one.\n");} is executed. Here, fprintf is C's standard input/output function for printing to a file, and yyout is the output file currently being used by the lex-generated code. So the program generated by lex reads its input and, whenever it sees the string abc, prints the line Found one.

Code generated by lex copies any unmatched characters to the output. Our lex program avoids this default behavior by including a second regexp, .|\n, which matches every character. (The regexp . matches every character except the end-of-line marker, and the regexp \n matches the end-of-line marker.) The C code associated with the second regexp is the empty statement, meaning that no action is taken when this match is found. The lex-generated code finds the longest match it can, so it won't match a single character a using the second rule if it is the start of an abc that matches the first rule.

If the lex program above is stored in a file named abc.l, we can build a program from it with the following Unix commands:

```
% flex abc.l
% gcc lex.yy.c -o abc –ll
%
```

The first command uses flex (the Gnu implementation of lex) to compile the lex program into DFA-based C code, which is stored in a file named lex.yy.c. The second command then runs the C compiler on the flex-generated code; the –o abc tells it to put the executable program in a file named abc, and –ll tells it to link with the special library for lex. To test the resulting program, we make a file named abctest that contains these lines:

```
abc
aabbcc
abcabc
```

Now we run abc giving it abctest as input:

```
% abc < abctest
Found one.
Found one.
Found one.
%
```

It finds the three instances of the substring abc.

For one last example, let's again implement the divisible-by-three filter, this time using lex. This lex program uses the same regexp we used in the previous section with egrep. Any line matching that regexp is echoed to the output; any unmatched characters are ignored. (The lex variable `yytext` gives us a way to access the substring that matched the regexp.)

```
%%
^(0|1(01*0)*1)*$    {fprintf(yyout, "%s\n", yytext);}
.|\n                {}
%%
```

This lex program can be compiled into a filter program, just like the `Mod3Filter` we implemented in Java:

```
% flex mod3.l
% gcc lex.yy.c -o mod3 -ll
%
```

The result is a directly executable file that filters its input file, echoing the divisible-by-three lines to the standard output. Here, as before, the file `numbers` contains the binary representations of the numbers zero through ten.

```
% mod3 < numbers
0
11
110
1001
%
```

For simple applications like those above, the code produced by lex can be compiled as a standalone program. For compilers and other large applications, the code produced by lex is used as one of many source files from which the full application is compiled.

8.6 Further Reading

For more information about lex, see

Levine, John R., Tony Mason, and Doug Brown. *lex & yacc*. Sebastopol, CA: O'Reilly & Associates, 1992.

For more information about regexp tools in general, an excellent resource is

Friedl, Jeffrey E. F. *Mastering Regular Expressions*. Sebastopol, CA: O'Reilly & Associates, 2002.

Exercises

EXERCISE 1

Show an egrep command that reads the standard input and echoes only those lines that are in the language $L(01^*0)$.

EXERCISE 2

Show an egrep command that reads the standard input and echoes only those lines over the alphabet $\{a, b\}$ that have an odd number of *a*s.

EXERCISE 3

Show an egrep command that reads the standard input and echoes only those lines that are binary representations of numbers that are divisible by four.

EXERCISE 4

Show an egrep command that reads the standard input and echoes only those lines that contain somewhere a decimal number in this form: one or more decimal digits, followed by a decimal point, followed by one or more additional digits. You will need to find more information about egrep's regexp syntax to solve this problem correctly, and still more to solve it elegantly.

EXERCISE 5

The lexical structure of Java programs includes the following three elements:

- Single-line comments, using `//`, as in these examples:

```
int x = 1; // NOT starting from 0 here
// s = "Hello";
```

- Literals of the `String` type, such as

```
"hello"
"Comments start with /*"
"He said, \"Hello.\""
```

- Literals of the `char` type, such as

```
'a'
'\n'
'\''
'\34'
```

If you are not a Java expert, you will need to refer to Java documentation for precise definitions of these three elements. You'll also need to learn more about egrep's regexp syntax to solve these problems.

a. Show an egrep command that reads the standard input and echoes only those lines that contain single-line comments. You may assume that no `String` or `char` literals and no block comments using `/*` and `*/` are present.

b. Show an egrep command that reads the standard input and echoes only those lines that contain valid String literals. You may assume that no comments or char literals are present.

c. Show an egrep command that reads the standard input and echoes only those lines that contain valid char literals. You may assume that no comments or String literals are present.

EXERCISE 6

The lexical structure of Java programs includes the following four elements:

- Single-line comments, as described in the previous exercise.
- Literals of the String type, as described in the previous exercise.
- Literals of the char type, as described in the previous exercise.
- Traditional comments using /* and */, as in these examples:

```
/* fix
   this */
for (int i = 1; /* fix this */ i<100; i++)
/* hi /* // "hello" */
```

If you are not a Java expert, you will need to refer to Java documentation for precise definitions of these elements. You'll also need to learn more about the Java regexp syntax and about the methods of the Matcher and Pattern classes. *One hint:* A common beginner's mistake when using the regex package is to forget to double up the backslashes. Because a regexp is expressed as a Java String literal, a regexp like \s (the character class for white space) must be written in the string as "\\s".

a. Write a Java application that reads a Java source file from standard input. It examines the file one line at a time, identifies single-line comments by using the regex package, and echoes only those comments (not necessarily the whole line) to standard output. You may assume that none of the other elements described above is present.

b. Write a Java application that reads a Java source file from standard input. It examines the file one line at a time, identifies String literals by using the regex package, and echoes only those literals to standard output, one on each line of output. (Note that the input file may contain more than one String literal on a line; you must find them all.) You may assume that none of the other elements described above is present. Your program need not be well behaved if the input file contains errors (such as unclosed String literals).

c. Write a Java application that reads a Java source file from standard input. It examines the file one line at a time, identifies char literals by using the regex package, and echoes only those literals to standard output, one on each line of output. (Note that the input file may contain more than one `char` literal on a line; you must find them all.) You may assume that none of the other elements described above is present. Your program need not be well behaved if the input file contains errors (such as unclosed `char` literals).

d. Write a Java application that takes the name of a Java source file as a command-line argument. It reads the entire file into a string, identifies traditional comments by using the `regex` package, and echoes only those comments to standard output. You may assume that none of the other elements described above is present. Note that these may be multiline comments, so it is not sufficient to process each input line in isolation. Your program need not be well behaved if the input file contains errors (such as unclosed traditional comments).

e. Write a Java application that takes the name of a Java source file as a command-line argument. It reads the entire file into a string, identifies all four elements described above by using the `regex` package, and echoes only them to standard output. Your output should identify which of the four parts each element is. For example, if the file `Test.java` contains this text:

```
char /* type is char */ p = '\''; // p is the quote
String s = "\"It's nice to be here,\" he said.";
/* Commented out:
   String x = "Normal comments start with /*"; */
String w = "Single-line comments start with //";
```

the result should look like this:

```
% java AllFour Test.java
Traditional comment: /* type is char */
Char literal: '\''
Single-line comment: // p is the quote
String literal: "\"It's nice to be here,\" he said."
Traditional comment: /* Commented out:
   String x = "Normal comments start with /*"; */
String literal: "Single-line comments start with //"
```

Your program need not be well behaved if the input file contains errors. *Hint:* it may sound difficult to handle comments that contain strings, strings that contain comments, and so on, but it isn't. With minor modifications, you can simply combine the four regexps from the previous four parts into one regexp. If your final regexp has four sets of capturing parentheses, one for each of the four elements, you'll be able to identify which of the four was responsible for each match by using the `group` method of the `Matcher` class.

EXERCISE 7

Write a Java application that examines a text file, looking for words that occur twice in a row, separated by white space. (Define a "word" to be a string of one or more letters, upper- or lowercase; define "white space" as in the Java regexp `\s` character class.) Your application should take the file name as input on the command line. For each duplicated word found, your application should print the word, the line number (counting the first line as line 1), and the character position of the start of the first of the two copies (counting the first character as character 1).

For example, if the file `speech.txt` contains this:

```
Four score and seven seven years ago our fathers
brought forth on on this continent, a new nation,
conceived in Liberty, and dedicated to the
the proposition that all men are created equal.
```

then the result should look like this:

```
% java Doubles speech.txt
Line 1, char 16: seven
Line 2, char 15: on
Line 3, char 40: the
```

In your application, read the entire file into a string, then use the `regex` package to find the duplicated words. (As the example shows, you must catch pairs that are split from one line to the next.) You'll need to learn more about the Java regexp syntax and about the methods of the `Matcher` and `Pattern` classes. Be careful that your application does not find partial word pairs, like "ago our" in the example above or "equal equality."

9

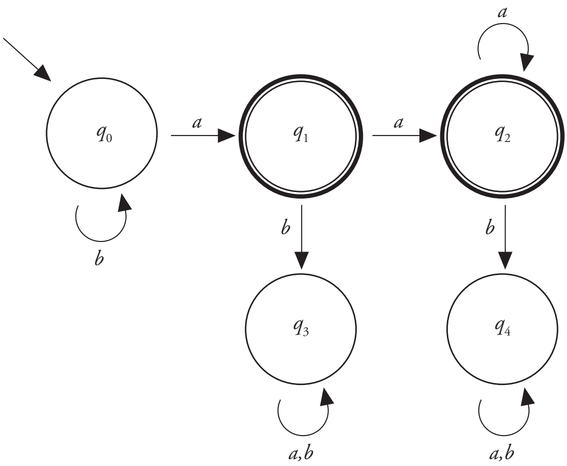
Advanced Topics in Regular Languages

There are many more things to learn about finite automata than are covered in this book. There are many variations with interesting applications, and there is a large body of theory. Especially interesting, but beyond the scope of this book, are the various algebras that arise around finite automata. This chapter gives just a taste of some of these advanced topics.

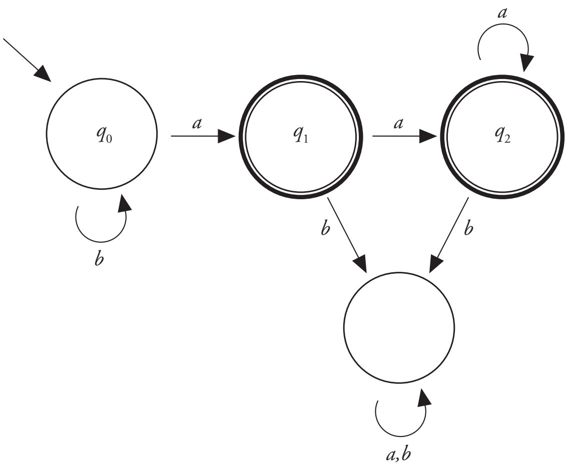
9.1 DFA Minimization

Given a DFA, can we find one with fewer states that accepts the same language? Can we find the smallest DFA for a given language? Is the smallest DFA for a given language unique, or might there be more than one? These are important questions for efficiency as well as elegance.

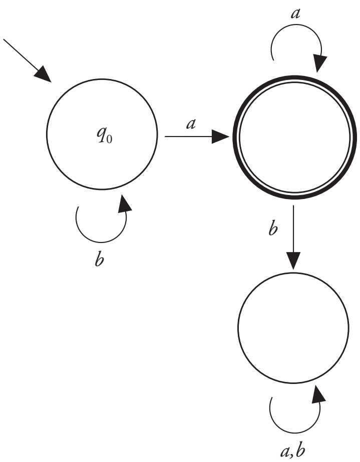
Unreachable states, like some of those introduced by the subset construction, can obviously be eliminated without affecting the language accepted. Even the reachable states of a DFA can sometimes be combined or eliminated. For example, consider this DFA:



This machine has two "trap" states, q_3 and q_4. When in state q_3, the machine will ultimately reject the string, no matter what the unread part contains. State q_4 has exactly the same property, so it is equivalent to q_3 in an important sense and can be merged with q_3 without changing the language accepted by the machine. So then we have



The states q_1 and q_2 are equivalent in that same sense. When in state q_2, the machine will accept if and only if the rest of the string consists of zero or more *a*s. State q_1 has exactly the same property, so it is equivalent to q_2 and can be merged with it. We then have



The result is a minimum-state DFA for the language $\{xay \mid x \in \{b\}^* \text{ and } y \in \{a\}^*\}$.

That example used an important idea about the equivalence of two states. Informally, we said that two states were equivalent when the machine's future decision, after any remaining input, was going to be the same from either state. Formally, let's define a little language $L(M, q)$ for each state q, which is the language that would be accepted by M if q were used as the start state:

$$L(M, q) = \{x \in \Sigma^* \mid \delta^*(q, x) \in F\}$$

Now we can formally define our idea of the equivalence of two states: "q is equivalent to r" means $L(M, q) = L(M, r)$. In our original DFA above, we had

$$\begin{aligned}
L(M, q_0) &= \{xay \mid x \in \{b\}^* \text{ and } y \in \{a\}^*\}\\
L(M, q_1) &= \{x \mid x \in \{a\}^*\}\\
L(M, q_2) &= \{x \mid x \in \{a\}^*\}\\
L(M, q_3) &= \{\}\\
L(M, q_4) &= \{\}
\end{aligned}$$

Thus q_1 was equivalent to q_2, and q_3 was equivalent to q_4.

A general procedure for minimizing DFAs is this:

1. Eliminate states that are not reachable from the start state.
2. Combine all the equivalent states, so that no two remaining states are equivalent to each other.

Formally, Step 2 can be described as the construction of a new DFA whose states are the equivalence classes of the states of the original DFA. This is sometimes called the *quotient construction*. But we will not give a formal definition of the quotient construction. Instead, we will just state without proof the important property of the whole minimization procedure:

> **Theorem 9.1:** Every regular language has a unique minimum-state DFA, and no matter what DFA for the language you start with, the minimization procedure finds it.

The minimum-state DFA is "unique" in a structural sense. Mathematicians say that the minimized DFA is unique *up to isomorphism*—in this case, unique except perhaps for the names of the states, which could of course be changed without affecting the structure of the DFA materially. Thus our minimization procedure is both safe and effective: safe, in that it does not change the language accepted by the DFA; effective, in that it arrives at the structurally unique, smallest DFA for that language.

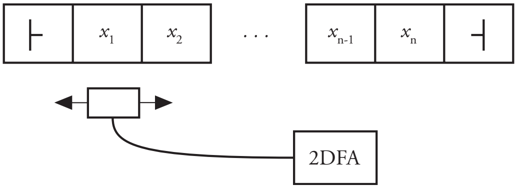
As described above, our minimization procedure is adequate for simple exercises done by hand, but it is not obvious how to write a program to do it. Is there an algorithm that can efficiently detect equivalent states and so perform the minimization? The answer is yes. The basic strategy is to start with a partition of the states into two classes, accepting and nonaccepting, and then repeatedly divide these into smaller partitions as states are discovered to be nonequivalent. The Further Reading section below has references for this algorithm.

Minimization results are weaker for NFAs. With NFAs, you can eliminate unreachable states and combine states using an equivalence relation similar to the one we saw for DFAs. But the resulting NFA is not necessarily a unique minimum-state NFA for the language.

9.2 Two-way Finite Automata

DFAs and NFAs read their input once, from left to right. One way to try to make finite automata more powerful is to allow rereading. A *two-way finite automaton* treats the input like a tape; the automaton has a read-only "head" that reads a symbol at the current position on the input tape. On each transition, the automaton moves the head either to the left or to the right. There are two-way deterministic finite automata (2DFAs) and two-way nondeterministic finite automata (2NFAs).

A 2DFA can be pictured like this:



The picture above shows an input tape containing the input string $x_1x_2 \dots x_{n-1}x_\text{n}$. The input string is framed on the tape by special symbols that serve as left and right end-markers. Like a DFA, a 2DFA defines state transitions based on the current symbol and state. But the transition function δ has an expanded role, since it now returns two things: a new state and a direction to move the head, either L for left or R for right. For example, $\delta(q, a) = (s, L)$ says that if the 2DFA is in the state q and the head is currently reading the symbol a, it then enters the state s and moves the head one place to the left. (On every transition the head moves one place, either left or right.) The transition function cannot move the head left past the left end-marker or right past the right end-marker.

In a DFA the end of the computation is reached when the last input symbol is read, but a 2DFA needs some other way to tell when the computation is finished. So instead of a set of accepting states, a 2DFA has a single accepting state t and a single rejecting state r. Its computation is finished when it enters either of these two states, signaling that it has reached its decision about the input string. (It can also get into an infinite loop, so it might never reach a decision.)

We state without proof the important results about two-way finite automata:

Theorem 9.2.1: There is a 2DFA for language L if and only if L is regular.

Theorem 9.2.2: There is a 2NFA for language L if and only if L is regular.

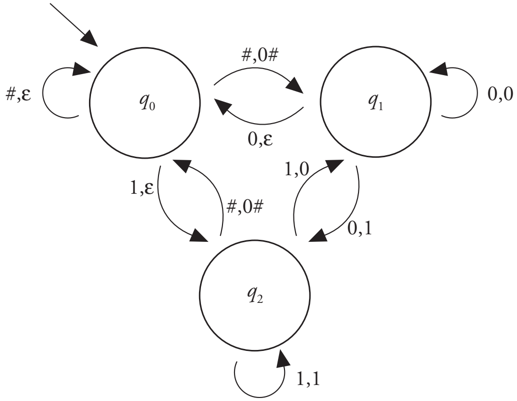
So adding two-way reading to finite automata does not increase their power; they can still recognize exactly the regular languages.

The interesting thing is that with just a little more tweaking we can get automata that are much more powerful, as we will see in later chapters. Adding the ability to write the tape as well as read it yields a kind of machine called a *linear bounded automaton*, which can define far more than just the regular languages. Adding the ability to write and to move unboundedly far past at least one of the end-markers produces a kind of machine called a Turing machine, which is more powerful still.

9.3 Finite-state Transducers

The machines we have seen may be thought of as computing functions with an arbitrary string as input and a single bit as output: either *yes*, the string is in the language, or *no* it is not. A variety of applications of finite-state machines actually want more output than this, which leads to the idea of *finite-state transducers*.

Here is an example of a deterministic finite-state transducer that produces output on each transition:



The alphabet for this machine is {0, 1, #}. It works like a DFA, except that on every transition, the machine not only reads one input symbol, but also produces a string of zero or more output symbols. Each transition is labeled with a pair *a*, *x*. The first element in the pair, *a*, is the input symbol read when the transition is made; the second element in the pair, *x*, is an output string generated when the transition is made. There is no accepting state, because the machine is not trying to recognize a language—it is trying to transform input strings into output strings.

Let's see what this machine does given the input 10#11#. Starting in q_0, initially no input has been read and no output produced. The following chart shows how this changes after each move:

Input Read by M	State of M	Output So Far
ε	q_0	ε
1	q_2	ε
10	q_1	1
10#	q_0	10#
10#1	q_2	10#
10#11	q_2	10#1
10#11#	q_0	10#10#

Given the input 10#11#, the machine produced the output 10#10#. In general, if the input to this machine is a string of binary numbers, each terminated by one or more # symbols, the output is a string of the same binary numbers *rounded down to the nearest even number*, each terminated by a single # symbol.

As this example suggests, finite-state transducers can be used as signal processors. In particular, they are used in a variety of natural-language processing, speech recognition, and speech-synthesis applications. Finite-state transducers come in many flavors. Some are deterministic, some nondeterministic; some associate an output string with each transition, others with each state.

9.4 Advanced Regular Expressions

When you write regular expressions, you quickly discover that there are many correct answers: many different regular expressions that actually define the same language. It is useful to be able to compare two regular expressions and decide whether they are equivalent. We have already seen a rough outline of how this can be done. You can convert your two expressions into NFAs, using the construction of Appendix A; then convert the NFAs into DFAs, using the subset construction; and then minimize the DFAs, using the techniques of this chapter. The minimum-state DFA for a language is unique, so your original regular expressions are equivalent if and only if your two minimized DFAs are identical. This is quite an expensive procedure, because the constructions involved can produce very large DFAs from comparatively small regular expressions. The problem of deciding regular-expression equivalence belongs to a class of problems known as *PSPACE-complete*. Without going into the definition now, we'll just note that PSPACE-complete problems are generally believed (but not proved) to require exponential time. Certainly the algorithm just outlined has that property; deciding whether two regular expressions of length n are equivalent using that approach takes, in the worst case, time at least proportional to 2^n.

We can consider extending regular expressions in various ways. As we have seen, many practical regexp languages already do this. But there are several regular-expression extensions that have theoretical interest as well as practical value.

One simple extension is *squaring*. We use the basic three atomic expressions and the basic three compound expressions, but then add one more kind of compound expression: $(r)^2$, denoting the same language as $(r)(r)$. Obviously, this extension does not add the ability to define new languages. But it does add the ability to express some regular languages much more compactly. For example, consider the language $\{0^n \mid n \bmod 64 = 0\}$. Using basic regular expressions, we would have to write this as

(0000000000000000000000000000000000000000000000000000000000000000)*

With squaring, we could write it more compactly as $(((((0000)^2)^2)^2)^2)^*$.

The question of whether two regular expressions are equivalent, when squaring is allowed, becomes more difficult to answer. It belongs to a class of problems known as *EXSPACE-complete*. Without going into the definition now, we'll just note that *EXSPACE*-complete problems require exponential memory space and at least exponential time and are generally believed (but not proved) to require more than exponential time.

Another possible addition to regular expressions is *complement*. For this extension, we start with basic regular expressions and then allow one extra kind of compound expression: $(r)^C$ denotes the complement of a language (with respect to some fixed alphabet), so that

$$L((r)^C) = \{x \in \Sigma^* \mid x \notin L(r)\}$$

We know that basic regular expressions can define all the regular languages, and we know that the regular languages are closed for complement. This means that our extension does not add the ability to define new languages. But, once again, we get the ability to define some languages much more compactly; and, once again, that makes the problem of deciding the equivalence of two regular expressions harder—much harder. (To be precise, the problem now requires *NONELEMENTARY TIME*. Complexity classes like *PSPACE*, *EXPSPACE*, and *NONELEMENTARY TIME* are discussed further in Chapter 20.)

In a regular expression, one subexpression using the Kleene star may be nested inside another. Informally, the *star height* of a regular expression is the depth of this nesting: $a + b$ has star height 0, $(a + b)^*$ has star height 1, $(a^* + b^*)^*$ has star height 2, and so on. Technically, we can define the star height $h(r)$ for any regular expression r inductively:

$h(r) = 0$ for any atomic regular expression r
$h(r + s) = \max(h(r), h(s))$
$h(rs) = \max(h(r), h(s))$
$h((r)^*) = h(r) + 1$
$h((r)^2) = h(r)$
$h((r)^C) = h(r)$

It is often possible, and desirable, to simplify expressions in a way that reduces their star height. For example, the expression $(a^* + b^*)^*$, star height 2, defines the same language as $(a + b)^*$, star height 1. Attempts to reduce the star height of regular expressions lead to some interesting questions:

- Is there some fixed star height, perhaps 1 or 2, that suffices for any regular language?
- Is there an algorithm that can take a regular expression and find the minimum possible star height for any expression for that language?

For basic regular expressions, the answers to these questions are known: no, no fixed height suffices for all regular languages; yes, there is an algorithm for finding the minimum star height for a regular language. However, when complement is allowed, these questions are still open. In particular, when complement is allowed, it is not known whether there is a regular language that requires a star height greater than 1! This *generalized star-height problem* is one of the most prominent open problems surrounding regular expressions.

9.5 Further Reading

There is a good introduction to the theory of DFA and NFA minimization in

Kozen, Dexter. Automata and Computability. New York: Springer-Verlag, 1997.

Kozen also has a good discussion of 2DFAs and 2NFAs, with proofs of their important properties.

The basic algorithm that minimizes a DFA with the alphabet Σ and the state set Q in $O(|\Sigma|\,|Q|^2)$ time is presented in

Hopcroft, John, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Menlo Park, CA: Addison-Wesley, 2001.

Using the fancier partitioning algorithm due to Hopcroft, a DFA can be minimized in $O(|\Sigma|\,|Q| \log |Q|)$ time. See:

Hopcroft, John. "An *n* log *n* algorithm for minimizing the states in a finite automaton." Published in *The Theory of Machines and Computations*, ed. Z. Kohavi. New York: Academic Press, 1972, 189–96.

Finite-state transducers have fallen out of most textbooks on formal languages and automata, even as they have grown as an active area of research. See, for example,

Mohri, Mehryar. "Finite-State Transducers in Language and Speech Processing." *Computational Linguistics* 23 (1997): 269-311.

The star-height problem is simple to describe, but very difficult to solve. Unfortunately, most of the work in this area is inaccessible to most undergraduates. If you are intested in the problem, and if your mathematical background includes abstract algebra, combinatorics, and computational complexity, you should begin by studying formal languages from the algebraic perspective, as in

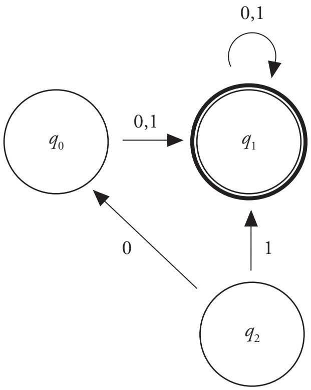
Holcombe, W. M. L. *Algebraic Automata Theory*. Cambridge: Cambridge University Press, 2004.

Exercises

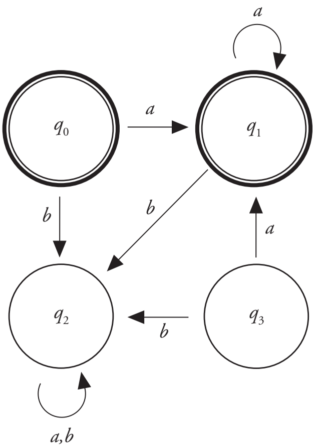
EXERCISE 1

For each of the following DFAs, list the unreachable states if any, show $L(M, q)$ for each $q \in Q$, and construct the minimized DFA using the procedure of Section 9.1.

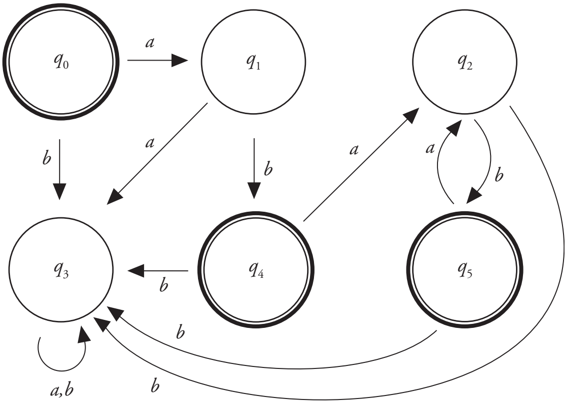
a.


0,1
q_0
0,1
q_1
0
1
q_2

b.


a
q_0
a
q_1
b
b
a
q_2
b
q_3
a,b

c.


q_0
a
q_1
q_2
b
a
b
a
a
b
q_3
b
q_4
q_5
b
a,b
b

EXERCISE 2

Write a Java class that implements the finite-state transducer example from Section 9.3. Include a `main` method that tests it on input from the command line.

EXERCISE 3

Using regular expressions extended with complement, find an expression with a star height of 0 for each of these languages. Assume that $\Sigma = \{a, b\}$.

a. Σ^*

b. $L((a + b)^*aa(a + b)^*)$

c. $\{x \in \Sigma^* \mid x \text{ has no two } a\text{s in a row}\}$

d. $L(b^*)$

e. For any two extended regular expressions r_1 and r_2 of star height 0, the language $L(r_1) \cap L(r_2)$.

f. For any two extended regular expressions r_1 and r_2 of star height 0, the language $L(r_1) - L(r_2)$.

g. $L((ab)^*)$. *Hint:* Almost all of these are strings that begin with *ab*, end with another *ab*, and have no two *a*s or *b*s in a row.

10 Grammars

Grammar *is another of those common words for which the study of formal language introduces a precise technical definition. For us, a grammar is a certain kind of collection of rules for building strings. Like DFAs, NFAs, and regular expressions, grammars are mechanisms for defining languages rigorously.*

A simple restriction on the form of these grammars yields the special class of right-linear grammars. The languages that can be defined by right-linear grammars are exactly the regular languages. There it is again!

10.1 A Grammar Example for English

Let's start with an example of a grammar for a familiar language: English. An *article* can be the word *a* or *the*. We use the symbol *A* for *article* and express our definition this way:

$A \rightarrow a$
$A \rightarrow the$

A *noun* can be the word *dog*, *cat*, or *rat*:

$N \rightarrow dog$
$N \rightarrow cat$
$N \rightarrow rat$

A *noun phrase* is an article followed by a noun:

$P \rightarrow AN$

A *verb* can be the word *loves*, *hates*, or *eats*:

$V \rightarrow loves$
$V \rightarrow hates$
$V \rightarrow eats$

A *sentence* is a noun phrase, followed by a verb, followed by another noun phrase:

$S \rightarrow PVP$

Put them all together and you have a grammar that defines a small subset of unpunctuated English. Call this grammar G_1:

$S \rightarrow PVP$
$P \rightarrow AN$
$V \rightarrow loves$
$V \rightarrow hates$
$V \rightarrow eats$
$A \rightarrow a$
$A \rightarrow the$
$N \rightarrow dog$
$N \rightarrow cat$
$N \rightarrow rat$

How does G_1 define a language? Think of each arrow as a rule that says how to modify strings by substitution. These rules are called *productions.* A production of the form $x \rightarrow y$ says that wherever you see the substring x, you may substitute y. For example, if we are given the string *thedogeatsP*, the production $P \rightarrow AN$ permits us to replace the P with AN, producing *thedogeatsAN*. By starting from S and following the productions of G_1 repeatedly, you can derive a variety of unpunctuated English sentences. For example:

$$\begin{aligned} S &\Rightarrow PVP \Rightarrow ANVP \Rightarrow \mathit{theNVP} \Rightarrow \mathit{thecatVP} \Rightarrow \mathit{thecateatsP} \Rightarrow \mathit{thecateatsAN} \\ &\Rightarrow \mathit{thecateatsaN} \Rightarrow \mathit{thecateatsarat} \end{aligned}$$

$$\begin{aligned} S &\Rightarrow PVP \Rightarrow ANVP \Rightarrow \mathit{aNVP} \Rightarrow \mathit{adogVP} \Rightarrow \mathit{adoglovesP} \Rightarrow \mathit{adoglovesAN} \\ &\Rightarrow \mathit{adoglovestheN} \Rightarrow \mathit{adoglovesthecat} \end{aligned}$$

$$\begin{aligned} S &\Rightarrow PVP \Rightarrow ANVP \Rightarrow \mathit{theNVP} \Rightarrow \mathit{thecatVP} \Rightarrow \mathit{thecathatesP} \Rightarrow \mathit{thecathatesAN} \\ &\Rightarrow \mathit{thecathatestheN} \Rightarrow \mathit{thecathatesthedog} \end{aligned}$$

The language defined by G_1 is the language of all the lowercase strings you can get by applying the productions of G_1, starting from S. The three examples above show that the language of G_1 includes the strings *thecateatstherat*, *adoglovesthecat*, and *thecathatesthedog*.

Often there is more than one place in a string where a production could be applied. For example, if we are given the string *PlovesP*, there are two places where the production $P \rightarrow AN$ can be used. Replacing the leftmost P gives $\mathit{PlovesP} \Rightarrow \mathit{ANlovesP}$; replacing the rightmost one gives $\mathit{PlovesP} \Rightarrow \mathit{PlovesAN}$. Because of this, there is often more than one way to produce the same final string:

$$\begin{aligned} S &\Rightarrow PVP \Rightarrow ANVP \Rightarrow \mathit{aNVP} \Rightarrow \mathit{adogVP} \Rightarrow \mathit{adoglovesP} \Rightarrow \mathit{adoglovesAN} \\ &\Rightarrow \mathit{adoglovestheN} \Rightarrow \mathit{adoglovesthecat} \end{aligned}$$

$$\begin{aligned} S &\Rightarrow PVP \Rightarrow PVAN \Rightarrow \mathit{PVAcat} \Rightarrow \mathit{PVthecat} \Rightarrow \mathit{Plovesthecat} \Rightarrow \mathit{ANlovesthecat} \\ &\Rightarrow \mathit{Adoglovesthecat} \Rightarrow \mathit{adoglovesthecat} \end{aligned}$$

These two different paths both produce the same string, *adoglovesthecat*.

When a grammar contains more than one production with the same left-hand side, those productions can be written in a compressed form, with the common left-hand side first, then the arrow, then the list of alternatives for the right-hand side, separated by vertical lines. Using this notation, our G_1 looks like this:

$$\begin{aligned} &S \rightarrow PVP \\ &P \rightarrow AN \\ &V \rightarrow \mathit{loves} \mid \mathit{hates} \mid \mathit{eats} \\ &A \rightarrow a \mid \mathit{the} \\ &N \rightarrow \mathit{dog} \mid \mathit{cat} \mid \mathit{rat} \end{aligned}$$

This makes the grammar easier to read, but it is merely shorthand. G_1 is not changed by being written this way and still has a total of ten separate productions.

For a more abstract example, consider this grammar, which we'll call G_2:

$$S \rightarrow aS$$
$$S \rightarrow X$$
$$X \rightarrow bX$$
$$X \rightarrow \varepsilon$$

The final production for X says that an X may be replaced by the empty string, so that for example the string *abbX* can become the string *abb*. Written in the more compact way, G_2 is

$$S \rightarrow aS \mid X$$
$$X \rightarrow bX \mid \varepsilon$$

Here are some derivations using G_2:

$$S \Rightarrow aS \Rightarrow aX \Rightarrow a$$
$$S \Rightarrow X \Rightarrow bX \Rightarrow b$$
$$S \Rightarrow aS \Rightarrow aX \Rightarrow abX \Rightarrow abbX \Rightarrow abb$$
$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaaX \Rightarrow aaabX \Rightarrow aaabbX \Rightarrow aaabb$$

All derivations for G_2 follow a simple pattern: first they use $S \rightarrow aS$ zero or more times, then they use $S \rightarrow X$ once, then they use $X \rightarrow bX$ zero or more times, and then they use $X \rightarrow \varepsilon$ once. So the string that is generated always consists of zero or more *a*s followed by zero or more *b*s. Thus, the language generated by G_2 is a regular language: $L(a^*b^*)$.

In all our examples, we have used productions with a single uppercase letter on the left-hand side. Not all grammars are this simple. Productions can have any nonempty string on the left-hand side. Exactly the same mechanism of substitution still applies; for example, the production $Sb \rightarrow bS$ says that wherever you have the substring *Sb*, you can substitute *bS*. Productions of this kind can be very powerful, and we'll explore them briefly in a later chapter. But for now, all our examples and exercises will continue to use productions with a single nonterminal symbol on the left-hand side.

10.2 The 4-Tuple

Up to this point, we have followed the convention of using *S* as the start symbol, with derivations from there ending only when the resulting string is fully lowercase. This special treatment for the Latin alphabet with its uppercase and lowercase letters is not sufficiently general for a formal treatment of grammars. To switch to a formal treatment, we have to include definitions of two separate sets of symbols: the *terminal symbols* (corresponding to our lowercase letters) and the *nonterminal symbols* (corresponding to our uppercase letters). Counting these symbol sets, a grammar has four key parts, so we describe it mathematically as a 4-tuple.

A grammar G is a 4-tuple $G = (V, \Sigma, S, P)$, where

V is an alphabet, the *nonterminal alphabet*

Σ is another alphabet, the *terminal alphabet*, disjoint from V

$S \in V$ is the *start symbol*

P is a finite set of productions, each of the form $x \rightarrow y$, where x and y are strings over $\Sigma \cup V$ and $x \neq \varepsilon$.

The first part of the grammar is an alphabet, the nonterminal alphabet V. The nonterminal alphabet contains the symbols we have been writing as uppercase letters—the symbols that are used in derivations but do not occur in the string that is finally derived. The second part of a grammar is the terminal alphabet. We use the same symbol Σ that we have used throughout this book for the alphabet of current interest; these are the symbols that we have been writing as lowercase letters. The sets Σ and V are *disjoint*, meaning that no symbol occurs in both. The third part of the grammar is a special nonterminal symbol S, the start symbol. The final part of a grammar is the set of productions. Each production is of the form $x \rightarrow y$, where x and y are both strings over $\Sigma \cup V$ and x is not permitted to be the empty string.

For example, consider our previous grammar for the language $L(a^*b^*)$:

$$S \rightarrow aS \mid X$$
$$X \rightarrow bX \mid \varepsilon$$

Formally, the grammar is $G = (V, \Sigma, S, P)$, where $V = \{S, X\}$, $\Sigma = \{a, b\}$, and the set P of productions is

$$\{S \rightarrow aS, S \rightarrow X, X \rightarrow bX, X \rightarrow \varepsilon\}$$

A grammar is a 4-tuple: it is a mathematical structure with four parts given in order. The names given to those four parts in the definition above—V, Σ, S, and P—are conventional, but the grammar G could just as well have been defined as $G = (\{S, X\}, \{a, b\}, S, \{S \rightarrow aS, S \rightarrow X, X \rightarrow bX, X \rightarrow \varepsilon\})$, without naming all the parts. The important thing is to specify the four parts in the required order.

10.3 The Language Generated by a Grammar

For DFAs, we defined a one-step transition function δ and then used it to define the zero-or-more-step extended transition function δ^*; that let us define the language of a DFA. For NFAs, we defined a one-step relation $\mapsto$, then extended it to a zero-or-more-step relation $\mapsto^*$; that let us define the language of an NFA. For grammars we will do something similar: we will first define a one-step derivation relation $\Rightarrow$ and then use it to define a zero-or-more-step extended derivation relation $\Rightarrow^*$.

The following four definitions are all made with respect to some fixed grammar $G = (V, \Sigma, S, P)$. Each production in the grammar specifies a possible substring substitution;

the production $x \rightarrow y$ says that wherever you see the substring x, you may substitute y. When a string w can be transformed into a string z in this way, we write $w \Rightarrow z$. (This is read as "*w derives z*.") Formally:

$\Rightarrow$ is a relation on strings, with $w \Rightarrow z$ if and only if there exist strings u, x, y, and v over $\Sigma \cup V$, with $w = uxv$, $z = uyv$, and $(x \rightarrow y) \in P$.

A sequence of $\Rightarrow$-related strings, $x_0 \Rightarrow x_1 \Rightarrow \ldots \Rightarrow x_n$, is called a *derivation*:

An *n-step derivation*, for any $n \geq 0$, is a sequence of strings x_0 through x_n such that for all $0 \leq i < n$, $x_i \Rightarrow x_{i+1}$.

Using n-step derivations, we can define the extended derivation relation $\Rightarrow^*$:

$\Rightarrow^*$ is a relation on strings, with $w \Rightarrow^* z$ if and only if there is a derivation of zero or more steps that starts with w and ends with z.

Notice here that $\Rightarrow^*$ is reflexive: for any string x, $x \Rightarrow^* x$ by a zero-step derivation. Using the $\Rightarrow^*$ relation, we can define the language generated by G:

The language generated by the grammar G is $L(G) = \{x \in \Sigma^* \mid S \Rightarrow^* x\}$

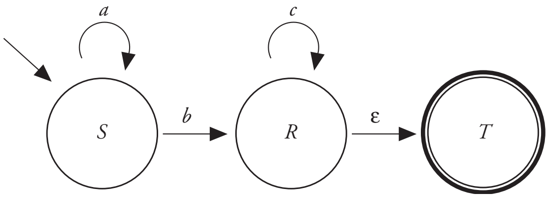
Notice here the restriction that $x \in \Sigma^*$. The intermediate strings in a derivation can involve both terminal and nonterminal symbols, but only the fully terminal strings derivable from the start symbol are in the language generated by the grammar.

All four of the preceding definitions were made with respect to some fixed grammar $G = (V, \Sigma, S, P)$. Clearly, different grammars produce different $\Rightarrow$ and $\Rightarrow^*$ relations. In those rare cases when we need to work with two or more grammars at once, we use subscripts to identify the relations of each: $\Rightarrow_G$, $\Rightarrow_H$, and so on.

10.4 Every Regular Language Has a Grammar

In exercises and examples we have seen many regular languages generated by grammars. Is there a grammar for every regular language? The answer is yes, and the proof is by construction. We will show how to take any NFA M and build a grammar G for which $L(M) = L(G)$.

First, an example of this construction. Consider the language $L(a^*bc^*)$. This is an NFA for the language:



We will construct a grammar that generates the same language. Not only will it generate the same language, but its derivations will exactly mimic the behavior of the NFA. To make the construction clearer, the states in the NFA are named *S*, *R*, and *T*, instead of the usual q_0, q_1, and q_2. That's because in our construction, each state of the NFA will become a nonterminal symbol in the grammar, and the start state of the NFA will become the start symbol in the grammar. That settles the two alphabets and the start state for the new grammar; all that remains is to construct the set of productions.

For each possible transition $Y \in \delta(X, z)$ in the NFA (where z is any single symbol from Σ or $z = \varepsilon$), we add a production $X \to zY$ to the grammar. For our example this is

Transition of *M*	**Production in *G***
$\delta(S, a) = \{S\}$	$S \to aS$
$\delta(S, b) = \{R\}$	$S \to bR$
$\delta(R, c) = \{R\}$	$R \to cR$
$\delta(R, \varepsilon) = \{T\}$	$R \to T$

In addition, for each accepting state in the NFA, the grammar contains an ε-production for that nonterminal. For our example this is

Accepting State of *M*	**Production in *G***
T	$T \to \varepsilon$

So the complete grammar constructed for this example is

$S \to aS$
$S \to bR$
$R \to cR$
$R \to T$
$T \to \varepsilon$

To see how this *G* simulates *M*, let's compare the behavior of *M* as it accepts the string *abc* with the behavior of *G* as it generates the same string. Here are the accepting sequence of instantaneous descriptions of *M* and the derivation of *G*, side by side:

$$\begin{array}{ccccccccccc} (S, abc) & \mapsto & (S, bc) & \mapsto & (R, c) & \mapsto & (R, \varepsilon) & \mapsto & (T, \varepsilon) & & \\ S & \Rightarrow & aS & \Rightarrow & abR & \Rightarrow & abcR & \Rightarrow & abcT & \Rightarrow & abc \end{array}$$

M starts in the state *S*; *G* starts with the start symbol *S*. After reading the first symbol *a*, *M* is in the state *S*; after the first derivation step *G* has produced the string *aS*. After reading the first two symbols, *ab*, *M* is in the state *R*; after the first two derivation steps *G* has produced the string *abR*. After the first three symbols, *abc*, *M* is in the state *R*; after the first three derivation steps *G* has produced the string *abcR*. As its fourth move, *M* makes an ε-transition to state *T*; after the first four derivation steps *G* has produced the string *abcT*. Now *M* has finished reading the string *abc* and ends in an accepting state; *G* applies a final derivation step using $T \to \varepsilon$, deriving the fully terminal string *abc*. Except for this final derivation step, there is a simple relation between the NFA and the grammar: every time the NFA reads a symbol, the grammar generates that symbol. The NFA can be in the state *Y* after reading a string *x* if and only if the grammar can derive the string *xY*.

That's how the general construction will work. Every derivation from *S* of a fully terminal string has the same general form:

$$S \Rightarrow z_1R_1 \Rightarrow z_1z_2R_2 \Rightarrow z_1z_2z_3R_3 \Rightarrow \ldots \Rightarrow z_1 \ldots z_nR_n \Rightarrow z_1 \ldots z_n$$

At every step but the last, z_i (which is a single symbol or the empty string) is added to the right-hand side of the growing terminal string, along with a rightmost nonterminal symbol. In the last step, one of the ε-productions is used to eliminate the nonterminal symbol. This exactly traces a possible behavior of the NFA when it reads the same string.

Theorem 10.1: Every regular language is generated by some grammar.

Proof: By construction. Every regular language is accepted by some NFA, so let $M = (Q, \Sigma, \delta, S, F)$ be any NFA. (For convenience, we have named the start state *S*, instead of the more common q_0.) Let *G* be the grammar $G = (Q, \Sigma, S, P)$, using *M*'s set of states as the nonterminal alphabet, *M*'s alphabet as the terminal alphabet, and *M*'s start state *S* as the start symbol. The set *P* of productions is defined as follows: wherever *M* has $Y \in \delta(X, \omega)$, *P* contains the production $X \to \omega Y$, and for each $X \in F$, *P* contains $X \to \varepsilon$.

By construction, *G* has $X \to \omega Y$ whenever *M* has $(X, \omega) \mapsto (Y, \varepsilon)$. We can extend this property to all strings $z \in \Sigma^*$: *G* has a derivation $X \Rightarrow^* zY$ if and only if *M* has $(X, z) \mapsto^* (Y, \varepsilon)$. (This can be proved by induction on the length of the derivation.) By construction, the grammar has $Y \to \varepsilon$ whenever the NFA has $Y \in F$. So for all strings $z \in \Sigma^*$, $\delta^*(S, z)$ contains at least one element of *F* if and only if $S \Rightarrow^* z$. Thus $L(M) = L(G)$.

10.5 Right-Linear Grammars

The construction of Theorem 10.1 generates productions of a particularly restricted form: *single-step form*.

A grammar $G = (V, \Sigma, S, P)$ is *single step* if and only if every production in P is in one of these two forms:

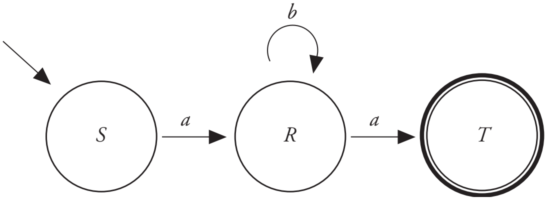
$$X \to \omega Y$$
$$X \to \varepsilon$$

where $X \in V$, $Y \in V$, and $\omega \in \Sigma \cup \{\varepsilon\}$.

If you're given a grammar that happens to be single step, you can easily reverse the construction and build a corresponding NFA. For instance, here's a single-step grammar that generates the language $L(ab^*a)$:

$$S \to aR$$
$$R \to bR$$
$$R \to aT$$
$$T \to \varepsilon$$

We can make an NFA with three states, S, R, and T. The first three productions correspond to three transitions in the NFA diagram, and the final production tells us to make T an accepting state. The resulting NFA is



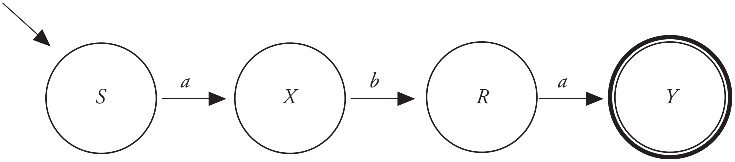
So it is easy enough to convert a single-step grammar to an NFA. Even if the productions are not of the right form, it is sometimes possible to massage them until they are. Here's a grammar that does not follow our restrictions on production forms—a grammar for the language $\{aba\}$:

$$S \to abR$$
$$R \to a$$

The production $S \to abR$ does not qualify because it has the string *ab* instead of just a single symbol before the final nonterminal R. But this is easy to correct; we can just substitute the two productions $S \to aX$ and $X \to bR$, both of which are single step. The production $R \to a$ does not qualify either, but again, it is easy to correct; we can just substitute the two productions $R \to aY$ and $Y \to \varepsilon$, both of which are single step. The resulting single-step grammar is

$$\begin{aligned}&S \to aX\\&X \to bR\\&R \to aY\\&Y \to \varepsilon\end{aligned}$$

This can then be converted into an NFA using our construction:



The grammar we started with this time, although not quite in the form we wanted, was still fairly close; every production had a single nonterminal as its left-hand side and a string containing at most one nonterminal as its right-hand side. Moreover, that nonterminal on the right-hand side was always the rightmost symbol in the string. A grammar that follows this restriction is called a *right-linear grammar*.

A grammar $G = (V, \Sigma, S, P)$ is *right linear* if and only if every production in P is in one of these two forms:

$$\begin{aligned}&X \to zY\\&X \to z\end{aligned}$$

where $X \in V$, $Y \in V$, and $z \in \Sigma^*$.

Every right-linear grammar generates a regular language and can be converted fairly easily into an NFA that accepts that language. As in the example above, the trick is to first rewrite the grammar to make it single step. This is always possible.

Lemma 10.1: Every right-linear grammar G is equivalent to some single-step grammar G'.

Proof: By construction. Let $G = (V, \Sigma, S, P)$ be any right-linear grammar. Each production in P is of the form $X \to z_1 \ldots z_n \omega$, where $\omega \in V$ or $\omega = \varepsilon$. For each such production let P' contain these $n + 1$ productions:

$$\begin{aligned}&X \to z_1 K_1\\&K_1 \to z_2 K_2\\&\ldots\end{aligned}$$

$K_{n-1} \to z_n K_n$
$K_n \to \omega$

where each K_i is a new nonterminal symbol. Now let $G' = (V', \Sigma, S, P')$, where V' is the set of nonterminals used in the new productions. The new grammar G' is single step; it remains to be proven that $L(G) = L(G')$. This is straightforward. Any derivation of a terminal string in G can be converted to a derivation of the same terminal string in G' simply by replacing each step that used a rule $X \to z_1 \ldots z_n \omega$ with the $n + 1$ corresponding steps in G' and vice versa.

The construction given above makes more new productions than strictly necessary. For example, it converts a production $A \to bC$ into the two productions $A \to bK_1$ and $K_1 \to C$. Conversion is unnecessary in that case, of course, since the original production was already in the required form. But it doesn't hurt, and it makes the proof simpler to use a single, general transformation for every production in the original grammar.

10.6 Every Right-Linear Grammar Generates a Regular Language

Using the techniques just introduced, we can take any right-linear grammar and convert it to an NFA. This shows that the language generated by a right-linear grammar is always a regular language.

Theorem 10.2: For every right-linear grammar G, $L(G)$ is regular.

Proof: By construction, using Lemma 10.1 and the reverse of the construction of Theorem 10.1.

Incidentally, there is parallel definition for *left-linear grammars*:

A grammar $G = (V, \Sigma, S, P)$ is *left linear* if and only if all productions in P are of one of these two forms:

$X \to Yz$ or
$X \to z$,

where $X \in V$, $Y \in V$, and $z \in \Sigma^*$.

With a little more work, one can show that the language generated by a left-linear grammar is also always a regular language. A grammar that is either left linear or right linear is sometimes called a *regular grammar*.

A grammar does not have to be regular to generate a regular language. Consider this grammar:

$S \to aSaa \mid \varepsilon$

This is neither right linear nor left linear, yet the language it generates is a simple regular language: $L((aaa)^*)$. If a grammar is regular, you can conclude that the language it generates is regular. But if a grammar is not regular, you can't conclude anything about the language it generates; the language may still be regular.

All of this leads to the next important question: are there grammars that generate languages that are *not* regular? This is answered in the next chapter.

Exercises

EXERCISE 1

These questions concern grammar G_1 from section 10.1:

a. Show a derivation from S for the string *thedogeatsthecat*.

b. List all the lowercase strings that can be derived from P.

c. How many lowercase strings are in the whole language defined by G_1—the language of all lowercase strings that can be derived from S? Show your computation.

EXERCISE 2

Modify grammar G_1 from Section 10.1 to make it allow at most one of the adjectives *red*, *little*, *black*, or *cute*, as part of any noun phrase. For example, your grammar should generate the sentence *thereddoglovestheblackcat*. Make sure it also generates all the original sentences, without adjectives.

EXERCISE 3

These questions concern grammar G_2 from Section 10.1:

c. Show a derivation from S for the string *bbb*.

d. Show a derivation from S for the string *aabb*.

e. Show a derivation from S for the empty string.

EXERCISE 4

Give a regular expression for the language generated by each of these grammars.

a. $S \rightarrow abS \mid \varepsilon$

b. $S \rightarrow aS \mid aA$ (*Hint:* This one is tricky—be careful!)
$A \rightarrow aS \mid aA$

c. $S \rightarrow smellA \mid fishA$
$A \rightarrow y \mid \varepsilon$

d. $S \rightarrow aaSa \mid \varepsilon$

EXERCISE 5

Give a grammar for each of the following languages. In each case, use S as the start symbol.

a. $L(a^*)$

b. $L(aa^*)$

c. $L(a^*b^*c^*)$

d. $L((abc)^*)$

e. The set of all strings consisting of one or more digits, where each digit is one of the symbols 0 through 9.

EXERCISE 6

Give a DFA that accepts the language generated by this grammar:

$S \to A \mid B \mid C$
$A \to aB \mid \varepsilon$
$B \to bC \mid \varepsilon$
$C \to cA \mid \varepsilon$

EXERCISE 7

State each of the following grammars formally, as a 4-tuple. (Assume, as usual, that the terminal alphabet is the set of lowercase letters that appear in productions, the nonterminal alphabet is the set of uppercase letters that appear in productions, and the start symbol is *S*.)

a. $S \to aS \mid b$

b. $S \to abS \mid A$
$A \to baA \mid \varepsilon$

c. $S \to A \mid B \mid C$
$A \to aB \mid \varepsilon$
$B \to bC \mid \varepsilon$
$C \to cA \mid \varepsilon$

EXERCISE 8

According to the definition, what is the smallest possible grammar—with the smallest possible set in each of the four parts? What language does it generate?

EXERCISE 9

Consider this grammar:

$S \to aS \mid bbS \mid X$
$X \to cccX \mid \varepsilon$

For each of the following, show all strings that can be derived from the given string in one step. For example, for the string *cS*, your answer would be the three strings *caS*, *cbbS*, and *cX*.

a. *S*

b. *aXa*

c. *abc*

d. *ScS*

e. *aXbSc*

EXERCISE 10

Consider this grammar:

$$S \to AB \mid \varepsilon$$
$$A \to aA \mid \varepsilon$$
$$B \to bB \mid \varepsilon$$

For each of the following pairs related by $\Rightarrow^*$, show a derivation. For instance, for the pair $S \Rightarrow^* a$, one correct answer is the four-step derivation $S \Rightarrow AB \Rightarrow aAB \Rightarrow aB \Rightarrow a$.

a. $aA \Rightarrow^* aaaA$
b. $aB \Rightarrow^* abbB$
c. $Bb \Rightarrow^* bb$
d. $S \Rightarrow^* \varepsilon$
e. $S \Rightarrow^* a$
f. $S \Rightarrow^* aabb$

EXERCISE 11

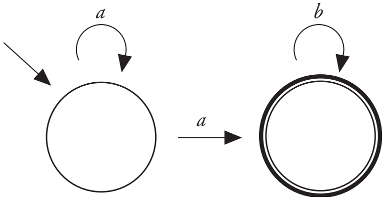
Give a grammar for each of the following languages. You need not state it formally; just list the productions, and use S as the start symbol.

a. The set of all strings consisting of zero or more *a*s with a *b* after each one.
b. The set of all strings consisting of one or more *a*s with a *b* after each one.
c. The set of all strings consisting of one or more *a*s, with a *b* between each *a* and the next. (There should be no *b* before the first or after the last *a*.)
d. The set of all strings consisting of zero or more *a*s, with a *b* between each *a* and the next. (There should be no *b* before the first or after the last *a*.)
e. The set of all strings consisting of an open bracket (the symbol [) followed by a list of zero or more digits separated by commas, followed by a closing bracket (the symbol]). (A digit is one of the characters 0 through 9.)

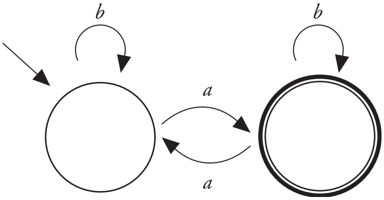
EXERCISE 12

Using the construction of Theorem 10.1, make a right-linear grammar that generates the language accepted by each of these NFAs.

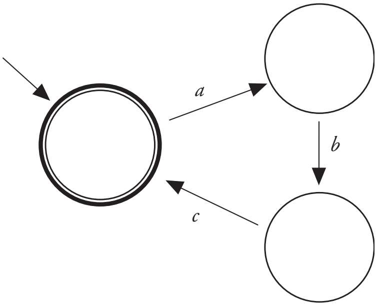
a.



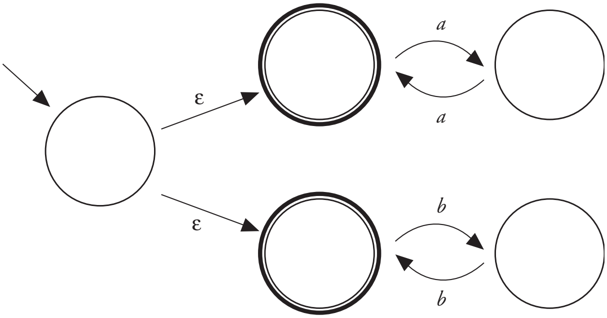
b.



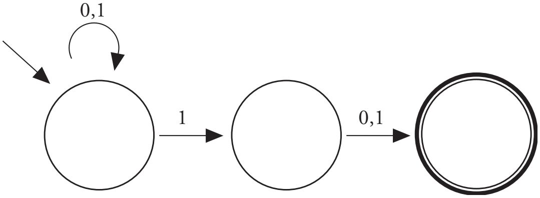
c.



d.



e.



EXERCISE 13

Using the construction of Theorem 10.2, make an NFA that accepts the language generated by each of these right-linear grammars.

a. $S \rightarrow bS \mid aX$
   $X \rightarrow aX \mid \varepsilon$

b. $S \rightarrow bS \mid X \mid \varepsilon$
   $X \rightarrow aX \mid cY$
   $Y \rightarrow \varepsilon$

c. $S \rightarrow bR \mid \varepsilon$
   $R \rightarrow baR \mid \varepsilon$

d. $S \rightarrow aS \mid bbS \mid \mathrm{X}$
   $X \rightarrow cccX \mid d$

EXERCISE 14

Given any single-step grammar $G = (Q, \Sigma, S, P)$, give a formal construction for an NFA M with $L(M) = L(G)$. (This is the reverse of the construction used to prove Theorem 10.1.) You don't have to show a proof that the languages are equal.

EXERCISE 15

Consider this grammar G:

$$S \rightarrow Sa \mid Sb \mid \varepsilon$$

Prove by induction on $|x|$ that for any string $x \in \{a, b\}^*$, $S \Rightarrow^* Sx$. Then use this result to prove that $L(G) = \{a, b\}^*$.

EXERCISE 16

Rewrite the proof of Theorem 10.1. This time, include a full inductive proof that G has a derivation $X \Rightarrow^* zY$ if and only if M has $(X, z) \mapsto^* (Y, \varepsilon)$. *Hint:* It is easier to prove a slightly stronger lemma, that G has an n-step derivation $X \Rightarrow^* zY$ if and only if M has an n-step sequence $(X, z) \mapsto^* (Y, \varepsilon)$.

11

Nonregular Languages

We have now encountered regular languages in several different places. They are the languages that can be recognized by a DFA. They are the languages that can be recognized by an NFA. They are the languages that can be denoted by a regular expression. They are the languages that can be generated by a right-linear grammar. You might begin to wonder, are there any languages that are ***not*** *regular?*

In this chapter, we will see that there are. There is a proof tool that is often used to prove languages nonregular. It is called the **pumping lemma***, and it describes an important property of all regular languages. If you can show that a given language does not have this property, you can conclude that it is not a regular language.*

11.1 The Language $\{a^nb^n\}$

Consider the language of strings that consist of any number of *a*s followed by the same number of *b*s. The language contains the strings ε, *ab*, *aabb*, *aaabbb*, and so on. In set-former notation this language can be written simply as $\{a^nb^n\}$.

It is simple enough to give a grammar for $\{a^nb^n\}$:

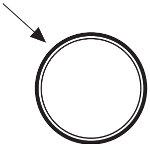
$$S \rightarrow aSb \mid \varepsilon$$

Every derivation in this grammar of a fully terminal string has the same form:

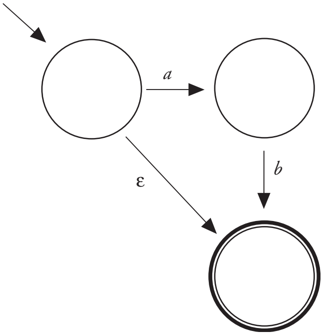
$$S \Rightarrow aSb \Rightarrow a^2Sb^2 \Rightarrow a^3Sb^3 \Rightarrow \ldots \Rightarrow a^nSb^n \Rightarrow a^nb^n$$

At every step but the last, one *a* and one *b* are generated, ensuring that the number of *a*s is always the same as the number of *b*s. In the final derivation step, the ε production is used to produce a fully terminal string a^nb^n.

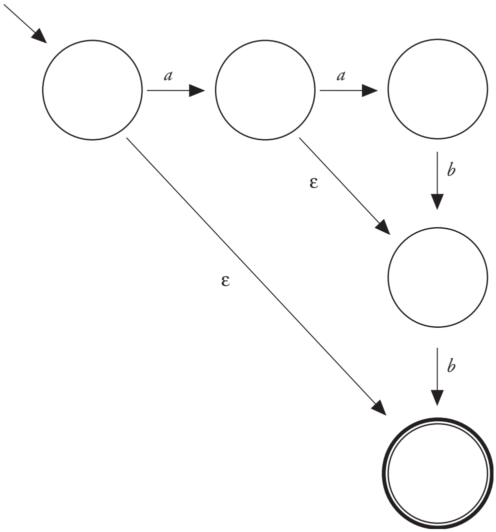
Is $\{a^nb^n\}$ a regular language? If it is, there is an NFA for it, so let's try to develop one. Here is an NFA for the subset $\{a^nb^n \mid n \leq 0\}$, which contains just the empty string:



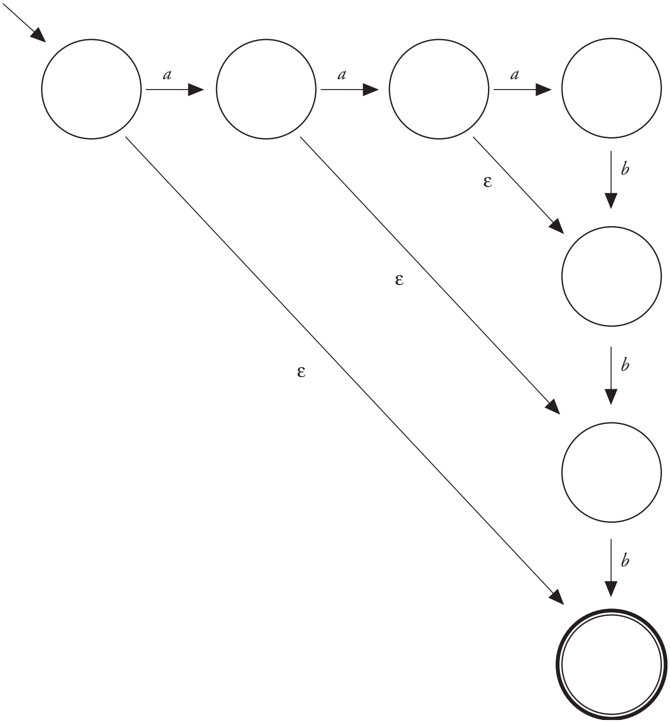
Here is one for the larger subset $\{a^nb^n \mid n \leq 1\}$:



Here is one for the larger subset $\{a^n b^n \mid n \leq 2\}$:



For the next larger subset $\{a^n b^n \mid n \leq 3\}$, we could use this:



Clearly, this is not going to be a successful pattern on which to construct an NFA for the whole language $\{a^n b^n\}$, since for each larger value of n we are adding two more states. In effect, we are using the states of the NFA to count how many *a*s were seen, then to verify that the same number of *b*s follows. This won't work in general, since the *F* in NFA stands for *finite*, but no finite number of states will be enough to count the unbounded n in $\{a^n b^n\}$.

Of course, this failure does not prove that $\{a^n b^n\}$ is not regular, but it contains the germ of the idea for a proof—the intuition that no fixed, finite number of states can be enough. A formal proof follows below.

Theorem 11.1: $\{a^n b^n\}$ is not regular.

Proof: Let $M = (Q, \{a, b\}, \delta, q_0, F)$ be any DFA over the alphabet $\{a, b\}$; we will show that $L(M) \neq \{a^n b^n\}$. Consider the behavior of M on an arbitrarily long string of *as*. As it reads the string, M visits a sequence of states: first $\delta^*(q_0, \varepsilon)$, then $\delta^*(q_0, a)$, then $\delta^*(q_0, aa)$, and so on. Eventually, since M has only finitely many states, it must revisit a state; that is, there must be some i and j with $i < j$ for which $\delta^*(q_0, a^i) = \delta^*(q_0, a^j)$. Now by appending b^j to both strings, we see that $\delta^*(q_0, a^i b^j) = \delta^*(q_0, a^j b^j)$. Thus M ends up in the same final state for both $a^i b^j$ and $a^j b^j$. If that is an accepting state, M accepts both; if it is a rejecting state, M rejects both. But this means that $L(M) \neq \{a^n b^n\}$, since $a^j b^j$ is in $\{a^n b^n\}$ while $a^i b^j$ is not. Since we have shown that $L(M) \neq \{a^n b^n\}$ for any *DFA M,* we conclude that $\{a^n b^n\}$ is not regular.

An interesting thing about this proof is how much one can infer about the behavior of a completely unknown DFA M. Using only the fact that as a DFA M must have a finite number of states, one can infer that there must be some i and j with $i < j$ for which M either accepts both $a^i b^j$ and $a^j b^j$ or rejects both. The basic insight is that with a sufficiently long string you can force any DFA to repeat a state. This is the basis of a wide variety of nonregularity proofs.

11.2 The Language $\{xx^R\}$

Consider the language of strings that consist of any string over a given alphabet Σ, followed by a reversed copy of the same string. For $\Sigma = \{a, b\}$, the language contains the strings ε, *aa, bb, abba, baab, aaaa, bbbb*, and so on. For any string x, the notation x^R refers to the reverse of x; the language in question is just $\{xx^R\}$.

It is simple enough to give a grammar for $\{xx^R\}$. Here is one for $\Sigma = \{a, b\}$:

$$S \rightarrow aSa \mid bSb \mid \varepsilon$$

Every derivation in this grammar of a fully terminal string has the same form:

$$S \Rightarrow x_1Sx_1^R \Rightarrow x_2Sx_2^R \Rightarrow x_3Sx_3^R \Rightarrow \ldots \Rightarrow x_nSx_n^R \Rightarrow x_nx_n^R$$

where, for all i, $|x_i| = i$. At every step but the last, a matching pair of *a*s or *b*s is generated, ensuring that the terminal string to the left of S is always the reverse of the terminal string to the right of S. In the final derivation step, the ε production is used to produce a fully terminal string $x_nx_n^R$.

After seeing the $\{a^nb^n\}$ example, you may already have doubts about whether $\{xx^R\}$ can be a regular language. A finite state machine would have to use its current state to keep track of the string x, then check that it is followed by a matching x^R. Since there is no bound on the length of x, this suggests that no finite number of states will suffice.

Theorem 11.2: $\{xx^R\}$ is not regular for any alphabet with at least two symbols.

Proof: Let $M = (Q, \Sigma, \delta, q_0, F)$ be any DFA with $|\Sigma| \geq 2$; we will show that $L(M) \neq \{xx^R\}$. The alphabet Σ contains at least two symbols; call two of these a and b. Consider the behavior of M on an arbitrarily long string of *a*s. As it reads the string, M visits a sequence of states: first $\delta^*(q_0, \varepsilon)$, then $\delta^*(q_0, a)$, then $\delta^*(q_0, aa)$, and so on. Eventually, since M has only finitely many states, it must revisit a state; that is, there must be some i and j with $i < j$ for which $\delta^*(q_0, a^i) = \delta^*(q_0, a^j)$. Now by appending bba^j to both strings, we see that $\delta^*(q_0, a^ibba^j) = \delta^*(q_0, a^jbba^j)$. Thus M ends up in the same final state for both a^ibba^j and a^jbba^j. If that is an accepting state, M accepts both; if it is a rejecting state, M rejects both. But this means that $L(M) \neq \{xx^R\}$, since a^jbba^j is in $\{xx^R\}$ while a^ibba^j is not. Since we have shown that $L(M) \neq \{xx^R\}$ for any DFA M, we conclude that $\{a^nb^n\}$ is not regular.

This proof was almost identical to the previous proof. We used the same insight that by giving a sufficiently long string we could force the DFA to repeat a state—and by so doing, we could force it to end up in the same final state for two strings that ought to be treated differently.

11.3 Pumping

Both of the last two proofs worked by choosing a string long enough to make any given DFA repeat a state. Both proofs just used strings of *a*s, showing that for any given DFA there must be some i and j with $i < j$ for which $\delta^*(q_0, a^i) = \delta^*(q_0, a^j)$.

For those proofs it was enough to find those two strings—two different strings that put the DFA in the same state. For other proofs of nonregularity it is sometimes necessary to carry the same argument further: to show that there are not just two but infinitely many different strings, all of which leave the DFA in the same state. It is actually a fairly simple extension. Let r be the state that repeats, so $r = \delta^*(q_0, a^i)$. We know that r is visited again after $j - i$ more *a*s: $\delta^*(q_0, a^j) = \delta^*(q_0, a^{i+(j-i)}) = r$. In fact, every time the machine reads an additional $j - i$ *a*s, it will return to state r:

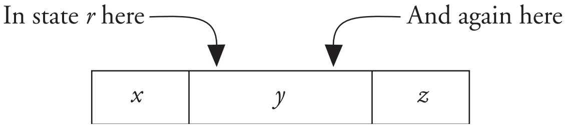
$$\begin{aligned} r &= \delta^*(q_0, a^i) \\ &= \delta^*(q_0, a^{i+(j-i)}) \\ &= \delta^*(q_0, a^{i+2(j-i)}) \\ &= \delta^*(q_0, a^{i+3(j-i)}) \\ &= \ldots \end{aligned}$$

and so on. That little substring of $j - i$ additional *a*s can be "pumped" any number of times, and the DFA always ends up in the same state.

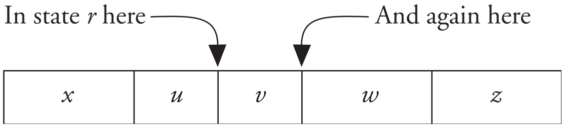
All regular languages have an important property involving pumping. Any sufficiently long string in a regular language must contain a "pumpable" substring—a substring that can be replicated any number of times, always yielding another string in the language. Formally:

> **Lemma 11.1:** (*The Pumping Lemma for Regular Languages*): For all regular languages L there exists some $k \in \mathcal{N}$ such that for all $xyz \in L$ with $|y| \geq k$, there exist $uvw = y$ with $|v| > 0$, such that for all $i \geq 0$, $xuv^iwz \in L$.
>
> **Proof:** Let L be any regular language, and let $M = (Q, \Sigma, \delta, q_0, F)$ be any DFA for it. For k we choose $k = |Q|$. Consider any x, y, and z with $xyz \in L$ and $|y| \geq k$. Since $|y| \geq |Q|$ we know there is some state r that M repeats while reading the y part of the string xyz. We can therefore divide y into three substrings $uvw = y$ so that $\delta^*(q_0, xu) = \delta^*(q_0, xuv) = r$. Now v can be pumped: for all $i \geq 0$, $\delta^*(q_0, xuv^i) = r$, and so $\delta^*(q_0, xuv^iwz) = \delta^*(q_0, xuvwz) = \delta^*(q_0, xyz) \in F$. Therefore, for all $i \geq 0$, $xuv^iwz \in L$.

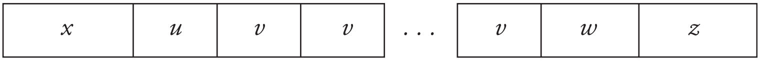
In this proof, nothing is known about the strings x, y, and z except that $xyz \in L$ and $|y| \geq k = |Q|$. Because there are at least as many symbols in y as states in Q, we know that some state r must be visited more than once as M reads the y part of the string xyz:



That gives us a way to divide y into those three further substrings $uvw = y$:



Now v is the pumpable substring. We can replace it with any number of copies, and we know the DFA will end up in the same state after each copy, so we know it will end in the same (accepting) state after the whole string is read. Thus it will accept, not just that original string $xyz = xuvwz$, but xuv^iwz for all i.



Take a closer look at the structure of the pumping lemma. It is a sequence of clauses that alternate "for all" and "there exist" parts: **for all** regular languages L **there exists** some $k \in N$ such that **for all** $xyz \in L$ with $|y| \geq k$, **there exist** $uvw = y$ with $|v| > 0$, such that **for all** $i \geq 0$, $xuv^iwz \in L$. Here is the overall structure of the pumping lemma, using the symbols $\forall$ ("for all") and $\exists$ ("there exists"):

1. $\forall\ L$ ...
2. $\exists\ k$ ...
3. $\forall\ xyz$ ...
4. $\exists\ uvw$ ...
5. $\forall\ i$ ...

Our proof shows how the $\exists$ values, k and uvw, can be constructed, starting from a DFA for the language L. But the construction used in the proof is not part of the lemma. From now on we will forget that k might be the number of states in a DFA for L; we will close the lid and treat the pumping lemma as a black box. The lemma merely says that suitable values k and uvw can be found; it does not say how.

11.4 Pumping-Lemma Proofs

The pumping lemma is very useful for proving that languages are not regular. For example, here is a pumping-lemma proof showing that $\{a^nb^n\}$ is not regular. The steps in the proof are numbered for future reference.

1. The proof is by contradiction using the pumping lemma for regular languages. Assume that $L = \{a^nb^n\}$ is regular, so the pumping lemma holds for L. Let k be as given by the pumping lemma.
2. Choose x, y, and z as follows:

   $x = a^k$
   $y = b^k$
   $z = \varepsilon$

   Now $xyz = a^kb^k \in L$ and $|y| \geq k$ as required.
3. Let u, v, and w be as given by the pumping lemma, so that $uvw = y$, $|v| > 0$, and for all $i \geq 0$, $xuv^iwz \in L$.

4. Choose $i = 2$. Since v contains at least one b and nothing but bs, uv^2w has more bs than uvw. So xuv^2wz has more bs than as, and $xuv^2wz \notin L$.
5. By contradiction, $L = \{a^nb^n\}$ is not regular.

The structure of this proof matches the structure of the pumping lemma. The alternating ∀ and ∃ parts of the lemma make every pumping-lemma proof into a kind of game. The ∃ parts (the natural number k and the strings u, v, and w) are merely guaranteed to exist. In effect, the pumping lemma itself makes these moves; it is a black box that produces values for k, u, v, and w. In Steps 1 and 3 we could only say these values are "as given by the pumping lemma." The ∀ parts, on the other hand, are the moves you get to make; you can choose any values for x, y, z, and i, since the lemma is supposed to hold for all such values. In Steps 2 and 4 we chose values for x, y, z, and i so as to reach a contradiction: a string xuv^iwz that is not in L, contradicting the pumping lemma.

Pumping-lemma proofs follow this pattern quite strictly. Here is another example: a pumping-lemma proof showing that $\{xx^R\}$ is not regular.

1. The proof is by contradiction using the pumping lemma for regular languages. Assume that $L = \{xx^R\}$ is regular, so the pumping lemma holds for L. Let k be as given by the pumping lemma.
2. Choose x, y, and z as follows:
   $x = a^kbb$
   $y = a^k$
   $z = \varepsilon$
   Now $xyz = a^kbba^k \in L$ and $|y| \geq k$ as required.
3. Let u, v, and w be as given by the pumping lemma, so that $uvw = y$, $|v| > 0$, and for all $i \geq 0$, $xuv^iwz \in L$.
4. Choose $i = 2$. Since v contains at least one a and nothing but as, uv^2w has more as than uvw. So xuv^2wz has more as after the bs than before them, and thus $xuv^2wz \notin L$.
5. By contradiction, $L = \{xx^R\}$ is not regular.

Notice that Steps 1, 3, and 5 are the same as before—only the language L has been changed. In fact, Steps 1, 3, and 5 are the same in all pumping-lemma proofs. The only steps that require creativity are Steps 2 and 4. In step 2, you choose xyz and show that you have met the requirements $xyz \in L$ and $|y| \geq k$; the trick here is to choose them so that pumping in the y part will lead to a contradiction, a string that is not in L. In Step 4 you choose i, the number of times to pump, and show that the contradiction has been achieved: $xuv^iwz \notin L$.

11.5 Strategies

Proving that a language L is not regular using the pumping lemma comes down to those four delicate choices: you must choose the strings xyz and the pumping count i and show that these choices lead to a contradiction. There are usually a number of different choices that successfully lead to a contradiction—and, of course, many others that fail.

Let $A = \{a^n b^j a^n \mid n \geq 0, j \geq 1\}$, and let's try to prove that A is not regular. The start is always the same:

1. The proof is by contradiction using the pumping lemma for regular languages. Assume that A is regular, so the pumping lemma holds for it. Let k be as given by the pumping lemma.
2. Choose x, y, and z ...

How to proceed? The following table shows some good and bad choices for x, y, and z:

$x = aaa$ $y = b$ $z = aaa$	Bad choice. The pumping lemma requires $\|y\| \geq k$; it never applies to fixed-size examples. Since k is not known in advance, y must be some string that is constructed using k, such as a^k.
$x = \varepsilon$ $y = a^k$ $z = a^k$	Bad choice. The pumping lemma applies only if the string $xyz \in A$. That is not the case here.
$x = a^n$ $y = b$ $z = a^n$	This is ill-formed, since the value of n is not defined. At this point the only integer variable that is defined is k.
$x = a^k$ $y = b^{k+2}$ $z = a^k$	This meets the requirements $xyz \in A$ and $\|y\| \geq k$, but it is a bad choice because it won't lead to a contradiction. Pumping within the string y will change the number of bs in the middle, but the resulting string can still be in A.
$x = a^k$ $y = bba^k$ $z = \varepsilon$	This meets the requirements $xyz \in A$ and $\|y\| \geq k$, but it is a bad choice because it won't lead to a contradiction. The pumping lemma can choose any $uvw = y$ with $\|v\| > 0$. If it chooses $u = b$, $v = b$, and $w = a^k$, there will be no contradiction, since for all $i \geq 0$, $xuv^iwz \in A$.
$x = a^k b$ $y = a^k$ $z = \varepsilon$	Good choice. It meets the requirements $xyz \in A$ and $\|y\| \geq k$, and it will lead to a contradiction because pumping anywhere in the y part will change the number of as after the b, without changing the number before the b.
$x = \varepsilon$ $y = a^k$ $z = ba^k$	An equally good choice.

Let's use that last choice and continue with Step 2:

2. Choose x, y, and z as follows:

   $x = \varepsilon$
   $y = a^k$
   $z = ba^k$

   Now $xyz = a^kba^k \in A$ and $|y| \geq k$ as required.

3. Let u, v, and w be as given by the pumping lemma, so that $uvw = y$, $|v| > 0$, and for all $i \geq 0$, $xuv^iwz \in A$.

4. Choose $i = \ldots$

Now, how to choose i? In this example it is easy, since there is only one bad choice. For $i = 1$, $xuv^iwz = xyz \in A$ and no contradiction is reached. For any other choice of i, however, the string xuv^iwz will have a different number of *a*s on each side of the b, so it will not be in A. So we will choose $i = 2$ and continue:

4. Choose $i = 2$. Since v contains at least one a and nothing but *a*s, uv^2w has more *a*s than uvw. So xuv^2wz has more *a*s after the b than before it, and thus $xuv^2wz \notin A$.

5. By contradiction, A is not regular.

11.6 Pumping and Finite Languages

The pumping lemma applies in a trivial way to all finite languages. Here is our statement of the pumping lemma again:

> For all regular languages L there exists some $k \in \mathcal{N}$ such that for all $xyz \in L$ with $|y| \geq k$, there exist $uvw = y$ with $|v| > 0$, such that for all $i \geq 0$, $xuv^iwz \in L$.

If the language L is finite then there must be some number which is greater than the length of the longest string in L. Choosing k to be this number, it is then obviously true that "for all $xyz \in L$ with $|y| \geq k$, ..."—since there are *no* strings in L with $|y| \geq k$. The rest of the lemma is, as mathematicians say, *vacuously true.*

Thus, if L is a finite language, we can't get a contradiction out of the pumping lemma. In fact, you may already have noticed the following:

> **Theorem 11.3:** All finite languages are regular.
>
> **Proof:** Let A be any finite language of n strings: $A = \{x_1, \ldots, x_n\}$. There is a regular expression that denotes this language: $A = L(x_1 + \ldots + x_n)$. (In case $n = 0$, $A = L(\varnothing)$.) Since A is denoted by some regular expression, we conclude that A is a regular language.

Alternative proofs using DFAs, NFAs, or right-linear grammars are only a little harder.

Exercises

EXERCISE 1

Prove that $\{a^n b^n c^n\}$ is not regular. *Hint:* Copy the proof of Theorem 11.1—only minor alterations are needed.

EXERCISE 2

Prove that $\{a^n b^* c^n\}$ is not regular. *Hint:* Copy the proof of Theorem 11.1—only minor alterations are needed.

EXERCISE 3

Show that $\{xx^R\}$ is regular when $|\Sigma| = 1$.

EXERCISE 4

Show that $\{xcx^R \mid x \in \{a, b\}^*\}$ is not regular. *Hint:* Copy the proof of Theorem 11.2—only minor alterations are needed.

EXERCISE 5

Show that $\{xx \mid x \in \{a, b\}^*\}$ is not regular. *Hint:* Copy the proof of Theorem 11.2—only minor alterations are needed.

EXERCISE 6

Let $A = \{a^n b^{2n} \mid n \geq 0\}$. Using the pumping lemma for regular languages, prove that A is not regular.

EXERCISE 7

Let $B = \{a^n b^n c^n \mid n \geq 0\}$. Using the pumping lemma for regular languages, prove that B is not regular.

EXERCISE 8

Let $C = \{0^n 1^m 0^p \mid n + m = p\}$. Using the pumping lemma for regular languages, prove that C is not regular.

EXERCISE 9

Let $D = \{a^n b^m \mid n \geq m\}$. Using the pumping lemma for regular languages, prove that D is not regular.

EXERCISE 10

Let $E = \{a^n b^m \mid m \geq n\}$. Using the pumping lemma for regular languages, prove that E is not regular.

EXERCISE 11

Let $F = \{a^n \mid n = b^2 \text{ for some integer } b\}$. Using the pumping lemma for regular languages, prove that F is not regular.

12 Context-free Languages

We defined the right-linear grammars by giving a simple restriction on the form of each production. By relaxing that restriction a bit, we get a broader class of grammars: the **context-free grammars**. *These grammars generate the* **context-free languages**, *which include all the regular languages along with many that are not regular.*

12.1 Context-free Grammars and Languages

Many of the nonregular languages we have seen can be generated by grammars. For example, here is a grammar for $\{a^n b^n\}$:

$$S \rightarrow aSb \mid \varepsilon$$

This is a grammar for $\{xx^R \mid x \in \{a, b\}^*\}$:

$$S \rightarrow aSa \mid bSb \mid \varepsilon$$

And this is a grammar for $\{a^n b^j a^n \mid n \geq 0, j \geq 1\}$:

$$\begin{aligned} &S \rightarrow aSa \mid R \\ &R \rightarrow bR \mid b \end{aligned}$$

These grammars are not right linear nor left linear, but they do obey another, simpler restriction: they are *context free*. A context-free grammar is one in which the right-hand side of every production is a single nonterminal symbol.

A grammar $G = (V, \Sigma, S, P)$ is a context-free grammar (CFG) if and only if every production in P is of the form

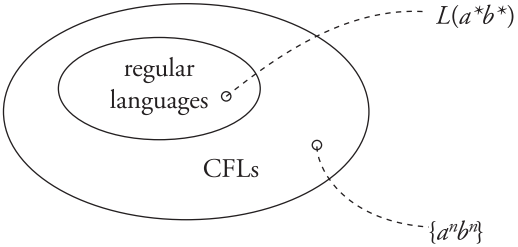
$$X \rightarrow z$$

where $X \in V$ and $z \in (V \cup \Sigma)^*$.

Why is this called *context free*? A production like $uRz \rightarrow uyz$ specifies that R can be replaced by y, but only in a specific context—only when there is a u to the left and a z to the right. In a context-free grammar, all productions look like $R \rightarrow y$, so they specify a substitution for a nonterminal that does not depend on the context of surrounding symbols in the string.

A context-free language (CFL) is one that is $L(G)$ for some CFG G.

Because every regular language has a right-linear grammar and every right-linear grammar is a CFG, it follows that every regular language is a CFL. But, as the examples above show, CFGs can generate languages that are not regular, so not every CFL is regular. The CFLs properly contain the regular languages, like this:



12.2 Writing CFGs

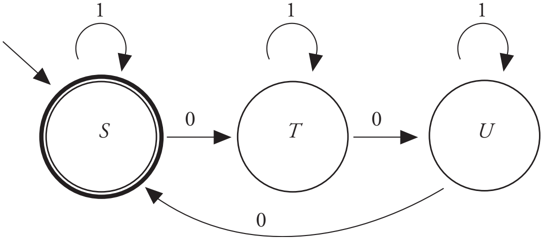
Writing a CFG is a bit like writing a program; it uses some of the same mental muscles. A program is a finite, structured, mechanical thing that specifies a potentially infinite collection of runtime behaviors. To write a program, you have to be able to imagine how the code you are crafting will unfold when it executes. Similarly, a grammar is a finite, structured, mechanical thing that specifies a potentially infinite language. To write a grammar, you have to be able to imagine how the productions you are crafting will unfold in derivations of terminal strings. The derivation mechanism is simple enough, but it takes practice to develop the skill of writing a CFG for a given CFL. This section gives some techniques and examples.

12.2.1 Regular Languages

If the language is regular, we already have a technique for constructing a CFG: start with an NFA and apply the construction of Theorem 10.1. For example, consider the language

$$L = \{x \in \{0, 1\}^* \mid \text{the number of 0s in } x \text{ is divisible by } 3\}$$

Here is an NFA for L:



Now we apply the construction of Theorem 10.1. Wherever the NFA has $Y \in \delta(X, z)$, we add the production $X \rightarrow zY$, and for each accepting state X in the NFA, we add $X \rightarrow \varepsilon$. The result is this grammar for L:

$S \rightarrow 1S \mid 0T \mid \varepsilon$
$T \rightarrow 1T \mid 0U$
$U \rightarrow 1U \mid 0S$

It may be possible to design a smaller or otherwise more readable CFG manually, without using the construction. Here is another grammar for L, not derived from an NFA:

$S \rightarrow T0T0T0S \mid T$
$T \rightarrow 1T \mid \varepsilon$

The construction of Theorem 10.1 does not always produce a pretty grammar, but it is simple and certain—assuming you already have an NFA!

12.2.2 Balanced Pairs

CFLs often seem to involve strings that contain balanced pairs. For example, in $\{a^n b^n\}$, each a can be paired with a b on the other side of the string; in $\{xx^R\}$, each symbol in x can be paired with its mirror image in x^R; in $\{a^n b^j a^n \mid n \geq 0, j \geq 1\}$; each a on the left can be paired with a matching a on the right. The intuition that CFLs involve strings with balanced pairs can be made rigorous, as we will see in a later chapter. For now, we just observe that this kind of balancing is accomplished in the grammar with a simple trick: a recursive production of the form $R \rightarrow xRy$. Such a production generates any number of xs, each of which is matched with a y on the other side. Look again at the grammars at the beginning of this chapter; each uses the same trick.

For another example, consider the language $\{a^n b^{3n}\}$. This is the subset of $L(a^* b^*)$ where the number of bs is three times the number of as. If you think of pairing each a with 3 bs, you can use the trick for balanced pairs to get this grammar:

$S \rightarrow aSbbb \mid \varepsilon$

One more example: consider the language

$$L = \{xy \mid x \in \{a, b\}^*, y \in \{c, d\}^*, \text{ and } |x| = |y|\}$$

If you think of pairing each symbol of x (which can be either a or b) with a symbol of y (which can be either c or d), you can use the trick for balanced pairs to get this grammar:

$S \rightarrow XSY \mid \varepsilon$
$X \rightarrow a \mid b$
$Y \rightarrow c \mid d$

12.2.3 Concatenations

A divide-and-conquer approach is often helpful for writing CFGs, just as for writing programs. For instance, consider the language $L = \{a^n b^n c^m d^m\}$. At first glance this may look daunting, but notice that we can easily write grammars for $\{a^n b^n\}$ and $\{c^m d^m\}$:

$S_1 \rightarrow aS_1b \mid \varepsilon$

$S_2 \rightarrow cS_2d \mid \varepsilon$

These are two separate grammars with two separate start symbols S_1 and S_2. Now every string in L consists of a string generated from S_1 followed by a string generated from S_2. So if we combine the two grammars and introduce a new start symbol, we get a full grammar for L:

$$\begin{aligned}&S \to S_1S_2\\&S_1 \to aS_1b \mid \varepsilon\\&S_2 \to cS_2d \mid \varepsilon\end{aligned}$$

In general, when you discover that a CFL L can be thought of as the concatenation of two languages L_1 and L_2

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

you can write a CFG for L by writing separate CFGs for L_1 and L_2, carefully keeping the two sets of nonterminal symbols separate and using two separate start symbols S_1 and S_2. A full grammar for L is then given by combining all the productions and adding a new start symbol S with the production $S \to S_1S_2$.

12.2.4 Unions

Another use of the divide-and-conquer approach is for a language that can be decomposed into the union of two simpler languages. Consider this language:

$$L = \{z \in \{a, b\}^* \mid z = xx^R \text{ for some } x\text{, or } |z| \text{ is odd}\}$$

Taken all at once this might be difficult, but notice that the definition of L can be expressed as a union:

$$L = \{xx^R \mid x \in \{a, b\}^*\} \cup \{z \in \{a, b\}^* \mid |z| \text{ is odd}\}$$

We can easily give CFGs for these two parts—again, being careful to use separate sets of nonterminal symbols and separate start symbols. A grammar for $\{xx^R \mid x \in \{a, b\}^*\}$ is

$$S_1 \to aS_1a \mid bS_1b \mid \varepsilon$$

And a grammar for $\{z \in \{a, b\}^* \mid |z| \text{ is odd}\}$ is

$$\begin{aligned}&S_2 \to XXS_2 \mid X\\&X \to a \mid b\end{aligned}$$

So a grammar for the whole language L is

$$\begin{aligned}&S \to S_1 \mid S_2\\&S_1 \to aS_1a \mid bS_1b \mid \varepsilon\\&S_2 \to XXS_2 \mid X\\&X \to a \mid b\end{aligned}$$

For a slightly more subtle example, consider the language $L = \{a^nb^m \mid n \neq m\}$. One way to build a grammar for this is to think of it as a union:

$$L = \{a^n b^m \mid n < m\} \cup \{a^n b^m \mid n > m\}$$

In fact, to carry the idea one step further, we can think of each of the two parts of that union as a concatenation:

$$L_1 = \{a^n b^n\}$$
$$L_2 = \{b^i \mid i > 0\}$$
$$L_3 = \{a^i \mid i > 0\}$$
$$L = L_1 L_2 \cup L_3 L_1$$

A string in L is a string in $\{a^n b^n\}$ followed by at least one additional b, or a string in $\{a^n b^n\}$ preceded by at least one additional a. This expresses L as a combination of three languages, each of which is easy to make a CFG for.

$S \to S_1 S_2 \mid S_3 S_1$
$S_1 \to a S_1 b \mid \varepsilon$ (S_1 generates L_1)
$S_2 \to b S_2 \mid b$ (S_2 generates L_2)
$S_3 \to a S_3 \mid a$ (S_3 generates L_3)

12.3 CFG Applications: Backus-Naur Form

Programming languages contain many “balanced pair” constructs, like the balanced parentheses in an arithmetic expression and the balanced braces or `begin`/`end` keyword pairs that delimit nested blocks. It is therefore not surprising to find that context-free grammars play a central role in the definition and implementation of programming languages.

Between 1959 and 1963, John Backus and Peter Naur developed a way to use grammars to define the syntax of programming languages. This notation was used in their reports on the Algol languages. It is now called Backus-Naur Form (BNF).

BNF grammars are really just CFGs. They have the same four parts: a set of terminal symbols (called *tokens*), a set of nonterminal symbols, a start symbol, and a set of productions. The productions are written with a slightly different notation:

- Instead of uppercase letters, nonterminal symbols are written as words enclosed in angle brackets.
- Instead of the arrow symbol (→), the symbol ::= separates the left and right sides of each production
- Instead of ε, the empty string is written as <*empty*>.

Except for these cosmetic differences, a BNF grammar is a CFG. (Interestingly, although Noam Chomsky had developed CFGs a few years earlier, Backus and Naur were unaware of Chomsky’s work and developed their notation independently.) Here is an example of a BNF definition for a little language of expressions:

<*exp*> ::= <*exp*> - <*exp*> | <*exp*> * <*exp*> | <*exp*> = <*exp*> | <*exp*> < <*exp*>
 | (<*exp*>) | a | b | c

This grammar generates expressions like `a<b` and `(a-(b*c))`. The same technique is used to define all the constructs of a language, as in this fragment of a grammar for a C-like language:

<*stmt*> ::= <*exp-stmt*> | <*while-stmt*> | <*compound-stmt*> |...
<*exp-stmt*> ::= <*exp*> ;
<*while-stmt*> ::= while (<*exp*>) <*stmt*>
<*compound-stmt*> ::= { <*stmt-list*> }
<*stmt-list*> ::= <*stmt*> <*stmt-list*> | <*empty*>

This grammar generates C-like statements such as

```
while (a<b) {
  c = c * a;
  a = a + a;
}
```

The BNF grammar for a full language may include hundreds of productions.

12.4 Parse Trees

Grammars and regular expressions *generate* languages; DFAs, NFAs, and other automata *recognize* languages. Either way, all these formalisms *define* languages in a purely syntactic way. They define sets of strings, without ascribing meaning to them.

A programming language, by contrast, is much more than just a definition of syntax. Whether by direct interpretation or by compilation into a lower-level language, a program specifies a computation and so has a meaning, a semantics. Programs are supposed to do something, and that makes them very different from the formal languages studied in this book. The BNF grammar for a programming language plays a dual role here, specifying not only syntax but a bit of semantics as well. To see how this works, we need to look at the idea of the *parse tree*.

We have treated the productions of a grammar as rules that specify how strings can be rewritten, so that a rule $R \rightarrow x$ specifies that wherever you see the nonterminal symbol R you can replace it with x. Another way to think of the productions in a CFG is as rules for building trees. Imagine building a tree with the start symbol at the root. Each production $R \rightarrow x$ specifies that the symbols in x may be installed as children of the nonterminal symbol R. You add children to nonterminal nodes, always following the productions of the grammar, and stop only when all the leaves are terminal symbols. You then read off the fringe of the tree from left to right. These two different ways of thinking of a grammar are equivalent in the sense that a string is produced by some derivation if and only if it is the fringe of some parse tree.

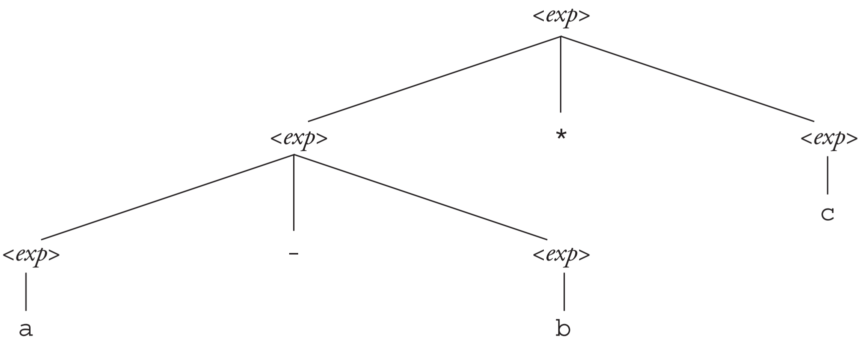
For example, consider again this grammar for a simple language of expressions:

```
<exp> ::= <exp> - <exp> | <exp> * <exp> | <exp> = <exp> | <exp> < <exp>
                | (<exp>) | a | b | c
```

The string `a-b*c` is in the language generated by this grammar. Our usual way to demonstrate this has been to give a derivation, like this:

```
<exp> ⇒ <exp> * <exp>
      ⇒ <exp> - <exp> * <exp>
      ⇒ a- <exp> * <exp>
      ⇒ a-b* <exp>
      ⇒ a-b*c
```

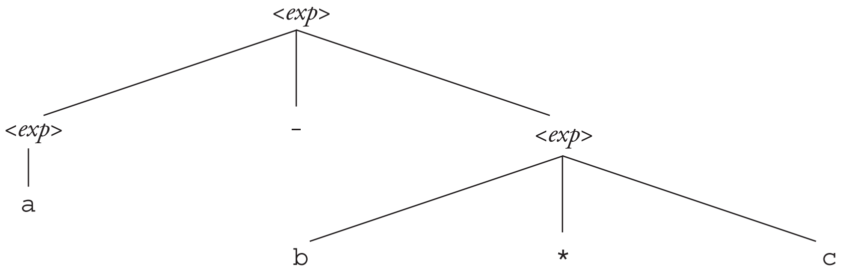
Here is a parse tree for the same string:



This parse tree is more than just a demonstration that the string `a-b*c` is in the language—it is also a plan for evaluating the expression when the program is run. It says to evaluate *a - b*, then multiply that value by *c*. (That's not how most programming languages handle `a-b*c`, of course—more about that below!) In general, a parse tree specifies not just the syntax of a program, but how the different parts of the program fit together. This in turn says something about what happens when the program runs. The parse tree (or a simplified version called the *abstract syntax tree*) is one of the central data structures of almost every compiler or other programming language system. To *parse* a program is to find a parse tree for it. Every time you compile a program, the compiler must first parse it. We will see more about algorithms for parsing in another chapter.

12.5 Ambiguity

The previous grammar for expressions is *ambiguous* in the sense that it permits the construction of different parse trees for the same string. For example, consider our string a-b*c. We built one parse tree for it, but here is another one:

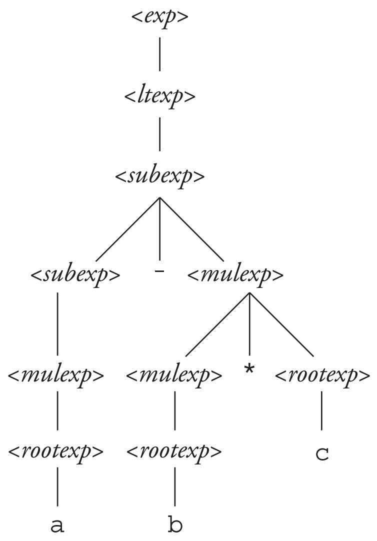


This parse tree suggests a different plan for evaluating the expression: first compute the value *b* × *c*, then subtract that value from *a*. This kind of ambiguity is unacceptable for programming languages; part of the definition of the language must be a clear decision about whether a-b*c means (*a* - *b*) × *c* or *a* - (*b* × *c*).

To resolve this problem, BNF grammars for programming languages are usually crafted to be *unambiguous.* They not only specify the intended syntax, but do so with a unique parse tree for each program, one that agrees with intended semantics. This is not usually difficult, but it generally means making the grammar more complicated. For example, the following grammar generates the same language as the previous one, but now does so unambiguously:

```
<exp> ::= <ltexp> = <exp> | <ltexp>
<ltexp> ::= <ltexp> < <subexp> | <subexp>
<subexp> ::= <subexp> - <mulexp> | <mulexp>
<mulexp> ::= <mulexp> * <rootexp> | <rootexp>
<rootexp> ::= (<exp>) | a | b | c
```

Using this grammar, a parse tree for a-b*c is



Like most programming language definitions, this gives the multiplication operator higher *precedence* than the subtraction operator, so that the expression a-b*c is computed as *a* - (*b* × *c*). It also makes subtraction *left associative*, so that the expression a-b-c is computed as (*a* - *b*) - *c*.

An alternative solution to the problem of ambiguity is to stick with the ambiguous, simple grammar, but add some text describing how to choose among the possible parse trees. In our case, we could stick with our first grammar but add a sentence saying that each operator is at a separate level of precedence, in order =, <, -, and * from lowest to highest, and that all except = are left associative. If the grammar is meant only for human consumption, this approach may make sense. But often, grammars are meant to be used directly by computer programs, like parsers and parser generators, about which we will see more in another chapter. For such uses it is generally better for the grammar to be unambiguous. (Incidentally, there are CFLs for which it is not possible to give an unambiguous grammar. Such languages are called *inherently ambiguous*. But programming languages rarely suffer from this problem.)

12.6 Extended BNF

Plain BNF has a few special symbols, like ::= and |, which are understood to be part of the BNF mechanism rather than part of the language being defined. Such special symbols are called *metasymbols*. A few more metasymbols can be added to BNF to help with common patterns of language definition. For example, we might add [,], {, }, (, and) as metasymbols with the following special meanings:

- [*something*] in the right-hand side of a production means that the *something* inside is optional.

- { *something* } in the right-hand side of a production means that the *something* inside can be repeated any number of times (zero or more). This is like the Kleene star for regular expressions.
- Parentheses are used to group things on the right-hand side, so that |, [], and {} can be used in the same production unambiguously.
- Single quotes are used around tokens when necessary to distinguish them from metasymbols, as in <*empty-list*> ::= '[]'.

With these new metasymbols, some common patterns can be defined more simply than with plain BNF. For example, an `if-then` statement with an optional `else` part might be defined like this:

<*if-stmt*> ::= `if` <*expr*> `then` <*stmt*> [`else` <*stmt*>]

Remember that the square brackets are not part of the language being defined; they are metasymbols that make the `else` part optional. A list of zero or more statements, each ending with a semicolon, might have a definition like this:

<*stmt-list*> ::= {<*stmt*> `;`}

Again, the curly brackets are not part of the language being defined; they are metasymbols that allow the <*stmt*> `;` part to repeat zero or more times. A list of zero or more things, each of which can be either a statement or a declaration and each ending with a semicolon, might have a definition like this:

<*thing-list*> ::= { (<*stmt*> | <*declaration*>) `;`}

The parentheses are metasymbols; they make it clear that the `;` token is not part of the choice permitted by the | metasymbol.

Plain BNF can define all these patterns easily enough, but extra metasymbols like those above make the grammar easier to read. Much like the | metasymbol from plain BNF, the new metasymbols allow you to express a collection of productions succinctly. Any grammar syntax that extends BNF in this way is called an EBNF (extended BNF). Many variations have been used, and there is no widely accepted standard. In formal language terms, an EBNF is a bit like a CFG augmented with regular expressions in the right-hand sides of the productions. Since we know that CFGs can already define all the regular languages, it is not surprising that this augmentation does not really add any power. Any language construct that can be defined with an EBNF can also be defined with plain BNF. But EBNF can define various language constructs more conveniently.

Although EBNF grammars can be easier for people to read and write, they are not without their drawbacks. In particular, the use of the {} metasymbols obscures the form of the parse tree. In a previous example we used the BNF productions:

<*mulexp*> ::= <*mulexp*> `*` <*rootexp*> | <*rootexp*>

This could be simplified using EBNF:

<mulexp> ::= *<rootexp>* {* *<rootexp>*}

But while the BNF grammar allows only a left-associative parse tree for a sequence of multiplications like `a*b*c`, the EBNF grammar is unclear about associativity. One can try to establish conventions—asserting that, by convention, an EBNF production of the form above implies left associativity—but many variations of such conventions have been used, and there is no widely accepted standard.

12.7 Exercises

EXERCISE 1

Give CFGs for the following languages.

a. $L(a^*b^*)$

b. $\{x \in \{0, 1\}^* \mid \text{the number of 0s in } x \text{ is divisible by } 3\}$

c. $\{x \in \{0, 1\}^* \mid |x| \text{ is divisible by } 3\}$

d. $\{x \in \{0, 1\}^* \mid x \text{ is a binary representation of a number divisible by } 3\}$

e. $\{x \in \{a, b\}^* \mid x \notin L(a^*b^*)\}$

f. $\{a^n b^m \mid m > 2n\}$

g. $\{a^n b^n \mid n \text{ is even}\}$

h. $\{(ab)^n c(de)^n\}$

i. $\{a^n b^n \mid n \text{ is odd}\}$

j. $\{a^i b^n a^j b^n a^k\}$

k. $\{a^n b^n a^i b^j\}$

l. $\{x \in \{0, 1\}^* \mid \text{the number of 0s in } x \text{ is divisible by } 3,$
$\text{or } |x| \text{ is divisible by } 3, \text{ or both}\}$

m. $\{a^n b^m \mid m > 2n \text{ or } m < n\}$

n. $\{x \in \{a, b\}^* \mid x \notin \{a^n b^n\}\}$

o. $\{a^n b^n c^j\}$

p. $\{a^n b^{2n}\}$

q. $\{a^n b^m \mathrm{c}^m d^n\}$

r. $\{a^n b^n \mathrm{c}^m d^m\}$

s. $\{x \in \{a, b\}^* \mid x = x^R\}$

EXERCISE 2

Give a BNF grammar for each of the languages below:

a. The set of all strings consisting of zero or more `a`s.

b. The set of all strings consisting of an uppercase letter followed by zero or more additional characters, each of which is either an uppercase letter or one of the characters `0` through `9`.

c. The set of all strings consisting of one or more `a`s.

d. The set of all strings consisting of one or more digits. (Each digit is one of the characters `0` through `9`.)

e. The set of all strings consisting of zero or more `a`s with a semicolon after each one.

f. The set of all strings consisting of the keyword `begin`, followed by zero or more statements with a semicolon after each one, followed by the keyword `end`. Use the nonterminal <*statement*> for statements, and do not give productions for it.

g. The set of all strings consisting of one or more `a`s with a semicolon after each one.

h. The set of all strings consisting of the keyword `begin`, followed by one or more statements with a semicolon after each one, followed by the keyword `end`. Use the nonterminal <*statement*> for statements, and do not give productions for it.

i. The set of all strings consisting of one or more `a`s, with a comma between each `a` and the next. (There should be no comma before the first or after the last `a`.)

j. The set of all strings consisting of an open bracket (the symbol `[`) followed by a list of one or more digits separated by commas, followed by a closing bracket (the symbol `]`).

k. The set of all strings consisting of zero or more `a`s, with a comma between each `a` and the next. (There should be no comma before the first or after the last `a`.)

l. The set of all strings consisting of an open bracket (the symbol `[`) followed by a list of zero or more digits separated by commas, followed by a closing bracket (the symbol `]`).

EXERCISE 3

Give an EBNF grammar for each of the languages of Exercise 1. Use the EBNF extensions wherever possible to simplify the grammars. In particular, you should eliminate recursion from the grammars wherever possible. Don't forget to put quotation marks around metasymbols when they are used as tokens.

EXERCISE 4

Show that each of the following BNF grammars is ambiguous. (To show that a grammar is ambiguous, you must demonstrate that it can generate two parse trees for the same string.)

a. This grammar for expressions:

```
<exp> ::= <exp> + <exp>
        | <exp> * <exp>
        | ( <exp> )
        | a | b | c
```

b. This grammar:

```
<person> ::= <woman> | <man>
<woman> ::= wilma | betty | <empty>
<man> ::= fred | barney | <empty>
```

c. The following grammar for strings of balanced parentheses. (A language of any number of different kinds of balanced parentheses is called a Dyck language. Dyck languages play an interesting role in the theory of formal language, because they isolate the balanced-pair concept that is the essence of the CFLs.)

<*s*> ::= <*s*> <*s*> | (<*s*>) | ()

d. This grammar:

<*s*> ::= <*round*> <*square*> | <*outer*>
<*round*> ::= (<*round*>) | ()
<*square*> ::= [<*square*>] | []
<*outer*> ::= (<*outer*>] | (<*inner*>]
<*inner*> ::=) <*inner*> [|) [

EXERCISE 5

For each of the grammars in Exercise 4 *except the last*, give an unambiguous grammar for the same language. (The last language in that exercise is a classic example of an inherently ambiguous language—the grammar cannot be fixed!)

EXERCISE 6

We claimed without proof that for this grammar G:

$$S \to aSb \mid \varepsilon$$

we have $L(G) = \{a^n b^n\}$. Prove it.

EXERCISE 7

We claimed without proof that for this grammar G:

$$S \to aSa \mid bSb \mid \varepsilon$$

we have $L(G) = \{xx^R \mid x \in \{a, b\}^*\}$. Prove it.

13

Stack Machines

Commonly, the word **stack** *refers to any orderly, vertical pile: a stack of books, plates, or poker chips. All the action in a stack is at the top, where items can be added or removed. Throughout computer science, the word* **stack** *is used in a related technical sense: a stack is a collection of data accessed in last-in-first-out order. Data items may be added to the top (***pushed** *onto the stack) or removed from the top (***popped** *off the stack).*

Stacks are ubiquitous in computer programming, and they have an important role in formal language as well. A **stack machine** *is a kind of automaton that uses a stack for auxiliary data storage. The size of the stack is unbounded—it never runs out of space—and that gives stack machines an edge over finite automata. In effect, stack machines have infinite memory, though they must use it in stack order.*

If you travel by two paths that seem to depart in different directions, it is a surprise to discover that they lead to the same destination. It makes that destination feel more important—an intersection rather than a dead end. That is the situation with context-free languages. Stack machines and CFGs seem like two very different mechanisms for language definition—two paths that depart in different directions. But it turns out that these two paths lead to the same place. The set of languages that can be defined using a stack machine is exactly the same as the set of languages that can be defined using a CFG: the context-free languages.

13.1 Stack Machine Basics

A stack machine is a very different kind of automaton for defining languages. Unlike a DFA or NFA, it makes no state transitions; instead, it maintains a stack of symbols. We will represent this stack as a string, treating the left end of the string as the top of the stack. For example, the string *abc* represents a stack with *a* on top and *c* on the bottom; popping the *a* off the top of *abc* leaves the stack *bc*, while pushing *b* onto *abc* produces the stack *babc*.

A stack machine is specified by a table that shows the moves it is allowed to make. For example:

read	pop	push
a	*c*	*abc*

The entry in the first column is an input symbol (or ε—more about this below). The entry in the second column is a stack symbol, and the entry in the third column is a string of stack symbols. The example above says

> If the current input symbol is *a*, and if the symbol on top of the stack is *c*, you may pop the *c* off the stack, push the string *abc* on in its place, and advance to the next input symbol.

Remember that the stack is represented as a string with the top symbol at the left end. So when you push a string of two or more symbols onto the stack, it is the leftmost one that becomes the new top. For example, suppose the stack machine's next input symbol is an *a*, and suppose its stack is *cd*. Then the move shown above can be used, leaving the string *abcd* on the stack. (It first pops the *c* off, leaving *d*, then pushes the string *abc* on, leaving *abcd*.) The new top symbol on the stack is the leftmost symbol, the *a*.

Every move of a stack machine pops one symbol off the stack, then pushes a string of zero or more symbols onto the stack. To specify a move that leaves the stack unchanged, you can explicitly push the popped symbol back on, like this:

read	pop	push
a	*c*	*c*

In this case, if the stack machine's next input symbol is an *a*, and its stack is *cd*, then the move shown above can be used, leaving the string *cd* on the stack. (It first pops the *c* off, leaving *d*, then pushes the *c* back on, leaving *cd*.)

Every move of a stack machine pushes some string of symbols onto the stack. To specify a move that pops but does not push, you can explicitly push the empty string, like this:

read	pop	push
a	*c*	ε

In this case, if the stack machine's next input symbol is an *a*, and its stack is *cd*, then the move shown above can be used, leaving the string *d* on the stack. (It first pops the *c* off, leaving *d*, then pushes the empty string back on, still leaving *d*.)

The entry in the first column can be ε. This encodes a move that can be made without reading an input symbol, just like an ε-transition in an NFA:

read	pop	push
ε	*c*	*ab*

The example above says that whenever the stack machine has a *c* on top of the stack, it may pop it off and push the string *ab* onto the stack. This move does not advance to the next input symbol and may even be made after all the input symbols have been read.

A stack machine starts with a stack that contains just one symbol *S*. On each move it can alter its stack, but only in stack order—only by popping from the top and/or pushing onto the top. If the stack machine decides to accept the input string, it signals this by leaving its stack empty, popping everything off including the original bottom-of-stack symbol *S*.

Like an NFA, a stack machine is potentially nondeterministic: it may have more than one legal sequence of moves on a given input string. It accepts if there is at least one legal sequence of moves that reads the entire input string and *ends with the stack empty*—the initial symbol *S*, and anything else that was pushed during the computation, must be popped off to signal that the input string is accepted.

Consider this stack machine:

	read	pop	push
1.	ε	*S*	*ab*
2.	*a*	*S*	*ef*
3.	*a*	*S*	ε

Suppose the input string is *a*. The initial stack is *S*, so all three moves are possible. In fact, there are three possible sequences of moves:

- If move 1 is used as the first move, no input is read and the stack becomes *ab*. From there no further move is defined. This is a rejecting sequence of moves, both because the input was not finished and because the stack is not empty.
- If move 2 is used as the first move, the *a* is read and the stack becomes *ef*; then no further move can be made. This too is a rejecting sequence of moves, because even though all the input was read, the stack is not empty.
- If move 3 is used as the first move, the *a* is read and the stack becomes empty. This is an accepting sequence of moves: the input was all read and the stack is empty at the end.

Because there is at least one accepting sequence of moves for *a*, *a* is in the language defined by this stack machine. (In fact, the language defined by this stack machine is just {*a*}.)

13.2 A Stack Machine for $\{a^nb^n\}$

Here's a general strategy for using a stack machine to recognize the language $\{a^nb^n\}$. Suppose the stack initially contains the symbol S. Now start reading the input string from left to right:

1. For each a you read, pop off the S, push a 1, and then push the S back on.
2. In the middle of the string, pop off the S. (At this point the stack contains just a list of zero or more 1s, one for each a that was read from the input string.)
3. For each b you read, pop a 1 off the stack.

This strategy permits only as until the S is popped and only bs after the S is popped. Moreover, it pushes a 1 for every a and pops a 1 for every b. Thus it finishes with an empty stack if and only if the input string is in $\{a^nb^n\}$.

Here is our strategy, translated into a table of moves for a stack machine:

	read	pop	push
1.	a	S	$S1$
2.	ε	S	ε
3.	b	1	ε

Let's look at the accepting sequence of moves this machine makes for the string $aaabbb$. Initially, the stack machine is reading the first symbol of $aaabbb$ and has S on its stack, as shown here:

$$\underline{a}aabbb \qquad \underline{S}$$

The current input symbol and the top-of-the-stack symbol are underlined; these determine which moves are possible. The possible moves are 1 and 2, but the accepting sequence starts with move 1. After that move, the stack machine has advanced to the second symbol of $aaabbb$ and has $S1$ on its stack, like this:

$$\begin{array}{lcl} \underline{a}aabbb & & \underline{S} \\ & \Downarrow & \text{move 1: pop } S\text{, push } S1\text{, and advance to next input symbol} \\ a\underline{a}abbb & & \underline{S}1 \end{array}$$

The full sequence is

$\underline{a}aabbb \qquad \underline{S}$

↓ move 1: pop S, push $S1$, and advance to next input symbol

$a\underline{a}abbb \qquad \underline{S}1$

↓ move 1: pop S, push $S1$, and advance to next input symbol

$aa\underline{a}bbb \qquad \underline{S}11$

↓ move 1: pop S, push $S1$, and advance to next input symbol

$aaa\underline{b}bb \qquad \underline{S}111$

↓ move 2: pop S

$aaa\underline{b}bb \qquad \underline{1}11$

↓ move 3: pop 1 and advance to next input symbol

$aaab\underline{b}b \qquad \underline{1}1$

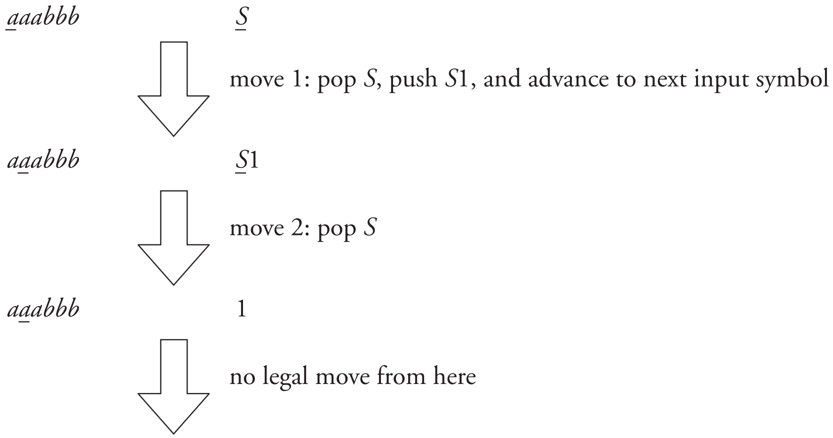
↓ move 3: pop 1 and advance to next input symbol

$aaabb\underline{b} \qquad \underline{1}$

↓ move 3: pop 1 and advance to next input symbol

$aaabbb\underline{\ } \qquad \varepsilon$

At the end, the stack machine has read all its input and emptied its stack. That is an accepting sequence of moves. As we have already seen, there are also many nonaccepting sequences, such as this one:



There is no next move here, so the stack machine terminates without reaching the end of the input string and without emptying its stack. But since there is at least one accepting sequence of moves, the string *aaabbb* is in the language defined by the machine.

13.3 A Stack Machine for $\{xx^R\}$

Here's a general strategy for using a stack machine to recognize the language $\{xx^R \mid x \in \{a, b\}^*\}$. Suppose the stack initially contains the symbol S. Now start reading the input string from left to right:

1. For each *a* you read, pop off the S, push *a*, and then push the S back on.
2. For each *b* you read, pop off the S, push *b*, and then push the S back on.
3. In the middle of the string, pop off the S. (At this point the stack contains just a sequence of *a*s and *b*s—the reverse of the string read so far.)
4. For each *a* you read, pop *a* off the stack.
5. For each *b* you read, pop *b* off the stack.

Before the S is popped, this strategy pushes (beneath the S) a copy of each symbol read. Because of the stack order, the result is that (beneath the S) the stack contains the reverse of the string read so far. After the S is popped, our strategy matches each input symbol against the top symbol of the stack, which is popped off. Thus it finishes with an empty stack if and only if the second part of the input string is the reverse of the first part—that is, if and only if the input string is in $\{xx^R \mid x \in \{a, b\}^*\}$.

Here is our strategy, translated into a table of moves for a stack machine:

	read	pop	push
1.	*a*	*S*	*Sa*
2.	*b*	*S*	*Sb*
3.	ε	*S*	ε
4.	*a*	*a*	ε
5.	*b*	*b*	ε

Here is an accepting sequence of moves for the string *abbbba*:

$\underline{a}bbbba$ $\underline{S}$

↓ move 1: pop *S*, push *Sa*, and advance to next input symbol

$a\underline{b}bbba$ $\underline{S}a$

↓ move 2: pop *S*, push *Sb*, and advance to next input symbol

$ab\underline{b}bba$ $\underline{S}ba$

↓ move 2: pop *S*, push *Sb*, and advance to next input symbol

$abb\underline{b}ba$ $\underline{S}bba$

↓ move 3: pop *S*

$abb\underline{b}ba$ $\underline{b}ba$

↓ move 5: pop *b* and advance to next input symbol

$abbb\underline{b}a$ $\underline{b}a$

↓ move 5: pop *b* and advance to next input symbol

$abbbb\underline{a}$ $\underline{a}$

↓ move 4: pop *a* and advance to next input symbol

*aaabbb*_ ε

Notice that this stack machine can use move 3 at any time (though, since it pops off the only S, it can be used only once). However, in an accepting sequence move 3 must occur exactly in the middle of the input string. We can think of the stack machine as making a guess about where the middle of the string is. When it guesses that it has found the middle, it applies move 3, then begins popping instead of pushing. This takes advantage of the nondeterminism of the stack machine. It accepts if there is any sequence of moves that reads the entire input string and ends with an empty stack. All those sequences that make the wrong guess about where the middle of the string is do not accept. But if the string is in our language, the one sequence that makes the right guess about where the middle of the string is does accept; and that one is all it takes.

13.4 Stack Machines, Formally Defined

We have been working with informal descriptions of stack machines. Now, as always, we need a way to describe them mathematically. This is easy to do, because our informal description using a table of moves is already almost rigorous enough; all that remains is to observe that the table encodes a function and the function captures the essential structure of the machine.

A stack machine M is a 4-tuple $M = (\Gamma, \Sigma, S, \delta)$, where

Γ is the stack alphabet
Σ is the input alphabet
$S \in \Gamma$ is the initial stack symbol
$\delta \in ((\Sigma \cup \{\varepsilon\}) \times \Gamma \to P(\Gamma^*))$ is the transition function

The stack alphabet may or may not overlap with the input alphabet.

The transition function δ takes two parameters. The first parameter is an input symbol or ε, and the second is a stack symbol. These parameters correspond to the first two columns of the tables we used in the previous chapter: the input symbol being read (or ε for ε-transitions) and the symbol currently on top of the stack. The output of the transition function is a set of strings that can be pushed onto the stack. It is a set of strings, rather than just a single string, because a stack machine is nondeterministic: at each stage there may be any number of possible moves. For example, this machine:

	read	pop	push
1.	ε	S	ab
2.	a	S	ef
3.	a	S	ε

has this transition function:

$\delta(\varepsilon, S) = \{ab\}$
$\delta(a, S) = \{\varepsilon, ef\}$

As a stack machine operates, it reads its input string and changes the contents of its stack. At any point in the stack machine's operation, its future depends on two things: that part of the input string that is still to be read, and the current contents of the stack. Taken together, these two pieces of information are an instantaneous description for the stack machine. The following definitions are all made with respect to some fixed stack machine $M = (\Gamma, \Sigma, S, \delta)$:

An instantaneous description for a stack machine is a pair (x, y) where:
$x \in \Sigma^*$ is the unread part of the input
$y \in \Gamma^*$ is the current stack contents
As always, the left end of the string y is considered to be the top of the stack.

The δ function for the stack machine determines a relation $\mapsto$ on IDs; we write $I \mapsto J$ if I is an ID and J is an ID that follows from I after one move of the stack machine.

$\mapsto$ is a relation on IDs, defined by the δ function for the stack machine as follows:

1. Regular transitions: $(ax, Bz) \mapsto (x, yz)$ if and only if $y \in \delta(a, B)$
2. ε-transitions: $(x, Bz) \mapsto (x, yz)$ if and only if $y \in \delta(\varepsilon, B)$.

Note that no move is possible when the stack machine's stack is empty; there is never an ID I with $(x, \varepsilon) \mapsto I$.

Next, as usual, we define an extended relation $\mapsto^*$ for sequences of zero or more steps:

$\mapsto^*$ is a relation on IDs, with $I \mapsto^* J$ if and only if there is a sequence of zero or more $\mapsto$ relations that starts with I and ends with J.

Notice here that $\mapsto^*$ is reflexive: for any ID I, $I \mapsto^* I$ by a sequence of zero moves. Using the $\mapsto^*$ relation, we can define the language accepted by M:

The language accepted by a stack machine M is
$L(M) = \{x \in \Sigma^* \mid (x, S) \mapsto^* (\varepsilon, \varepsilon)\}$

In this definition, the stack machine starts reading its input string x with S on the stack. It accepts the string x if it has some sequence of zero or more moves that ends with all the input used up and the stack empty.

For example, in Section 13.3 we gave an informal description of a stack machine for the language $\{xx^R \mid x \in \{a, b\}^*\}$:

	read	pop	push
1.	a	S	Sa
2.	b	S	Sb
3.	ε	S	ε
4.	a	a	ε
5.	b	b	ε

and we showed an excepting sequence of moves for the string *abbbba*. Formally, this machine is $M = (\{a, b, S\}, \{a, b\}, S, \delta)$, where

$$\begin{aligned}
&\delta(a, S) = \{Sa\}\\
&\delta(b, S) = \{Sb\}\\
&\delta(\varepsilon, S) = \{\varepsilon\}\\
&\delta(a, a) = \{\varepsilon\}\\
&\delta(b, b) = \{\varepsilon\}
\end{aligned}$$

and the accepting sequence of moves that we showed for *abbbba* is

$$\begin{aligned}
&(abbbba, S) \mapsto (bbbba, Sa) \mapsto (bbba, Sba) \mapsto (bba, Sbba)\\
&\quad \mapsto (bba, bba) \mapsto (ba, ba) \mapsto (a, a) \mapsto (\varepsilon, \varepsilon)
\end{aligned}$$

Thus $(abbbba, S) \mapsto^* (\varepsilon, \varepsilon)$ and so, by definition, $abbbba \in L(M)$.

That one example does not, of course, establish that $L(M) = \{xx^R \mid x \in \{a, b\}^*\}$. Proofs of that kind tend to be quite tricky. In the case of our current stack machine, however, we can sidestep explicit induction without too much difficulty. We first observe that only move 3 changes the number of S symbols in the stack, reducing it by 1. So move 3 must be used in any accepting sequence, exactly once. In fact, any accepting sequence must have the form

$$(xy, S) \mapsto^* (y, Sz) \mapsto_3 (y, z) \mapsto^* (\varepsilon, \varepsilon)$$

where x, y, and z are in $\{a, b\}^*$, only moves 1 and 2 occur before the use of move 3, and only moves 4 and 5 occur after it. Moves 1 and 2 push the symbol just read onto the stack; therefore $z = x^R$. Moves 4 and 5 pop *a*s and *b*s only if they match the input; therefore $z = y$. Thus any accepting sequence must have the form

$$(xx^R, S) \mapsto^* (x^R, Sx^R) \mapsto (x^R, x^R) \mapsto^* (\varepsilon, \varepsilon)$$

and there is such an accepting sequence for any string x. We conclude that $L(M) = \{xx^R \mid x \in \{a, b\}^*\}$.

13.5 An Example: Equal Counts

Consider this language:

$$A = \{x \in \{a, b\}^* \mid \text{the number of } a\text{s in } x \text{ equals the number of } b\text{s in } x\}.$$

This is not the same as $\{a^n b^n\}$, since it includes strings like *abab* that have the *a*s and *b*s mixed together.

For a first attempt at making a stack machine for this language, we can try the following strategy. Whenever the machine sees an *a*, it pushes a 0 onto the stack. Whenever is sees a *b*, it pops a 0 off the stack. So if the number of *a*s is the same as the number of *b*s, the machine ends up with just *S* on the stack. It pops that off as its last move. That strategy is embodied in this stack machine, which we'll call M_1:

	read	pop	push
1.	a	S	$0S$
2.	a	0	00
3.	b	0	ε
4.	ε	S	ε

Here, moves 1 and 2 allow it to push a 0 for each *a* read. (Two moves are required to handle the two possible symbols on top of the stack, *S* and 0.) Move 3 allows it to pop a 0 for each *b* read. Move 4 allows it to pop the S off. Since move 4 is the only way to get rid of the S, it must occur in any accepting sequence; and since it leaves the stack empty, no move can follow it. Thus it is always the last move in any accepting sequence.

Formally, this is $M_1 = (\{0, S\}, \{a, b\}, S, \delta)$, where the δ function is

$$\delta(a, S) = \{0S\}$$
$$\delta(a, 0) = \{00\}$$
$$\delta(b, 0) = \{\varepsilon\}$$
$$\delta(\varepsilon, S) = \{\varepsilon\}$$

Here is a sequence of IDs showing that M_1 accepts *abab*:

$$(abab, S) \mapsto (bab, 0S) \mapsto (ab, S) \mapsto (b, 0S) \mapsto (\varepsilon, S) \mapsto (\varepsilon, \varepsilon)$$

This sequence shows that we have $(abab, S) \mapsto^* (\varepsilon, \varepsilon)$, and so, by definition, $abab \in L(M_1)$.

It is clear that by using the stack as a counter, M_1 checks that the number of *a*s in the string is the same as the number of *b*s. Thus everything M_1 accepts is in our target language *A*. Unfortunately, not everything in *A* is accepted by M_1. Consider its behavior on the input string *abba*. Starting in (*abba*, *S*), it can proceed to (*bba*, 0*S*) and then to (*ba*, *S*). But from there it has no transition for reading the next input symbol, the *b*. Our strategy uses the number of 0s on the stack to represent the number of *a*s so far minus the number of *b*s so far. That will fail if, as in *abba*, there is some prefix with more *b*s than *a*s.

To handle this we need a more sophisticated strategy. We will still keep count using 0s on the stack, whenever the number of *a*s so far exceeds the number of *b*s so far. But when the number of *b*s so far exceeds the number of *a*s so far, we will keep count using 1s. That strategy is embodied in this stack machine, which we'll call M_2:

	read	pop	push
1.	a	S	$0S$
2.	a	0	00
3.	b	0	ε
4.	b	S	$1S$
5.	b	1	11
6.	a	1	ε
7.	ε	S	ε

Formally, this is $M_2 = (\{0, 1, S\}, \{a, b\}, S, \delta)$, where the δ function is

$\delta(a, S) = \{0S\}$
$\delta(a, 0) = \{00\}$
$\delta(b, 0) = \{\varepsilon\}$
$\delta(b, S) = \{1S\}$
$\delta(b, 1) = \{11\}$
$\delta(a, 1) = \{\varepsilon\}$
$\delta(\varepsilon, S) = \{\varepsilon\}$

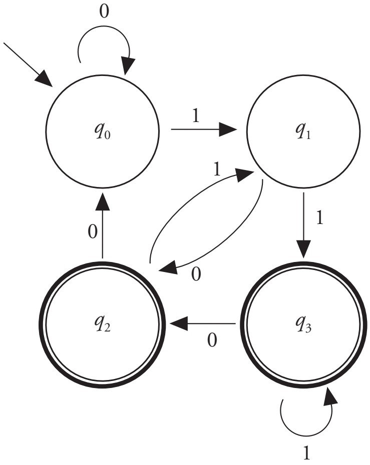
This is now flexible enough to handle strings like *abba*:

$$(abba, S) \mapsto (bba, 0S) \mapsto (ba, S) \mapsto (a, 1S) \mapsto (\varepsilon, S) \mapsto (\varepsilon, \varepsilon)$$

In fact, the machine now accepts our language, $L(M_2) = A$.

13.6 An Example: A Regular Language

A stack machine can easily simulate any finite automaton. Consider the language $\{x \in \{0, 1\}^* \mid \text{the next-to-last symbol in } x \text{ is } 1\}$, which has this DFA:



Our strategy for making a stack machine to simulate a DFA is simple. Until the last move, our machine will always have exactly one symbol on its stack, which will be the current state of the DFA. As its last move, if the DFA is in an accepting state, the stack machine will pop that state off, ending with an empty stack. For the DFA above, our stack machine is $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, q_0, \delta)$, where the δ function is

$\delta(0, q_0) = \{q_0\}$
$\delta(1, q_0) = \{q_1\}$
$\delta(0, q_1) = \{q_2\}$
$\delta(1, q_1) = \{q_3\}$
$\delta(0, q_2) = \{q_0\}$
$\delta(1, q_2) = \{q_1\}$
$\delta(0, q_3) = \{q_2\}$
$\delta(1, q_3) = \{q_3\}$
$\delta(\varepsilon, q_2) = \{\varepsilon\}$
$\delta(\varepsilon, q_3) = \{\varepsilon\}$

Notice that we are using the states of the DFA as the stack alphabet of the stack machine, and we are using the start state q_0 of the DFA as the initial stack symbol. The transition function of the stack machine exactly parallels that of the DFA, so that the stack machine simply simulates the actions of the DFA. For example, the DFA accepts the string 0110 by going through a series of states $(q_0, q_0, q_1, q_3, q_2)$ ending in an accepting state (q_2). The stack machine accepts the string 0110 through the corresponding sequence of IDs:

$$(0110, q_0) \mapsto (110, q_0) \mapsto (10, q_1) \mapsto (0, q_3) \mapsto (\varepsilon, q_2) \mapsto (\varepsilon, \varepsilon)$$

The same kind of construction works, not just for DFAs, but for NFAs as well. You can practice it in some of the following exercises. However, we will not give the formal construction here, since we will now prove an even stronger result.

13.7 A Stack Machine for Every CFG

Consider the string-rewriting process defined by a CFG. From the start symbol S a sequence of replacements is made; at each step, one nonterminal symbol in the string is replaced by a string of symbols as specified by one production of the grammar. The set of all fully terminal strings that can be derived in this way is the language defined by the CFG.

This string-rewriting process can be carried out by a stack machine, using the stack to hold the string being rewritten. At each step, one nonterminal symbol in the stack string is replaced by a string of symbols as specified by one production of the grammar. There's just one hitch: a stack machine can only operate on the top of the stack—the leftmost symbol in the string. If this leftmost symbol is a nonterminal, the stack machine can replace it with something else, tracing out a leftmost derivation. But what if the leftmost symbol is a terminal symbol? In that case the stack machine can simply pop it off, since it is a finished part of the string being derived.

Using this idea, we can take any CFG and construct a stack machine for the same language. The constructed machine starts, as always, with the start symbol S on its stack. It carries out a derivation in the language, repeatedly using two types of moves:

1. If the top symbol on the stack is a nonterminal, replace it using any one of the grammar's productions for that nonterminal. This is an ε-move, not consuming any symbols of the stack machine's input.
2. If the top symbol on the stack is a terminal that matches the next input symbol, pop it off.

Finally, when it has completed a derivation and popped off all the resulting terminal symbols, it is left with an empty stack, and it accepts the input string.

This is a highly nondeterministic machine. The moves of type 1 allow the machine to trace out any derivation from S permitted by the grammar. But the moves of type 2 allow progress only if the derived string matches the input string. The only way the stack can be emptied is if a fully terminal string was derived. The only way the stack machine will accept at that point is if the entire input string has been matched. In effect, the machine can try all possible derivations; it accepts if and only if at least one of them is a derivation of the input string.

For example, consider this CFG for $\{xx^R \mid x \in \{a, b\}^*\}$:

$$S \rightarrow aSa \mid bSb \mid \varepsilon$$

The constructed stack machine corresponding to this CFG is this:

	read	pop	push
1.	ε	S	aSa
2.	ε	S	bSb
3.	ε	S	ε
4.	a	a	ε
5.	b	b	ε

Moves 1 through 3 correspond to the three productions in the grammar. Moves 4 and 5 allow the stack machine to pop the terminals a and b off the stack if they match the input symbols; these moves would be the same in any stack machine constructed from a CFG whose terminal alphabet is $\{a, b\}$.

A string in the language defined by the CFG is *abbbba*. Here is a derivation for the string:

$$S \Rightarrow aSb \Rightarrow abSba \Rightarrow abbSbba \Rightarrow abbbba$$

Here is the corresponding accepting sequence of IDs for the stack machine. The subscripts on each step indicate which move was performed:

$$\begin{aligned}&(abbbba, S) \mapsto_1 (abbbba, aSa) \mapsto_4 (bbbba, Sa) \mapsto_2 (bbbba, bSba)\\&\quad \mapsto_5 (bbba, Sba) \mapsto_2 (bbba, bSbba) \mapsto_5 (bba, Sbba) \mapsto_3 (bba, bba)\\&\quad \mapsto_5 (ba, ba) \mapsto_5 (a, a) \mapsto_4 (\varepsilon, \varepsilon)\end{aligned}$$

We can generalize this construction in the following lemma and proof sketch.

Lemma 13.1: If $G = (V, \Sigma, S, P)$ is any context-free grammar, there is some stack machine M with $L(M) = L(G)$.

Proof sketch: By construction. Let M be the stack machine $M = (V \cup \Sigma, \Sigma, S, \delta)$, where the δ function is defined as follows:

- for all $v \in V$, $\delta(\varepsilon, v) = \{x \mid (v \rightarrow x) \in P\}$, and
- for all $a \in \Sigma$, $\delta(a, a) = \{\varepsilon\}$.

Now M accepts a string if and only if there is a derivation of that string in the grammar G. Thus $L(M) = L(G)$.

A complete proof would have to demonstrate in detail that M accepts a string if and only if there is a derivation of that string in the grammar G. We'll skip that step here, but there is a very similar proof in the next section.

13.8 A CFG for Every Stack Machine

We have just seen that it is possible to take any CFG and construct an equivalent stack machine. The reverse is also true: there is a way to take any stack machine and construct a CFG for the same language. In many ways this is like the previous construction; the executions of the stack machine are exactly simulated by derivations in the CFG.

For example, consider this stack machine from Section 13.5:

	read	pop	push
1.	a	S	$0S$
2.	a	0	00
3.	b	0	ε
4.	b	S	$1S$
5.	b	1	11
6.	a	1	ε
7.	ε	S	ε

The language it accepts is the language of strings over the alphabet $\{a, b\}$ in which the number of *a*s equals the number of *b*s. Here is a grammar that can simulate this stack machine using productions that mimic the stack machine's transitions:

1. $S \rightarrow a0S$
2. $0 \rightarrow a00$
3. $0 \rightarrow b$
4. $S \rightarrow b1S$
5. $1 \rightarrow b11$
6. $1 \rightarrow a$
7. $S \rightarrow \varepsilon$

The productions are numbered here to show how they match the moves of the stack machine. Wherever the stack machine has a move $t \in \delta(\omega, A)$, the grammar has a production $A \rightarrow \omega t$. Thus the derivations in the grammar simulate the executions of the stack machine: each time the stack machine reads an input symbol, the grammar generates that symbol, and the stack machine's stack is simulated in the nonterminal part of the string being rewritten. For example, this is an accepting sequence in the stack machine on the input *abab*:

$$(abab, S) \mapsto_1 (bab, 0S) \mapsto_3 (ab, S) \mapsto_1 (b, 0S) \mapsto_3 (\varepsilon, S) \mapsto_7 (\varepsilon, \varepsilon)$$

(The subscripts indicate which move was used at each step.) The grammar generates the string *abab* using the corresponding sequence of derivation steps:

$$S \Rightarrow_1 a0S \Rightarrow_3 abS \Rightarrow_1 aba0S \Rightarrow_3 ababS \Rightarrow_7 abab$$

At each stage in this derivation, the string has two parts *xy*: *x* is the portion of the input string that has been read by the stack machine, and *y* is the content of the stack. At the end of the derivation the stack machine's stack *y* is empty, and the string *x* is the fully terminal string that was the stack machine's input.

Lemma 13.2.1: If $M = (\Gamma, \Sigma, S, \delta)$ is any stack machine, there is context-free grammar G with $L(G) = L(M)$.

Proof: By construction. We assume that Γ and Σ have no elements in common. (This is without loss of generality since the symbols in Γ can be renamed away from Σ if necessary.) Let G be the grammar $G = (\Gamma, \Sigma, S, P)$, where

$$P = \{(A \rightarrow \omega t) \mid A \in \Gamma, \omega \in (\Sigma \cup \{\varepsilon\}), \text{ and } t \in \delta(\omega, A)\}.$$

Now leftmost derivations in G exactly simulate executions of M, in the sense that for any $x \in \Sigma^*$ and $y \in \Gamma^*$,

$$S \Rightarrow^* xy \text{ if and only if } (x, S) \mapsto^* (\varepsilon, y)$$

(A detailed proof of this is given below.) Setting $y = \varepsilon$, this gives

$S \Rightarrow^* x$ if and only if $(x, S) \mapsto^* (\varepsilon, \varepsilon)$

and so $L(G) = L(M)$.

In the construction, the stack symbols of the stack machine become nonterminals in the constructed grammar, and the input symbols become terminals. That's why we need the assumption that the two alphabets have no symbols in common; no symbol in a grammar can be both a terminal and a nonterminal. The assumption is "without loss of generality" because offending stack symbols can always be renamed if necessary.

For an example demonstrating this kind of renaming, consider this stack machine for the language $\{a^n b^{2n}\}$:

	read	pop	push
1.	a	S	Sbb
2.	ε	S	ε
3.	b	b	ε

This stack machine does not satisfy our assumption, because it uses the symbol b as both an input symbol and a stack symbol. If we applied our construction to this machine we would end up with a grammar containing the production $b \to b$, which is not legal in a CFG. But the problem is easy to solve by renaming: simply replace the stack symbol b with a new symbol such as B:

	read	pop	push
1.	a	S	SBB
2.	ε	S	ε
3.	b	B	ε

Notice that we changed *only the pop and push columns*—only the instances of b on the stack, not the instance of b as an input symbol. This modified stack machine now satisfies our assumption, and we can proceed with the construction, yielding this CFG:

$$S \to aSBB \mid \varepsilon$$
$$B \to b$$

The proof of Lemma 13.2.1 asserts that the leftmost derivations in G exactly simulate the executions of M. The previous examples should help to convince you that this is true, but we can also prove the assertion in more detail using induction. We have avoided most such proofs this far, but since this one is similar to the missing piece in our proof of Lemma 13.1, let's bite the bullet this time and fill in the details.

Lemma 13.2.2: For the construction of the proof of Lemma 13.2.1, for any $x \in \Sigma^*$ and $y \in \Gamma^*$, $S \Rightarrow^* xy$ if and only if $(x, S) \mapsto^* (\varepsilon, y)$.
Proof that if $S \Rightarrow^* xy$ then $(x, S) \mapsto^* (\varepsilon, y)$: By induction on the length of the derivation. In the base case the length of the derivation is zero. The only zero-length derivation $S \Rightarrow^* xy$ is with $xy = S$, and since $x \in \Sigma^*$ we must have $x = \varepsilon$ and $y = S$. For these values, by definition, $(x, S) \mapsto^* (\varepsilon, y)$ in zero steps.

In the inductive case, the length of the derivation is greater than zero, so it has at least one final step. Also, there must be some leftmost derivation of the same string, and a leftmost derivation in G always produces a string of terminals followed by a string of nonterminals. Thus, we have

$$S \Rightarrow^* x'Ay' \Rightarrow xy$$

for some $x' \in \Sigma^*$, $A \in \Gamma$, and $y' \in \Gamma^*$. By the inductive hypothesis (which applies since the derivation of $x'Ay'$ is one step shorter) we know that $(x', S) \mapsto^* (\varepsilon, Ay')$. The final step in the derivation uses one of the productions in P, which by construction are all of the form

$$(A \rightarrow \omega t) \mid A \in \Gamma,\ \omega \in (\Sigma \cup \{\varepsilon\}), \text{ and } t \in \delta(\omega, A)$$

Thus the derivation can be written as

$$S \Rightarrow^* x'Ay' \Rightarrow x'\omega ty' = xy$$

where $x'\omega = x$ and $ty' = y$. Now since $(x', S) \mapsto^* (\varepsilon, Ay')$ we also have $(x'\omega, S) \mapsto^* (\omega, Ay')$, and since $t \in \delta(\omega, A)$ we can add a final step for the stack machine:

$$(x, S) = (x'\omega, S) \mapsto^* (\omega, Ay') \mapsto (\varepsilon, ty') = (\varepsilon, y)$$

This gives $(x, S) \mapsto^* (\varepsilon, y)$ as required.

Proof of the other direction (*if* $(x, S) \mapsto^* (\varepsilon, y)$ *then* $S \Rightarrow^* xy$)**:** Similar, by induction on the number of steps in the execution.

Putting the previous results together, we have

Theorem 13.1: A language is context free if and only if it is $L(M)$ for some stack machine M.
Proof: Follows immediately from Lemmas 13.1 and 13.2.1.

So CFGs and stack machines have equivalent definitional power.

Exercises

EXERCISE 1

How would you change the stack machine of section 13.2 so that the language it accepts is $\{a^n b^n \mid n > 0\}$?

EXERCISE 2

Show the table of moves for a stack machine for the language $\{a^n c b^n\}$.

EXERCISE 3

How would you change the stack machine of Section 13.3 so that the language it accepts is $\{xx^R \mid x \in \{a, b\}^*, x \neq \varepsilon\}$?

EXERCISE 4

Show the table of moves for a stack machine for the language $\{xcx^R \mid x \in \{a, b\}^*\}$.

EXERCISE 5

Show the table of moves for a stack machine for the language $\{(ab)^n(ba)^n\}$.

EXERCISE 6

Give stack machines for the following languages:

a. $L(a^*b^*)$

b. $\{x \in \{0, 1\}^* \mid$ the number of 0s in x is divisible by 3$\}$

c. $\{x \in \{0, 1\}^* \mid |x|$ is divisible by 3$\}$

d. $\{x \in \{0, 1\}^* \mid x$ is a binary representation of a number divisible by 3$\}$

e. $\{x \in \{a, b\}^* \mid x \notin L(a^*b^*)\}$

f. $\{a^n b^m \mid m > 2n\}$

g. $\{a^n b^n \mid n$ is even$\}$

h. $\{(ab)^n c(de)^n\}$

i. $\{a^n b^n \mid n$ is odd$\}$

j. $\{a^i b^n a^j b^n a^k\}$

k. $\{a^n b^n a^i b^j\}$

l. $\{a^n b^m \mid m \geq n\}$

m. $\{a^n b^n c^j\}$

n. $\{a^n b^{2n}\}$

o. $\{a^n b^m c^m d^n\}$

p. $\{a^n b^n c^m d^m\}$

q. $\{x \in \{a, b\}^* \mid x = x^R\}$

EXERCISE 7

Using the construction of Lemma 13.1, give a stack machine for this grammar:

$$S \to S_1 \mid S_2$$
$$S_1 \to aS_1a \mid bS_1b \mid \varepsilon$$
$$S_2 \to XXS_2 \mid X$$
$$X \to a \mid b$$

Show accepting sequences of IDs for your stack machine for *abba* and *bbb*.

EXERCISE 8

Using the construction of Lemma 13.1, give a stack machine for this grammar:

$$S \to T0T0T0S \mid T$$
$$T \to 1T \mid \varepsilon$$

Show accepting sequences of IDs for your stack machine for 110010 and 00011000.

EXERCISE 9

Using the construction of Lemma 13.1, give a stack machine for this grammar:

$$S \to XSY \mid \varepsilon$$
$$X \to a \mid b$$
$$Y \to c \mid d$$

Show accepting sequences of IDs for your stack machine for *abcd* and *abbddd*.

EXERCISE 10

Using the construction of Lemma 13.1, give a stack machine for this grammar:

$$S \to S_1S_2 \mid S_3S_1$$
$$S_1 \to aS_1b \mid \varepsilon$$
$$S_2 \to bS_2 \mid b$$
$$S_3 \to aS_3 \mid a$$

Show accepting sequences of IDs for your stack machine for *aabbb* and *aaabb*.

EXERCISE 11

Give a more elegant grammar for the language recognized by the stack machine in Section 13.8.

EXERCISE 12

Using the construction of Lemma 13.2.1, give a CFG for this stack machine:

	read	pop	push
1.	a	S	$0S$
2.	a	0	00
3.	b	0	ε
4.	ε	S	ε

Show a derivation of *abab* in your grammar.

EXERCISE 13

Using the construction of Lemma 13.2.1, give a CFG for this stack machine:

	read	pop	push
1.	a	S	Sa
2.	b	S	Sb
3.	ε	S	ε
4.	a	a	ε
5.	b	b	ε

(Note that because this machine has stack and input alphabets that overlap, you will have to rename some of the symbols in the stack alphabet to make the construction work.) Show a derivation of *abba* in your grammar.

EXERCISE 14

Give an example of a stack machine for the language $\{a^nb^n\}$ that has exactly two distinct accepting sequences of IDs on every accepted input. Then give an example of a stack machine for the same language that has infinitely many distinct accepting sequences of IDs on every accepted input.

EXERCISE 15

Our proof of Lemma 13.2.2 ended like this:

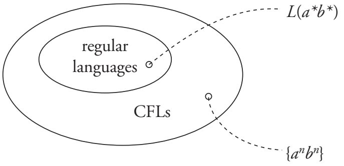
Proof of the other direction (*if* $(x, S) \mapsto^* (\varepsilon, y)$ *then* $S \Rightarrow^* xy$): Similar, by induction on the number of steps in the execution.

Fill in the missing details, proving by induction that if $(x, S) \mapsto^* (\varepsilon, y)$ then $S \Rightarrow^* xy$.

14

The Context-free Frontier

At this point we have two major language categories, the regular languages and the context-free languages, and we have seen that the CFLs include the regular languages, like this:



Are there languages outside of the CFLs? In this chapter we see that the answer is yes, and we see some simple examples of languages that are not CFLs.

We have already seen that there are many closure properties for regular languages. Given any two regular languages, there are many ways to combine them—intersections, unions, and so on—that are guaranteed to produce another regular language. The context-free languages also have some closure properties, though not as many as the regular languages. If regular languages are a safe and settled territory, context-free languages are more like frontier towns. Some operations like union get you safely to another context-free language; others like complement and intersection just leave you in the wilderness.

14.1 Pumping Parse Trees

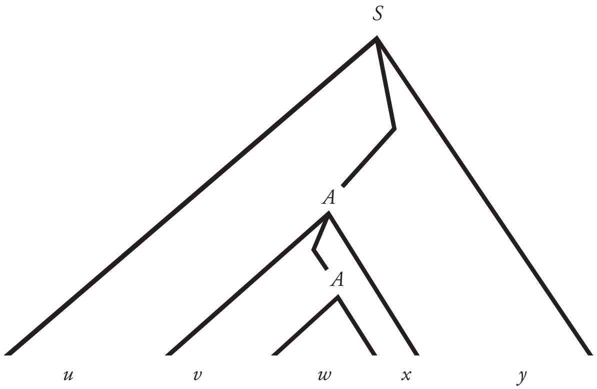
To show that there are some languages that are not regular, we reasoned that any sufficiently long string will force a DFA to repeat a state. For context-free languages there is a parallel argument, again involving repetition. We will see that we can force a parse tree to repeat a nonterminal symbol in a certain way. Such parse trees are called *pumping parse trees.*

A pumping parse tree for a CFG $G = (V, \Sigma, S, P)$ is a parse tree with two properties:

1. There is a node for some nonterminal symbol A, which has that same nonterminal symbol A as one of its descendants.
2. The terminal string generated from the ancestor A is longer than the terminal string generated from the descendant A.

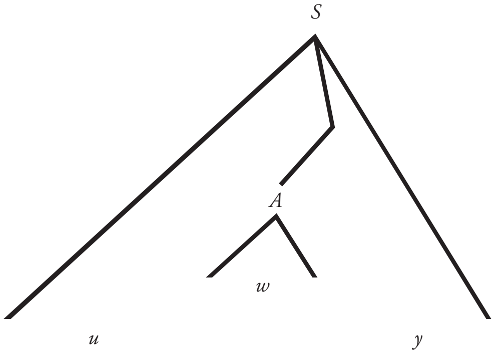
If you can find a pumping parse tree for a grammar, you know a set of strings that must be in the language—not just the string the pumping parse tree yields, but a whole collection of related strings.

Lemma 14.1.1: If a grammar G generates a pumping parse tree with a yield as shown:

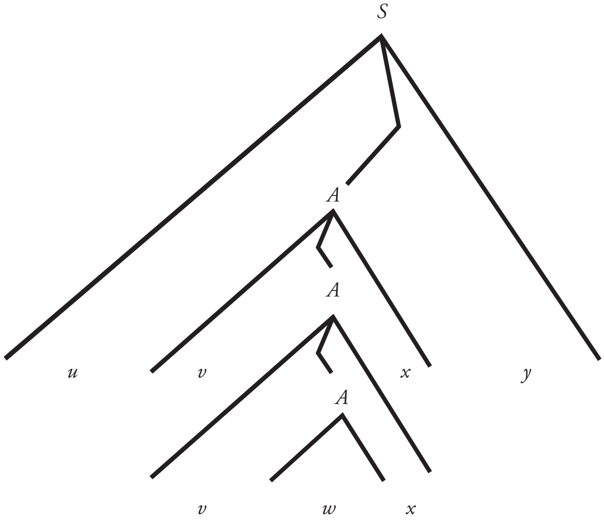


then $L(G)$ includes uv^iwx^iy for all i.

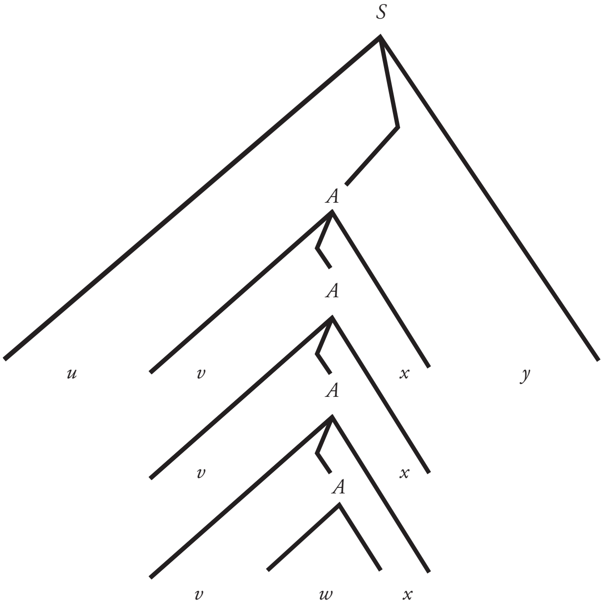
Proof: The string *uvwxy* is the whole terminal string derived by the pumping parse tree. As shown, A is the nonterminal symbol that occurs as its own descendant in the tree. The string *vwx* is the terminal string derived from the ancestor A, and the string w is the terminal string derived from the descendant *A*. (Satisfying the second property of pumping parse trees, *v* and *x* are not both ε.) It is the nature of context-free grammars that any *A* in the tree can be replaced with either of the two subtrees rooted at *A*, producing other legal parse trees for the grammar. For example, we could replace the *vwx* subtree with the *w* subtree, producing this parse tree for *uwy*:



Or we could replace the *w* subtree with the *vwx* subtree, producing this parse tree for uv^2wx^2y:



Then we could again replace the *w* subtree with the *vwx* subtree, producing this parse tree for uv^3wx^3y:



Repeating this step any number of times, we can generate a parse tree for uv^iwx^iy for any *i*.

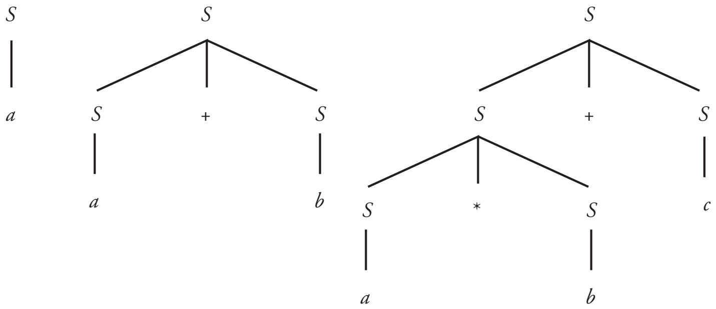
The previous lemma shows that pumping parse trees are very useful—find one, and you can conclude that not only $uvwxy \in L(G)$, but $uv^iwx^iy \in L(G)$ for all *i*. The next lemma shows that they are not at all hard to find. To prove it, we will need to refer to the height of a parse tree, which is defined as follows:

The *height* of a parse tree is the number of edges in the longest path from the start symbol to any leaf.

Consider this (ambiguous) grammar, for example:

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

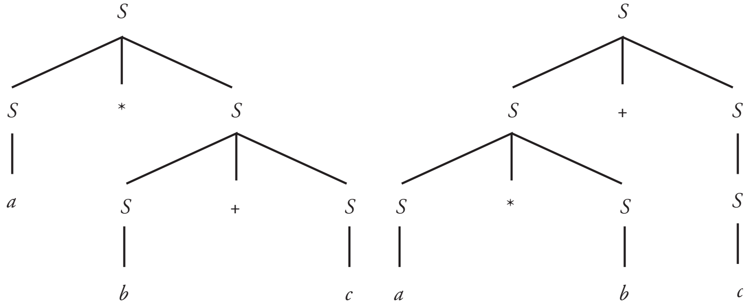
These are parse trees of heights 1, 2, and 3, respectively:



We will also need the idea of a minimum-size parse tree for a given string.

A *minimum-size* parse tree for the string *x* in a grammar *G* is a parse tree in *G* that generates *x* and has no more nodes than any other parse tree in *G* that generates *x*.

The need for this definition arises because an ambiguous grammar can generate the same string using more than one parse tree, and these parse trees might even have different numbers of nodes. For example, the previous grammar generates the string *a*b+c* with the parse tree shown above, but also with these ones:



All three parse trees generate the same string, but the last one is not minimum size, since it has one more node than the others. Every string in the language generated by a grammar has at least one minimum-size parse tree; for an ambiguous grammar, some strings may have more than one.

Now we're ready to see why pumping parse trees are easy to find.

Lemma 14.1.2: Every CFG $G = (V, \Sigma, S, P)$ that generates an infinite language generates a pumping parse tree.

Proof: Let $G = (V, \Sigma, S, P)$ be any CFG that generates an infinite language. Each string in $L(G)$ has at least one minimum-size parse tree; therefore, G

generates infinitely many minimum-size parse trees. There are only finitely many minimum-size parse trees of height $|V|$ or less; therefore, G generates a minimum-size parse tree of height greater than $|V|$ (indeed, it generates infinitely many). Such a parse tree must satisfy property 1 of pumping parse trees, because on a path with more than $|V|$ edges there must be some nonterminal A that occurs at least twice. And such a parse tree must also satisfy property 2, because it is a minimum-size parse tree; if it did not satisfy property 2, we could replace the ancestor A with the descendant A to produce a parse tree with fewer nodes yielding the same string. Thus, G generates a pumping parse tree.

This proof actually shows that every grammar for an infinite language generates not just one, but infinitely many pumping parse trees. However, one is all we'll need for the next proof.

14.2 The Language $\{a^nb^nc^n\}$

We have seen that the language $\{a^nb^n\}$ is context free but not regular. Now we consider the language $\{a^nb^nc^n\}$: any number of *a*s, followed by the same number of *b*s, followed by the same number of *c*s.

Theorem 14.1: $\{a^nb^nc^n\}$ is not a CFL.

Proof: Let $G = (V, \Sigma, S, P)$ be any CFG over the alphabet $\{a, b, c\}$, and suppose by way of contradiction that $L(G) = \{a^nb^nc^n\}$. By Lemma 14.1.2, G generates a pumping parse tree. Using Lemma 14.1.1, this means that for some k we have $a^kb^kc^k = uvwxy$, where v and x are not both ε and uv^2wx^2y is also in $L(G)$. The substrings v and x must each contain only *a*s, only *b*s, or only *c*s, because otherwise uv^2wx^2y is not even in $L(a^*b^*c^*)$. But in that case uv^2wx^2y has more than k copies of one or two symbols, but only k of the third. Thus $uv^2wx^2y \notin \{a^nb^nc^n\}$. By contradiction, $L(G) \neq \{a^nb^nc^n\}$.

The tricky part of this proof is seeing that no matter how you break up a string $a^kb^kc^k$ into substrings $uvwxy$, where v and x are not both ε, you must have $uv^2wx^2y \notin \{a^nb^nc^n\}$. If the substrings v and/or x contain a mixture of symbols, as in this example:

a a a	*a a b b*	*b*	*b b c*	*c c c c*
u	*v*	*w*	*x*	*y*

then the resulting string uv^2wx^2y would have *a*s after *b*s and/or *b*s after *c*s; for the example above it would be *aaaaabbaabbbbbcbbccccc*, clearly not in $\{a^nb^nc^n\}$. On the other hand, if the substrings v and x contain at most one kind of symbol each, as in this example:

a a a	*a a*	*b b b*	*b*	*b c c c c c*
u	*v*	*w*	*x*	*y*

then the resulting string uv^2wx^2y would no longer have the same number of *a*s, *b*s, and *c*s; for the example above it would be *aaaaaaabbbbbbccccc*, clearly not in $\{a^nb^nc^n\}$. Either way, $uv^2wx^2y \notin \{a^nb^nc^n\}$.

14.3 Closure Properties for CFLs

Context-free languages are closed for some of the same common operations as regular languages, including union, concatenation, Kleene star, and intersection with regular languages. Closure under union and concatenation can be proved using constructions that we saw, informally, back in Chapter 12.

Theorem 14.2.1: If L_1 and L_2 are any context-free languages, $L_1 \cup L_2$ is also context free.

Proof: By construction. If L_1 and L_2 are context free then, by definition, there exist grammars $G_1 = (V_1, \Sigma_1, S_1, P_1)$ and $G_2 = (V_2, \Sigma_2, S_2, P_2)$ with $L(G_1) = L_1$ and $L(G_2) = L_2$. We can assume without a loss of generality that V_1 and V_2 are disjoint. (Symbols shared between V_1 and V_2 could easily be renamed.) Now consider the grammar $G = (V, \Sigma, S, P)$, where S is a new nonterminal symbol and

$$
\begin{aligned}
&V = V_1 \cup V_2 \cup \{S\},\\
&\Sigma = \Sigma_1 \cup \Sigma_2, \text{ and}\\
&P = P_1 \cup P_2 \cup \{(S \to S_1), (S \to S_2)\}
\end{aligned}
$$

Clearly $L(G) = L_1 \cup L_2$, so $L_1 \cup L_2$ is a context-free language.

Theorem 14.2.2: If L_1 and L_2 are any context-free languages, L_1L_2 is also context free.

Proof: By construction. If L_1 and L_2 are context free then, by definition, there exist grammars $G_1 = (V_1, \Sigma_1, S_1, P_1)$ and $G_2 = (V_2, \Sigma_2, S_2, P_2)$ with $L(G_1) = L_1$ and $L(G_2) = L_2$. We can assume without a loss of generality that V_1 and V_2 are disjoint. (Symbols shared between V_1 and V_2 could easily be renamed.) Now consider the grammar $G = (V, \Sigma, S, P)$, where S is a new nonterminal symbol and

$V = V_1 \cup V_2 \cup \{S\}$,
$\Sigma = \Sigma_1 \cup \Sigma_2$, and
$P = P_1 \cup P_2 \cup \{(S \to S_1S_2)\}$

Clearly $L(G) = L_1L_2$, so L_1L_2 is a context-free language.

We can define the Kleene closure of a language in a way that parallels our use of the Kleene star in regular expressions:

The Kleene closure of any language L is $L^* = \{x_1x_2 \ldots x_n \mid n \geq 0$, with all $x_i \in L\}$.

The fact that the context-free languages are closed under Kleene star can then be proved with a CFG construction, as before:

Theorem 14.2.3: If L is any context-free language, L^* is also context free.

Proof: By construction. If L is context free then, by definition, there exists a grammar $G = (V, \Sigma, S, P)$ with $L(G) = L$. Now consider the grammar $G' = (V', \Sigma, S', P')$, where S' is a new nonterminal symbol and

$V' = V \cup \{S'\}$, and
$P' = P \cup \{(S' \to SS'), (S' \to \varepsilon)\}$

Now $L(G') = L^*$, so L^* is a context-free language.

One more closure property: the CFLs are closed under intersection with regular languages:

Theorem 14.2.4: If L_1 is any context-free language and L_2 is any regular language, then $L_1 \cap L_2$ is context free.

Proof sketch: This can be proved by a (somewhat hairy) construction that combines a stack machine M_1 for L_1 with an NFA M_2 for L_2, producing a new stack machine for $L_1 \cap L_2$. It works a bit like the product construction: if M_1's stack alphabet is Γ, and M_2's state set is Q, the new stack machine can use an element of $\Gamma \times Q$ as the symbol on the top of its stack to record both the M_1's current top symbol and M_2's current state.

14.4 Nonclosure Properties

In the previous section we saw that the CFLs are closed for some of the same operations for which the regular languages are closed. Not all, however: the CFLs are not closed for intersection or complement.

Theorem 14.3.1: The CFLs are not closed for intersection.

Proof: By counterexample. Consider these two CFGs G_1 and G_2:

$$
\begin{aligned}
&S_1 \rightarrow A_1B_1\\
&A_1 \rightarrow aA_1b \mid \varepsilon\\
&B_1 \rightarrow cB_1 \mid \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
&S_2 \rightarrow A_2B_2\\
&A_2 \rightarrow aA_2 \mid \varepsilon\\
&B_2 \rightarrow bB_2c \mid \varepsilon
\end{aligned}
$$

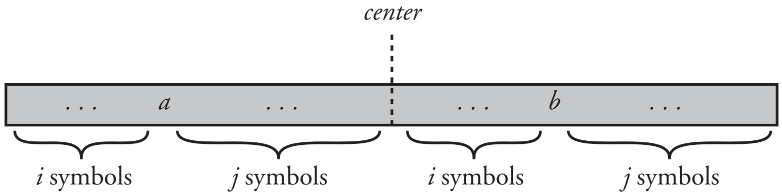
Now $L(G_1) = \{a^nb^nc^m\}$, while $L(G_2) = \{a^mb^nc^n\}$. The intersection of the two languages is $\{a^nb^nc^n\}$, which we know is not context free. Thus, the CFLs are not closed for intersection.

This nonclosure property does *not* mean that every intersection of CFLs fails to be a CFL. In fact, we have already seen many cases in which the intersection of two CFLs is another CFL. For example, we know that any intersection of regular languages is a regular language—and they are all CFLs. We also know that the intersection of a CFL with a regular language is a CFL. All the theorem says is that the intersection of two CFLs is *not always* a CFL. Similarly, the complement of a CFL is sometimes, but not always, another CFL:

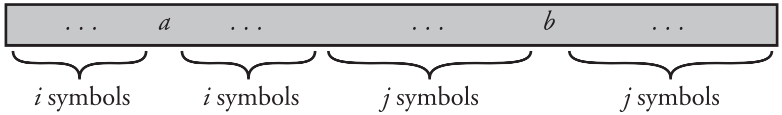
Theorem 14.3.2: The CFLs are not closed for complement.

Proof 1: By Theorem 14.2.1 we know that CFLs are closed for union. Suppose by way of contradiction that they are also closed for complement. By DeMorgan's laws we have $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. This defines intersection in terms of union and complement, so the CFLs are closed for intersection. But this is a contradiction of Theorem 14.3.1. By contradiction, we conclude that the CFLs are not closed for complement.

Proof 2: By counterexample. Let L be the language $\{xx \mid x \in \{a, b\}^*\}$, and consider its complement, $\overline{L} = \{x \in \{a, b\}^* \mid x \notin L\}$. Obviously, all odd-length strings are in $\overline{L}$. Even-length strings are in $\overline{L}$ if and only if they have at least one symbol in the first half that is not the same as the corresponding symbol in the second half; that is, either there is an *a* in the first half and a *b* at the corresponding position in the second half, or the reverse. In the first case the string looks like this, for some *i* and *j*:



At first this looks daunting: how could a grammar generate such strings? How can we generate *waxybz*, where $|w| = |y| = i$ and $|x| = |z| = j$? It doesn't look context free, until you realize that because the *x* and *y* parts can both be any string in $\{a, b\}^*$, we can swap them, instead picturing it as:



Now we see that this actually is context free; it's $\{way \mid |w| = |y|\}$, concatenated with $\{xbz \mid |x| = |z|\}$. For the case with a *b* in the first half corresponding to an *a* in the second half, we can just swap the two parts, getting $\{xbz \mid |x| = |z|\}$ concatenated with $\{way \mid |w| = |y|\}$.

To summarize, we can express $\overline{L}$ as the union of three sets:

$$\begin{aligned}\overline{L} = \{x \in \{a, b\}^* \mid |x| \text{ is odd}\} \\ \cup\ (\{way \mid |w| = |y|\} \text{ concatenated with } \{xbz \mid |x| = |z|\}) \\ \cup\ (\{xbz \mid |x| = |z|\} \text{ concatenated with } \{way \mid |w| = |y|\})\end{aligned}$$

This CFG generates the language: *O* generates the odd-length part, and *AB* and *BA* generate the two even-length parts.

$$\begin{aligned}&S \rightarrow O \mid AB \mid BA \\ &A \rightarrow XAX \mid a\end{aligned}$$

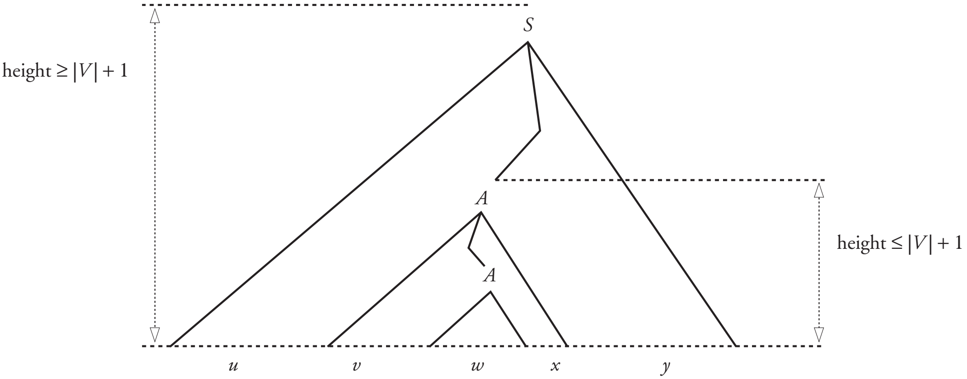
$B \to XBX \mid b$
$O \to XXO \mid X$
$X \to a \mid b$

We conclude that $\overline{L}$ is context free. But as we will show in Section 14.7, the complement of $\overline{L}$, $L = \{xx \mid x \in \{a, b\}^*\}$ is not context free. We conclude that the CFLs are not closed for complement.

14.5 A Pumping Lemma

Let's return to pumping parse trees and try to make a general-purpose pumping lemma for CFLs. We have argued that any minimum-size tree of a height greater than $|V|$ must be a pumping parse tree, since on any path of more than $|V|$ edges there must be some nonterminal A that occurs at least twice. Of course, there may be many different places where a nonterminal occurs as its own descendant. To make a useful pumping lemma for context-free languages, we need to be a little more specific about how we are dividing the tree.

Lemma 14.2.1: For every grammar $G = (V, \Sigma, S, P)$, every minimum-size parse tree of a height greater than $|V|$ can be expressed as a pumping parse tree with the properties shown:



Proof: Choose any path from root to leaf with more than $|V|$ edges. Then, working from the leaf back toward the root on this path, choose the first two nodes that repeat a nonterminal, and let these be the nodes labeled A in the diagram. As shown in the proof of Lemma 14.1.2, the result is a pumping parse tree. Furthermore, the nonterminal A must have repeated within the first $|V| + 1$ edges, so the height of the subtree generating vwx is $\leq |V| + 1$.

Being able to give a bound on the height of a tree or subtree also enables us to give a bound on the length of the string generated.

> **Lemma 14.2.2:** For every CFG $G = (V, \Sigma, S, P)$ there exists some k greater than the length of any string generated by any parse tree or subtree of height $|V| + 1$ or less.
>
> **Proof 1:** For any given grammar, there are only finitely many parse trees and subtrees of height $|V| + 1$ or less. Let k be one greater than the length of the longest string so generated.
>
> **Proof 2:** Let b be the length of the longest right-hand side of any production in P. Then b is the maximum branching factor in any parse tree. So a tree or subtree of height $|V| + 1$ can have at most $b^{|V|+1}$ leaves. Let $k = b^{|V|+1} + 1$.

These two proofs give different values for k. That is not a problem, because what matters here is not the value of k, but the existence of some k with the right properties. This was also true of the value k we used in the pumping lemma for regular languages. To prove the pumping lemma, we let k be the number of states in some DFA for the language. But once the pumping lemma for regular languages was proved, the value of k used in the proof was irrelevant, since the lemma itself claims only that some suitable k exists.

Putting these lemmas together with our previous results about pumping parse trees, we get the pumping lemma for context-free languages.

> **Lemma 14.2.3** *(The Pumping Lemma for Context-free Languages)*: For all context-free languages L there exists some $k \in \mathcal{N}$ such that for all $z \in L$ with $|z| \geq k$, there exist $uvwxy$ such that
>
> 1. $z = uvwxy$,
> 2. v and x are not both ε,
> 3. $|vwx| \leq k$, and
> 4. for all i, $uv^iwx^iy \in A$.
>
> **Proof:** We are given some CFL L. Let G be any CFG with $L(G) = L$, and let k be as given for this grammar by Lemma 14.2.2. We are given some $z \in L$ with $|z| \geq k$. Since all parse trees in G for z have a height greater than $|V| + 1$, that includes a minimum-size parse tree for z. We can express this as a pumping parse tree for z as given in Lemma 14.2.1. Since this is a parse tree for z, property 1 is satisfied; since it is a pumping parse tree, properties 2 and 4 are satisfied; and since the subtree generating vwx is of height $|V| + 1$ or less, property 3 is satisfied.

This pumping lemma shows once again how *matching pairs* are fundamental to the context-free languages. Every sufficiently long string in a context-free language contains a matching pair consisting of the two substrings v and x of the lemma. These substrings can be pumped in tandem, always producing another string uv^iwx^iy in the language. Of course, one or the other (but not both) of those substrings may be ε. In that way the pumping lemma for context-free languages generalizes the one for regular languages, since if one of the strings is ε, the other can be pumped by itself.

14.6 Pumping-Lemma Proofs

The pumping lemma is very useful for proving that languages are not context free. For example, here is a pumping-lemma proof showing that $\{a^nb^nc^n\}$ is not context free (a fact we already proved without using the pumping lemma). The steps in the proof are numbered for future reference.

1. Proof is by contradiction using the pumping lemma for context-free languages. Assume that $L = \{a^nb^nc^n\}$ is context free, so the pumping lemma holds for L. Let k be as given by the pumping lemma.
2. Choose $z = a^kb^kc^k$. Now $z \in L$ and $|z| \geq k$ as required.
3. Let *u, v, w, x, and y* be as given by the pumping lemma, so that $uvwxy = a^kb^kc^k$, v and x are not both ε, $|vwx| \leq k$, and for all i, $uv^iwx^iy \in L$.
4. Now consider pumping with $i = 2$. The substrings v and x cannot contain more than one kind of symbol each—otherwise the string uv^2wx^2y would not even be in $L(a^*b^*c^*)$. So the substrings v and x must fall within the string $a^kb^kc^k$ in one of these ways:

	a^k	b^k	c^k
1.	$\boxed{v}$ $\boxed{x}$		
2.	$\boxed{v}$	$\boxed{x}$	
3.		$\boxed{v}$ $\boxed{x}$	
4.		$\boxed{v}$	$\boxed{x}$
5.			$\boxed{v}$ $\boxed{x}$
6.	$\boxed{v}$		$\boxed{x}$

   But in all these cases, since v and x are not both ε, pumping changes the number of one or two of the symbols, but not all three. So $uv^2wx^2y \notin L$.

5. This contradicts the pumping lemma. By contradiction, $L = \{a^nb^nc^n\}$ is not context free.

The structure of this proof matches the structure of the pumping lemma itself. Here is the overall structure of the pumping lemma, using the symbols ∀ ("for all") and ∃ ("there exists"):

1. $\exists\ k$ …
2. $\forall\ z$ …
3. $\exists\ uvwxy$ …
4. $\forall\ i \geq 0$ …

The alternating $\forall$ and $\exists$ parts make every pumping-lemma proof into a kind of game, just as we saw using the pumping lemma for regular languages. The $\exists$ parts (the natural number k and the strings u, v, w, x, and y) are merely guaranteed to exist. In effect, the pumping lemma itself makes these moves, choosing k, u, v, w, x, and y any way it wants. In steps 1 and 3 we could only say these values are "as given by the pumping lemma." The $\forall$ parts, on the other hand, are the moves you get to make: you can choose any values for z and i, since the lemma holds for all such values. This pumping lemma offers fewer choices than the pumping lemma for regular languages, and that actually makes the proofs a little harder to do. The final step, showing that a contradiction is reached, is often more difficult, because one has less control early in the proof.

You may have noticed that in that table in the proof—the one that shows how the strings v and x can fall within the string $a^kb^kc^k$—line 6 is unnecessary. It does not hurt to include it in the proof, because all six cases lead to the same contradiction. But the substrings v and x actually cannot be as far apart as shown in line 6, because that would make the combined substring vwx more than k symbols long, while the pumping lemma guarantees that $|vwx| \leq k$. In many applications of the pumping lemma it is necessary to make use of this. The following pumping-lemma proof is an example:

> **Theorem 14.4:** $L = \{a^nb^mc^n \mid m \leq n\}$ is not context free.
>
> **Proof:** By contradiction using the pumping lemma for context-free languages. Assume that $L = \{a^nb^mc^n \mid m \leq n\}$ is context free, so the pumping lemma holds for L. Let k be as given by the pumping lemma. Choose $z = a^kb^kc^k$. Now $z \in L$ and $|z| \geq k$ as required. Let u, v, w, x, and y be as given by the pumping lemma, so that $uvwxy = a^kb^kc^k$, v and x are not both ε, $|vwx| \leq k$, and for all i, $uv^iwx^iy \in L$.
>
> Now consider pumping with $i = 2$. The substrings v and x cannot contain more than one kind of symbol each—otherwise the string uv^2wx^2y would not even be in $L(a^*b^*c^*)$. So the substrings v and x must fall within the string $a^kb^kc^k$ in one of these ways:

	a^k	b^k	c^k
1.	$\boxed{v}$ $\boxed{x}$		
2.	$\boxed{v}$	$\boxed{x}$	
3.		$\boxed{v}$ $\boxed{x}$	
4.		$\boxed{v}$	$\boxed{x}$
5.			$\boxed{v}$ $\boxed{x}$
6.	$\boxed{v}$		$\boxed{x}$

Because v and x are not both ε, uv^2wx^2y in case 1 has more *a*s than *c*s. In case 2 there are either more *a*s than *c*s, more *b*s than *c*s, or both. In case 3 there are more *b*s than *a*s and *c*s. In case 4 there are more *c*s than *a*s, more *b*s than *a*s, or both. In case 5 there are more *c*s than *a*s. Case 6 contradicts $|vwx| \leq k$. Thus, all cases contradict the pumping lemma. By contradiction, $L = \{a^nb^mc^n \mid m \leq n\}$ is not context free.

This proof is very similar to the previous one, using the same string z and pumping count i. This time, however, we needed to use the fact that $|vwx| \leq k$ to reach a contradiction in case 6. Without that, no contradiction would be reached, since pumping v and x together could make the number of *a*s and the number of *c*s grow in step, which would still be a string in the language L.

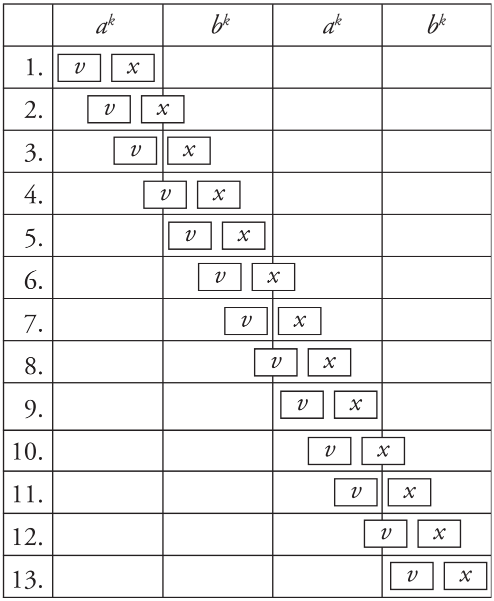
14.7 The Language {*xx*}

Consider the language of strings that consist of any string over a given alphabet Σ, followed by another copy of the same string: $\{xx \mid x \in \Sigma^*\}$. For $\Sigma = \{a, b\}$, the language contains the strings ε, *aa, bb, abab, baba, aaaa, bbbb*, and so on. We have seen that the related language $\{xx^R\}$ is context free, though not regular for any alphabet with at least two symbols.

Theorem 14.5: $\{xx \mid x \in \Sigma^*\}$ is not a CFL for any alphabet Σ with at least two symbols.

Proof: By contradiction using the pumping lemma for context-free languages. Let Σ be any alphabet containing at least two symbols (which we will refer to as *a* and *b*). Assume that $L = \{xx \mid x \in \Sigma^*\}$ is context free, so the pumping lemma holds for L. Let k be as given by the pumping lemma. Choose $z = a^kb^ka^kb^k$. Now $z \in L$ and $|z| \geq k$ as required. Let u, v, w, x, and y be as given by the pumping lemma, so that $uvwxy = a^kb^ka^kb^k$, v and x are not both ε, $|vwx| \leq k$, and for all i, $uv^iwx^iy \in L$.

Now consider how the substrings v and x fall within the string. Because $|vwx| \leq k$, v and x cannot be widely separated, so we have these 13 cases:



For cases 1 through 5, we can choose $i = 0$. Then the string uv^0wx^0y is some sa^kb^k where $|s| < 2k$. In this string the last symbol of the first half is an a, while the last symbol of the second half is a b. So $uv^0wx^0y \notin L$.

For cases 9 through 13, we can again choose $i = 0$. This time the string uv^0wx^0y is some a^kb^ks where $|s| < 2k$. In this string the first symbol of the first half is an a, while the first symbol of the second half is a b. So, again, $uv^0wx^0y \notin L$.

Finally, for cases 6, 7, and 8, we can again choose $i = 0$. Now the string uv^0wx^0y is some a^ksb^k where $|s| < 2k$. But such an a^ksb^k can't be rr for any r, because if r starts with k as and ends with k bs, we must have $|r| \geq 2k$ and so $|rr| \geq 4k$, while our $|a^ksb^k| < 4k$. So, again, $uv^0wx^0y \notin L$.

In every case we have a contradiction. By contradiction, L is not a context-free language.

In Chapter 12 we saw the important role CFGs play in defining the syntax of programming languages. Many languages, including Java, require variables to be declared before they are used:

```
int fred = 0;
while (fred==0) {
    ...
}
```

The requirement that the same variable name `fred` occur in two places, once in a declaration and again in a use, is not part of any grammar for Java. It is a non-context-free construct in the same way that $\{xx \mid x \in \Sigma^*\}$ is a non-context-free language—think of x here as the variable name which must be the same in two places. Java compilers can enforce the requirement easily enough, but doing so requires more computational power than can be wired into any context-free grammar.

One final note about pumping-lemma proofs. In the proof above, as in all those shown in this chapter, the choice of i was static. The proof above always chose $i = 0$, regardless of the values of the other variables. This is not always possible. The pumping lemma permits the choice of i to depend on the values of k, u, v, w, x, and y, just as the choice of z must depend on k. For more challenging proofs it is often necessary to take advantage of that flexibility. In terms of the pumping-lemma game, it is often necessary to see what moves the pumping lemma makes before choosing the i that will lead to a contradiction.

Exercises

EXERCISE 1

If our definition of pumping parse trees included only the first, but not the second property, the proof of Theorem 14.1 would fail to reach a contradiction. Explain exactly how it would fail.

EXERCISE 2

Construct a grammar for an infinite language that generates a parse tree of a height greater than $|V|$ that is not a pumping parse tree. Show both the grammar and the parse tree.

EXERCISE 3

Prove that $\{a^n b^n c^n d^n\}$ is not a CFL.

EXERCISE 4

Prove that $\{a^n b^{2n} c^n\}$ is not a CFL.

EXERCISE 5

Show that $\{a^n b^n c^p d^q\}$ is a CFL by giving either a stack machine or a CFG for it.

EXERCISE 6

Show that $\{a^n b^m c^{n+m}\}$ is a CFL by giving either a stack machine or a CFG for it.

EXERCISE 7

Give a CFG for $\{a^n b^n\} \cup \{xx^R \mid x \in \{a, b\}^*\}$.

EXERCISE 8

Give a stack machine for $\{a^n b^n\} \cup \{xx^R \mid x \in \{a, b\}^*\}$.

EXERCISE 9

Give a CFG for the set of all strings xy such that $x \in \{0^n 1^{2n}\}$ and y is any string over the alphabet {0, 1} that has an even number of 1s.

EXERCISE 10

Give a stack machine for the set of all strings xy such that $x \in \{0^n 1^{2n}\}$ and y is any string over the alphabet {0, 1} that has an even number of 1s.

EXERCISE 11

Give a CFG for $\{a^i b^n a^j b^n a^k\}^*$.

EXERCISE 12

Give a stack machine for $\{a^i b^n a^j b^n a^k\}^*$.

EXERCISE 13

Give a CFG for $\{a^n b^n\} \cap \{x \in \{a, b\}^* \mid$ the number of *a*s in x is even}.

EXERCISE 14

Give a stack machine for $\{a^n b^n\} \cap \{x \in \{a, b\}^* \mid$ the number of *a*s in x is even}.

EXERCISE 15

Using the pumping lemma for context-free languages, prove that $\{a^n b^m c^n \mid m \geq n\}$ is not a CFL.

EXERCISE 16

Show that $\{a^n b^m c^n\}$ is a CFL by giving either a stack machine or a CFG for it.

EXERCISE 17

Using the pumping lemma for context-free languages, prove that $\{a^n b^m c^p \mid n \geq m$ and $m \geq p\}$ is not a CFL.

EXERCISE 18

Show that $\{a^n b^m c^p \mid n \geq m\}$ is a CFL by giving either a stack machine or a CFG for it.

EXERCISE 19

Let $A = \{x \mid x \in \{a, b, c\}^*$ and x contains the same number of each of the three symbols}. Using the pumping lemma for context-free languages, prove that A is not a CFL.

EXERCISE 20

Let $B = \{x \mid x \in \{a, b, c\}^*$ and x contains the same number of at least two of the three symbols}. Show that B is a CFL by giving either a stack machine or a CFG for it.

15
Stack Machine Applications

The parse tree (or a simplified version, the abstract syntax tree) is one of the central data structures of almost every compiler or other programming language system. To parse a program is to find a parse tree for it. Every time you compile a program, the compiler must first parse it. Parsing algorithms are fundamentally related to stack machines, as this chapter illustrates.

15.1 Top-Down Parsing

In section 13.7 we saw a general procedure for constructing a stack machine from any CFG. The constructed stack machine verifies that a string is in the language by finding a derivation for that string in the given CFG. That makes it a useful point of departure for applications that require parsing. The constructed stack machine works by repeatedly applying productions until a fully terminal string is derived from the start symbol S. That is a kind of *top-down* parsing: it builds a parse tree starting from S and working down. Of course, our constructed stack machine doesn't actually build anything; it just accepts or rejects by following the steps of a leftmost derivation in the grammar. But a program implementing such a stack machine could record the production applied for each nonterminal and thus create the corresponding parse tree.

The construction of Section 13.7 is a good point of departure, but it is not directly useful for parsing applications because the constructed stack machine is highly nondeterministic. Whenever there is a nonterminal on top of the stack, the constructed stack machine can apply any one of the productions for that nonterminal. It does not choose deterministically which production to use for each nonterminal in the parse tree; it just relies on nondeterminism, accepting the string if there is some choice that eventually works. For parsing applications this could be implemented using backtracking of the kind we used when implementing NFAs. But it would be much better for a compiler to be able to find the parse tree in one deterministic pass over the input program, without having to search around for it.

For a certain class of grammars the nondeterminism of the top-down-parsing stack machine can easily be avoided. For example, consider the CFL $\{xcx^R \mid x \in \{a, b\}^*\}$. It is generated by this CFG:

$$S \rightarrow aSa \mid bSb \mid c$$

and the construction of Section 13.7 would give this stack machine for it:

	read	pop	push
1.	ε	S	aSa
2.	ε	S	bSb
3.	ε	S	c
4.	a	a	ε
5.	b	b	ε
6.	c	c	ε

Although this stack machine is nondeterministic, closer inspection reveals that it is always quite easy to choose the next move. For example, suppose the input string is *abbcbba*. There are three possible first moves:

$(abbcbba, S) \mapsto_1 (abbcbba, aSa) \mapsto \ldots$
$(abbcbba, S) \mapsto_2 (abbcbba, bSb) \mapsto \ldots$
$(abbcbba, S) \mapsto_3 (abbcbba, c) \mapsto \ldots$

But only the first of these has any future. The other two leave, on top of the stack, a terminal symbol that is not the same as the next input symbol. From there, no further move will be possible.

We can formulate simple rules about when to use those three moves:

- Use move 1 when the top stack symbol is S and the next input symbol is a.
- Use move 2 when the top stack symbol is S and the next input symbol is b.
- Use move 3 when the top stack symbol is S and the next input symbol is c.

These rules say when to apply those moves that derived from productions in the original grammar. (The other stack-machine moves, the ones that read an input symbol and pop a matching stack symbol, are already deterministic.) Our rules can be expressed as a two-dimensional *look-ahead table*.

	a	b	c	$\$$
S	$S \rightarrow aSa$	$S \rightarrow bSb$	$S \rightarrow c$	

The entry at *table*[A][ω] tells which production to use when the top of the stack is A and the next input symbol is ω. The final column of the table, *table*[A][$\$$], tells which production to use when the top of the stack is A and all the input has been read. For our example, *table*[S][$\$$] is empty, showing that if you still have S on top of the stack but all the input has been read, there is no way to proceed.

Using the table requires one symbol of look-ahead: when there is a nonterminal symbol on top of the stack, we need to look ahead to see what the next input symbol will be, without actually consuming it. With the help of this table our stack machine becomes deterministic. Here is the idea in pseudocode:

```
void predictiveParse(table, S) {
  initialize a stack containing just S
  while (the stack is not empty) {
    A = the top symbol on stack
    c = the current symbol in input (or $ at the end)
    if (A is a terminal symbol) {
      if (A != c) the parse fails
      pop A and advance input to the next symbol
    }
    else {
      if table[A][c] is empty the parse fails
12.                 pop A and push the right-hand side of table[A][c]
13.             }
14.         }
15.         if input is not finished the parse fails
16.     }
```

This treats `input` as a global variable: the source of the input string to be parsed. The code scans the input from left to right, one symbol at a time, maintaining (implicitly) the current position of the scan. Line 5 peeks at the current input symbol, but only line 8 advances to the next symbol. We assume there is a special symbol, not the same as any terminal symbol, marking the end of the string; call this symbol \$. The parameter `table` is the two-dimensional table of productions, as already described.

Our `predictiveParse` is an example of a family of top-down techniques called *LL(1) parsers*. The first *L* signifies a left-to-right scan of the input, the second *L* signifies that the parse follows the order of a leftmost derivation, and the *1* signifies that one symbol of input look-ahead is used (along with the top symbol of the stack) to choose the next production. It is a simple parsing algorithm; the catch is the parse table it requires. Such parse tables exist for some grammars but not for others—which means that LL(1) parsing is possible for some grammars but not for others. *LL(1) grammars* are those for which LL(1) parsing is possible; *LL(1) languages* are those that have LL(1) grammars.

LL(1) grammars can often be constructed for programming languages. Unfortunately, they tend to be rather contorted. Consider this simple grammar and the little language of expressions it generates:

$$S \to (S) \mid S + S \mid S * S \mid a \mid b \mid c$$

As we saw in Chapter 12, that grammar is ambiguous, and no ambiguous grammar is LL(1). But even this reasonably simple unambiguous grammar for the language fails to be LL(1):

$$S \to S + R \mid R$$
$$R \to R * X \mid X$$
$$X \to (S) \mid a \mid b \mid c$$

In this case the grammar's problem is that it is *left recursive*: it includes productions like $S \to S + R$ that replace a nonterminal with a string starting with that same nonterminal. No left-recursive grammar is LL(1). This LL(1) grammar for the same language manages to avoid left recursion, at some cost in clarity:

$$S \to AR$$
$$R \to +AR \mid \varepsilon$$
$$A \to XB$$

$B \rightarrow *XB \mid \varepsilon$
$X \rightarrow (S) \mid a \mid b \mid c$

Once we have an LL(1) grammar, we still need to construct the LL(1) parse table for it. That construction can be rather subtle. Here, for example, is the parse table that goes with the grammar above:

	a	*b*	*c*	+	*	(	)	$
S	$S \rightarrow AR$	$S \rightarrow AR$	$S \rightarrow AR$			$S \rightarrow AR$		
R				$R \rightarrow +AR$			$R \rightarrow \varepsilon$	$R \rightarrow \varepsilon$
A	$A \rightarrow XB$	$A \rightarrow XB$	$A \rightarrow XB$			$A \rightarrow XB$		
B				$B \rightarrow \varepsilon$	$B \rightarrow *XB$		$B \rightarrow \varepsilon$	$B \rightarrow \varepsilon$
X	$X \rightarrow a$	$X \rightarrow b$	$X \rightarrow c$			$X \rightarrow (S)$		

It is instructive to consider these entries carefully. For example, the production $S \rightarrow AR$ is in `table`[*S*][*a*] not because it generates the terminal symbol *a* directly, but because using it can eventually generate a string that starts with *a*. The production $B \rightarrow \varepsilon$ is in `table`[*B*][)] not because it generates the terminal symbol), but because it can reveal a) beneath the *B* on the stack.

There is an algorithm for constructing an LL(1) parse table, given an LL(1) grammar. We will not see that algorithm here, but the interested reader should see "Further Reading" (Section 15.5) at the end of this chapter.

15.2 Recursive-Descent Parsing

An LL(1) parser can also be implemented using recursive functions, with one function for parsing each nonterminal in the grammar. Consider again our simple grammar for $\{xcx^R \mid \mathrm{x} \in \{a, b\}^*\}$:

$S \rightarrow aSa \mid bSb \mid c$

We first implement a utility routine called `match`:

```
void match(x) {
  c = the current symbol in input
  if (c!=x) the parse fails
  advance input to the next symbol
}
```

All `match` does is check that the expected nonterminal really does occur next in the input and advance past it. Now we implement a function for deriving a parse tree for *S*:

```
void parse_S() {
  c = the current symbol in input (or $ at the end)
  if (c=='a') { // production $S \to aSa$
    match('a'); parse_S(); match('a');
  }
  else if (c=='b') { // production $S \to bSb$
    match('b'); parse_S(); match('b');
  }
  else if (c=='c') { // production $S \to c$
    match('c');
  }
  else the parse fails
}
```

This code is an LL(1) parser, not really that different from `predictiveParse`, our previous, table-driven implementation. Both are left-to-right, top-down parsers that choose a production to apply based on the current nonterminal and the current symbol in the input. But now the information from the parse table is incorporated into the code, which works recursively. This is called a *recursive-descent* parser.

This is part of a recursive-descent parser for our LL(1) grammar of expressions:

```
void parse_S() {
  c = the current symbol in input (or $ at the end)
  if (c=='a' || c=='b' ||
      c=='c' || c=='(') { // production $S \to AR$
    parse_A(); parse_R();
  }
  else the parse fails
}


void parse_R() {
  c = the current symbol in input (or $ at the end)
  if (c=='+') { // production $R \to +AR$
    match('+'); parse_A(); parse_R();
```

```
   }
   else if (c==')' || c=='$') { // production R → ε
   }
   else  the parse fails
 }
```

Additional functions `parse_A`, `parse_B`, and `parse_X` constructed on the same lines would complete the parser.

Compare the recursive-descent parser with the previous table-driven LL(1) parser. Both are top-down, LL(1) techniques that work only with LL(1) grammars. The table-driven method uses an explicit parse table, while recursive descent uses a separate function for each nonterminal. The table-driven method uses an explicit stack—but what happened to the stack in the recursive-descent method? It's still there, but hidden. Where a table-driven, top-down parser uses an explicit stack, a recursive-descent parser uses the language system's call stack. Each function call implicitly pushes and each return implicitly pops the call stack.

15.3 Bottom-Up Parsing

In the last two sections we saw some top-down parsing methods, which develop a parse tree by starting at the root. It is also possible to build a parse tree from the bottom up. An important bottom-up parsing method is *shift-reduce* parsing, so called because it involves two basic kinds of moves:

1. (shift) Push the current input symbol onto the stack and advance to the next input symbol.
2. (reduce) There is a string on top of the stack that is the right-hand side of some production. Pop it off and replace it with the nonterminal symbol from the left-hand side of that production.

The reduce move is exactly the reverse of what our predictive parser did. The predictive parser replaced a nonterminal symbol on the stack with the right-hand side of some production for that nonterminal. A bottom-up parser replaces the right-hand side of some production on the stack with its nonterminal symbol.

For example, a shift-reduce parser for our grammar

$$S \rightarrow aSa \mid bSb \mid c$$

would parse the string *abbcbba* using a sequence of moves like this:

Input	Stack	Next Move
$\underline{a}bbcbba\$$	ε	shift
$a\underline{b}bcbba\$$	a	shift
$ab\underline{b}cbba\$$	ba	shift
$abb\underline{c}bba\$$	bba	shift
$abbc\underline{b}ba\$$	$\underline{c}bba$	reduce by $S \rightarrow c$
$aaac\underline{b}bb\$$	$Sbba$	shift
$abbcb\underline{b}a\$$	$\underline{bSb}ba$	reduce by $S \rightarrow bSb$
$abbcb\underline{b}a\$$	Sba	shift
$abbcbb\underline{a}\$$	$\underline{bSb}a$	reduce by $S \rightarrow bSb$
$abbcbb\underline{a}\$$	Sa	shift
$abbcbba\underline{\$}$	$\underline{aSa}$	reduce by $S \rightarrow aSa$
$abbcbba\underline{\$}$	S	

In this table the current input symbol is underlined and, for reduce moves, the salient substring of the stack is underlined as well. As you can see, the parser does not get around to the top of the parse tree, the root symbol *S*, until the final step. It builds the parse tree from the bottom up.

A popular kind of shift-reduce parser is the *LR(1) parser*. The *L* signifies a left-to-right scan of the input, the *R* signifies that the parse follows the order of a rightmost derivation in reverse, and the *1* signifies that one symbol of input look-ahead is used (along with the string on top of the stack) to select the next move. These techniques are considerably trickier than LL(1) techniques. One difficulty is that reduce moves must operate on the top-of-the-stack string, not just the top-of-the-stack symbol. Making this efficient requires some close attention. (One implementation trick uses stacked DFA state numbers to avoid expensive string comparisons in the stack.)

Grammars that can be parsed this way are called *LR(1) grammars*, and languages that have LR(1) grammars are called *LR(1) languages*. In spite of their complexity, LR(1) parsers are quite popular, chiefly because the class of LR(1) grammars is quite broad. The LR(1) grammars include all the LL(1) grammars, and many others as well. Programming language constructs can usually be expressed with LR(1) grammars that are reasonably readable; making a grammar LR(1) usually does not require as many contortions as making it LL(1). (For example, LR(1) grammars need not avoid left-recursive productions.)

LR(1) parsers are complicated—almost always too complicated to be written without the help of special tools. There are many tools for generating LR(1) parsers automatically.

A popular one is the Unix tool called yacc, which works a bit like the lex tool we saw in Chapter 8. It converts a context-free grammar into C code for an LR(1) parser. In the yacc input file, each production in the grammar is followed by a piece of C code, the *action*, which is incorporated into the generated parser and is executed by the parser whenever a reduce step is made using that production. For most compilers, this action involves the construction of a parse tree or abstract syntax tree, which is used in subsequent (hand-coded) phases of compilation.

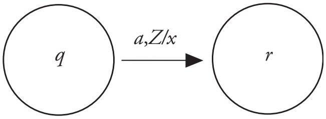
Although they are complicated, LR(1) techniques have very good efficiency—like LL(1) techniques, they take time that is essentially proportional to the length of the program being parsed. Beyond LR(1) techniques there are many other parsing algorithms, including some that have no restrictions at all and work with any CFG. The Cocke-Kasami-Younger (CKY) algorithm, for example, parses deterministically and works with any CFG. It is quite a simple algorithm too—much simpler than LR(1) parsing. The drawback is its lack of efficiency; the CKY algorithm takes time proportional to the *cube* of the length of the string being parsed, and that isn't nearly good enough for compilers and other programming-language tools.

15.4 PDAs, DPDAs, and DCFLs

The DFA and NFA models are quite standardized, and most textbooks give more or less the same definitions for those formalisms. For stack-based automata the differences are more noticeable.

One widely studied kind of automaton is the *push-down automaton* (*PDA*). A PDA is like a finite-state machine augmented with an unbounded stack of symbols. Each move made by a PDA depends not only on the current state and input symbol, but also on the symbol that appears on the top of its stack. Each move made by a PDA not only changes the current state like a finite-state machine, but also changes the stack like a stack machine.

PDAs can be illustrated with state-transition diagrams. A state transition must show not only the input symbol *a* to be read, but also the symbol *Z* to be popped off the stack and the string of symbols *x* to be pushed back on in its place. Such a transition might be illustrated like this:



This says that if the PDA is in the state *q*, and the current input symbol is *a*, and the symbol on top of the stack is *Z*, the PDA can go to the state *r* and replace the *Z* on the stack with the string of symbols *x*.

There is considerable variety among PDA models. Some accept a string by emptying their stack, like a stack machine; some accept by ending in an accepting state, like an NFA; some must do both to accept. Some start with a special symbol on the stack; some start with a special symbol marking the end of the input string; some do both; some do neither. All these minor variations end up defining the same set of languages; the languages that can be recognized by a PDA are exactly the CFLs.

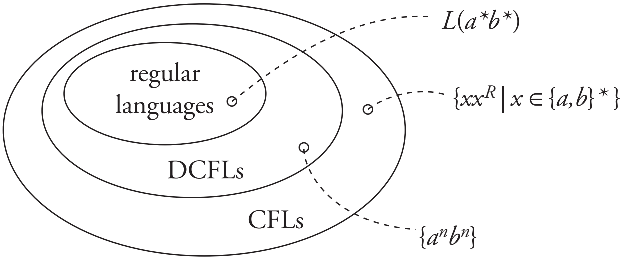
The formal definition for a PDA is more complicated than the formal definition for a stack machine, particularly in the transition function. The set of languages that can be recognized is the same as for a stack machine. So why bother with PDAs at all? One reason is because they are useful for certain kinds of proofs. For example, the proof that the CFLs are closed for intersection with regular languages is easier using PDAs, since you can essentially perform the product construction on the two state machines while preserving the stack transitions of the PDA. Another reason is that they have a narrative value: they make a good story. You start with DFAs. Add nondeterminism, and you have NFAs. Does that enlarge the class of languages you can define? No! Then add a stack, and you have PDAs. Does that enlarge the class of languages you can define? Yes! What is the new class? The CFLs!

But perhaps the most important reason for studying PDAs is that they have an interesting deterministic variety. When we studied finite automata, we saw that the transition function for an NFA gives zero or more possible moves from each configuration, while the transition function for a DFA always gives exactly one move until the end of the string is reached. One can think of DFAs as a restricted subset of NFAs. Restricting PDAs to the deterministic case yields another formalism for defining languages: the deterministic PDA (DPDA). For technical reasons, DPDAs are usually defined in a way that allows them to get stuck in some configurations, so a DPDA always has *at most* one possible move. But like a DFA, it never faces a choice of moves, so it defines a simple computational procedure for testing language membership.

When we studied finite automata, we discovered that adding nondeterminism did not add any definitional power; NFAs and DFAs can define exactly the same set of languages, the regular languages. For PDAs, the case is different; DPDAs are strictly weaker than PDAs. DPDAs define a separate, smaller class of languages: the deterministic context-free languages.

A *deterministic context-free language* (*DCFL*) is a language that is $L(M)$ for some DPDA M.

The DCFLs include all the regular languages and more:



For example, the language $\{xx^R \mid x \in \{a, b\}^*\}$ is a CFL but not a DCFL. That makes intuitive sense; a parser cannot know where the center of the string is, so it cannot make a deterministic decision about when to stop pushing and start popping. By contrast, the language $\{xcx^R \mid x \in \{a, b\}^*\}$ is a DCFL.

The DCFLs have their own closure properties, different from the closure properties of CFLs. Unlike the CFLs, the DCFLs are not closed for union: the union of two DCFLs is not necessarily a DCFL, though of course it is always a CFL. Unlike the CFLs, the DCFLs *are* closed for complement: the complement of a DCFL is always another DCFL. These different closure properties can be used to help prove that a given CFL is not a DCFL. Such proofs are usually quite tricky, since there is no tool like the pumping lemma specifically for DCFLs.

Part of the reason why we view the regular languages and the CFLs as important language classes is that they keep turning up. Regular languages turn up in DFAs, NFAs, regular expressions, and right-linear grammars, as well as many other places not mentioned in this book. CFLs turn up in CFGs, stack machines, and PDAs. DCFLs get this kind of validation as well. It turns out that they are the same as a class we have already met; the DCFLs are the same as the LR(1) languages. These are exactly the languages that can be quickly parsed using deterministic, bottom-up techniques.

15.5 Further Reading

There is a vast literature on parsing techniques. For a good place to start, see the classic

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Menlo Park, CA: Addison-Wesley, 1986.

For more information about yacc, a useful reference is

Levine, John R., Tony Mason, and Doug Brown. *lex & yacc*. Sebastopol, CA: O'Reilly & Associates, 1992.

A classic treatment of PDAs and DPDAs may be found in

Hopcroft, John, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Menlo Park, CA: Addison-Wesley, 2001.

Exercises

EXERCISE 1

Implement a predictive parser in Java for the language of expressions discussed in Section 15.1, using the grammar and parse table as given. Your Java class `PParse` should have a `main` method that takes a string from the command line and reports the sequence of productions (used at line 12 in the `predictiveParse` pseudocode) used to parse it. If the parse fails, it should say so. For example, it should have this behavior:

```
> java PParse a+b*c
S -> AR
A -> XB
X -> a
B ->
R -> +AR
A -> XB
X -> b
B -> *XB
X -> c
B ->
R ->
> java PParse a++
S -> AR
A -> XB
X -> a
B ->
R -> +AR
parse fails
```

(Unix systems usually preprocess input from the command line and give special treatment to symbols like `(`, `)`, and `*`. On these systems, you should enclose the argument to `PParse` in quotation marks, as in `java PParse "a+b*c"`.)

EXERCISE 2

Complete the pseudocode for the recursive-descent parser begun in Section 15.2.

EXERCISE 3

Implement a recursive-descent parser in Java for the language of expressions discussed in Sections 15.1 and 15.2, using the grammar given. Your Java class `RDParse` should have a `main` method whose visible behavior is exactly like that of the main method of `PParse` in Exercise 1.

EXERCISE 4

A stack machine is deterministic if it never has more than one possible move. To be deterministic, a stack machine must meet two requirements:

1. For all $\omega \in \Sigma \cup \{\varepsilon\}$ and $A \in \Gamma$, $|\delta(\omega, A)| \leq 1$. (No two entries in the table have the same "read" and "pop" columns.)
2. For all $A \in \Gamma$, if $|\delta(\varepsilon, A)| > 0$ then for all $c \in \Sigma$, $|\delta(c, A)| = 0$. (If there is an ε-move for some stack symbol A, there can be no other move for A.)

For example, consider this machine for the language $L(a^*b)$:

	read	pop	push
1.	ε	S	aS
2.	ε	S	b
3.	a	a	ε
4.	b	b	ε

This is not deterministic, because moves 1 and 2 are both possible whenever there is an S on top of the stack. A deterministic stack machine for the same language would be

	read	pop	push
1.	a	S	S
2.	b	S	ε

For each of the following languages, give a deterministic stack machine:

a. $L(01^*0)$
b. $\{xcx^R \mid x \in \{a, b\}^*\}$
c. $\{a^nb^n \mid n > 0\}$

EXERCISE 5

Deterministic stack machines, as defined in the previous exercise, are not powerful enough to recognize all the DCFLs, or even all the regular languages.

a. Prove that the language $\{a, \varepsilon\}$ cannot be defined by a deterministic stack machine.

b. Let L be any language for which there are two strings x and y, with $x \in L$, $xy \in L$, and $y \neq \varepsilon$. Prove that L cannot be defined by a deterministic stack machine. (Languages with no two such strings are said to have the *prefix property*.)

EXERCISE 6

To increase the power of the deterministic stack machines defined in Exercise 4, add an additional kind of move:

	read	pop	push
1.	$	A	x

Here, $ in the "read" column signifies a move that can be made only when all the input has been read. The entry above says that if all the input has been read and if there is an A on top of the stack, you may pop the A and push x in its place. Using this additional kind of move (along with the others), give deterministic stack machines for the following languages. (These languages lack the prefix property, so they cannot be defined using plain deterministic stack machines.)

a. $L((0 + 1) * 0)$
b. $\{a^n b^n\}$

EXERCISE 7

Using the extended definition of a deterministic stack machine from Exercise 6, prove by construction that every regular language is defined by some deterministic stack machine. (*Hint:* Start from a DFA for the language.) Include a detailed proof that the stack machine you construct works as required.

EXERCISE 8

(*Note to instructors:* Read the sample solution before assigning this one!)

In this exercise you will write a recursive-descent parser for a language of quantified Boolean formulas. Your parser will compile each string into an object representing the formula. Each such object will implement the following interface:

```
/*
 * An interface for quantified boolean formulas.
 */
public interface QBF {
  /*
   * A formula can convert itself to a string.
   */
  String toString();
}
```

The simplest of these formulas are variables, which are single, lowercase letters, like `x` and `y`. These will be compiled into objects of the `Variable` class:

```
/*
 * A QBF that is a reference to a variable, v.
 */
public class Variable implements QBF {
  private char v; // the variable to which we refer
  public Variable(char v) {
    this.v = v;
  }
  public String toString() {
    return "" + v;
  }
}
```

Any QBF can be logically negated using the ~ operator, as in ~a. Logical negations will be compiled into objects of the Complement class:

```
/*
 * A QBF for a complement: not e.
 */
public class Complement implements QBF {
  private QBF e; // the QBF we complement
  public Complement(QBF e) {
    this.e = e;
  }
  public String toString() {
    return "~(" + e + ")";;
  }
}
```

Any two QBFs can be logically ANDed using the * operator, as in a*b. These formulas will be compiled into objects of the Conjunction class:

```
/*
 * A QBF for a conjunction: lhs and rhs.
 */
public class Conjunction implements QBF {
  private QBF lhs; // the left operand
  private QBF rhs; // the right operand
  public Conjunction(QBF lhs, QBF rhs) {
    this.lhs = lhs;
    this.rhs = rhs;
  }
  public String toString() {
    return "(" + lhs + "*" + rhs + ")";
  }
}
```

Similarly, any two QBFs can be logically ORed using the + operator, as in a+b. These formulas will be compiled into objects of the Disjunction class:

```
/*
 * A QBF for a disjunction: lhs or rhs.
 */
public class Disjunction implements QBF {
  private QBF lhs; // the left operand
  private QBF rhs; // the right operand
  public Disjunction(QBF lhs, QBF rhs) {
    this.lhs = lhs;
    this.rhs = rhs;
  }
  public String toString() {
    return "(" + lhs + "+" + rhs + ")";
  }
}
```

Any QBF can be placed in the scope of a universal quantifier, as in Ax(x+~x). Syntactically, this is the symbol A, followed by a variable, followed by a QBF. The formula is true if for all bindings of x (that is, for both x=true and x=false) the inner formula is true. Universally quantified formulas are compiled into objects of the Universal class:

```
/*
 * A QBF that is universally quantified: for all v, e.
 */
public class Universal implements QBF {
  private char v; // our variable
  private QBF e; // the quantified formula
  public Universal(char v, QBF e) {
    this.v = v;
    this.e = e;
  }
  public String toString() {
    return "A" + v + "(" + e + ")";
  }
}
```

Similarly, any QBF can be placed in the scope of an existential quantifier, as in `Ex(x+~x)`. Syntactically, this is the symbol `E`, followed by a variable, followed by a QBF. The formula is true if for some binding of `x` (that is, for at least one of `x=true` and `x=false`) the inner formula is true. Existentially quantified formulas are compiled into objects of the `Existential` class:

```
/*
 * A QBF that is existentially quantified:
 * there exists v such that e.
 */
public class Existential implements QBF {
  private char v; // our variable
  private QBF e; // the quantified formula
  public Existential(char v, QBF e) {
    this.v = v;
    this.e = e;
  }
  public String toString() {
    return "E" + v + "(" + e + ")";
  }
}
```

This is a BNF grammar for the language, using <*QBF*> as the starting nonterminal:

<*QBF*> ::= `A` <*V*> <*QBF*> | `E` <*V*> <*QBF*> | <*BF*>
<*BF*> ::= <*A*> <*R*>
<*R*> ::= `+` <*A*> <*R*> | <*empty*>
<*A*> ::= <*X*> <*B*>
<*B*> ::= `*` <*X*> <*B*> | <*empty*>
<*X*> ::= `~` <*X*> | `(` <*QBF*> `)` | <*V*>
<*V*> ::= `a` | `b` | `c` | … | `z`

The alphabet of the language is exactly as shown, with no spaces. The grammar establishes the precedence: ~ has highest precedence, and quantifiers have lowest precedence. Notice that a QBF may be a parenthesized subformula, as in `(a+b)*c`. Parentheses serve to override the operator precedence, but they are not compiled into separate objects; the formulas `Axx` and `Ax(x)` should compile into exactly the same objects.

Write a Java application `QBFParser`, using the classes and grammar above, that compiles a QBF formula entered on the command line. Use a recursive-descent parser, with one parsing method for each nonterminal in the BNF grammar. Write it so that each of the parsing methods for a nonterminal returns an object of one of the classes above, representing the subformula compiled from that nonterminal. Your application should print out the final object compiled, using this as the `main` method:

```
public static void main(String[] args) {
  QBF q = parse(args[0]);
  System.out.println(q);
}
```

The `QBF` classes already have `toString` methods that will convert them back into strings for printing. So, if your compiler works correctly, the printed output will look much like the input, except that there will be exactly one pair of parentheses for each operator and each quantifier. Thus, you should see

```
> java QBFParser 'Axx'
Ax(x)
> java QBFParser 'AxEyAzx+y*z'
Ax(Ey(Az((x+(y*z)))))
> java QBFParser '((x))'
x
```

Your code will compile QBF formulas, but it won't do anything with them other than print them out. Just throw an `Error` if the string does not parse. You should not have to add anything to the `QBF` classes. (A later exercise, at the end of Chapter 20, will add methods to the `QBF` classes, so that a formula can decide whether or not it is true.)

16

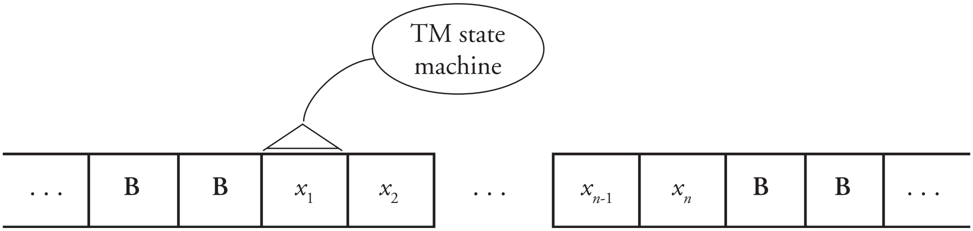
Turing Machines

We now turn to the most powerful kind of automaton we will study: the **Turing machine**. *Although it is only slightly more complicated than a finite-state machine, a Turing machine can do much more. It is, in fact, so powerful that it is the accepted champion of automata. No more powerful model exists.*

The Turing machine is the strongest of computational mechanisms, the automaton of steel. But like all superheroes it does have one weakness, revealed in this chapter: it can get stuck in an infinite loop.

16.1 Turing Machine Basics

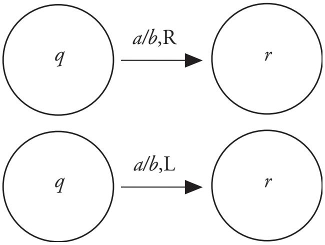
A Turing machine (TM) can be pictured like this:



The TM's input is presented on a tape with one symbol at each position. The tape extends infinitely in both directions; those positions not occupied by the input contain a special blank-cell symbol **B.** The TM has a head that can read and write symbols on the tape and can move in both directions. The head starts at the first symbol in the input string.

A state machine controls the read/write head. Each move of the TM is determined by the current state and the current input symbol. To make a move, the TM writes a symbol at the current position, makes a state transition, and moves the head one position, either left or right. If the TM enters an accepting state, it halts and accepts the input string. It does not matter what the TM leaves on its tape or where it leaves the head; it does not even matter whether the TM has read all the input symbols. As soon as the TM enters an accepting state, it halts and accepts. This mechanism for accepting is quite different from that of DFAs and NFAs. DFAs and NFAs can make transitions out of their accepting states, proceeding until all the input has been read, and they often have more than one accepting state. In a TM, transitions leaving an accepting state are never used, so there is never any real need for more than one accepting state.

We will draw TMs using state-transition diagrams. Each TM transition moves the head either to the right or to the left, as illustrated in these two forms:



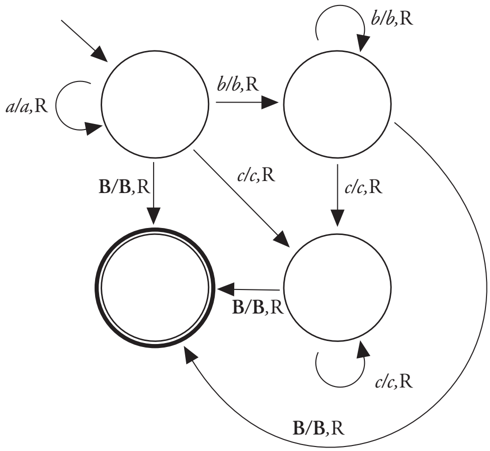
The first says that if the TM is in the state *q* and the current tape symbol is *a*, the TM can write *b* over the *a*, move the head one place to the right, and go to the state *r*. The second is the same except that the head moves one place to the left.

As you can see, TMs are not much more complicated than DFAs. TMs are deterministic and have no ε-transitions. Unlike a stack machines, TMs do not have a separate location for storage. They use one tape both for input and for scratch memory. But they have a far richer variety of behaviors than DFAs or stack machines and (as we will see) can define a far broader class of languages.

16.2 Simple TMs

TMs can easily handle all regular languages. In fact, a TM that always moves the head to the right works much like a DFA.

For example, consider the language $L(a^*b^*c^*)$. This TM for the language just makes one state transition on each symbol, much like a DFA:

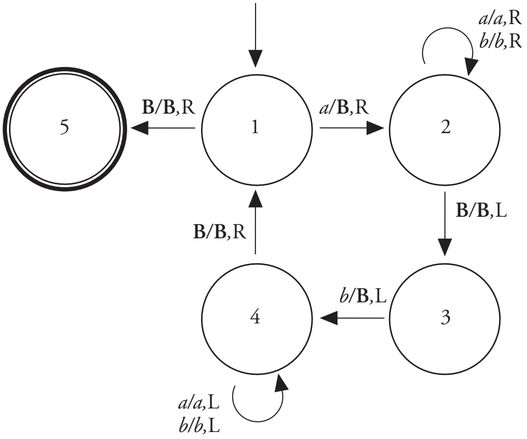


This machine does not take advantage of a TM's ability to move the head in both directions; it always moves right. It also does not take advantage of a TM's ability to write on the tape; it always rewrites the same symbol just read. Of course, since it never moves to the left, it makes no difference what it writes on the tape. Any symbols written are left behind and never revisited. So it could just as well write a **B** over every symbol, erasing as it reads.

TMs can also easily handle all context-free languages. For example, consider the language $\{a^nb^n\}$. Here is a series of steps for a TM that accepts this language:

1. If the current symbol is **B**, go to Step 5. If the current symbol is an *a*, write a **B** over it and go to Step 2.
2. Move right past any *a*s and *b*s. At the first **B**, move left one symbol and go to Step 3.
3. If the current symbol is *b*, write a **B** over it and go to Step 4.
4. Move left past any *a*s and *b*s. At the first **B**, move right one symbol and go to Step 1.
5. Accept.

This TM works by repeatedly erasing the first symbol (*a*) and the last symbol (*b*) from the input string. If the input string was in $\{a^nb^n\}$, this process eventually erases the entire input string, leaving a blank tape. Here is an illustration of the TM, with states numbered to match the steps above.



It is also possible to take any stack machine and convert it into an equivalent TM that uses the infinite tape as a stack. But, as in the example above, it is often easier to find some non-stack-oriented approach.

16.3 A TM for $\{a^nb^nc^n\}$

Consider the language $\{a^nb^nc^n\}$, which as we have seen is not a context-free language. Here is a series of steps for a TM that accepts this language:

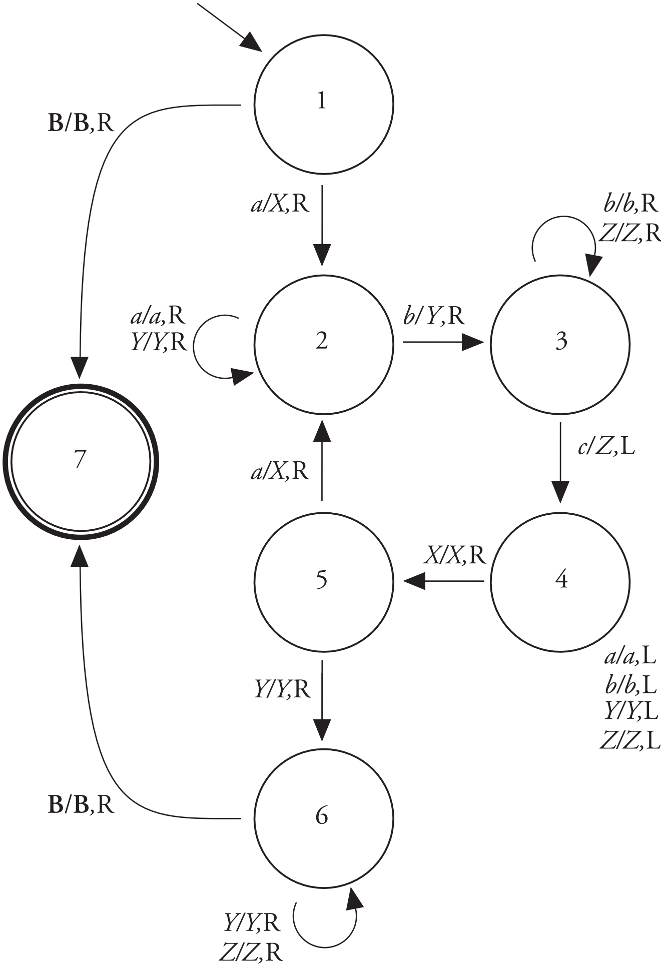
1. If the current symbol is **B**, go to Step 7. If the current symbol is *a*, write an *X* over it and go to Step 2.
2. Move right past any *a*s and *Y*s. At the first *b*, write *Y* over it and go to Step 3.
3. Move right past any *b*s and *Z*s. At the first *c*, write *Z* over it and go to Step 4.
4. Move left past any *a*s, *b*s, *Z*s, and *Y*s. At the first *X*, move right one symbol and go to Step 5.
5. If the current symbol is *a*, write an *X* and go to step 2. If the current symbol is *Y* go to Step 6.
6. Move right past any *Y*s and *Z*s. At the first **B**, go to Step 7.
7. Accept.

The strategy followed by this TM is a simple one. Suppose for example that our input string is *aaabbbccc*. In Steps 1, 2, and 3 it overwrites the first *a* with *X*, then overwrites the first *b* with *Y* and the first *c* with *Z*. The resulting string is *XaaYbbZcc.* In Step 4 the TM moves back left to the *X*, then one place right to the first remaining *a*. Now, using Steps 5, 2, and 3, it overwrites another *a*, *b,* and *c*, producing the string *XXaYYbZZc*. Then the whole process repeats one more time, producing the string *XXXYYYZZZ*. After that, Step 5 finds that the first symbol to the right of the *X*s is a *Y*. This indicates that all the *a*s have been overwritten.

Step 6 then checks that there is nothing left but *Y*s and *Z*s, indicating that the number of *b*s and *c*s was the same as the number of *a*s. Then in step 7 the machine accepts.

If the input string is not in $\{a^n b^n c^n\}$ the TM gets stuck somewhere and does not reach Step 7. For example, if the input string is *acb*, the TM overwrites the *a* with an *X*, but it cannot handle the *c* it sees in Step 2 and gets stuck there. Try it for yourself on some other strings like *aababbccc* and *aaabbbbccc*, and you will see that if the input string does not have the same number of *a*s, *b*s, and *c*s, or if it is not in $L(a^*b^*c^*)$ at all, the TM cannot reach Step 7.

Here is an illustration of the TM for $\{a^n b^n c^n\}$ that we just outlined. The states are numbered to match the steps in the outline.



16.4 The 7-Tuple

A TM M is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, \mathbf{B}, q_0, F)$, where

Q is the finite set of states
Σ is the input alphabet
Γ is the tape alphabet, with $\Sigma \subset \Gamma$ and $Q \cap \Gamma = \{\}$
$\delta \in (Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\})$ is the transition function
$\mathbf{B}$ is the blank symbol, $\mathbf{B} \in \Gamma$, $\mathbf{B} \notin \Sigma$
$q_0 \in Q$ is the start state
$F \subseteq Q$ is the set of accepting states

Note that the tape alphabet Γ includes all of the input alphabet, plus at least one additional symbol **B**. The requirement that $Q \cap \Gamma = \{\}$ means no state is also a symbol—not an onerous requirement, but a necessary one for the definition of IDs that follows.

The transitions of the TM are defined by the transition function δ. The δ function takes as parameters a state $q \in Q$ (the current state) and a symbol $X \in \Gamma$ (the symbol at the current position of the head). The value $\delta(q, X)$ is a triple (p, Y, D), where $p \in Q$ is the next state, $Y \in \Gamma$ is the symbol to write at the current head position, and D is one of the directions L or R, indicating whether the head should move left or right. Note that a TM is deterministic in the sense that it has at most one transition at any point. However, the δ function need not be defined over its whole domain, so there may be some q and X with no move $\delta(q, X)$.

16.5 The Languages Defined by a TM

An instantaneous description for a TM represents its entire configuration in a single string: the contents of the tape, the state, and the position of the head. The following definitions are all made with respect to some fixed TM $M = (Q, \Sigma, \Gamma, \delta, \mathbf{B}, q_0, F)$:

An instantaneous description for a TM is a string xqy, where $x \in \Gamma^*$ represents the tape *to the left of* the head, q is the current state, and $y \in \Gamma^*$ represents the tape *at and to the right of* the head. In x, leading **B**s from the tape are omitted, except when there are only **B**s to the left of the head, in which case $x = \mathbf{B}$. Likewise, in y, trailing **B**s from the tape are omitted, except when there are only **B**s at and to the right of the head, in which case $y = \mathbf{B}$. We define a function *idfix*(z) that normalizes an ID string z in this way, removing (or adding) leading and trailing **B**s as necessary.

Although the tape extends infinitely in both directions, it contains only **B**s in those positions that are outside the input string and not yet written by the TM. So, because the leading **B**s on the left and the trailing **B**s on the right are suppressed, an ID is always a finite string. The state q in the ID string tells what state the TM is in and also shows the position of the head.

The δ function for a TM determines a relation $\mapsto$ on IDs; we write $I \mapsto J$ if I is an ID and J is an ID that follows from I after one move of the TM.

$\mapsto$ is a relation on IDs, defined by the δ function for the TM. For any $x \in \Gamma^*$, $c \in \Gamma$, $q \in Q$, $a \in \Gamma$, and $y \in \Gamma^*$, we have

1. Left moves: if $\delta(q, a) = (p, b, \text{L})$ then $xcqay \mapsto \mathit{idfix}(xpcby)$
2. Right moves: if $\delta(q, a) = (p, b, \text{R})$ then $xcqay \mapsto \mathit{idfix}(xcbpy)$

For example, consider making a left move. Suppose the machine is in the configuration given by the ID *xcqay*. So it is in the state q, reading the tape symbol a. Thus $\delta(q, a)$ gives the transition. If that transition is $\delta(q, a) = (p, b, \text{L})$, the machine can write a b over the a, go to the state p, and move the head one position to the left. So after the move the ID is *xpcby*: the a has been replaced by a b, and the head has moved left so that it is now reading the tape symbol c.

Next, as usual, we define an extended relation $\mapsto^*$ for sequences of zero or more steps:

$\mapsto^*$ is a relation on IDs, with $I \mapsto^* J$ if and only if there is a sequence of zero or more $\mapsto$ relations that starts with I and ends with J.

Notice here that $\mapsto^*$ is reflexive: for any ID I, $I \mapsto^* I$ by a sequence of zero moves. Using the $\mapsto^*$ relation, we can define the language accepted by M:

The language accepted by a TM M is

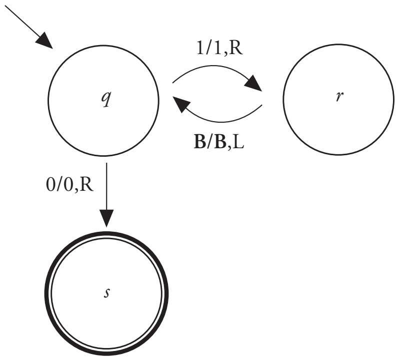
$$L(M) = \{x \in \Sigma^* \mid \mathit{idfix}(q_0x) \mapsto^* ypz \text{ for some } p \in F\}$$

In this definition, the TM starts in its start state reading the first symbol of the input string x (or reading **B**, if x is ε—the initial ID normalized by *idfix* will be either $\mathbf{B}q_0x$ if $x \neq \varepsilon$ or $\mathbf{B}q_0\mathbf{B}$ if $x = \varepsilon$). It accepts x if it has a sequence of zero or more moves that goes to an accepting state, regardless of what is left on the tape, and regardless of the final position of the head.

16.6 To Halt or Not to Halt

By the definition above, a TM accepts its input if it ever reaches an accepting state. Although nothing in the definition of a TM specifically forbids transitions that leave accepting states, there is no point in including such transitions, since they do not affect the language accepted. The machine is considered to *halt* if it ever reaches an accepting state. It also halts on a given input if it gets stuck—if it enters an ID I in a nonaccepting state, for which there is no ID J with $I \mapsto J$. There is also a third possibility: a TM may run forever on a given input. The TM may always have a move, yet never reach an accepting state.

Consider this TM, for example:



Formally this is $M = (\{q, r, s\}, \{0, 1\}, \{0, 1, \mathbf{B}\}, \delta, \mathbf{B}, q, \{s\})$, where the transition function δ is given by

$$\delta(q, 1) = (r, 1, \mathrm{R})$$
$$\delta(q, 0) = (s, 0, \mathrm{R})$$
$$\delta(r, \mathbf{B}) = (q, \mathbf{B}, \mathrm{L})$$

Given the input string 0, *M* accepts (and thus halts):

$$\mathbf{B}q0 \mapsto 0s\mathbf{B}$$

In general, *M* will accept a string if and only if it begins with a 0, so $L(M)$ is the regular language $L(0(0 + 1)^*)$. But *M* also exhibits the other two possible outcomes. Given the input string ε, *M* rejects by halting in a nonaccepting state—the start state, since there is no move from there:

$$\mathbf{B}q\mathbf{B} \mapsto ?$$

Given the input string 1, *M* does not accept or reject, but runs forever without reaching an accepting state:

$$\mathbf{B}q1 \mapsto 1r\mathbf{B} \mapsto \mathbf{B}q1 \mapsto 1r\mathbf{B} \mapsto \mathbf{B}q1 \mapsto \dots$$

Of course, it is possible to make a TM for the language $L(0(0 + 1)^*)$ that halts on all inputs. In general, though, we will see that it is not always possible for TMs to avoid infinite loops. We might say that the possibility of running forever is the price TMs pay for their great power.

In previous chapters we have seen occasional glimpses of nonterminating computations. Stack machines and NFAs can in some sense run forever, since they can contain cycles of ε-transitions. But those nonterminating computations are never necessary. We saw how to construct an equivalent DFA for any NFA—and DFAs always terminate. Similarly, it is possible to show that for any stack machine with a cycle of ε-transitions you can construct an equivalent one without a cycle of ε-transitions—which therefore always terminates. Thus, the possibility of having a computation run forever has been a minor nuisance, easily avoidable.

With TMs, however, the situation is different. There are three possible outcomes when a TM is run: it may accept, it may reject, or it may run forever. A TM M actually partitions Σ^* into three subsets: those that make M accept (which we have named $L(M)$), those that make M reject, and those that make M run forever. Instead of just defining $L(M)$, then, a TM really defines *three* languages:

The language accepted by a TM M is

$$L(M) = \{x \in \Sigma^* \mid idfix(q_0x) \mapsto^* ypz \text{ for some } p \in F\}.$$

The language rejected by a TM M is

$$R(M) = \{x \in \Sigma^* \mid x \notin L(M) \text{ and there is some ID } I \text{ with } idfix(q_0x) \mapsto^* I \text{ and no } J \text{ with } I \mapsto J\}.$$

The language on which a TM M runs forever is

$$F(M) = \{x \in \Sigma^* \mid x \notin L(M) \text{ and x} \notin R(M)\}.$$

That third possibility is critical. There is a special name for TMs that halt on all inputs; they are called *total TMs*.

A TM M is a total TM if and only if $F(M) = \{\}$.

In the next few chapters we will see some important results about the nonterminating behavior of TMs. We will see that if we restrict our attention to total TMs we lose power. In other words, we will see that these two sets of languages are not the same:

A *recursively enumerable* (*RE*) language is one that is $L(M)$ for some TM M.
A *recursive language* is one that is $L(M)$ for some total TM M.

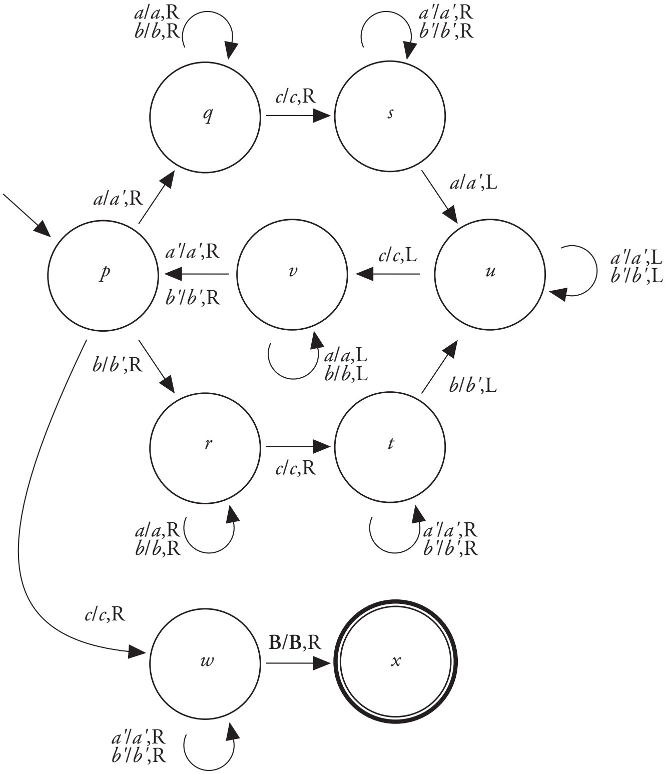
These may seem like odd names—what does recursion have to do with TMs? But it turns out that these two sets of languages were independently identified by a number of different researchers working in different areas of mathematics. Although we are defining these sets in terms of TMs, the standard names come from mathematical studies of computability using the theory of recursive functions.

16.7 A TM for $\{xcx \mid x \in \{a, b\}^*\}$

Consider the non-context-free language $\{xcx \mid x \in \{a, b\}^*\}$. A TM for this language can work by checking each symbol in the first x against the corresponding symbol in the second x. To keep track of where it is in this process, our TM will mark the symbols that have been checked. (To mark a symbol, the TM just overwrites it with a marked version of the same symbol, for instance by overwriting a with a'.) Here is the plan:

1. If the first symbol is a *c*, go to Step 6.
2. Mark the first symbol and remember what it was.
3. Move right up to the *c*, then right past any marked symbols to the first unmarked *a* or *b*.
4. Mark it and check that it is the same symbol we saw in Step 2.
5. Move back left to the first unmarked symbol in the string, and go to Step 1.
6. Check that all input after the *c* has been marked.

Here is a TM that implements this strategy.



This TM always halts, showing that $\{xcx \mid x \in \{a, b\}^*\}$ is a recursive language.

This example demonstrates two important higher-level techniques of TM construction. The first is the idea of *marking* cells of the tape. As the machine illustrates, a cell can be marked simply by overwriting the symbol there with a marked version of that symbol. In our machine, the input alphabet is $\{a, b, c\}$, but the tape alphabet is $\{a, b, c, a', b', \mathbf{B}\}$—it includes marked versions of the symbols *a* and *b*.

The second is the technique of using the TM states to record some finite information. In our example, we needed the machine to remember whether *a* or *b* was marked in the state *p*. So we constructed the machine with two paths of states from *p* to *u*. The machine is in the state *q* or *s* if it saw *a* and is expecting to see a matching *a* in the second half of the string; it is in the state *r* or *t* if it saw *b* and is expecting to see a matching *b*. In a high-level description of the machine we can simply say that it "remembers" whether *a* or *b* was seen; we can always implement any finite memory using the states of the TM.

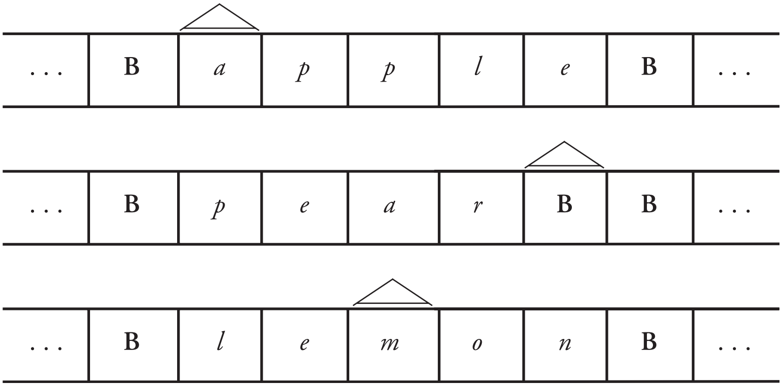
16.8 Three Tapes

To handle more challenging problems, we will introduce a larger cousin of the basic TM. This one works just like the basic model, except that it has three independent tapes, each with its own independent read/write head. Recall that a one-tape TM is a 7-tuple like this:

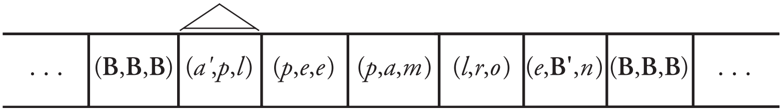
$$M = (Q, \Sigma, \Gamma, \delta, \mathbf{B}, q_0, F)$$

A three-tape TM may be defined with the same kind of 7-tuple, changing only the transition function. For a three-tape TM, the transition function would take four inputs—the current state and the symbol being read by the each of the three heads—and the value of $\delta(q, X_1, X_2, X_3) = (p, Y_1, D_1, Y_2, D_2, Y_3, D_3)$ would give the new state, plus a symbol to write and a direction to move for each of the three heads.

The three-tape TM model is easier to program than our basic model, but no more powerful. For any three-tape TM we can construct an equivalent basic TM that simulates it. Imagine the three tapes stretched out side by side, each with its own head position:



We can encode all this information on one tape, using an enlarged tape alphabet:


... (B,B,B) (a',p,l) (p,e,e) (p,a,m) (l,r,o) (e,B',n) (B,B,B) ...

Each symbol in this new alphabet is a triple, encoding all the information about three symbols from the original alphabet and using markings to indicate the presence or absence of the head at that position. For example, the symbol (*a*', *p*, *l*) indicates that, in the three-tape machine, there is an *a* on the first tape, a *p* on the second tape, and an *l* on the third tape; the head of the first tape is at this location, while the heads of the other two tapes are elsewhere. This alphabet of triples is, of course, much larger than the original alphabet. The original alphabet Γ is first doubled in size by the addition of a marked version of each symbol; for example, $\Gamma = \{a, \mathbf{B}\}$ becomes $\Gamma_2 = \{a, a', \mathbf{B}, \mathbf{B}'\}$. This enlarged alphabet is then *cubed* in size by being formed into 3-tuples; for example, $\Gamma_2 = \{a, a', b, b'\}$ becomes

$$\begin{aligned}\Gamma_3 &= (\Gamma_2 \times \Gamma_2 \times \Gamma_2)\\ &= \{(a, a, a), (a, a, a'), (a, a, \mathbf{B}), (a, a, \mathbf{B}'), (a, a', a), (a, a', a'), (a, a', \mathbf{B}), (a, a', \mathbf{B}'), \ldots\}\end{aligned}$$

and so on, for a total of 64 symbols.

Now let M_3 be any three-tape TM. We can construct an ordinary TM M_1 to simulate M_3. M_1 will use the alphabet of triples, as shown above, and will use $(\mathbf{B}, \mathbf{B}, \mathbf{B})$ as its blank symbol. To simulate one move of the three-tape TM, M_1 will use a two-pass strategy like this:

1. Make a left-to-right pass over the tape until all three marked symbols have been found. Use our state to record M_3's state and the symbols being read at each of M_3's three heads. This information determines the move M_3 will make. If M_3 halts, halt M_1 as well—in an accepting state if M_3 was in an accepting state or in a nonaccepting state if not.
2. Make a right-to-left pass over the tape, carrying out M_3's actions at each of its three head positions. (This means writing what M_3 would write and also moving the marks to record M_3's new head positions.) Leave the head at the leftmost cell containing any marked symbol. Go back to step 1 for the next move.

By repeating these passes, M_1 simulates M_3. Of course, it makes far more moves than M_3. It uses a far larger alphabet than M_3. It uses far more states than M_3. But none of that matters for questions of computability. M_1 reaches exactly the same decisions as M_3 for all input strings: accepting where M_3 accepts, rejecting where M_3 rejects, and running forever where M_3 runs forever. Using this construction we can conclude the following:

Theorem 16.1: For any given partion of a Σ^* into three subsets L, R, and F, there is a three-tape TM M_3 with $L(M_3) = L$, $R(M_3) = R$, and $F(M_3) = F$, if and only if there is a one-tape TM M_1 with $L(M_1) = L$, $R(M_1) = R$, and $F(M_1) = F$.

Proof sketch: As indicated above, given any three-tape TM, we can construct a one-tape TM with the same behaviors. In the other direction, any one-tape TM can easily be simulated by a three-tape TM that simply ignores two of its three tapes.

16.9 Simulating DFAs

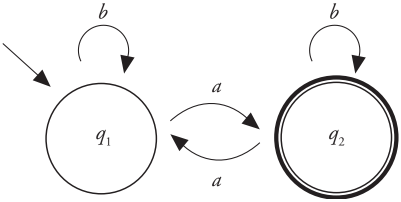
We have given examples of TMs that define a variety of languages: regular, context free, and noncontext free. We can now demonstrate what these examples suggest: that all regular languages and all context-free languages are recursive.

Ordinarily, we would prove such claims by simple constructions. For example, we might give a construction that takes any DFA and constructs an equivalent TM. Such constructions are certainly possible, but we'll tackle something more interesting: a single TM that can simulate any DFA. This TM will take a pair of inputs: a DFA (encoded as a string) and the string for that DFA to read. It will decide whether the given DFA accepts the given string. We'll specify it as a three-tape TM, knowing that (by Theorem 16.1) there is an equivalent one-tape TM.

Our TM will use the alphabet {0, 1}, like any good computer. First, we need to represent the alphabet of the DFA. We'll number the symbols of the DFA's alphabet arbitrarily as $\Sigma = \{\sigma_1, \sigma_2, ...\}$. The TM will use the string 1^i to represent each symbol σ_i. Using 0 as a separator, we can represent any string over any alphabet this way. For example, suppose the DFA's alphabet is $\{a, b\}$. We'll number a as σ_1 and b as σ_2. Then the string *abba* can be represented in our TM's alphabet as $1^1 0 1^2 0 1^2 0 1^1 = 101101101$.

Next, we need to represent the states of the DFA. We'll number the states as $Q = \{q_1, q_2, ...\}$, choosing the start state for q_1 and numbering the other states arbitrarily. Then we can use the string 1^i to represent state q_i.

Next, we need to encode the transition function of the DFA as a string. We will encode each transition $\delta(q_i, \sigma_j) = q_k$ as a string $1^i 0 1^j 0 1^k$ and encode the entire function δ as a list of such transitions, in any order, using 0 as a separator. For example, consider this DFA M:



Numbering a as σ_1 and b as σ_2, we have this transition function for M:

$$\delta(q_1, \sigma_1) = q_2$$
$$\delta(q_1, \sigma_2) = q_1$$
$$\delta(q_2, \sigma_1) = q_1$$
$$\delta(q_2, \sigma_2) = q_2$$

that is encoded as this string:

101011 0 101101 0 110101 0 11011011

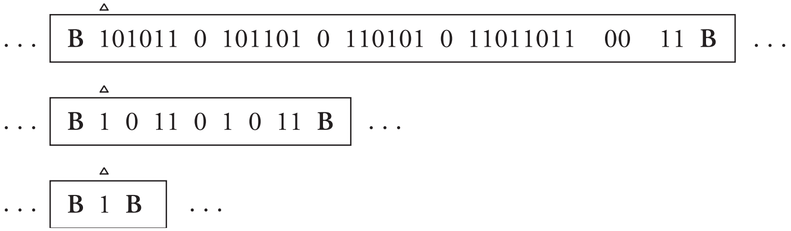
(The spaces shown are not part of the string, but emphasize the division of the string into a list of four transitions separated by 0s.)

Finally, we need to represent the set of accepting states. We are already representing each state q_i as 1^i, so we can represent the set of accepting states as a list of such strings, using 0 as the separator. Putting this all together, we can represent the entire DFA by concatenating the transition-function string with the accepting-states string, using 00 as a separator. For the example DFA *M*, this is

101011 0 101101 0 110101 0 11011011 00 11

This gives us a way to encode a DFA as a string of 1s and 0s. It should come as no surprise that this can be done—programmers are accustomed to the idea that everything physical computers manipulate is somehow encoded using the binary alphabet. Our goal now is to define a TM that takes one of these encoded DFAs along with an encoded input string for that DFA and decides by a process of simulation whether the DFA accepts the string.

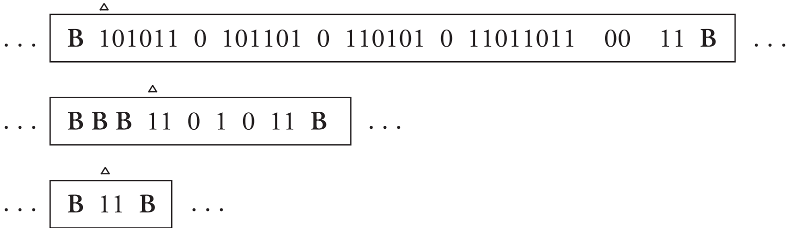
We'll express this as a three-tape TM (knowing that there is an equivalent, though more complex, one-tape TM). Our TM will use the first tape to hold the DFA being simulated and use the second tape to hold the DFA's input string. Both tapes are encoded using the alphabet {0, 1}, as explained in the previous section. The third tape will hold the DFA's current state, representing each state q_i as 1^i. This shows the initial configuration of the simulator, using the example DFA *M*, defined above, and the input string *abab*.



```
       △
...  B 101011 0 101101 0 110101 0 11011011 00 11 B  ...

       △
...  B 1 0 11 0 1 0 11 B  ...

       △
...  B 1 B  ...
```

As before, the spaces are added in the illustration to emphasize the encoding groups.

To simulate a move of the DFA, our TM will perform one state transition and erase one encoded symbol of input. In our example above, the machine is in the state q_1 (as shown by the third tape) reading the input $\sigma_1 = a$ (as shown by the second tape). The first transition recorded on the first tape says what to do in that situation: go to the state σ_2. So after one simulated move, the machine's state will be this:



```
       △
...  B 101011 0 101101 0 110101 0 11011011 00 11 B  ...

           △
...  B B B 11 0 1 0 11 B  ...

       △
...  B 11 B  ...
```

Of course, it takes many moves of the TM to accomplish this simulation of one move of the DFA. That one-move simulation is repeated until all the input has been erased; then the final state of the DFA is checked to see whether or not the DFA accepts the string. This is the strategy in more detail:

1. If the second tape is not empty, go to Step 2. Otherwise, the DFA's input has all been read and its computation is finished. Search the list of accepting states on the first tape for a match with the DFA's current state on the third tape. If a match is found, halt in an accepting state; if it is not found, halt in a nonaccepting state.
2. Search the first tape for a delta-function move $1^i01^j01^k$, where 1^i matches the current state on the third tape and 1^j matches the first input symbol on the second tape. (Since a DFA has a move for every combination of state and symbol, this search will always succeed.) The 1^k part now gives the next state for the DFA.
3. Replace the 1^i on the third tape with 1^k, and erase the 1^j and any subsequent separator from the second tape, by writing **B**s over them. This completes one move of the DFA; go to Step 1.

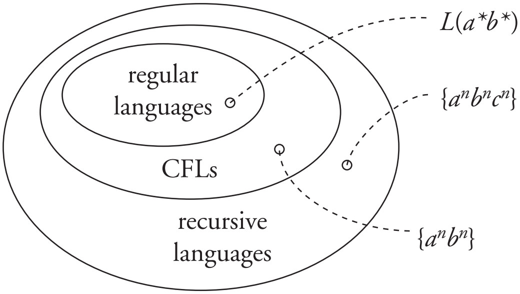
The TM that simulates a DFA uses only a fixed, finite portion of each of its tapes. It takes the encoded machine as input on its first tape and never uses more; it takes the encoded input string on its second tape and never uses more; and it stores only one encoded state on its third tape, which takes no more cells than there are states in the DFA. The existence of this TM shows, indirectly, that all regular languages are recursive. (For any regular language, we could construct a specialized version of the DFA-simulating TM, with a DFA for that language wired in.)

One important detail we are skipping is the question of what to do with improper inputs. What should the DFA simulator do with all the strings in {0, 1}* that don't properly encode DFAs and inputs? If our simulator needs to reject such strings, it will have to check the encoding before beginning the simulation. For example, since the input is supposed to be a DFA, it will have to check that there is exactly one transition from every state on every symbol in the alphabet.

16.10 Simulating Other Automata

An interpreter is a program that takes another program as input and carries out the instructions of that input program. This is a familiar concept from software development; there are BASIC interpreters, Java bytecode interpreters, and Perl interpreters, written in diverse languages. We have just seen how to construct a TM that acts as a DFA interpreter. The same techniques can also be used to build interpreters for all our other kinds of automata.

We could, for example, make a three-tape TM that interprets any stack machine. (Use the first tape for an encoding of the stack machine to be simulated, the second for the input string, and the third for the working stack.) The existence of this TM shows, indirectly, that all context-free languages are recursive. Because we have already seen an example of a language that is recursive but not context free, we conclude that the recursive languages are a proper superset of the CFLs. This picture captures our language classes so far:



Using the same techniques, a TM can even act as a TM-interpreter. Such a TM is called a *universal Turing machine*. To construct a universal TM, we first design an encoding of TMs as strings of 1s and 0s, whose principal content is an enumeration of the transition function. Then we can construct a three-tape universal TM, taking an encoded TM as input on the first tape and encoded input string on the second tape and using the third tape to record its current state. The second tape, containing the input to the simulated TM, also serves as that TM's working tape, so we need to use marked symbols there to record the location of the simulated TM's head. Simulating a move is simply a matter of looking up the appropriate transition on the first tape, making the necessary change on the second tape, and updating the state on the third tape. If the simulated TM has no next move, we make our simulator halt—in an accepting state if the simulated TM accepted or in a rejecting state if it rejected. Of course, if the simulated TM runs forever, our simulating TM will too.

Like all the TMs described in the last few sections, our universal TM is merely sketched here. In fact, we have not given a detailed construction of any TM since Section 16.7. We did not give a complete, formal construction for converting three-tape TMs to standard TMs; we just presented a sketch. We did not really construct any TMs to simulate other automata; we just sketched how certain three-tape TMs could be constructed and then relied on the previous construction to conclude that equivalent standard TMs exist. In effect, we demonstrated that all these TMs could be constructed, without actually constructing them.

When you want to show that something can be done using a Turing machine, rough outlines are often more effective than detailed constructions. That is because it is very difficult to figure out what a TM does just by inspection—very difficult even for small TMs, and inhumanly difficult for large ones. Merely showing a large TM and claiming that it recognizes a certain language is not a good way to convince anyone that the language can be recognized by a TM. Instead, it is more convincing to give less detail—to describe in outline how a TM can be constructed. Once you're convinced it can be done there is no point in actually doing it!

No point, that is, except for fun. People have actually constructed many universal TMs in detail. It serves virtually no purpose, but it is an interesting puzzle, especially if you try to do it with the smallest possible number of states and the smallest possible alphabet. There is a fine introduction to this puzzle in Marvin Minsky's classic book, noted below.

16.11 Further Reading

Marvin Minsky illustrates a small (but not record-holding) universal Turing machine using seven states and an alphabet of four symbols in this out-of-print classic:

Minsky, Marvin. *Computation: Finite and Infinite Machines*. Upper Saddle River, NJ: Prentice Hall, Inc., 1967.

Although slightly dated, it is an elegant and very rewarding book.

Exercises

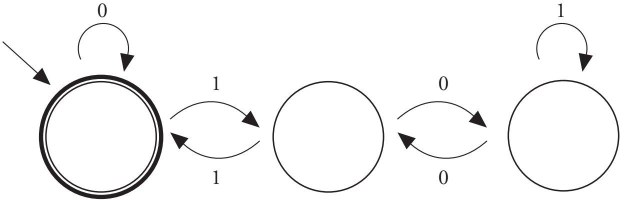
These exercises refer to ordinary, one-tape TMs. Show completed TMs in detail. (Yes, we just asserted that high-level descriptions are more readable and that there is no point in constructing large TMs in detail, except for fun. Conclusion: These exercises must be fun.)

EXERCISE 1

Give a TM for the language $L(a^*b^*)$.

EXERCISE 2

Give a TM for the language accepted by this DFA:



EXERCISE 3

Give a TM for $\{x \in \{a, b\}^* \mid x$ contains at least three consecutive as$\}$.

EXERCISE 4

Give a TM for the language generated by this grammar:

$$S \rightarrow aSa \mid bSb \mid c$$

EXERCISE 5

The TM in Section 16.3 for $\{a^nb^nc^n\}$ has two states than can be merged. Rewrite it with these states merged, so that it has one less state in total.

EXERCISE 6

Give a TM for $\{a^nb^nc^{2n}\}$.

EXERCISE 7

Give a TM that accepts the same language as the TM in Section 16.6 but always halts.

EXERCISE 8

Construct a TM that accepts the language $\{a^n b^n c^n\}$ and runs forever on all strings that are not accepted. Thus, for your TM M you will have $L(M) = \{a^n b^n c^n\}$, $R(M) = \{\}$, and $F(M) = \{x \in \{a, b, c\}^* \mid x \notin \{a^n b^n c^n\}\}$. *Hint:* Modify the machine from Section 16.3.

EXERCISE 9

Construct a TM that accepts the language $\{x \in \{a, b, c\}^* \mid x \notin \{a^n b^n c^n\}\}$. *Hint:* Modify the machine from Section 16.3.

EXERCISE 10

The TM of Section 16.7 was chosen to illustrate the concept of marking cells of the tape. A simpler TM for the same language can be implemented if the symbols in the first half of the string are not marked but simply erased (by being overwritten with a **B**). Using this idea, reimplement the TM. (You should be able to eliminate a state.)

EXERCISE 11

Construct a TM for the language $\{0^i 1^j 0^k \mid i > 0, j > 0, \text{and } i + j = k\}$.

EXERCISE 12

Construct a TM for the language $\{a^n b^n a^i b^j\}$.

EXERCISE 13

Construct a TM for the language $L(a^*b^* + b^*a^*)$.

EXERCISE 14

Construct a TM that runs forever on inputs in $L(a^*b^* + b^*a^*)$, but halts on all other inputs in $\{a, b\}^*$. (It doesn't matter whether or not it accepts the other inputs.)

EXERCISE 15

Consider the language of encoded DFAs from Section 16.9. By removing the requirement that there be exactly one transition from every state on every symbol in the alphabet, we have a similar language of encoded NFAs. Give a regular expression for this language.

EXERCISE 16

Construct a TM that accepts the language of strings that encode NFAs that have at least one accepting state; that is, your TM must check its input for membership in the regular language of the previous exercise and must check that the NFA encoded has at least one accepting state.

EXERCISE 17

Construct a TM that takes an input of the form 1^i0x, where x is an encoding of a DFA as in Section 16.9. Your TM should halt leaving only the string 1^j on the tape, where $\delta(q_1, \sigma_i) = q_j$. Other than this string 1^j, the final tape should contain nothing but **B**s. The behavior of your TM on improperly formatted inputs does not matter, and it does not matter whether the TM accepts or rejects.

EXERCISE 18

Construct a TM that takes an input of the form 1^i0x, where x is an encoding of a DFA as in Section 16.9. Your TM should accept if and only if the DFA accepts the one-symbol string σ_i. The behavior of your TM on improperly formatted inputs does not matter.

17 Computability

Computability *is the mysterious border of our realm. Within the border is our neatly ordered kingdom, occupied by solvable problems. These problems range from the trivially simple to the hideously complicated, but all are algorithmically solvable given unbounded time and resources. The shape of this border was first clearly perceived in the 1930s.*

17.1 Turing-Computable Functions

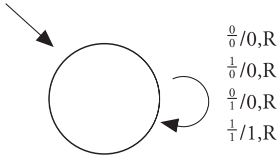
Consider any total TM—any TM that always halts. But instead of focusing on the state in which it halts (accepting or not), think of the final contents of the tape as the TM's output. Viewed this way, any total TM implements a function with string input and output. This doesn't change any of the details of TM operation that we saw in the previous chapter; it only changes how we look at the result of the TM's execution.

A function $f: \Sigma^* \to \Gamma^*$ is *Turing computable* if and only if there is some total TM M such that for all $x \in \Sigma^*$, if M starts with x on the tape, M halts with $f(x)$ on the tape.

In this definition, the final state of the TM is not considered, nor does it matter where the TM's head is positioned when it halts.

Let's look at an example. In Java and the other C-family languages we write the expression `x&y` to express the bitwise AND of two integer operands x and y. We'll show that this is Turing computable, and to make this more interesting, we'll extend it to allow the operands to be arbitrarily long strings of bits. Imagine first that the TM's input is presented with the operands already stacked up, using $\Sigma=\{\frac{0}{0},\frac{0}{1},\frac{1}{0},\frac{1}{1}\}$ as the input alphabet. To show that this *and* function is Turing computable, we need to find a TM that, given any $x \in \Sigma^*$ as input, halts with *and*(x) on its tape. For example, given $\frac{0}{0}\frac{1}{1}\frac{1}{0}\frac{0}{1}\frac{1}{1}\frac{1}{0}$ as input, our TM should halt with 010010 on its tape, since *and* $(\frac{0}{0}\frac{1}{1}\frac{1}{0}\frac{0}{1}\frac{1}{1}\frac{1}{0})=010010$.

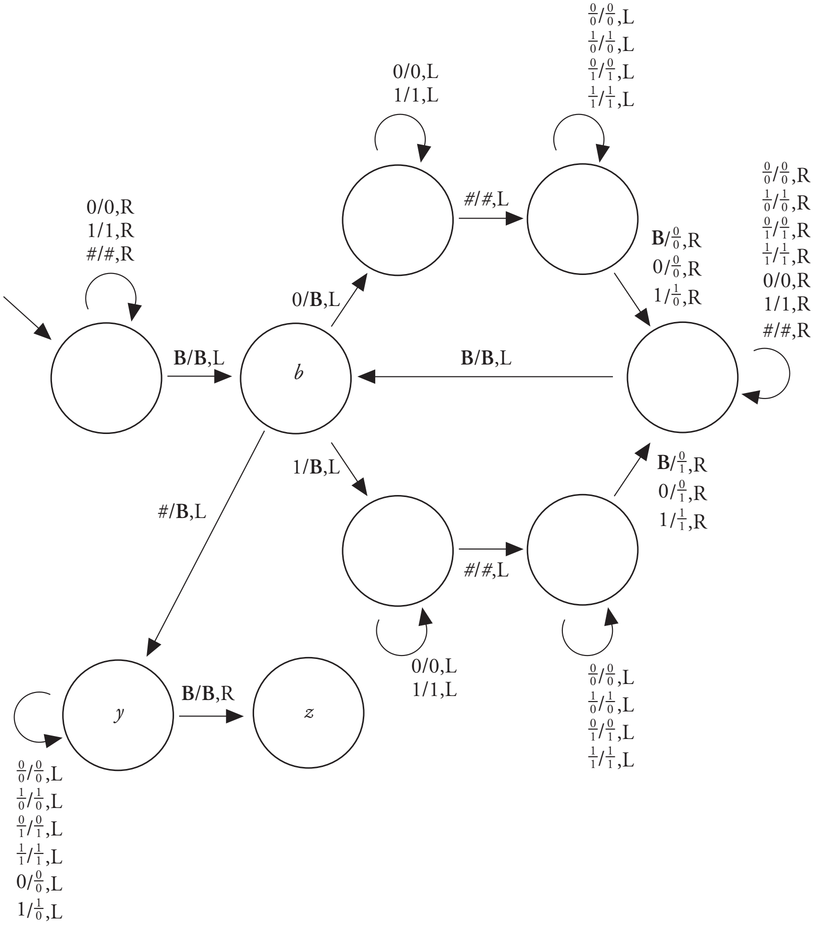
But nothing could be simpler:



This *and* machine simply proceeds from left to right over its input, overwriting each stacked-up pair of input bits with the corresponding output bit. When the end of the input is reached, the machine halts, having no transition on a **B**. This machine implements our *and* function, demonstrating that the function is Turing computable. Notice that the *and* TM has no accepting states, so $L(\textit{and}) = \{\}$. But we're not interested in the language it defines; we're interested in the function it implements.

17.2 TM Composition

You might argue that we cheated in that last example by assuming that the input was presented in stacked form, using an input alphabet completely different from the output alphabet. What if the input is presented as two, linear, binary operands? What if our input is not $\frac{0}{0}\frac{1}{1}\frac{1}{0}\frac{0}{1}\frac{1}{1}\frac{1}{0}$ but 011011#010110, where some special symbol # separates the two operands? It's simple: we can preprocess the input into the stacked form we need. We can define a function *stacker* that converts linear inputs into stacked form, so that for example *stacker*(011011#010110)=$\frac{0}{0}\frac{1}{1}\frac{1}{0}\frac{0}{1}\frac{1}{1}\frac{1}{0}$. This function is Turing computable; here's a machine to compute it:



0/0,R
1/1,R
#/#,R
B/**B**,L
b
0/**B**,L
0/0,L
1/1,L
#/#,L
$\frac{0}{0}/\frac{0}{0}$,L
$\frac{1}{0}/\frac{1}{0}$,L
$\frac{0}{1}/\frac{0}{1}$,L
$\frac{1}{1}/\frac{1}{1}$,L
B/$\frac{0}{0}$,R
0/$\frac{0}{0}$,R
1/$\frac{1}{0}$,R
$\frac{0}{0}/\frac{0}{0}$,R
$\frac{1}{0}/\frac{1}{0}$,R
$\frac{0}{1}/\frac{0}{1}$,R
$\frac{1}{1}/\frac{1}{1}$,R
0/0,R
1/1,R
#/#,R
B/**B**,L
1/**B**,L
#/**B**,L
#/#,L
0/0,L
1/1,L
B/$\frac{0}{1}$,R
0/$\frac{0}{1}$,R
1/$\frac{1}{1}$,R
$\frac{0}{0}/\frac{0}{0}$,L
$\frac{1}{0}/\frac{1}{0}$,L
$\frac{0}{1}/\frac{0}{1}$,L
$\frac{1}{1}/\frac{1}{1}$,L
y
B/**B**,R
z
$\frac{0}{0}/\frac{0}{0}$,L
$\frac{1}{0}/\frac{1}{0}$,L
$\frac{0}{1}/\frac{0}{1}$,L
$\frac{1}{1}/\frac{1}{1}$,L
0/$\frac{0}{0}$,L
1/$\frac{1}{0}$,L

The *stacker* machine uses the following strategy. In the start state it skips to the rightmost symbol of the string. Then for each symbol in the second operand it makes a pass through the main loop of the machine, starting and ending in the state *b*. On each such pass it erases the rightmost remaining symbol of the second operand and overwrites the rightmost 0 or 1 of the first operand with the corresponding stacked symbol. (It treats **B** as equivalent to 0 in the first operand, in case it is shorter than the second operand.)

When the second operand has been erased, the separating # is also erased and the machine enters the state *y*. There it makes a final pass over the first operand. If the first operand was longer than the second, it will still contain some unstacked 0s and/or 1s; in the state *y* they are replaced with stacked versions, as if the second operand had been padded with leading 0s.

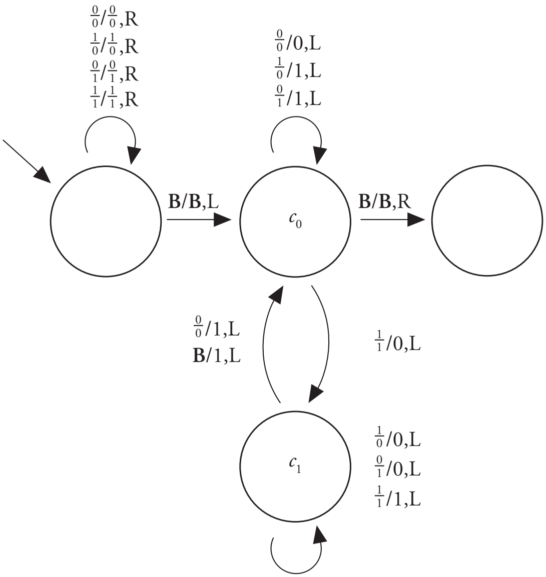
In its final state *z*, the TM halts, leaving the stacked version of its input on the tape. (If the input is improperly formatted—if it does not include exactly one separator #—then all bets are off, and the output of the *stacker* TM will not be a neatly stacked binary string. But we don't care about the output for such cases.) In the state *z* the *stacker* TM's head is at the leftmost symbol of the stacked string, which is exactly where our *and* TM wants it. So we can simply compose the two machines, replacing the final state of the *stacker* TM with the start state of the *and* TM. This gives us a TM that implements the composition of the two functions: $linearAnd(y) = and(stacker(y))$.

Using TM composition is a bit like using subroutines in a high-level language. It is an important technique for building computers of more elaborate functions.

17.3 TM Arithmetic

We turn now to the problem of adding binary numbers. As before, we'll assume that the operands are stacked binary numbers of arbitrary length. The *add* function will take an input of that form and produce the simple binary sum as output. For example, $add\left(\begin{smallmatrix}0&1&1&0&1&1\\0&1&0&1&1&0\end{smallmatrix}\right)$=110001 (or, in decimal, 27 + 22 = 49).

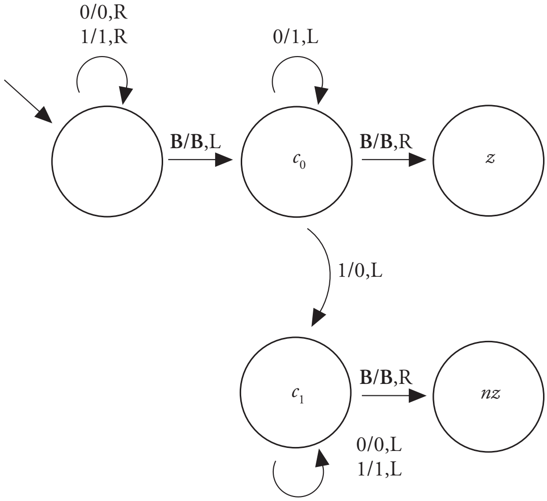
The *add* function is Turing computable:



The *add* machine uses the following strategy. In the start state it skips to the rightmost symbol of the string. Then for each pair of stacked symbols, working from right to left, it overwrites that pair with the corresponding sum bit. The state of the TM records the carry-in from the previous bit; it is in the state c_0 if the carry-in was 0 and the state c_1 if the carry-in was 1. When all the bits have been added the TM encounters the **B** to the left of the input. If the final carry-out was a 1, it writes a leftmost 1 over the **B**; then it halts.

As before, we can combine this machine with the *stacker* machine to make an adder that takes its operands linearly: *linearAdd*(x) = *add*(*stacker*(x)).

To *decrement* a binary number (that is, to subtract one from it) is a simple operation closely related to addition. We can decrement a binary number simply by adding a string of 1s to it and ignoring the carry-out from the final bit position. For example, *decrement*(100) = 011; this is the same as *linearAdd*(100#111) = 1011, except for the final carry-out. If you start with the *add* machine above, but assume that the bottom bit in each stacked pair is a 1 and ignore the final carry-out, you get this *decrement* machine:



The extra final state in the *decrement* machine may come in handy, since it indicates whether the number that has just been decremented was already zero. If it was zero, there was no carry-out from the leftmost bit, so the final state is z. If it was not zero, the final state is nz. So our *decrement* machine can be used not only as a *decrement*-function computer, but also as a decrement-and-test component in larger machines.

17.4 TM Random Access

A TM's memory is like a tape. To get to a particular cell you have to move the TM's head there, one step at a time. By contrast, modern physical computers use addressable memories. To refer to a particular memory word you just give its integer address. This seems like a significant difference; can TMs implement this kind of addressing?

Let i and s be any two binary strings, and define *ith*(i#s) = the symbol at the position i in s, treating i as a binary number and indexing s from the left starting at 0. (If $i \geq |s|$ then define *ith*(i#s) = ε.) For example,

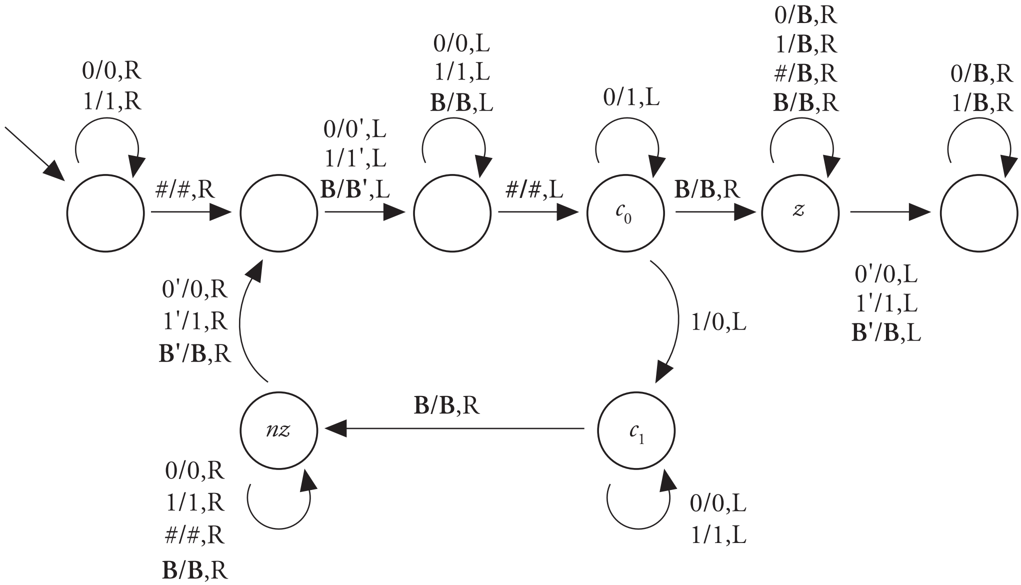
$ith(0\#100) = 1$ (bit 0, counting from the left, in 100)

$ith(1\#100) = 0$ (bit 1, counting from the left, in 100)

$ith(10\#100) = 0$ (bit 2, counting from the left, in 100)

$ith(11\#100) = \varepsilon$ (bit 3—no such bit)

The *ith* function is Turing computable:



Notice that our *decrement* machine is embedded here, in the states labeled c_0, c_1, *nz*, and *z*. The *ith* machine starts by marking the first symbol of the string *s*. Then, in its main loop, it repeatedly decrements *i* and moves the mark one place to the right. When the decrementing step finds that *i* was zero (entering the state *z*), the currently marked symbol is the one that should be output; the *ith* machine then erases everything except the marked symbol and halts.

17.5 Functions and Languages

Our main interest in this book is to define languages formally, but in this chapter we took a detour to examine a related problem: how to use TMs to implement general functions on strings. The TM model we used was exactly the same, and the only difference was in how we looked at the results of a TM's execution.

In many ways, function implementation and language definition are just two different perspectives on the same computational problem. For every language *L* we can define a corresponding function, such as

$$f(x) = 1 \text{ if } x \in L,\ 0 \text{ if } x \notin L$$

With such a correspondence, L is recursive if and only if f is Turing computable. The constructions that prove this are simple. For example, given a total TM with $L(M) = L$, we could build a new TM M' that imitates M up to the point where M halts. Then, if M halts in an accepting state, M' erases its tape, writes a 1, and halts; while if M halts in a nonaccepting state, M' erases its tape, writes a 0, and halts. This new M' computes f.

Similarly, for every function f we can define a corresponding language, such as

$$\{x\#y \mid y = f(x)\}$$

Again, it is easy to show that the language is recursive if and only if the function is Turing computable.

The examples of this chapter illustrated that TMs can be used to

- implement Boolean logic;
- get the effect of subroutine calls by composition;
- perform binary arithmetic; and
- index memory using binary addresses.

These operations are among the most important building blocks of modern computer systems. The fact that these primitives can be implemented using TMs is evidence for the extraordinary power of TMs, whether for language definition or for function implementation. In spite of their apparent simplicity, it turns out that TMs can do anything that can be done with a high-level programming language on a modern computer.

17.6 The Church-Turing Thesis

Many familiar parts of mathematics are concerned with computational procedures: given two numbers in $\mathcal{N}$, there is a procedure for finding their greatest common denominator; given a decimal number, there is a procedure for finding its inverse; given a second-degree polynomial, there is a procedure for finding its roots; given a compass and a straight edge, there is a procedure for constructing a regular pentagon. But in the early 1900s, the attention of mathematicians was drawn to problems that seemed to resist all efforts to find effective computational procedures: given an assertion in first-order logic, where's the procedure for deciding whether it is true or false? Given a polynomial equation, where's the procedure for finding whether it has integer solutions? (See the next chapter for the answers.)

Wrestling with such questions led to the more fundamental question: what exactly is an effective computational procedure? During the 1920s and 1930s, a number of mathematicians around the world were struggling with this problem. A number of very different formalisms were developed. Emil Post developed Post systems; Kurt Gödel worked on the theory of μ-recursive functions; Alonzo Church and Stephen Kleene developed the λ-calculus; Moses Schönfinkel and Haskell Curry developed combinator logic; and, in 1936, Alan Turing invented Turing machines. These different formalisms operate on different kinds of data. For instance, Turing machines work on strings, the λ-calculus works on λ-terms, and μ-recursive functions work on numbers in $\mathcal{N}$.

Although these mathematicians all started out in different directions, they all arrived at the same place. With suitable conversions between the different kinds of data—for instance, representing the natural numbers of μ-recursive functions as strings of digits for Turing machines—it turns out that all those formalisms for computation are interconvertible. Any Turing machine can be simulated by a Post system and vice versa, any Post system can be simulated by a λ-term and vice versa, and so on. People say that any formalism for computation that is interconvertible with Turing machines is *Turing equivalent*. (It would also be correct to say that they are "Post-system equivalent," "λ-calculus equivalent," and so on, but Turing machines are the most popular of the formalisms, perhaps because they are the most like physical computers.) All Turing-equivalent formalisms have the same computational power as Turing machines; they also have the same lurking possibility of infinite computation.

In 1936, Alonzo Church and Alan Turing both suggested that their equivalent formalisms had in fact captured the elusive idea of "effective computational procedure." In effect, they were suggesting that computability by a total TM be taken as the definition of computability. (Note that the definition of Turing computable requires a total Turing machine. A computational procedure that sometimes goes on forever is naturally not considered to be effective!) This has come to be known as *Church's Thesis*, or the *Church-Turing Thesis*. It is called a thesis, not a theorem, because it is not the kind of thing that is subject to proof or disproof. But it is a definition that has come to be generally accepted, and today it marks the border of our happy realm. One thing that makes Church's Thesis so compelling is the effect we have mentioned so often before: it just keeps turning up. Many researchers walking down different paths of mathematical inquiry found the same idea of effective computation waiting for them at the end. That was true in the 1930s, and it is even more true today—because today we have the additional evidence of all modern programming languages and all the physical computer systems that run them. They too are interconvertible with Turing machines, as we will see in the next section. Java, ML, Prolog, and all the rest just add more support for Church's Thesis; they are Turing-equivalent formalisms for computation. "Computable by a Java program that always halts" is the same as "Turing computable," and according to Church's Thesis, that's what computable means, period.

There are several different words in common use for this same concept of computability. A function that is Turing computable (like addition on binary numbers) is called a *computable* function. A language that is recognized by some total Turing machine (like $\{a^n b^n c^n\}$) is called a *recursive* language. When there is some total Turing machine that decides whether an input has a given property (like being a prime number or the square of an integer), we say the property is *decidable*. But beware: some authors prefer a different usage, and some just invent new terms!

An important related term is *algorithm*. Its meaning is quite close to that of *effective computational procedure*, as defined by Church's Thesis. The usage of *algorithm* tends to be broader, however. There are fault-tolerant, distributed algorithms; probabilistic algorithms; and interactive algorithms. These kinds of algorithms don't compute functions, so they're not a good fit for Church's Thesis. In casual use the word *algorithm* sometimes even refers to things like recipes, which are essentially noncomputational.

17.7 TM and Java Are Interconvertible

For any Java program there is an equivalent Turing machine, and for any Turing machine there is an equivalent Java program. This can be proved in both directions by construction, but the constructions are not at all symmetric. High-level languages are much more conveniently expressive than TMs, and that makes the construction invitingly simple in one direction and forbiddingly complicated in the other direction. We'll start by illustrating the inviting direction: a Java implementation of a TM.

We'll make a Java implementation of the TM from Section 16.3 for the language $\{a^n b^n c^n\}$. First, we need to encode the machine's definition:

```
/**
 * A Turing machine for the language a^n b^n c^n.
 */
public class TManbncn {
  /*
   * A constant for the tape blank.
   */
  private static final char B = 0;

  /*
   * The transition function. A char is used to
   * identify each state, and one of the chars
   * 'R' or 'L' is used to identify the head directions.
   * Each transition delta(q,a)=(p,b,D), where q is the
   * current state, a is the symbol being read, p is
   * the next state, b is the symbol to write, and D is
   * the direction to move the head, is represented by
   * a char array of length five: {q,a,p,b,D}.
   * Individual move arrays occur in the delta array in
   * no particular order, so looking up a move requires
   * a linear search.
   */
  private static final char[][] delta = {
          {1,'a',2,'X','R'}, // delta(q1,a) = (q2,X,R)
```

```
        {1,B,7,B,'R'},     // etc.
        {2,'a',2,'a','R'},
        {2,'Y',2,'Y','R'},
        {2,'b',3,'Y','R'},
        {3,'b',3,'b','R'},
        {3,'Z',3,'Z','R'},
        {3,'c',4,'Z','L'},
        {4,'a',4,'a','L'},
        {4,'b',4,'b','L'},
        {4,'Z',4,'Z','L'},
        {4,'Y',4,'Y','L'},
        {4,'X',5,'X','R'},
        {5,'a',2,'X','R'},
        {5,'Y',6,'Y','R'},
        {6,'Y',6,'Y','R'},
        {6,'Z',6,'Z','R'},
        {6,B,7,B,'R'}
};

/*
 * A String containing the char for each accepting
 * state, in any order.
 */
private static final String accepting = "\7";

/*
 * The initial state.
 */
private static final char initial = 1;
```

Formally, a TM is $M = (Q, \Sigma, \Gamma, \delta, \mathbf{B}, q_0, F)$. In this simulation, we encode only the last four parts. The tape alphabet is taken to be Java's `char` type, and the input alphabet is taken to be `char` – {**B**}. The state set is not explicity represented, but the states are represented by `char` values. Our machine uses the states 1 through 7.

The class definition continues:

```
/*
 * The TM's current tape and head position. We always
 * maintain 0 <= head < tape.length(), adding blanks
 * to the front or rear of the tape as necessary.
```

```
     */
    private String tape;
    private int head;

    /*
     * The current state.
     */
    private char state;

    /**
     * Decide whether the TM accepts the given string. We
     * run the TM until it either enters an accepting state
     * or gets stuck. If the TM runs forever on the given
     * string, this will of course never return.
     * @param s the String to test
     * @return true if it accepts, false if not
     */
    public boolean accepts(String s) {
      state = initial;
      head = 0;
      tape = s;

      // establish 0 <= head < tape.length()
      if (head==tape.length()) tape += B;

      while (true) {
        if (accepting.indexOf(state)!=-1) return true;
        char[] move = lookupMove(state,tape.charAt(head));
        if (move==null) return false;
        executeMove(move[2],move[3],move[4]);
      }
    }
```

The `accepts` method simulates the TM on a given input string. Its `while` loop simulates one move of the TM for each iteration. The loop has two exits. If the machine is in an accepting state, the simulation halts and accepts; if there is no next move from the current configuration, the simulation halts and rejects. (To test for an accepting state, it uses the `indexOf` method of the `String` class: `s.indexOf(c)` will return -1 if and only if the `char c` does not occur in the `String s`.)

To complete the class definition, we need the methods `executeMove` and `lookupMove`:

```
/**
 * Execute one move of the TM.
 * @param newstate the next state
 * @param symbol the char to write
 * @param dir the direction ('L' or 'R') to move
 */
void executeMove(char newstate, char symbol, char dir) {

  // write on the tape
  tape = tape.substring(0,head) + symbol +
             tape.substring(head+1);

  // move the head, maintaining the invariant
  // 0 <= head < tape.length()

  if (dir=='L') {
    if (head==0) tape = B + tape; else head -= 1;
  }
  else {
    head += 1;
    if (head==tape.length()) tape += B;
  }

  // go to the next state
  state = newstate;

}

/**
 * Find the move for the given state and symbol.
 * @param state current state
 * @param symbol symbol at current head position
 * @return the five-element char[] from the delta table
 */
char[] lookupMove(char state, char symbol) {
  for(int i = 0; i<delta.length; i++) {
        char[] move = delta[i];
        if (move[0]==state && move[1]==symbol) return move;
      }
      return null;
    }

  }
```

That concludes the implementation of the TManbncn simulator. To test the String s for membership in $\{a^nb^nc^n\}$, we just create a TManbncn object and use its accepts method, like this:

```
TManbncn m = new TManbncn();
if (m.accepts(s)) ...
```

Of course, there are much more efficient ways for a Java program to test a string for membership in $\{a^nb^nc^n\}$. Previous chapters examined implementations of DFAs, NFAs, and stack machines, and those techniques had important practical applications. Unlike them, our TM implementation has no practical value. It is just an exercise that helps you understand how TMs work, and it illustrates one direction of the Turing-equivalence construction.

In the previous chapter we showed how to construct TMs that interpret other automata encoded as input strings. Ultimately, we saw that it is possible to construct universal TMs that interpret other TMs encoded as input strings. Our Java implementation is not quite like that, since the TM being interpreted is hardwired into the program, in the form of those definitions of the variables B, delta, accepting, and initial. But it would be easy enough to modify the code to read the values of those variables from a file. Then it would be an implementation of a universal Turing machine.

The other direction—constructing a TM equivalent to a given Java program—is also possible, but requires a great deal more work. A TM for Java could have two parts: a Java compiler (converting the Java source to a machine language) and an interpreter for that machine language. The Java compiler could be written in machine language, so we only really need the second part, the machine-language interpreter. We have already seen that TMs can perform the basic operations of such an interpreter: Boolean logic, binary arithmetic, indexed addressing, and so on. Following this outline, it would be possible (though inhumanly complicated) to build a TM that serves as a Java interpreter, just as it was possible (and rather simple) to build a Java program that serves as a TM interpreter.

Of course, there's nothing in all this that is unique to Java. A similar argument about interconvertibility can be made for any high-level programming language. Our demonstration of this has been informal; rigorously proving that TMs are equivalent in power to a particular high-level language is a long and tedious chore. But such proofs have been done and are universally accepted. There is nothing that high-level languages can do that TMs cannot do. All modern programming languages turn out to be Turing equivalent, adding still more support to the Church-Turing Thesis.

17.8 Infinite Models, Finite Machines

Actually, a Turing machine can do a *better* job of implementing high-level languages than an ordinary computer. A physical computer is always limited by its finite memory. You may install as much memory as the machine can hold; you may buy a big array of disks and implement an even larger virtual memory; but no matter what you do, there will still be programs that demand more memory (and more time) than you have. Any implementation of Java on a physical computer introduces memory limitations. In the abstract, high-level languages like Java are Turing equivalent; the physical computers that implement them are not.

Any physical computer has finite memory, and therefore has finitely many possible states. In that sense, physical computers are more like DFAs than Turing machines—DFAs with enormous, but still finite, state sets. Nothing we can build has unbounded memory, nor do we have unbounded time for computation. Yet almost everyone would agree that, at an intuitive level, TMs are much better than DFAs as models for physical computers. It's a curious situation. Unbounded models are patently unrealistic, so why do they feel so natural?

Part of the explanation is that the number of states that can be taken on by a physical computer is so large that it usually appears unbounded to the human user. For example, a word processor running in finite memory can represent only finitely many different texts. Every possible text, profound or nonsensical, in every state of completion, corresponds to one of the states that can be taken on by the machine on which it is being written. But this limitation is of theoretical interest only. In practice, it does not chafe—a writer does not think of writing as the act of selecting one of these finitely many representable texts. Similarly, programmers do not think of programs as DFAs that drive the machine through finitely many possible states.

In any event, all of mathematics is filled with such idealizations. Our ordinary experience of the world includes no perfect points, lines, or circles, but that does not make geometry less useful. It includes no infinite numbers and no infinitesimals, but that does not make calculus less useful. Though it includes no infinite computations, the concept of computability captured by the Turing machine is no less important. In particular, the notion of *uncomputability* captured by the Turing machine is a strong one—for if a computation is impossible for an idealized, unbounded machine, it is certainly impossible for a physical computer as well. We will consider the question of uncomputability more deeply in the next chapter.

Exercises

EXERCISE 1

The *linearAnd* function can be implemented with fewer states and a smaller tape alphabet, if implemented directly and not as a composition of *and* with *stacker*. Show how.

EXERCISE 2

Let *increment* be the binary increment function, *increment*(x) = *add(stacker*(x#1)). Give a TM that implements *increment*. Note that the definition requires it to treat ε as a representation for 0, so that *increment* (ε) = 1. *Hint:* Actually using *stacker* in your implementation would be doing it the hard way. The *increment* function can be implemented with just three states, using just {0, 1, **B**} as the tape alphabet.

EXERCISE 3

Let *roundup* be the binary, round-up-to-even function, *roundup*(x) = x if x represents an even number, or *increment* (x) if x represents an odd number. Give a TM that implements *roundup*. Treat ε as a representation for zero, so that *roundup*(ε) = ε.

EXERCISE 4

Let *rounddown* be the binary, round-down-to-even function, *rounddown*(x) = x if x represents an even number, or *decrement* (x) if x represents an odd number. Give a TM that implements *rounddown*. Treat ε as a representation for zero, so that *rounddown*(ε) = ε.

EXERCISE 5

For any $x \in \{a, b\}^*$, let *mid*(x) be the middle symbol of x or ε if x has even length. Give a TM that implements *mid*.

EXERCISE 6

Give a TM that implements subtraction on unsigned, binary numbers, assuming stacked inputs as for the *add* TM of Section 17.3. Assume that the second operand is less than or equal to the first, so that the result can be represented as an unsigned, binary number.

EXERCISE 7

Give a TM that implements multiplication by two on unary numbers, using $\Sigma = \{1\}$. Given an input string 1^i, your TM should halt leaving 1^{2i} (and nothing else) on the tape.

EXERCISE 8

Give a TM that implements division by two on unary numbers, using $\Sigma = \{1\}$. Given an input string 1^i, your TM should halt leaving $1^{i/2}$ (and nothing else) on the tape. If i is odd, round up.

EXERCISE 9

For any $x \in \{0, 1\}^*$, let val(x) be the natural number for which x is a binary representation; for completeness, let val(ε) = 0. Give a TM that implements a binary-to-unary conversion function, $btu(x) = 1^{\text{val}(x)}$.

EXERCISE 10

Let *utb* be a unary-to-binary conversion function (an inverse of *btu* from the previous exercise). Give a TM that implements *utb*.

EXERCISE 11

Write a Java method `anbncn`, so that `anbncn(s)` returns `true` if `String s` is in $\{a^n b^n c^n\}$ and returns `false` if not. Make it as efficient as possible (which of course means not implementing it as a TM simulation). Then answer in as much detail as you can, how does the speed of your `anbncn` compare with that of the `accepts` method in `TManbncn`?

EXERCISE 12

Write a Java implementation of the TM implementing the *linearAdd* function, as described in Sections 17.2 and 17.3. Include a `main` method that reads the input string from the command line and writes the output string (that is, the final contents of the tape, not including leading and trailing **B**s) to standard output. For example, your program should have this behavior:

```
> java TMlinearAdd 1010#1100
10110
```

EXERCISE 13

Write a Java implementation of a universal Turing machine. Start with the code for `TManbncn`, but alter it to make it read the TM to be simulated from a file. You can design your own encoding for storing the TM in a file, but it should be in a text format that can be read and written using an ordinary text editor. Your class should have a main method that reads a file name and a text string from the command line, runs the TM encoded in the given file on the given string, and reports the result. For example, if the file `anbncn.txt` contains the encoding of a TM for the language $\{a^n b^n c^n\}$, then your program would have this behavior:

```
> java UTM anbncn.txt abca
reject
> java UTM anbncn.txt aabbcc
accept
```

EXERCISE 14

Complete the following short story:

The judge sighed as he sat down before the computer screen. He was administering his eighth Turing test of the day, and he was tired of it. Six had been machines, their classification easily revealed by their responses to simple conversational gambits like "`So ... you married?`" or "`How 'bout them Yankees?`" There had been only one human this morning, and the judge had been sure of the classification in less than a minute. He had always had a knack for this kind of testing, and the more he did, the quicker he got.

But this morning it was getting tedious. One more test, he thought, and I can break for lunch. He rubbed his eyes, sat up straighter in his chair, and quickly typed:

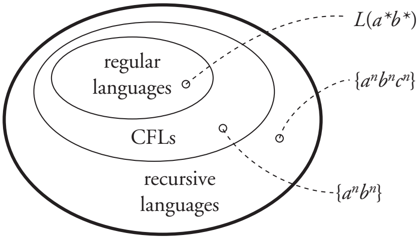
```
Would you say that your mind is finite?
```

Slowly, the response came back ...

18

Uncomputability

The Church-Turing Thesis gives a definition of computability, like a border surrounding the algorithmically solvable problems.



Beyond that border is a wilderness of uncomputable problems. This is one of the great revelations of 20th-century mathematics: the discovery of simple problems whose algorithmic solution would be very useful but is forever beyond us.

18.1 Decision and Recognition Methods

In this chapter, to make examples that are easier for practical programmers to read, we will switch from using Turing machines to using a more familiar Java-like syntax. Although this syntax looks like Java, there is a critical difference: we will assume that there are none of the limitations of actual Java. In particular, we will assume that there are no bounds on the length of a string or the size of an integer. So be aware that from here on, our code examples are intended to be illustrative rather than strictly executable.

Total TMs correspond to *decision methods* in our Java-like notation.

A decision method takes a `String` parameter and returns a `boolean` value. (It always returns, and it does not run forever.)

Here's one for the language $\{ax \mid x \in \Sigma^*\}$:

```
boolean ax(String p) {
  return (p.length()>0 && p.charAt(0)=='a');
}
```

(For those not familiar with Java, the expression `p.length()` is the length of the `String` object referred to by `p`, and the expression `p.charAt(i)` yields the `i`th character of the `String p`, counting from zero.) Here are decision methods for the languages {} and Σ*:

```
boolean emptySet(String p) {
  return false;
}

boolean sigmaStar(String p) {
  return true;
}
```

As with Turing machines, we will refer to the language accepted by a decision method `m` as L(`m`). So L(`emptySet`) = {} and L(`sigmaStar`) = Σ*. Decision methods give us an alternative, equivalent definition of a recursive language: a language is recursive if and only if it is L(`m`) for some decision method `m`.

For methods that might run forever, we will use the broader term *recognition method*. These correspond to TMs that are not necessarily total.

A recognition method takes a `String` parameter and either returns a `boolean` value or runs forever.

With these definitions, a decision method is a special kind of recognition method, just as a total TM is a special kind of TM. Here is a recognition method for the language $\{a^n b^n c^n\}$:

```
boolean anbncn1(String p) {
  String as = "", bs = "", cs = "";
  while (true) {
    String s = as+bs+cs;
    if (p.equals(s)) return true;
    as += 'a'; bs += 'b'; cs += 'c';
  }
}
```

This is a highly inefficient way of testing whether the string `p` is in $\{a^n b^n c^n\}$, but we are not concerned with efficiency. We are at times concerned with avoiding nontermination; the recognition method `anbncn1` loops forever if the string is not in the language. Thus it only demonstrates that $\{a^n b^n c^n\}$ is RE. We know that $\{a^n b^n c^n\}$ is a recursive language, so there is some decision method for it, such as this one:

```
boolean anbncn2(String p) {
  String as = "", bs = "", cs = "";
  while (true) {
    String s = as+bs+cs;
    if (s.length()>p.length()) return false;
    else if (p.equals(s)) return true;
    as += 'a'; bs += 'b'; cs += 'c';
  }
}
```

Recognition methods give us an alternative, equivalent definition of a recursively enumerable language: a language is RE if and only if it is L(`m`) for some recognition method `m`.

We will need the high-level-language equivalent of a universal TM. Real Java programs, whether applets or applications, consist of one or more complete class defininitions, which are compiled in a separate step before being run. We'll assume something simpler: that individual methods are run on individual input strings, interpreting directly from the source. We'll assume the existence of a `run` method with this specification:

```
/**
 * run(rSource, in) runs a recognition method, given
 * its source code. If rSource contains the source
 * code for a recognition method r, and in is any
```

```
   * string, then evaluating run(rSource, in) has the
   * same effect as evaluating r(in).
   */
  boolean run(String rSource, String in) {
    ...
  }
```

For example, in this fragment:

```
  String s = "boolean sigmaStar(String p) {return true;}";
  run(s,"abc");
```

run would return true, because that's what sigmaStar returns when run with the input string "abc". But in this fragment:

```
  String s =
    "boolean ax(String p) {                       " +
    "  return (p.length()>0 && p.charAt(0)=='a'); " +
    "}                                            ";
  run(s,"ba");
```

run would return false, since ax("ba") returns false. Finally, in this fragment:

```
  String s =
    "boolean anbncn1(String p) {               " +
    "  String as = \"\", bs = \"\", cs = \"\"; " +
    "  while (true) {                          " +
    "    String s = as+bs+cs;                  " +
    "    if (p.equals(s)( return true;         " +
    "    as += 'a'; bs += 'b'; cs += 'c';      " +
    "  }                                       " +
    "}                                         ";
  run(s,"abbc");
```

run would never return, because anbncn1("abbc") runs forever. Implementing run would be a lot of work—see any textbook on compiler construction for the details—but it is clearly possible.

The run method doesn't quite fit our definition of a recognition method, because it takes two input strings. We could make it fit by redefining so that it takes one delimited input string: something like run(p+'#'+in) instead of run(p,in). That's the kind of trick we had to use in Chapter 17 to give a Turing machine more than one input; recall *linearAdd*(*x*#*y*). But to keep the code more readable, we will just relax our definitions a bit, allowing recognition and decision methods to take more than one parameter of the `String` type. With this relaxation, the `run` method qualifies as a recognition (but not a decision) method.

18.2 The Language L_u

Programmers quickly learn about the perils of infinite computation. Perhaps you have written a program with a bug like this one:

```
int j = 0;
for (int i = 0; i < 100; j++) {
  j += f(i);
}
```

Oops! That postincrement expression should read `i++`, not `j++`. As it stands, the variable `i` is never assigned any value other than 0, so the loop never stops. Not noticing this mistake, you run the program. Perhaps that function `f` is something complicated, so you expect the computation to take a few minutes anyway. Ten minutes go by, then twenty, then thirty. Eventually you ask yourself the question programmers around the world have been asking since the dawn of the computer age: is this stuck in an infinite loop, or is it just taking a long time? There is, alas, no sure way for a person to answer such questions—and, as it turns out, no sure way for a computer to find the answer for you.

The existence of our `run` method shows that a particular language is RE:

$L(\texttt{run}) = \{(\texttt{p, in}) \mid \texttt{p}$ is a recognition method and $\texttt{in} \in L(\texttt{p})\}$

The language is RE because we have a recognition method that accepts it—the `run` method. A corresponding language for TMs is this:

$\{m\#x \mid m$ encodes a TM and x is a string it accepts$\}$

The language is RE because we have a TM that accepts it—any universal TM. In either case, we'll call the language L_u—remember *u* for *universal.*

Is L_u recursive, that is, is it possible to write a *decision* method with this specification?

```
/**
 * shortcut(p,in) returns true if run(p,in) would
 * return true, and returns false if run(p,in)
 * would return false or run forever.
 */
boolean shortcut(String p, String in) {
  ...
}
```

The shortcut method would be just like the run method, but it would always produce an answer—not run forever, even when run(p,in) would. For example, in this fragment:

```
String x =
  "boolean anbncn1(String p) {                    " +
  "  String as = \"\", bs = \"\", cs = \"\"; " +
  "  while (true) {                                " +
  "    String s = as+bs+cs;                        " +
  "    if (p.equals(s)) return true;               " +
  "    as += 'a'; bs += 'b'; cs += 'c';            " +
  "  }                                             " +
  "}                                               ";
shortcut(x,"abbc");
```

shortcut would return false, even though anbncn1("abbc") runs forever.

How would you go about implementing something like shortcut? Presumably, it would have to simulate the input program as run does, but somehow detect infinite loops. Some infinite loops are easy to detect, of course. It is simple enough for an interpreter to notice constructs like while(true) {}. But most infinite loops are subtler than that, which is why they are such a plague for programmers. In the case of anbncn1, it is possible to imagine a program clever enough to reason that the loop will not terminate once s is longer than p; it could then insert the extra exit,

```
if (s.length()>p.length()) return false;
```

But is it possible to write shortcut so that it can detect and avoid all possible infinite loops? It would certainly be valuable; imagine a debugging tool that could reliably alert you to infinite computations! Unfortunately, we can prove that no such shortcut method exists.

Such proofs are tricky; it isn't enough to say that we tried really hard but just couldn't think of a way to implement shortcut. We need a proof that no such implementation is possible. The proof is by contradiction. Assume by way of contradiction that L_u is recursive, so some implementation of shortcut exists. Then we could use it to make this decision method:

```
/**
 * nonSelfAccepting(p) returns false if run(p,p)
 * would return true, and returns true if run(p,p)
 * would return false or run forever.
 */
```

```
  boolean nonSelfAccepting(String p) {
    return !shortcut(p,p);
  }
```

This decision method determines what the program would decide, given itself as input—then returns the opposite. So the language defined by `nonSelfAccepting` is simply that set of recognition methods that do not accept, given themselves as input. For example,

```
  nonSelfAccepting(
    "boolean sigmaStar(String p) {return true;}"
  );
```

returns false. That's because `sigmaStar` accepts all strings, including the one that starts `boolean sigmaStar ...`. In other words, `sigmaStar` accepts itself, and therefore `nonSelfAccepting` returns false given `sigmaStar` as input. On the other hand,

```
  nonSelfAccepting(
    "boolean ax(String p) {                          " +
    "  return (p.length()>0 && p.charAt(0)=='a');   " +
    "}                                               "
  );
```

tests whether the string `boolean ax ...` is accepted by the decision method `ax`. Since the string begins with `b`, `ax` returns false. It does not accept itself, and therefore `nonSelfAccepting` returns true given `ax` as input.

Now comes the tricky part. What happens if we call `nonSelfAccepting`, giving it itself as input? We can easily arrange to do this:

```
  nonSelfAccepting(
    "boolean nonSelfAccepting(p) { " +
    "  return !shortcut(p,p);      " +
    "}                             "
  )
```

What does `nonSelfAccepting` return, given itself as input? If it accepts itself, that means `shortcut` determined it was not self-accepting—which is a contradiction. If it rejects itself, that means `shortcut` determined it was self-accepting—also a contradiction. Yet it must either accept or reject; it cannot run forever, because `shortcut` is a decision method. We're left with a contradiction in all cases. By contradiction, our original assumption must be false: no program satisfying the specifications of `shortcut` exists. In other words:

Theorem 18.1: L_u is not recursive.

This is our first example of a problem that lies outside the borders of computability. L_u is not recursive; equivalently, we can say that the `shortcut` function is not computable and the machine-*M*-accepts-string-*x* property is not decidable. This verifies our earlier claim that total TMs are weaker than general TMs. No total TM can be a universal TM.

18.3 The Halting Problems

Consider the sentence "This sentence is false." It's a well-known paradox: if it's true it must be false, if it's false it must be true. It achieves this paradox through self-reference. This is easy to do in English: a sentence can refer to itself as "this sentence." It's also fairly easy to do with computational procedures of all kinds: we can pass a method its own source as input or give a TM the string encoding of itself as input. Like the "this sentence is false" paradox, our proof that L_u is not recursive made use of self-reference.

For modern programmers, the idea of self-reference is not a big stretch. Every day we work with programs that take other programs as input: compilers, interpreters, debuggers, and the like. It is not uncommon to write such tools in the same language on which they operate; there are C compilers written in C, ML compilers written in ML, and so on. Such a compiler could be given itself as input without causing the modern programmer any existential discomfort. In the 1930s, however, the idea that effective computational procedures might have the power and the paradox that comes with self-reference was a major insight.

Here's another example using a similar self-reference contradiction. Consider this language:

{(`p`, `in`) | `p` is a recognition method and `in` is a string for which it halts}

It is easy to see that this language is RE:

```
/**
 * haltsRE(p,in) returns true if run(p,in) halts.
 * It just runs forever if run(p,in) runs forever.
 */
boolean haltsRE(String p, String in) {
  run(p,in);
  return true;
}
```

A corresponding language for TMs is this:

{*m*#*x* | *m* encodes a TM and *x* is a string on which it halts}

In either case, we'll call the language L_h—remember *h* for *halting*.

Is L_h recursive; that is, is it possible to write a decision method with this specification?

```
/**
 * halts(p,in) returns true if run(p,in) halts, and
 * returns false if run(p,in) runs forever.
 */
boolean halts(String p, String in) {
  ...
}
```

The `halts` method would be just like the `haltsRE` method, but it would always produce an answer—not run forever, even when `run(p,in)` would.

From our results about L_u, you might guess that L_h is not going to be recursive either. Intuitively, the only way to tell what `p` will do when run on `in` is to simulate it—and if that simulation runs forever, we won't get an answer. But that kind of intuitive argument is not strong enough to constitute a proof. How do we know there isn't some other way of determining whether `p` halts, a way that doesn't involve actually running it?

The proof that this is impossible is by contradiction. Assume by way of contradiction that L_h is recursive, so some implementation of `halts` exists. Then we could use it to make this program:

```
/**
 * narcissist(p) returns true if run(p,p) would
 * run forever, and runs forever if run(p,p) would
 * halt.
 */
boolean narcissist(String p) {
  if (halts(p,p)) while(true) {}
  else return true;
}
```

This method determines whether the program `p` will contemplate itself forever—that is, it defines the language of recognition methods that run forever, given themselves as input.

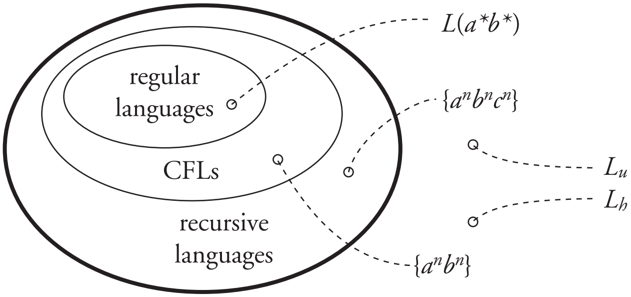
Now comes that trick using self-reference. What happens if we call `narcissist`, giving it itself as input? We can easily arrange to do this:

```
narcissist(
  "boolean narcissist(p) {           " +
  "  if (halts(p,p)) while(true) {} " +
      "  else return true;                  " +
      "}                                    "
    )
```

Does it run forever? No, that can't be right; if it did we would have a contradiction, since it would be saying that it halts. So then does it halt (returning true)? No, that can't be right either; if it did it would be saying that it runs forever. Either way we have a contradiction. By contradiction, we conclude that no program satisfying the specifications of `halts` exists. This proves the following:

Theorem 18.2: L_h is not recursive.

At this point, we have identified our first two languages that are not recursive, and our picture looks like this:



The non-recursive languages don't stop there, however. It turns out that there are uncountably many languages beyond the computability border. The question of whether a program halts on a given input is a classic undecidable problem: a *halting problem.* It has many variations: does a program halt on a given input? Does it halt on any input? Does it halt on every input? That last variation would be the most useful. It would be nice to have a program that could check over your code and warn you about all possible infinite loops. Unfortunately, all these variations of the halting problem are undecidable. In fact, as we will see below, most questions about the runtime behavior of TMs (or computer programs) turn out to be undecidable.

18.4 Reductions Proving a Language Is Recursive

Suppose you are planning a trip from your house to a friend's house. You formulate a plan:

1. I will drive my car to the airport.
2. I will fly to my friend's airport.
3. My friend will pick me up.

Suppose that you know steps 1 and 3 are possible; that leaves you with step 2. You have reduced the original problem *A* (making a trip from house to house) to another problem *B* (finding a flight from airport to airport). The reduction shows that if you can get a flight, you can make the trip.

In other words, reducing problem *A* to problem *B* shows that *A* is no harder than *B*. Of course, we have not ruled out the possibility that *A* might be easier than *B*, because there might be other, simpler solutions. For example, you and your friend might already be in the same city. In that case driving to the airport, getting on a plane, flying around, landing at the same airport, and driving to your friend's house is a correct solution but far from optimal.

The same idea is used in the design of algorithms: solve one problem by using the existing solution to another. Given an original problem *A*, a *reduction* is a solution of this form:

1. Convert the instance of problem *A* into an instance of problem *B*.
2. Solve that instance of problem *B*.
3. Convert the solution of the instance of problem *B* back into a solution of the original instance of problem *A*.

If steps 1 and 3 are no harder than step 2, we can conclude that problem *A* is no harder than problem *B*. (But problem *A* might still be easier than problem *B*, since there may be an easier, completely different algorithm.)

Such reductions can be used to prove that a language is recursive. Suppose we are given a language L_1. One way to prove that L_1 is recursive is to write a decision method (or a total TM) for it. But another way is to give a reduction to some language L_2 that we already know is recursive:

1. Given a string x_1 to be tested for membership in L_1, convert it into another string x_2 to be tested for membership in L_2.
2. Decide whether $x_2 \in L_2$.
3. Convert that decision about x_2 back into a decision about x_1.

If steps 1 and 3 are computable—if those conversions can be computed effectively, without infinite looping—and if L_2 is already known to be recursive, a reduction like this proves that L_1 is recursive too.

For example, let's use a reduction to prove that $L_1 = \{x \in \{a, b, d\}^* \mid x \notin \{a^n b^n d^n\}\}$ is recursive. We'll reduce it to the decision problem for $L_2 = \{a^n b^n c^n\}$.

```
a.    boolean decideL1(String x1) {
b.      String x2="";
c.      for (int i = 0; i < x1.length(); i++) {
d.        char ith = x1.charAt(i);
e.        if (ith=='d') x2+='c';
f.        else x2+=ith;
   g.          }
   h.          boolean b = anbncn2(x2);
   i.          return !b;
   j.      }
```

Lines b through g are step 1; they translate the string x_1 into a new string x_2, in this case by replacing the *d*s with *c*s. Line h is step 2; it checks whether $x_2 \in L_2$, in this case using our previously implemented decision method `anbncn2`. (In general, we don't need to show all the code for step 2; if we know that L_2 is recursive, we know some decision method for it exists, and that's enough.) Line i is step 3; it converts the answer for $x_2 \in L_2$ into an answer for $x_1 \in L_1$, in this case by logical negation. Steps 1 and 3 obviously cannot run forever, so this reduction shows that if L_2 is recursive (which we know it is) then L_1 is recursive.

Here is another example of a proof using a reduction: we'll prove that $\{a^nb^n\}$ is recursive by reduction to the recognition problem for $\{a^nb^nc^n\}$:

```
a.      boolean anbn(String x1) {
b.        String x2=x1;
c.        for (int i = 0; i < x1.length()/2; i++)
d.          x2+='c';
e.        boolean b = anbncn2(x2);
f.        return b;
g.      }
```

This reduces the problem of deciding membership in $\{a^nb^n\}$ to the problem of deciding membership in $\{a^nb^nc^n\}$. As we have already shown that $\{a^nb^nc^n\}$ is recursive, we conclude that $\{a^nb^n\}$ is recursive. Of course, we already knew that; in fact, $\{a^nb^n\}$ is not only recursive but also context free. This illustrates the one-way nature of the inference you can draw from a reduction. By reducing from a new problem to a problem with a known solution, we show that the new problem is no harder than the known one. That reduction does not rule out the possibility that the new problem might be easier.

18.5 Reductions Proving a Language Is Not Recursive

A reduction from *A* to *B* shows that *A* is no harder than *B*. *Equivalently, we can say the reduction shows that B is no easier than A*. This can be used to prove that a language is not recursive. Suppose we are given a language L_1. One way to prove that is not recursive is by contradiction, as we did in the previous sections for L_u and L_h. But another, often easier, way is to give a reduction from some language L_2 that we already know is not recursive. A reduction in that direction—a reduction *from* a nonrecursive language L_2 *to* the new language L_1—lets us conclude that L_1 is not recursive, since it is no easier than L_2. For example, define

L_e = {p | p is a recognition method that never returns true}

In other words, L_e is the set of recognition methods p for which L(p) = {} (remember *e* for empty). Then we have the following:

Theorem 18.3.1: L_e is not recursive.

Proof: By reduction from the halting problem. Assume by way of contradiction that L_e is recursive; then there is some decision method empty for it. We can use it to write the following decision method:

```
boolean halts(String p, String x) {
  String x2 =
    "boolean f(String z) {         " +
    "  run(\""+p+"\",\""+x+"\");    " +
    "  return true;                " +
    "}                             ";
  boolean b = empty(x2);
  return !b;
}
```

For any given program p and input string x, halts constructs a string x2 containing the source of a new decision method f. The f method ignores its input string z. Instead of looking at z, it runs p on x, then returns true. Now if p runs forever on x, f will not return, so the language of strings z it accepts is {}. But if p halts on x, f will return true, so the language of strings z it accepts is Σ^*. Thus x2 $\in L_e$ if and only if p runs forever on x. We conclude that halts is a decision method for L_h. But this is a contradiction, since L_h is not recursive. By contradiction, we conclude that L_e is not recursive.

The reduction shows that L_e is no easier to decide than L_h. But we know L_h is not recursive; so we conclude that L_e is not recursive. This is a very important application for reductions. To show that a language is recursive is comparatively easy: you only need to give a program that recognizes it and always halts. But to show that a language is not recursive is generally more difficult. Often, the best choice is to give a reduction from a problem that is already known to be nonrecursive.

Here's another example along the same lines. Define

L_r = {p | p is a recognition method and L(p) is regular}

For example, the string

```
boolean sigmaStar(String p) {return true;}
```

is in L_r, because L(sigmaStar) = Σ^*, a regular language, while our previous decision method anbn is not in L_r, because L(anbn) is not a regular language.

Theorem 18.3.2: L_r is not recursive.

Proof: By reduction from the halting problem. Assume by way of contradiction that L_r is recursive; then there is some decision method regular for it. We can use it to write the following decision method:

```
boolean halts(String p, String x) {
  String x2 =
    "boolean f(String z) {          " +
    "  run(\""+p+"\",\""+x+"\");    " +
    "  return anbn(z);              " +
    "}                              ";
  boolean b = regular(x2);
  return !b;
}
```

For any given program p and input string x, halts constructs a string x2 containing the source of a new decision method f. The f method runs p on x, then returns anbn(z). Now if p runs forever on x, f will not return, so the language of strings z it accepts is {}—a regular language. But if p halts on x, f will return anbn(z), so the language of strings z it accepts is $\{a^n b^n\}$—not a regular language. Thus x2 $\in L_r$ if and only if p runs forever on x. We conclude that halts is a decision method for L_h. But this is a contradiction, since L_h is not recursive. By contradiction, we conclude that L_r is not recursive.

18.6 Rice's Theorem

The previous two proofs clearly have considerable similarities. In both cases we used a reduction from the halting problem to show that some set of programs was not recursive. This can be generalized to a very powerful theorem, Rice's theorem, which is usually stated like this: *All nontrivial properties of the RE languages are undecidable*. Using our terminology, we'll state it like this:

Theorem 18.4 (Rice's theorem): For all nontrivial properties α, the language

{p | p is a recognition method and L(p) has the property α}

is not recursive.

Here are some examples of languages covered by Rice's theorem:

$L_e = \{\texttt{p} \mid \texttt{p}$ is a recognition method and $L(\texttt{p})$ is empty$\}$

$L_r = \{\texttt{p} \mid \texttt{p}$ is a recognition method and $L(\texttt{p})$ is regular$\}$

$\{\texttt{p} \mid \texttt{p}$ is a recognition method and $L(\texttt{p})$ is context free$\}$

$\{\texttt{p} \mid \texttt{p}$ is a recognition method and $L(\texttt{p})$ is recursive$\}$

$\{\texttt{p} \mid \texttt{p}$ is a recognition method and $|L(\texttt{p})| = 1\}$

$\{\texttt{p} \mid \texttt{p}$ is a recognition method and $|L(\texttt{p})| \geq 100\}$

$\{\texttt{p} \mid \texttt{p}$ is a recognition method and $\mathit{hello} \in L(\texttt{p})\}$

$\{\texttt{p} \mid \texttt{p}$ is a recognition method and $L(\texttt{p}) = \Sigma^*\}$

Of course, Rice's theorem also applies to the corresponding languages expressed in terms of TMs (or any other Turing-equivalent formalism).

Rice's theorem requires the property in question to be *nontrivial*. A property is *trivial* if no RE languages have it or if all RE languages have it; for trivial properties the language in question is either the empty set or the set of all recognition methods, both of which are recursive. For example, Rice's theorem does not apply to these two examples:

$\{\texttt{p} \mid \texttt{p}$ is a recognition method and $L(\texttt{p})$ is RE$\}$

$\{\texttt{p} \mid \texttt{p}$ is a recognition method and $L(\texttt{p}) \supset \Sigma^*\}$

The first uses a property which is by definition true of all RE languages, so it is just the set of all recognition methods. The second uses a property that is true of no RE languages, so it is just the empty set.

The proof of Rice's theorem is a generalization of the proofs we just saw for L_e and L_r.

Proof: Let α be any nontrivial property of the RE languages, and define

$A = \{\texttt{p} \mid \texttt{p}$ is a recognition method and $L(\texttt{p})$ has the property $\alpha\}$

We will show that A is nonrecursive, by reduction from the halting problem. Assume by way of contradiction that A is recursive. Then there is some decision method `falpha` that recognizes A. Now either {} has the α property or it does not; we will deal with these two cases separately.

First, suppose {} has the α property. Because α is nontrivial, there must be at least one RE language Y that does not have the α property, and since Y is RE there is some decision method `fy` that decides Y. Now construct

```
boolean halts(String p, String x) {
  String x2 =
    "boolean f(String z) {          " +
4.         "  run(\""+p+"\",\""+x+"\");    " +
5.         "  return fy(z);                " +
6.         "}                              ";
7.      boolean b = falpha(x2);
8.      return !b;
9.    }
```

For any given program `p` and input string `x`, `halts` constructs a string `x2` containing the source of a new recognition method `f`. The `f` method runs `p` on `x`, then returns `fy(z)`. Now if `p` runs forever on `x`, `f` will not return, so the language of strings it accepts is {}—a language with the α property. But if `p` halts on `x`, `f` will return `fy(z)`, so the language of strings `z` it accepts is Y—a language that does not have the α property. Thus `x2` ∈ A if and only if `p` runs forever on `x`. Under the assumption that `falpha` decides A, we conclude that `halts` decides L_h. But this is a contradiction, since L_h is not recursive. By contradiction, we conclude that A is not recursive.

That takes care of the case in which {} has the α property. The other possibility is that {} does not have the α property. In that case, there must be at least one RE language Y that *does* have the α property, and since Y is RE there is some decision method `fy` that decides Y. Using `fy`, we can construct `halts` as before, but return `b` instead of `!b` on line 8. Again, by contradiction, we conclude that A is not recursive.

We can appeal to Rice's theorem to make simple nonrecursiveness proofs for many languages. One of our previous examples was

{`p` | `p` is a recognition method and $|L(\texttt{p})| = 1$}

To show that this is not recursive, we merely observe that it is a language of the form covered by Rice's theorem and that the property in question ($|L(\texttt{p})| = 1$) is nontrivial, because some RE languages have one element and others don't.

Rice's theorem is a very useful tool. It applies to languages that are sets of recognition methods (defined using TMs, high-level languages, or any other Turing-equivalent formalism), selected for some nontrivial property of the language they accept. Rice's theorem covers many nonrecursive sets of programs, including the whole list of examples at the beginning of this section. But although it can be very useful, it is not the right tool for every nonrecursiveness proof. For example, consider these languages:

{`p` | `p` is a method that prints `"hello world"`}

{`p` | `p` is a method that never gets an uncaught exception}

{`p` | `p` is a method that produces no output}

These languages are not recursive, but you can't prove them so using Rice's theorem, because they are not sets of recognition methods and they are not defined in terms of some property of the language they accept. Each one needs a custom proof, like those we developed earlier in this chapter.

Most questions about the runtime behavior of programs turn out to be undecidable, even if you can't apply Rice's theorem directly. By contrast, most questions about program *syntax* turn out to be decidable. For example, the question of whether a given method contains the infinitely looping statement `while(true){}` is easy to decide—indeed, the set of strings that contain `while(true){}` is a regular language. Likewise, questions about the structure of TMs are generally decidable, like the question of whether a given TM contains ten states.

To summarize: questions about program behavior are usually undecidable, and questions about program syntax are usually decidable. But this is only a rough guideline, and there are exceptions either way. Consider this language:

{(`p`, `x`) | `p` is a method that executes at least ten statements
when run with the input `x`}

The question of whether a method `p` executes at least ten statements when run with a given input is easy to decide. You just start simulating `p` on `x` and count the number of statements executed. If `p` returns before you get to 10, it isn't in the language; if you get to 10, it is in the language (and you can stop simulating). It's a test that always gives you an answer within ten statements, so there's no trouble about infinite loops. The language is recursive, even though it is a set of programs defined in terms of their runtime behavior.

18.7 Enumerators

We have generally treated Turing machines as language *recognizers*: machines that take an input string and try to determine its membership in a language. When Alan Turing published the 1936 paper introducing his model for effective computation, his concept was slightly different. He envisioned his tape-memory automata as language *enumerators*: machines that take no input but simply generate, on an output tape, a sequence of strings.

This is just another way of defining a language formally: $L(M)$ for such a machine is the language of strings that eventually appear on the output tape:

$$L(M) = \{x \mid \text{for some } i,\ x \text{ is the } i\text{th string in } M\text{'s output sequence}\}$$

Like all Turing machines, enumerators may run forever. In fact, they must, if they enumerate infinite languages; and they may even if they enumerate finite languages. In Java-like notation we can think of these original Turing machines as *enumerator objects*:

An *enumerator class* is a class with an instance method `next` that takes no input and returns a string (or runs forever).

An enumerator object may preserve state across calls of `next`, so `next` may (and generally does) return a different string every time it is called. For an enumerator class `C`, we'll take L(`C`) to be the set of strings returned by an infinite sequence of calls to the `next` method of an object of the `C` class.

For example, this is an enumerator class `AStar` with L(`AStar`) = $\{a\}^*$:

```
class AStar {
  int n = 0;

  String next() {
    String s = "";
    for (int i = 0; i < n; i++) s += 'a';
    n++;
    return s;
  }
}
```

For this enumerator, the jth call to `next` (counting from 0) will return a string of j *a*s. In general, an enumerator class won't be limited to enumerating the strings in order of their length, but this one just happens to be specified that way.

For a more interesting example, suppose we have a method `isPrime` that tests whether a number is prime. Then we can implement this enumerator class:

```
class TwinPrimes {
  int i = 1;

  String next() {
    while (true) {
      i++;
      if (isPrime(i) && isPrime(i+2))
        return i + "," + (i+2);
    }
  }
}
```

This is an enumerator for a language of twin primes. A *twin prime* is a pair of numbers (i, i+2) that are both prime. So a series of calls to `next` would return

```
3,5
5,7
```

```
11,13
17,19
29,31
```

and so on. There is a famous conjecture that there are infinitely many twin primes, but no one has been able to prove it. So the language enumerated by `TwinPrimes` may or may not be infinite. If it is not, there is some largest pair of twin primes, and a call made to `next` after that largest pair has been returned will run forever.

Here's an enumeration problem whose solution we'll need later on: make an enumerator class for the set of all pairs of numbers from $\mathcal{N}$, $\{(j, k) \mid j \geq 0, k \geq 0\}$. (As usual, we'll represent the natural numbers j and k as decimal strings.) This is a bit trickier. This enumerates all the pairs $(0, k)$:

```
class BadNatPairs1 {
  int k = 0;

  String next() {
    return "(0," + k++ + ")";
  }
}
```

But that's not good enough. This enumerates all the pairs (j, k) where $j = k$, and that's not good enough either:

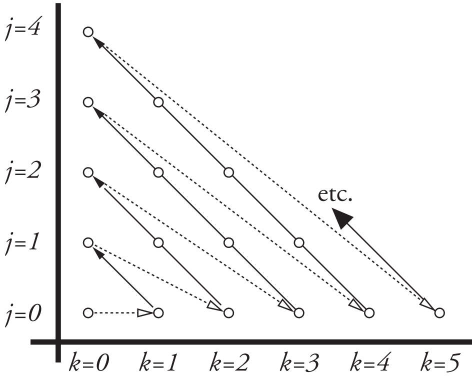
```
class BadNatPairs2 {
  int j = 0;
  int k = 0;

  String next() {
    return "(" + j++ + "," + k++ + ")";
  }
}
```

Instead we need to visit the (j, k) pairs in an order that eventually reaches every one. This is one way:

```
class NatPairs {
  int n = 0;
  int j = 0;
```

```
      String next() {
        String s = "(" + j + "," + (n-j) + ")";
        if (j<n) j++;
        else {j=0; n++;}
        return s;
      }
    }
```

It enumerates the (*j*, *k*) pairs in a triangular order, like this:



so every point in the space, every pair (*j*, *k*) in the language, is reached eventually.

For a last example, imagine defining an enumerator class `SigmaStar` that enumerates Σ*. If Σ = {*a*, *b*}, a `SigmaStar` object might return these strings for a sequence of calls to `next`:

```
""
"a"
"b"
"aa"
"ab"
"ba"
"bb"
"aaa"
```

and so on. The exact order in which strings are returned doesn't matter for our purposes. `SigmaStar` is not too difficult to implement, and this is left as an exercise.

Enumerators generate the strings in a language. You can also think of them as *numbering* the strings in the language—since the strings are generated in a fixed order. For example, we could define a method `sigmaStarIth(i)` that takes a natural number `i` and returns the `String` that is the `i`th one enumerated by `SigmaStar`:

```
String sigmaStarIth(int i) {
  SigmaStar e = new SigmaStar();
  String s = "";
  for (int j = 0; j<=i; j++) s = e.next();
  return s;
}
```

By the way, nothing in our definitions requires enumerators to generate a unique string on every call, so `sigmaStarIth` does not necessarily give a one-to-one mapping from $\mathcal{N}$ to Σ^*. In mathematical terms it is merely a *surjection* (an *onto* function): for every string $s \in \Sigma^*$ there is at least one i such that `sigmaStarIth`(i) = s.

We'll make use of both `sigmaStarIth` and `NatPairs` in the next section.

18.8 Recursively Enumerable Languages

It turns out that Turing's original enumeration machines are equivalent to our TM model, in the following sense:

> **Theorem 18.5:** A language is RE if and only if it is $L(M)$ for some enumeration machine M.

The term *recursively enumerable* makes more sense in this light!

We know that a language is RE if and only if it has a recognition method. So to prove the theorem we will give two constructions. First, given an enumerator class `AEnumerate` with L(`AEnumerate`) = A, we construct a recognition method `aRecognize`, showing that A is an RE language. This is straightforward.

```
boolean aRecognize(String s) {
  AEnumerate e = new AEnumerate();
  while (true)
    if (s.equals(e.next())) return true;
}
```

This method returns true if and only if `s` is eventually enumerated by `AEnumerate`. Thus L(`aRecognize`) = L(`AEnumerate`).

The construction in the other direction is a bit trickier. Given a recognition method `aRecognize` for some RE language A, we need to show that we can construct an enumerator class AEnumerate for the same language. Here is an attempt that doesn't quite work. It uses the SigmaStar enumerator to generate each possible string in Σ*. To find the next string in *A*, it tries successive strings in Σ* until it finds one that aRecognize accepts:

```
class BadAEnumerate {
  SigmaStar e = new SigmaStar();

  String next() {
    while (true) {
      String s = e.next();
      if (aRecognize(s)) return s;
    }
  }
}
```

That would work just fine if aRecognize were total—a decision method and not just a recognition method. But if aRecognize runs forever on one of the strings generated by SigmaStar, next will get stuck. It will never get around to testing any of the subsequent strings.

To solve this problem we will introduce a time-limited version of run. Define runWithTimeLimit(p,in,j) so that it returns true if and only if p returns true for in within j steps of the simulation. (Various definitions of a "step" are possible here: a statement, a virtual machine instruction, and so on. It doesn't matter which one you use. For TMs the meaning of "step" would be more obvious: one move of the TM.) This runWithTimeLimit can be total, because it can return false if p exceeds j steps without reaching a decision about in.

Using runWithTimeLimit, we can make an enumerator for *A*. For each (*j*, *k*) pair, we check whether the *j*th string in Σ* is accepted by aRecognize within *k* steps. This is where we can use the NatPairs enumerator and the sigmaStarIth method from the previous section. NatPairs enumerates all the (*j*, *k*) pairs we need, and sigmaStarIth finds indexed strings in Σ*.

```
class AEnumerate {
  NatPairs e = new NatPairs();

  String next() {
    while (true) {
      int (j,k) = e.next();
      String s = sigmaStarIth(j);
```

```
        if (runWithTimeLimit(aRecognize,s,k)) return s;
      }
    }
  }
```

(This uses some syntax that is not proper Java. The `NatPairs` method `next` returns a string giving a pair of natural numbers such as `"(13,7)"`, and it would take a few real Java statements to break that string apart and convert the numeric strings back into integer values `j` and `k`.)

Clearly `AEnumerate` only enumerates strings accepted by `aRecognize`. Further, every string `s` accepted by `aRecognize` is eventually enumerated by `AEnumerate`, because every such `s = sigmaStarIth(j)` for some `j`, and every such `s` is accepted by `aRecognize` within some number of steps `k`, and `AEnumerate` tries all such pairs `(j,k)`. Thus, L(`AEnumerate`) = L(`aRecognize`). That completes the proof of Theorem 18.5.

18.9 Languages That Are Not RE

We have seen examples of nonrecursive languages like L_h and L_u. Are there languages that are not only nonrecursive, but also nonRE? Yes, and they are easy to find.

Theorem 18.6: If a language is RE but not recursive, its complement is not RE.

Proof: Let L be any language that is RE but not recursive. Assume by way of contradiction that $\overline{L}$ is also RE. Then there exist recognition methods `lrec` and `lbar` for both languages, so we can implement

```
boolean ldec(String s) {
  for (int j = 1; ; j++) {
    if (runWithTimeLimit(lrec,s,j)) return true;
    if (runWithTimeLimit(lbar,s,j)) return false;
  }
}
```

If `s` $\in L$, then `runWithTimeLimit(lrec,s,j)` returns true for some `j`, so `ldec(s)` returns true. If `s` $\notin L$, then

`runWithTimeLimit(lbar,s,j)`

returns true for some `j`, so `ldec(s)` returns false. Thus `ldec` is a decision method for L. But this is a contradiction, because L is not recursive. By contradiction, $\overline{L}$ is not RE.

From this we conclude that the RE languages are not closed for complement. Notice that the recursive languages *are* closed for complement; given a decision method `ldec` for L, we can construct a decision method for $\overline{L}$ very simply:

```
boolean lbar(String s) {return !ldec(s);}
```

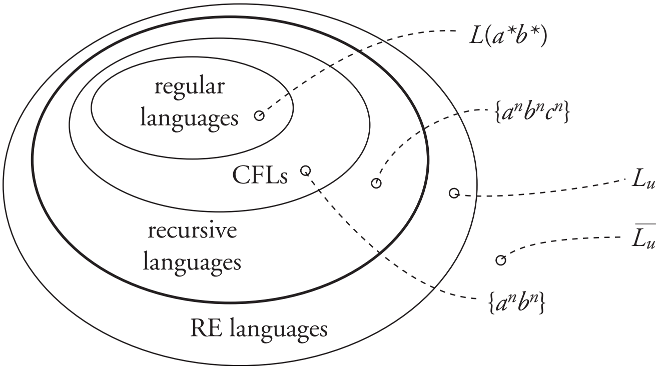
But when a language is RE but not recursive, this kind of construction fails. If the recognition method `lrec(s)` runs forever, `!lrec(s)` will too.

At the beginning of this chapter we saw that the languages L_h and L_u are RE but not recursive. From Theorem 18.6 it follows that these complements are not RE:

$\overline{L_u}$ = {(`p`, `s`) | `p` is *not* a recognition method that returns true for `s`}
$\overline{L_h}$ = {(`p`, `s`) | `p` is *not* a recognition method that halts given `s`}

18.10 Language Classifications Revisited

Here's a summary picture of our language categories, with an example language for each of the five regions:



When a language is recursive, there is an effective computational procedure that can definitely categorize all strings. Given a positive example it will decide yes; given a negative example it will decide no. A language that is *recursive*, a property that is *decidable*, a function that is *computable*: these terms all refer to computations like those performed by total TMs, computations that always halt.

By contrast, languages that are RE but not recursive are those for which only the positive examples can be definitely categorized. A TM for such a language will definitely answer *yes*, given any string in the language. But given a string that is not in the language, the TM may run indefinitely without reaching a decision. You get a definite *yes* if the answer is yes, but you don't get a definite *no* if the answer is no. A property like that is sometimes called *semi-decidable*. For example, the language L_h is RE but not recursive, and the corresponding property (`p` halts on `s`) is said to be semi-decidable. If `p` does halt on `s`, a simple simulation will always give you a definite *yes* answer, but if not, neither simulation nor any other approach will always give you a definite *no* answer.

When a language is not RE, that means that there is no procedure for categorizing strings that gives a definite *yes* answer on all positive examples. Consider the language $\overline{L_h}$, for example. One kind of positive example for this language would be a pair (`p`, `s`) where `p` is a recognition method that runs forever given `s`. But there is no way to identify such pairs—obviously, you can't simulate `p` on `s`, see if it runs forever, and then say *yes*!

18.11 Grammars and Computability

We defined grammars using productions of the general form $x \to y$, where x and y are any strings with $x \neq \varepsilon$. But all our examples have followed a more restricted form: they have been context-free grammars, grammars with a single nonterminal symbol on the left-hand side of every production. Using productions that are not context free, it is possible to define more languages.

For example, consider this grammar:

$$S \to aBSc \mid abc \mid \varepsilon$$
$$Ba \to aB$$
$$Bb \to bb$$

Here are some derivations for this grammar:

$$S \Rightarrow \varepsilon$$

$$S \Rightarrow abc$$

$$S \Rightarrow aBSc \Rightarrow aBabcc \Rightarrow aaBbcc \Rightarrow aabbcc$$

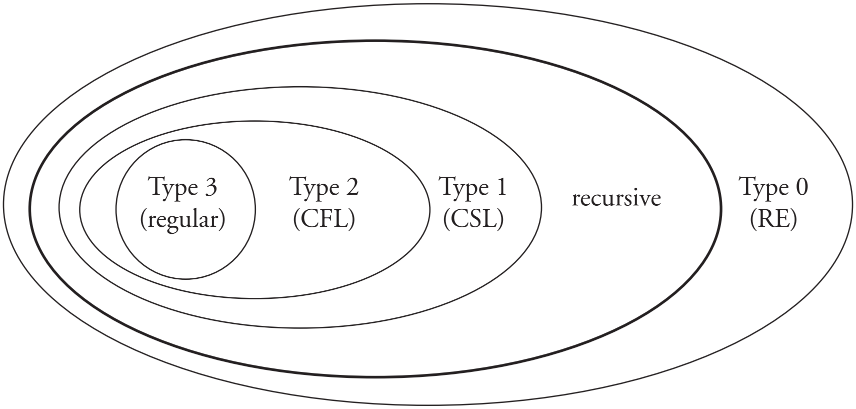
$$S \Rightarrow aBSc \Rightarrow aBaBScc \Rightarrow aBaBabccc \Rightarrow aaBBabccc \Rightarrow aaBaBbccc \Rightarrow aaaBBbccc \Rightarrow aaaBbbccc \Rightarrow aaabbbccc$$

The language generated is $\{a^n b^n c^n\}$: recursive but not context free.

In the late 1950s, Noam Chomsky proposed four classifications for grammars, using syntactic restrictions on the forms of productions. These are

- Type 0 (unrestricted): All forms allowed.
- Type 1 (context sensitive): All productions are of the form $xAz \to xyz$, where $y \neq \varepsilon$. Such a production is called *context sensitive* because it allows A to be replaced with y, but only in a particular context, with x to the left and z to the right. (The production $S \to \varepsilon$ is also allowed if S does not appear on the right side of any production.)
- Type 2 (context free): As in our definition in Section 12.1.
- Type 3 (right linear): As in our definition in Section 10.5.

Grammars of these four types define a hierarchy of four classes of languages called the *Chomsky hierarchy*:



As the illustration suggests, there is a remarkable correspondence between the Chomsky grammar classifications and some of the major classes of formal languages. We have already seen that a language is regular if and only if it is generated by some type-3 grammar and that a language is context free if and only if it is generated by some type-2 grammar. The type-0, unrestricted grammars also generate a familiar class of languages: the RE languages. Thus unrestricted grammars are yet another Turing-equivalent formalism. In fact they are closely related to the Post systems already mentioned.

Type-1 grammars also generate an interesting class of languages, one that we have not mentioned before: the *context-sensitive languages* (*CSLs*). The CSLs are a superset of the CFLs and a subset of the recursive languages. Recursive languages encountered in practice usually turn out to be CSLs. There are languages that are recursive but not context sensitive, but they're hard to find.

There is another interesting way to define the CSLs. Start with the TM model, add nondeterminism, and impose the restriction that writing anything over a **B** is not permitted. In effect, the TM's infinite-tape feature is eliminated—it is restricted to using that part of the tape initially occupied by the input string. The resulting automaton is called a *nondeterministic linear-bounded automaton* (*NLBA*). It turns out that a language is accepted by some NLBA if and only if it is a CSL! So just like our other language classes, the CSLs arise unexpectedly in several different ways.

Even the CFLs, familiar as they are, harbor some computability surprises. Rice's theorem shows that for all nontrivial properties α, the language

$$\{\texttt{p} \mid \texttt{p} \text{ is a recognition method and } L(\texttt{p}) \text{ has the property } \alpha\}$$

is not recursive. Although there is nothing as categorical as Rice's theorem for CFGs, there are a number of interesting properties α for which the language

$\{G \mid G \text{ is a CFG and } L(G) \text{ has the property } \alpha\}$

is not recursive. For example, these languages are not recursive:

$\{G \mid G \text{ is a CFG and } L(G) = \Sigma^*\}$

$\{G \mid G \text{ is a CFG and } \overline{L(G)} \text{ is a CFL}\}$

Similarly, the question of whether two CFGs generate the same language is undecidable, as is the question of whether the intersection of two CFGs is context free.

Formal language is an area where computer science and linguistics overlap. Although the Chomsky hierarchy is well known in computer science, Noam Chomsky is even better known in linguistics, where his theories on human language acquisition have been very influential.

18.12 Oracles

Here are two languages that Rice's theorem shows are nonrecursive:

$L_e = \{\texttt{p} \mid \texttt{p} \text{ is a recognition method and } L(\texttt{p}) = \{\}\}$

$L_f = \{\texttt{p} \mid \texttt{p} \text{ is a recognition method and } L(\texttt{p}) = \Sigma^*\}$

The first is the set of recognition methods that always return false or run forever; the second is the set of recognition methods that always return true. Neither one is recursive, and neither is RE. But there is a sense in which one is much harder to recognize than the other.

To the purely practical mind, that last statement is meaningless. Undecidable is undecidable; everything beyond the computability border is out of reach, so why worry about additional distinctions? But for those with active mathematical imaginations, the unreachable territory beyond the computability border has a fascinating structure.

Earlier in this chapter, we proved that L_h is not recursive. Later, we showed that L_e is not recursive, using a reduction from L_h. In other words, we showed that if there were a way to decide L_e, we could use that to decide L_h. Our conclusion at the time was that, since we already know there is no decision method for L_h, there couldn't be one for L_e either. But what if there *were* some way of deciding membership in L_e? We know that `empty` can't be a decision method in our ordinary sense, but what if it decides L_e somehow anyway? Then our construction would allow us to decide L_h as well.

Turing machines with such impossible powers are called *oracle* machines. They are just like ordinary Turing machines, except that they have a one-step way of checking a string's membership in a particular language. The language of this oracle can be anything we choose. Giving a TM an oracle for a nonrecursive language like L_e increases the range of languages it can decide. In particular, our previous construction shows that given an oracle for L_e, the language L_h is recursive. With a different construction you can also show the reverse: given an oracle for L_h, the language L_e is recursive.

But an oracle for L_h would not put an end to uncomputability. In particular, although it can decide the halting problem for ordinary Turing machines, it cannot decide the halting problem for Turing machines with L_h oracles! That would require a stronger oracle, whose addition to TMs would make the halting problem even harder, requiring a still stronger oracle, and so on. The result is an infinite hierarchy of oracles. It turns out that L_e is recursive given an oracle for L_h, but L_f is not—it requires a more powerful oracle. In that sense, L_f is harder to decide than L_e.

18.13 Mathematical Uncomputabilities

All the nonrecursive languages we have looked at have been languages of programs, like the languages covered by Rice's theorem. These languages are of natural interest to programmers, but uncomputability turns up in many other domains as well. In particular, it turns up at the foundations of mathematics.

Mathematics can be thought of as a structure of theorems, each of which can be proved from simpler theorems by mechanically following the rules of logic. The whole edifice of mathematics can be built up in this way, starting from the foundational axioms of arithmetic. The foundational axioms are not proved, but are accepted as true because they are simple and self-evident. If you think of mathematics this way, it is important to know that the axiom foundation is firm. The axioms should be *consistent*, meaning that they lead to no false theorems. They should be *complete*, meaning that all true theorems can be proved.

David Hilbert was a champion of this view of mathematics. He is widely regarded as one of the greatest mathematicians of his time. He contributed to many different branches of mathematics, but he is especially well known for a list of 23 open problems that he presented in a lecture at an international conference in Paris in 1900. Hilbert's list drove a great deal of 20th-century mathematical research, and a solution to any problem on the list brought fame to the mathematician who solved it. One of the problems on Hilbert's list (the second) was the problem of proving that the foundational axioms of mathematics are consistent. He was convinced that such a proof was possible. In general, he was confident that finite proof or disproof was always possible for well-formed mathematical conjectures. At the time, most of the world's mathematicians would have agreed with him.

But it turned out far otherwise. In 1930, Kurt Gödel published his two famous incompleteness theorems. First, he showed that there is a mathematical assertion that says, in effect,

This assertion has no proof.

That kind of thing is easy enough to write in English and not hard to do with a computer program. But Gödel's genius was to express that self-referential idea in the extremely limited language of formal mathematical assertions about arithmetic in $\mathcal{N}$. If the assertion is false, then you have a proof of it—a proof of something false, showing that the axioms are not consistent. If the assertion is true, then you have an assertion that is true but has no proof, showing that the axioms are not complete. Gödel's results apply not just to a particular axiomatic system, but to any axiomatic system capable of expressing the same basic things about arithmetic. No such system can be both consistent and complete. Gödel's second incompleteness theorem then goes on to show that no consistent axiomatic system that captures arithmetic can prove its own consistency. Gödel's results came as quite a surprise to those who expected a quick solution to Hilbert's second problem.

For long years before Hilbert, mathematicians had been wishing for an effective computational procedure to find proofs. It was a dream of Gottfried Wilhelm von Leibnitz, who demonstrated a mechanical calculator in 1673 and hoped to bring its computational advantages to the realm of logic. Hilbert considered this to be the central challenge of mathematical logic. Gödel's results showed that some true theorems must remain unprovable, but left open the question of whether the provable ones could be identified computationally. In 1936, both Alonzo Church and Alan Turing independently added a related result: the provability of theorems of arithmetic is undecidable. Turing's proof related arithmetic to Turing machines, showing that you can write

Turing machine M halts on input x.

as a mathematical assertion about arithmetic in $\mathcal{N}$.

Since then, many other mathematical problems have been found to lie on the far side of the computability border. One example is the problem of finding solutions to Diophantine equations. A Diophantine equation is a polynomial equation using integer variables and constants; an example is the equation $x^2 + y^2 = z^2$, whose integer solutions are the Pythagorean triples like (x=3, y=4, z=5). The problem of finding solutions to such equations was another of Hilbert's (the tenth) and remained open until 1970, when Yuri Matiyasevich showed that it could not be solved. Specifically, he showed that for every Turing machine M there is some Diophantine equation with a variable x that has solutions exactly where $x \in L(M)$. In effect, such equations cannot really be *solved*; they encode computational procedures, possibly nonterminating ones, that can only be *simulated*. This shows again the close ties between computer science and the foundations of mathematics.

Exercises

EXERCISE 1

Implement Java decision methods for the following languages:

a. $\{abc\}$

b. $L(a^*b^*)$

c. $\{a^n b^{2n}\}$

d. $\{xx \mid x \in \{a, b\}^*\}$

e. $\{x \in \{0, 1\}^* \mid |x| \text{ is a prime number}\}$

f. $\{x \in \{0, 1\}^* \mid x \text{ is a binary representation of an odd number}\}$

EXERCISE 2

For each of these Java decision methods, give a total Turing machine for the same language.

a.
```
boolean ax(String p) {
  return (p.length()>0 && p.charAt(0)=='a');
}
```

b.
```
boolean emptySet(String p) {
  return false;
}
```

c.
```
boolean sigmaStar(String p) {
  return true;
}
```

d.
```
boolean d1(String p) {
  String s = "";
  int n = p.length();
  for (int i = 0; i<n; i++) s = p.charAt(i) + s;
  return p.equals(s);
}
```

EXERCISE 3

For each of these Java recognition methods, give a Java decision method that accepts the same language.

a.
```
boolean r1(String p) {
  while (true) {}
  return true;
}
```

b.
```
boolean r2(String p) {
  if (p.length()<10) return true;
  else return r2(p);
}
```

c.
```
boolean r3(String p) {
  while (true) {
    if (p.length()<10) p += 'a';
  }
  return (p.length()>=10);
}
```

d.
```
boolean r4(String p) {
    String s = "";
    while (true) {
      s += 'a';
      if (s.equals(p)) return true;
    }
    return false;
  }
```

EXERCISE 4

Given the `shortcut` and `halts` methods as described in Sections 18.2 and 18.3, what would be the outcome of these code fragments? (Of course, we proved no such methods exist; but assume here that we have some way of meeting their specifications.)

a.
```
String s =
  "boolean sigmaStar(String p) {  " +
  "  return true;                 " +
  "}                              ";
shortcut(s,"fourscore and seven years ago");
```

b.
```
String s =
  "boolean sigmaStar(String p) {  " +
  "  return true;                 " +
  "}                              ";
halts(s,"fourscore and seven years ago");
```

c.
```
String s =
  "boolean emptySet(String p) {  " +
  "  return false;               " +
  "}                             ";
shortcut(s,s);
```

d.
```
String s =
  "boolean emptySet(String p) {  " +
  "  return false;               " +
  "}                             ";
halts(s,s);
```

e.
```
String s =
  "boolean r1(String p) { " +
  "  while (true) {}      " +
  "  return true;         " +
  "}                      ";
shortcut(s,"abba");
```

f.
```
String s =
    "boolean r1(String p) { " +
    "  while (true) {}      " +
    "  return true;         " +
    "}                      ";
halts(s,"abba");
```

EXERCISE 5

Consider the `nonSelfAccepting` method used in Section 18.2 to prove by contradiction that `shortcut` cannot be implemented. If we reimplement it like this:

```
boolean nonSelfAccepting(String p) {
  return !run(p,p);
}
```

does it lead in the same way to a contradiction proving that `run` cannot be implemented? Carefully explain why or why not.

EXERCISE 6

Prove that this language is not recursive:

{`p` | `p` is a recognition method that returns false for the empty string}

Hint: Follow the pattern in Section 18.5; you can't use Rice's theorem for this one.

EXERCISE 7

Prove that this language is not recursive:

{`p` | `p` is a method that takes no parameters and returns the string "Hello world"}

Hint: Follow the pattern in Section 18.5; you can't use Rice's theorem for this one.

EXERCISE 8

For each of the following languages, decide whether Rice's theorem applies, and explain why it does or does not.

a. {`p` | `p` is a recognition method and L(`p`) contains the empty string}

b. {`p` | `p` is a recognition method and L(`p`) $\subseteq \Sigma^*$}

c. {m | m encodes a Turing machine M with $abba \in L(M)$}

d. {`p` | `p` is a recognition method that contains the statement `x=1;`}

e. {`p` | `p` is a recognition method that, when run on the empty string, executes the statement `x=1;`}

f. {m | m encodes a Turing machine M for which $L(M)$ is finite}

EXERCISE 9

Classify the following languages as either

A. regular,
B. context free but not regular,
C. recursive but not context free, or
D. not recursive,

and explain for each why you chose that answer. You do not have to give detailed proofs—just give one or two sentences that outline how you would prove your classification.

a. $\{m\#x \mid m$ encodes a TM and x is a string it accepts$\}$
b. $\{a^n b^n c^n\}$
c. $\{x \in \{a, b\}^* \mid$ the number of as in x is the same as the number of bs in $x\}$
d. $\{a^n b^{2n}\}$
e. {`p` | `p` is a recognition method that is syntactically legal (that is, it can be parsed using a standard grammar for Java)}
f. the set of strings that contain the Java statement `while(true){}`
g. {(`p, x`) | `p` is a recognition method that, when run on the input `x`, executes the statement `while(true){}`}
h. {`p` | `p` is a recognition method and $L($`p`$) \neq \{\}$}
i. $\{m \mid m$ encodes a TM M with $L(M) \neq \{\}\}$
j. $\{xx \mid x \in \{a, b, c\}^* \}$
k. $\{(m, x) \mid m$ encodes a TM M that, when run on the input x, visits a state $q_7\}$
l. $L(a^*b^*) \cap L((a + ba^*b)^*)$
m. $\{xx^R \mid x \in \{0, 1\}^*\}$

EXERCISE 10

Taking the alphabet to be $\Sigma = \{a, b\}$, implement a Java enumerator class `SigmaStar`. Generate the strings in order of their length: first ε, then the strings of length 1, then the strings of length 2, and so on. Among strings of the same length any fixed order is acceptable.

EXERCISE 11

Repeat Exercise 10, but add a `String` parameter to the `SigmaStar` constructor, and take Σ to be the set of characters in that `String`. You may assume that the string is not empty, so $\Sigma \neq \{\}$.

EXERCISE 12

Implement a Java decision method for $\{xx \mid x \in \{a, b\}^*\}$.

EXERCISE 13

Implement a Java enumerator class for $\{xx \mid x \in \{a, b\}^*\}$.

EXERCISE 14

In the proof of Theorem 18.5, how often will each string in the language recognized by `aRecognize` be enumerated by `AEnumerate`? Explain.

EXERCISE 15

Suppose you're given a Java decision method `javaRec` that decides the language {`p` | `p` is a Java recognition method}. (The `javaRec` method would verify that the string `p` compiles without error as a Java method, that it has a single `String` parameter, and that it returns a `boolean` value.) Using this, write an enumerator class for {`p` | `p` is a Java recognition method}.

EXERCISE 16

Using `javaRec` defined as in Exercise 15, write an enumerator class for the nonrecursive language {`p` | `p` is a Java recognition method and $\varepsilon \in L$(`p`)}. *Hint:* This is like the construction of `AEnumerate` in Section 18.8. You'll need `NatPairs`, `runWithTimeLimit`, and your solution to Exercise 15.

EXERCISE 17

Using `javaRec` defined as in Exercise 15, write an enumerator class for the nonrecursive language {`p` | `p` is a Java recognition method and L(`p`) ≠ {}}. *Hint:* You may assume you have an enumerator class `NatTriples` that enumerates natural-number triples (i, j, k).

EXERCISE 18

Let `AEnumerate` be an enumerator class with the special property that it enumerates the strings of an infinite language A in order of their length, shortest first. Give a decision method for A, thus proving that A is recursive.

EXERCISE 19

Give a grammar for the language $\{a^n b^n c^n d^n\}$, and show a derivation of the string *aaabbbcccddd*.

EXERCISE 20

Give a grammar for the language $\{xx \mid x \in \{a, b\}^*\}$, and show a derivation of the string *aabaab*.

EXERCISE 21

Using decision methods, show that an oracle for L_h makes L_u recursive.

EXERCISE 22

Using decision methods, show that an oracle for L_u makes L_h recursive.

EXERCISE 23

Using decision methods, show that an oracle for L_e makes L_u recursive.

EXERCISE 24

Using decision methods, show that an oracle for L_u makes L_e recursive. (*Hint:* See the construction of `AEnumerate` in Section 18.8.)

19 Cost Models

Up to this point, we have focused on computability without considering efficiency. As practical programmers, however, we are naturally interested in understanding the resource requirements of the code we write. How much time a program needs and how much memory space it needs are important practical questions. A **cost model** *is a tool for answering such questions systematically. In this chapter, we define cost models for TMs and for our Java-like programs.*

19.1 Asymptotic Notation

If you've studied algorithms and data structures, you're probably already familiar with the big-O notation and its close cousins, Θ and Ω. If not, here's a quick introduction.

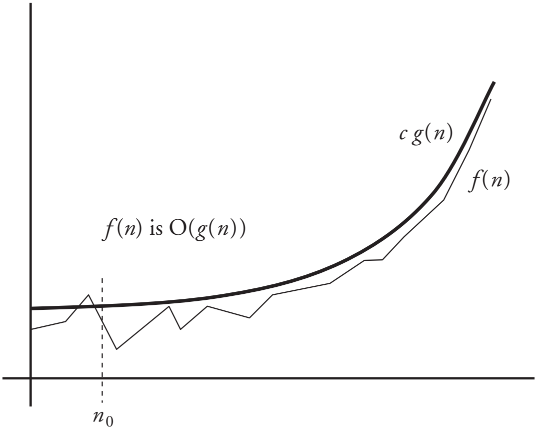
The big-O notation is used to say something trenchant about a complicated function f by relating it to a simpler function g.

Let f and g be any two functions over $\mathcal{N}$. We say

$$f(n) \text{ is } O(g(n))$$

if and only if there exist natural numbers c and n_0 so that for every $n \geq n_0$, $f(n) \leq c\, g(n)$.

Typically, the function f is the complicated function whose rate of growth we're trying to characterize; the function g is some simple function, like $g(n) = n^2$, whose rate of growth is easy to grasp. The assertion *f(n) is O(g(n))* is a way of saying that f grows no faster than g. If you multiply $g(n)$ by some constant c, and plot the result, it will be above the plot of $f(n)$ almost everywhere—like this:



Some exceptions to $f(n) \leq c\, g(n)$ may occur among small values of n, those less than n_0; the assertion "$f(n)$ is $O(g(n))$" says that some such constant n_0 exists, but does not say what it is, just as it does not say what the constant of proportionality c is. As the illustration suggests, the big-O notation describes *upper bounds*. When you assert that $f(n)$ is $O(g(n))$ you are saying that f grows no faster than g; that doesn't rule out the possibility that f actually grows much slower than g. For example, if $f(n)$ is $O(n^2)$, then by definition it is also true (though less informative) to say that $f(n)$ is $O(n^3)$, $O(n^4)$, or $O(n^{100})$.

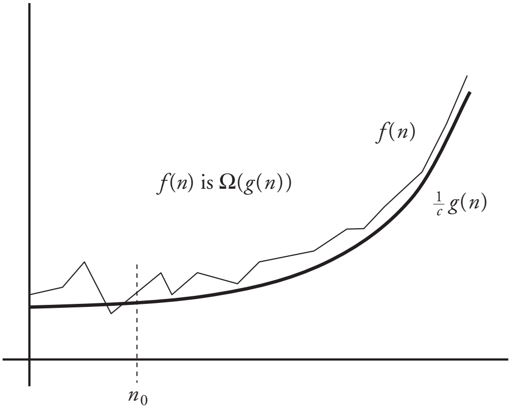
When you want to describe a lower bound on a function—to say that f grows at least as fast as g—there is a related asymptotic notation using the symbol Ω.

Let f and g be any two functions over $\mathcal{N}$. We say

$$f(n) \text{ is } \Omega(g(n))$$

if and only if there exist natural numbers c and n_0 so that for every $n \geq n_0$, $f(n) \geq \frac{1}{c}g(n)$.

Thus, if you divide $g(n)$ by some constant c and plot the result, it will be below the plot of $f(n)$ almost everywhere—like this:



Again, some exceptions may occur among small values of n, those less than n_0.

As the illustration suggests, the Ω notation describes *lower bounds*. When you assert that $f(n)$ is $\Omega(g(n))$ you are saying that f grows at least as fast as g; that doesn't rule out the possibility that f actually grows much faster than g. For example, if $f(n)$ is $\Omega(n^3)$, then by definition it is also true (though less informative) to say that $f(n)$ is $\Omega(n^2)$, $\Omega(n)$, or $\Omega(1)$.

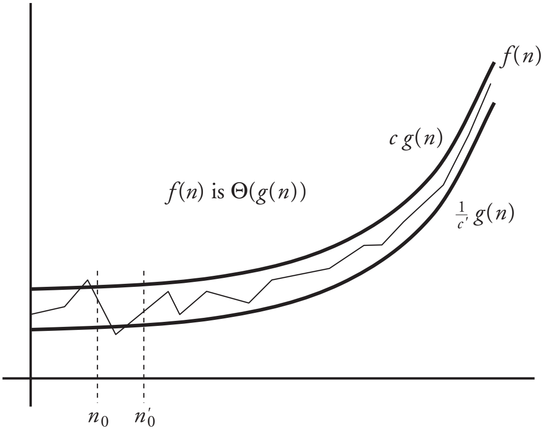
If you inspect the definitions for O and Ω, you'll see that it is possible for a function $f(n)$ to be both $O(g(n))$ and $\Omega(g(n))$ for the same function g. In fact, this is a very important situation: when your upper and lower bounds are the same, you know that you have an asymptotic characterization of the function f that is as informative as possible. There is a special notation for this:

Let f and g be any two functions over $\mathcal{N}$. We say

$$f(n) \text{ is } \Theta(g(n))$$

if and only if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

The important thing to observe here is that the constants involved, c and n_0, are *not* the same for both the upper bound and the lower bound. For the upper bound $O(g(n))$ we have one pair of constants c and n_0, and for the lower bound $\Omega(g(n))$ we have a different pair of constants—call them c' and n_0':



The Θ notation is used to describe *tight* bounds. When you assert that $f(n)$ is $\Theta(g(n))$ you are saying that f grows at the same rate as g, neither faster nor slower.

19.2 Properties of Asymptotics

The asymptotic notations have properties that allow many functions to be simplified. That's exactly what makes them useful; they can describe a function that is difficult to understand by relating it to one that is easy to understand. One important simplifying property is that constant factors can generally be eliminated.

Theorem 19.1.1, O version: For any nonzero $k \in \mathcal{N}$, $f(n)$ is $O(k \cdot g(n))$ if and only if $f(n)$ is $O(g(n))$.

Proof: If $f(n)$ is $O(k \cdot g(n))$ then, by definition, there exist natural numbers c and n_0 so that for every $n \geq n_0$, $f(n) \leq ck \cdot g(n)$. Let $c' = ck$. Now, for every $n \geq n_0$, $f(n) \leq c' \cdot g(n)$. Thus, by definition, $f(n)$ is $O(g(n))$. (The other direction of the proof is trivial, because $g(n) \leq k \cdot g(n)$.)

This property says, for example, that $O(3n^2)$ and $O(n^2)$ are equivalent. Thus you should never rest with the conclusion that a function is $O(3n^2)$, because $O(n^2)$ is equivalent and simpler.

The same theorem also applies for Ω and Θ, with similar proofs:

Theorem 19.1.1, Ω version: For any nonzero $k \in \mathcal{N}$, $f(n)$ is $\Omega(k \cdot g(n))$ if and only if $f(n)$ is $\Omega(g(n))$.

Theorem 19.1.1, Θ version: For any nonzero $k \in \mathcal{N}$, $f(n)$ is $\Theta(k \cdot g(n))$ if and only if $f(n)$ is $\Theta(g(n))$.

Another important simplification is that in a sum of terms, all but the fastest growing can be eliminated.

Theorem 19.1.2, O version: Let $\{g_1, \ldots, g_m\}$ be any finite set of functions over $\mathcal{N}$ in which g_1 is maximal, in the sense that every $g_i(n)$ is $O(g_1(n))$. Then for any function f, $f(n)$ is $O(g_1(n) + g_2(n) + \ldots + g_m(n))$ if and only if $f(n)$ is $O(g_1(n))$.

Proof: If $f(n)$ is $O(g_1(n) + g_2(n) + \ldots + g_m(n))$ then, by definition, there exist natural numbers c_0 and n_0 so that for every $n \geq n_0$,

$$f(n) \leq c_0(g_1(n) + \cdots + g_m(n))$$

We are given that every term $g_i(n)$ is $O(g_1(n))$, so by definition there exist natural numbers $\{c_1, \ldots, c_m\}$ and $\{n_1, \ldots, n_m\}$ such that, for every i and every $n \geq \max(n_0, \ldots, n_m)$, $g_i(n) \leq c_i g_1(n)$. Therefore, for every $n \geq \max(n_0, \ldots, n_m)$,

$$\begin{aligned} f(n) &\leq c_0(g_1(n) + \cdots + g_m(n)) \\ &\leq c_0(c_1 g_1(n) + \cdots + c_m g_1(n)) \\ &= c_0(c_1 + \cdots + c_m) g_1(n) \end{aligned}$$

By choosing $c' = c_0(c_1 + \ldots + c_m)$ and $n' = \max(n_0, \ldots, n_m)$, we can restate this as follows: for every $n \geq n'$, $f(n) \leq c' \cdot g_1(n)$. Thus, by definition, $f(n)$ is $O(g_1(n))$. (In the other direction, the proof is trivial, using the fact that $g_1(n) \leq g_1(n) + g_2(n) + \ldots + g_m(n)$.)

Because of this property, sums of terms inside the big O can almost always be simplified. You just identify the fastest growing term and discard the others. For example, you would never conclude that a function is $O(n^2 + n + \log n + 13)$, because $O(n^2)$ is equivalent and simpler. If you ever find yourself writing "+" inside the big O, think twice; it's almost always possible to simplify using Theorem 19.1.2.

Theorem 19.1.2 also applies to the Ω and Θ forms:

Theorem 19.1.2, Ω version: Let $\{g_1, \ldots, g_m\}$ be any finite set of functions over $\mathcal{N}$ in which g_1 is maximal, in the sense that every $g_i(n)$ is $O(g_1(n))$. Then for any function f, $f(n)$ is $\Omega(g_1(n) + g_2(n) + \ldots + g_m(n))$ if and only if $f(n)$ is $\Omega(g_1(n))$.

Theorem 19.1.2, Θ version: Let $\{g_1, \ldots, g_m\}$ be any finite set of functions over $\mathcal{N}$ in which g_1 is maximal, in the sense that every $g_i(n)$ is $O(g_1(n))$. Then for any function f, $f(n)$ is $\Theta(g_1(n) + g_2(n) + \ldots + g_m(n))$ if and only if $f(n)$ is $\Theta(g_1(n))$.

Interestingly, we still select the maximal term to keep (defined using O), even in the Ω and Θ versions of this theorem. You'll see how this works out if you do Exercise 7.

Another property that is very useful for analyzing algorithms is this:

Theorem 19.1.3, O version: If $f_1(n)$ is $O(g_1(n))$, and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) \cdot f_2(n)$ is $O(g_1(n) \cdot g_2(n))$.

Theorem 19.1.3, Ω version: If $f_1(n)$ is $\Omega(g_1(n))$, and $f_2(n)$ is $\Omega(g_2(n))$, then $f_1(n) \cdot f_2(n)$ is $\Omega(g_1(n) \cdot g_2(n))$.

Theorem 19.1.3, Θ version: If $f_1(n)$ is $\Theta(g_1(n))$, and $f_2(n)$ is $\Theta(g_2(n))$, then $f_1(n) \cdot f_2(n)$ is $\Theta(g_1(n) \cdot g_2(n))$.

The proofs of these are left as an exercise (Exercise 4). They come in handy for analyzing loops; if you know that the time taken by each iteration of a loop is $O(g_1(n))$, and you know that the number of iterations is $O(g_2(n))$, then you can conclude that the total amount of time taken by the loop is $O(g_1(n) \cdot g_2(n))$.

One final property: the O and Ω forms are symmetric:

Theorem 19.1.4: For any functions f and g over $\mathcal{N}$, $f(n)$ is $O(g(n))$ if and only if $g(n)$ is $\Omega(f(n))$.

Proof: Follows directly from the definitions of O and Ω.

This symmetry is interesting but not often useful, because O and Ω are typically used to characterize complicated functions by relating them to simple functions. Reversing the direction is not usually a good idea; it might be mathematically correct to characterize a simple function by relating it to complicated function, but it's rarely helpful.

19.3 Common Asymptotic Functions

If you start with the functions typically encountered when analyzing the resource requirements of computer programs and you apply the simplifications described above, you end up with a kind of standard vocabulary of functions. These functions occur so often in asymptotic analysis that they have special names. Here are some of the most common. (The descriptions below refer to tight bounds, but the same terms are also used for upper and lower bounds.)

Θ(1): *constant*

Here, $g(n) = 1$: a constant function that does not grow as n grows. Theorem 19.1.1 tells us that $\Theta(1)$ is equivalent to $\Theta(k)$ for any constant k. But simpler is better, and the convention is to write $\Theta(1)$ instead of any other constant.

Θ(*log n*): *logarithmic*

In computer science we often use base-2 logarithms, $log_2\ n$; in calculus the natural log is more common, $log_e\ n$. But it turns out that for any two constant bases *a* and *b*, $log_a\ n$ and $log_b\ n$ differ by only a constant factor: $log_a\ n = (log_a\ b)\ log_b\ n$. Theorem 19.1.1 tells us that constant factors can be ignored, so the base of the logarithm is not significant inside asymptotics. The convention is to omit the base altogether and simply write $\Theta(log\ n)$.

Θ(*n*): *linear*

Here, $g(n) = n$. As always, constant factors are ignored. You would never write $\Theta(3n)$, because the equivalent $\Theta(n)$ is simpler.

Θ(*n log n*): *loglinear*

The product of *n* times *log n* arises quite often in the analysis of algorithms. It grows just slightly faster than *n*, but not nearly as fast as n^2.

Θ(n^k): *polynomial*

Important special cases of polynomial include $\Theta(n)$ (*linear*), $\Theta(n^2)$ (*quadratic*), and $\Theta(n^3)$ (*cubic*). Theorem 19.1.2 allows us to drop all but the highest-order term from any polynomial. You never write $\Theta(n^2 + 4n + 2)$, because the equivalent $\Theta(n^2)$ is simpler.

Θ(2^n): *exponential*

It is common to refer to this simply as *exponential*, without mentioning the base. That's actually a bit sloppy, because (unlike the base of logarithms) the base of the exponential is significant asymptotically: $\Theta(3^n)$ is not the same as $\Theta(2^n)$. We'll see a less sloppy definition of exponential time in the next chapter.

The common asymptotic functions above are listed in order of their rates of growth, from the slowest growing to the fastest growing. We state without proof these useful facts about them:

Theorem 19.2:

- For all positive real constants *a* and *b*, *a* is $O((log\ n)^b)$.
- For all positive real constants *a* and *b*, and for any logarithmic base, $(log\ n)^a$ is $O(n^b)$.
- For all real constants $a > 0$ and $c > 1$, n^a is $O(c^n)$.

Note that the theorem covers cases with fractional exponents. For example, $\sqrt{n} = n^{\frac{1}{2}}$ and so, by Theorem 19.2, *log n* is $O(\sqrt{n})$.

19.4 A TM Cost Model

We define the time and space required by a Turing machine M on input x as follows:

$$\begin{aligned}\textit{time}(M, x) &= \text{the number of moves made by } M \text{ on the input } x \in \Sigma^* \\ \textit{space}(M, x) &= \text{the length of tape visited by } M \text{ on the input } x \in \Sigma^*\end{aligned}$$

This cost model is obvious, and it's hard to think of any significantly different alternatives.

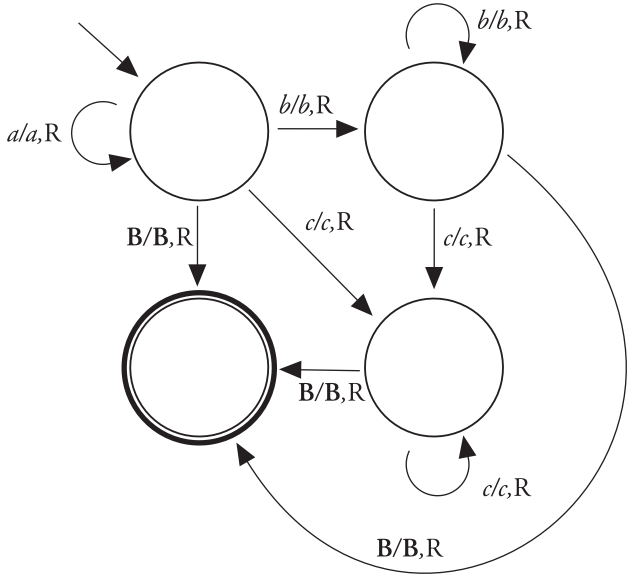
One simplification is to remove specific strings from the picture and concentrate only on the length of the input string. The worst-case cost is one way to do this:

$$\begin{aligned}\textit{worst-case-time}(M, n) &= \text{maximum } \textit{time}(M, x) \text{ of all } x \text{ with } |x| = n \\ \textit{worst-case-space}(M, n) &= \text{maximum } \textit{space}(M, x) \text{ of all } x \text{ with } |x| = n\end{aligned}$$

Similarly, we might also define cost functions that characterize the best case for strings of length n or functions that characterize an average over all strings of length n. Here, we will focus on worst-case cost. It is a widely used measure in practical programming, because it corresponds to the kind of performance guarantee we often want to make: an upper bound on the amount of time or space that a program will require.

Example: Linear Time and Space

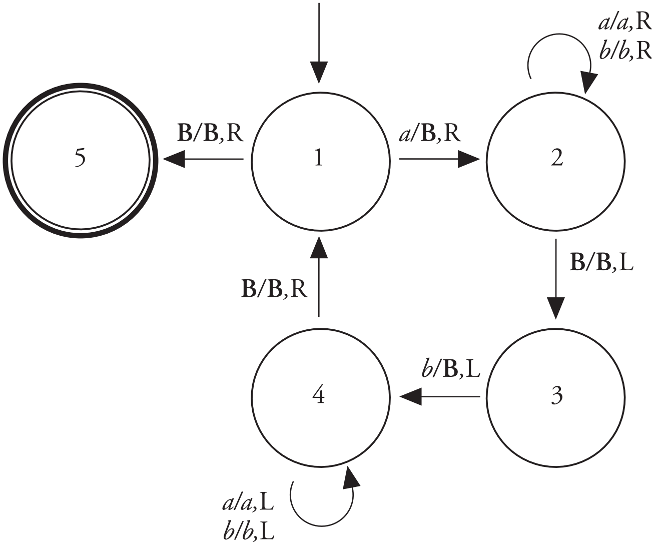
Consider this TM for $L(a^*b^*c^*)$, which was introduced in Chapter 16.



It is easy to see how many moves this machine will make on any $x \in L(M)$, because it always moves right and never gets stuck until after the first **B** at the end of the input. So for $x \in L(M)$, $\textit{time}(M, x) = |x| + 1$ and $\textit{space}(M, x) = |x| + 2$ (including as "visited" the final position of the head, from which the final state has no transition). On the other hand, the machine stops early when $x \notin L(M)$, because it gets stuck before reaching the accepting state. For example, it stops after one move on any input string beginning with *ba*, no matter how long: $time(M, bax) = 1$. We conclude that the worst-case time and space costs are linear: both $worst\text{-}case\text{-}time(M, n)$ and $worst\text{-}case\text{-}space(M, n)$ are $\Theta(n)$.

Example: Quadratic Time, Linear Space

Consider this TM for $\{a^n b^n\}$, which was introduced in Chapter 16.



The machine contains two states with self-edges: 2 and 4. For each state, the self-edges of that state all move the head in the same direction and all transition on a symbol that is not **B**. The machine never writes a non**B** over a **B**, so the number of non**B** symbols on the tape is never greater than the original length m. Therefore, when any state is entered, there will be $O(m)$ moves in the worst case before that state is exited. Aside from the self-edges, the machine has only one cycle: the states 1, 2, 3, and 4. On each iteration of this cycle, it overwrites one *a* and one *b*. It cannot iterate more than $m/2$ times before all the input symbols have been overwritten, so the number of iterations is $O(m)$ in the worst case. In the worst case, then, the machine makes $O(m)$ iterations of the cycle, each of which takes $O(m)$ time. Thus, $worst\text{-}case\text{-}time(M, m)$ is $O(m^2)$. The machine visits the whole part of the tape on which the input is presented, but not beyond that, so $worst\text{-}case\text{-}space(M, m)$ is $\Theta(m)$.

With a little more work it is possible to show that the time bound is tight: $worst\text{-}case\text{-}time(M, m)$ is $\Theta(m^2)$. It is even possible to derive an exact formula for *worst-case-time*, without using asymptotics (see Exercise 8). But finding exact formulas for worst-case time and space isn't easy, even for this relatively simple TM. For larger TMs it becomes dauntingly difficult, and for TMs that are described but not actually constructed it is impossible. Fortunately, it is often possible to give asymptotic bounds on the time and space resources required, based only on an informal description of the machine. For this, asymptotics are more than just a convenience—they are indispensable.

19.5 A Java Cost Model

For higher-level languages there are no cost models as obvious as those for TMs. How should we define *time*(`d`, *x*) and *space*(`d`, *x*) for a Java decision method `d` and input string *x*? This is a surprisingly delicate question. We want to model the time and space costs of a computation realistically, so that our conclusions will usefully predict experimental results on real computers. At the same time, we want our models to be simple and robust, so that our conclusions will not have to be discarded just because we've changed our experimental set-up, for example by installing more memory, a new processor, or a new compiler. Unfortunately, there is an irreducible conflict between these two goals, realism and robustness. The most realistic cost model for time would predict runtime exactly, not in terms of an abstraction like "moves" but in actual physical time, time in seconds. To do that, though, would require an extremely complex definition for *time*(`d`, *x*), accounting for every detail about the particular compiler, processor, memory, and operating system being modeled. At the other extreme, *time*(`d`, *x*) = 1 would be a very robust cost model for time, but it would be completely unrealistic, asserting that every computation takes the same amount of time! Somewhere in between these two extremes is the cost model we're looking for.

Our solution to this conundrum will be to use the asymptotic notation. It was developed long before there were computer programs to analyze, but it turns out to be just the right tool for this purpose. We'll won't try to give those exact, extremely complex definitions of *time*(`d`, *x*) and *space*(`d`, *x*). Instead, we'll state some simple assumptions about them asymptotically, using Θ. That way, we'll be able to reach robust conclusions about time and space requirements, conclusions that will be correct for any implementation that meets our asymptotic assumptions.

For the Java-like language we've been using, a key issue is the implementation of `int` values. Starting in Chapter 17, we have assumed that `int` values are somehow implemented without Java's usual 32-bit limit. This is very important for implementing recognition methods. For one thing, all interesting formal languages involve strings of unbounded length, and to manipulate these we need to have `int` values that can index them. In addition, it is convenient to be able to treat the `int` values themselves as strings being tested for language membership—to be able to write a recognition method that takes an `int` input and defines the language of even numbers, prime numbers, perfect numbers, or whatever.

So we'll assume that `int` values are implemented like `String` values, as references (pointers) to immutable blocks of memory that are allocated whenever a new `int` value is constructed. Thus, passing an `int` as a parameter takes constant time, because only the reference is passed. An `int` assignment, like a `String` assignment, takes constant time, because only the reference is assigned. But modifying an `int`, as in `x = x + 1`, necessitates the construction of a new `int` and takes time proportional to the memory size taken up by the `int`. (This is true even if only one bit of the `int` is modified; `x = x | 1` would still have to make a complete copy of `x`.)

With that in mind, here is a basic asymptotic cost model for our Java-like language:

1. *All values manipulated by a program take space proportional to the length of their representation as Java constants.* This means in particular that the space required by an integer constant is proportional to the number of digits required to represent it—so the space required to represent any $n \in \mathcal{N}$ is $\Theta(log\ n)$.
2. *The time required for any operator is* $\Theta(n)$, *where n is the total size of the data read and written.* For example, the time required to compute $a + 1$ is taken to be $\Theta(log\ a)$.[1]
3. *The overhead of making a method call is* $\Theta(1)$ *in time and space.* We assume that parameter passing, method dispatch, and returning a value take only constant time and that the initial allocation of memory for any method takes only constant space.
4. *Predefined methods have the following resource requirements:*
   - `s.length()` takes $\Theta(1)$ time and space. It just returns (by reference) the stored length of the string.
   - `s.charAt(i)` takes $\Theta(log$ `i`$)$ time and $\Theta(1)$ space. Clearly, it must take *log* `i` time just to look at all of the index value `i`; we assume that it does not need more than that to access the character at index `i`.

We'll state asymptotic assumptions about other predefined methods as we need them. This asymptotic model does not define *time*(`d`, *x*) and *space*(`d`, *x*) exactly. It only says that, whatever their exact definitions, these cost functions satisfy the given assumptions about asymptotic bounds. As we did with TMs, we'll also consider the corresponding worst-case cost functions: *worst-case-time*(`d`, *n*) is the maximum *time*(`d`, *x*) for any $|x| = n$, and likewise for *worst-case-space*(`d`, *n*).

Example: Constant Time and Space

Let's analyze some decision methods using this cost model. Here's one for the language $\{ax \mid x \in \Sigma^*\}$:

```
boolean ax(String p) {
  return (p.length()!=0 && p.charAt(0)=='a');
}
```

Let *n* be the input size, in this case simply *n* = |`p`|. Using our basic cost-model assumptions, we can break down the worst-case time cost like this:

[1] This is not quite realistic for integer multiplication and division. Simple algorithms for multiplication and division of *n*-bit operands require $\Theta(n^2)$ time. Advanced algorithms can improve on this, approaching $\Theta(n)$ time. To keep things simple here, we'll use $\Theta(n)$ for all operators, even including `*`, `/`, and `%`.

```
                                                Time Cost
 1. boolean ax(String p) {                        Θ(1)
 2.   return (                                    Θ(1)
 3.     p.length()                                Θ(1)
 4.       !=                                      Θ(1)
 5.     0                                         Θ(1)
 6.   &&                                          Θ(1)
 7.     p.charAt(0)                               Θ(1)
 8.       ==                                      Θ(1)
 9.     'a');                                     Θ(1)
10. }                                          +__________
                                                  Θ(1)
```

It turns out that the time complexity does not depend on *n*; every operation here takes constant time and is executed at most once. (In general, the `!=` operator takes time proportional to the size of the data it reads; but it is reasonable to assume that the `!=` operator on line 4 does not need not read the whole number to tell whether it is zero.) To get the total *worst-case-time*(`ax`, *n*), we add that column of asymptotics. Using Theorem 19.1.2, this is simply a matter of selecting the fastest-growing asymptotic in the column. The result: *worst-case-time*(`ax`, *n*) is Θ(1).

Turning to the space analysis, we can again consider this line by line.

```
                                                Space Cost
 1. boolean ax(String p) {                        Θ(1)
 2.   return (                                    Θ(1)
 3.     p.length()                                Θ(1)
 4.       !=                                      Θ(1)
 5.     0                                         Θ(1)
 6.   &&                                          Θ(1)
 7.     p.charAt(0)                               Θ(1)
 8.       ==                                      Θ(1)
 9.     'a');                                     Θ(1)
10. }                                          +__________
                                                  Θ(1)
```

The space taken by the string `p` is, of course, Θ(*n*). But in our model of execution this value is passed as a reference—it is not copied into new memory space. Similarly, the space required for `p.length()` on line 3 is Θ(log *n*). But in our model of execution the `length` method just returns, by reference, the stored length of the string, so no new memory space is used. We conclude that *worst-case-space*(`ax`, *n*) is also Θ(1).

Like `ax`, most methods manipulate a large (but constant) number of values that require constant space each. These add up to $\Theta(1)$ space, which is already given as the overhead of calling the method in the first place. So we can simplify the analysis of space complexity by ignoring all these constant-space needs. The analysis can focus instead on expressions that construct new values that require nonconstant space (like `String`, array, or unbounded `int` values) and on expressions that call methods that require nonconstant space. If there are none, as in `ax`, then the space complexity is $\Theta(1)$. In a similar way, the analysis of time complexity can be simplified by ignoring constant-time operations repeated for any constant number of iterations.

Example: Loglinear Time, Logarithmic Space

Here's another example, a decision method for $\{a^n b^n\}$:

```
boolean anbn(String x) {
  if (x.length()%2 != 0) return false;
  for (int i = 0; i < x.length()/2; i++)
    if (x.charAt(i) != 'a') return false;
  for (int i = x.length()/2; i < x.length(); i++)
    if (x.charAt(i) != 'b') return false;
  return true;
}
```

The previous example developed a tight bound on the worst-case complexity directly. This time, it will be more instructive to develop upper and lower bounds separately. First, a lower bound. Clearly, the worst case for this code is when `x` $\in \{a^n b^n\}$, so that all the loops run to completion and none of the early returns is taken. Assuming that `x` $\in \{a^n b^n\}$, we can state a lower bound for the time spent on each line:

		Time Cost	Iterations	Total
1.	`boolean anbn(String x) {`			
2.	`  if (x.length()%2 != 0)`	$\Omega(\log n)$	1	$\Omega(\log n)$
3.	`    return false;`			
4.	`  for (int i = 0;`			
5.	`       i < x.length()/2;`	$\Omega(\log n)$	$\Omega(n)$	$\Omega(n \log n)$
6.	`       i++)`	$\Omega(1)$	$\Omega(n)$	$\Omega(n)$
7.	`    if (x.charAt(i) != 'a')`	$\Omega(1)$	$\Omega(n)$	$\Omega(n)$
8.	`      return false;`			
9.	`  for (int i = x.length()/2;`	$\Omega(\log n)$	1	$\Omega(\log n)$
10.	`       i < x.length();`	$\Omega(\log n)$	$\Omega(n)$	$\Omega(n \log n)$
11.	`       i++)`	$\Omega(\log n)$	$\Omega(n)$	$\Omega(n \log n)$

```
12.       if (x.charAt(i) != 'b')        Ω(log n)     Ω(n)     Ω(n log n)
13.         return false;
14.     return true;
15.   }                                                       +_________
—
                                                               Ω(n log n)
```

The “Total” column here shows the total time spent on each line. In this case, it is computed for each line by simple multiplication: the minimum time for one iteration of the line multiplied by the minimum number of iterations of that line. This technique is very useful, but it does have its limitations. Consider, for example, lines 6 and 7. Because `i` can be as low as 0 there, we can't do better than Ω(1) as a lower bound on the time spent for a single iteration of those lines. (For the similar lines 11 and 12 we can claim the stronger Ω(*log n*) lower bound, because there we know that `i` ≥ *n*/2.) Of course, `i` doesn't stay 0—and the total time for lines 6 and 7 is more than just the minimum time for any iteration times the number of iterations. With a more detailed analysis we could get a stronger total lower bound for these lines. But in this case there is no need. Adding up the “Total” column is simply a matter of choosing the fastest growing function in the column (as Theorem 19.1.2 showed), and that is Ω(*n log n*). This turns out to be the strongest lower bound possible.

We know it is the strongest lower bound possible, because it matches the upper bound:

```
                                        Time Cost   Iterations     Total
 1.  boolean anbn(String x) {
 2.    if (x.length()%2 != 0)           O(log n)        1        O(log n)
 3.      return false;
 4.    for (int i = 0;
 5.         i < x.length()/2;           O(log n)      O(n)      O(n log n)
 6.         i++)                        O(log n)      O(n)      O(n log n)
 7.      if (x.charAt(i) != 'a')        O(log n)      O(n)      O(n log n)
 8.        return false;
 9.    for (int i = x.length()/2;       O(log n)        1        O(log n)
10.         i < x.length();             O(log n)      O(n)      O(n log n)
11.         i++)                        O(log n)      O(n)      O(n log n)
12.      if (x.charAt(i) != 'b')        O(log n)      O(n)      O(n log n)
13.        return false;
14.    return true;
15.  }                                                          +_________
—
                                                                O(n log n)
```

The upper bounds for lines involving `i` take advantage of the fact that `i` ≤ *n*. The total upper bound is O(*n log n*) time. This agrees with the lower bound, so we conclude that *worst-case-time*(`anbn`, *n*) is Θ(*n log n*).

This may come as a surprise. Clearly `anbn(x)` must, in the worst case, examine every symbol of `x`, so we expect it to require Ω(*n*) time. But where did that extra log factor come from? In this case, our cost model makes it unavoidable; accessing the `i`th character of a string takes time proportional to *log* `i`. If we introduced some way of iterating over the characters in a `String` without actually indexing each one, we might eliminate that extra log factor. (For example, we might model an implementation of strings as linked lists of characters.) But as it stands, we're stuck with it. It might be worse; don't forget that for the Turing machine in the previous section, deciding this language took Θ(n^2) time.

Turning to the space analysis, we can again examine this line by line. As already noted, constant-space needs can be ignored, because they are subsumed by the constant-space overhead of every method call. Also, the number of times each line is iterated is unimportant, because the memory used by a statement on one iteration can be reused on the next. (If only time could be reused the way space can!)

```
                                                Space Cost
 1. boolean anbn(String x) {
 2.   if (x.length()%2 != 0)
 3.     return false;
 4.   for (int i = 0;
 5.        i < x.length()/2;                     Θ(log n)
 6.        i++)                                  Θ(log n)
 7.     if (x.charAt(i) != 'a')
 8.       return false;
 9.   for (int i = x.length()/2;                 Θ(log n)
10.        i < x.length();
11.        i++)                                  Θ(log n)
12.     if (x.charAt(i) != 'b')
13.       return false;
14.   return true;
15. }                                          +__________
                                                 Θ(log n)
```

The values constructed at lines 5 and 9 are the same: *n*/2. The largest value constructed at line 6 is *n*/2, and the largest value constructed at line 11 is *n*. These all take Θ(*log n*) space. Thus the total space requirement, *worst-case-space*(`anbn`, *n*), is Θ(*log n*).

Example: Quadratic Time, Linear Space

Consider this method:

```
String matchingAs(String s) {
  String result = "";
  for (int i = 0; i < s.length(); i++)
    r += 'a';
  return r;
}
```

A line-by-line analysis of lower bounds on the worst-case time gives

		Time Cost	Iterations	Total
1.	`String matchingAs(String s) {`			
2.	`  String result = "";`			
3.	`  for (int i = 0;`			
4.	`       i < s.length();`	$\Omega(log\ n)$	$\Omega(n)$	$\Omega(n\ log\ n)$
5.	`       i++)`	$\Omega(1)$	$\Omega(n)$	$\Omega(n)$
6.	`    r += 'a';`	$\Omega(1)$	$\Omega(n)$	$\Omega(n)$
7.	`  return r;`			
8.	`}`			+_________
	—			
				$\Omega(n\ log\ n)$

As in the previous example, this produces some rather weak lower bounds. In this case, the weakness is significant. For upper bounds, we get

		Time Cost	Iterations	Total
1.	`String matchingAs(String s) {`			
2.	`  String result = "";`			
3.	`  for (int i = 0;`			
4.	`       i < s.length();`	$O(log\ n)$	$O(n)$	$O(n\ log\ n)$
5.	`       i++)`	$O(log\ n)$	$O(n)$	$O(n\ log\ n)$
6.	`    r += 'a';`	$O(n)$	$O(n)$	$O(n^2)$
7.	`  return r;`			
8.	`}`			+_________
	—			
				$O(n^2)$

The upper bounds for lines involving `i` and `r` take advantage of the fact that $i \le n$ and $|r| \le n$. The total upper bound is $O(n^2)$ time. This does not match our lower bound of $\Omega(n \log n)$.

It is usually desirable to refine an analysis until the upper and lower bounds meet. For subtle algorithms this can be very difficult. Sometimes, it isn't even clear whether to try refining the lower bound upward, the upper bound downward, or both. But in this case there is an obvious suspect: the weak analysis of the lower bound at line 6. The length of the string `r` there is initially 0, so for a single iteration of line 6 we can't state a stronger lower bound than $\Omega(1)$. But `r` doesn't stay that short, and the total time for line 6 is more than just the minimum time for any iteration times the number of iterations. Let $t(k)$ be the time taken for an iteration of line 6 when the length of the string contructed is k. Then the total worst-case time at line 6, for an input of length n, is

$$g(n)=\sum_{k=1}^{n} t(k)$$

where $t(k)$ is $\Omega(k)$. Using the definition of Ω, we know that for all sufficiently large n, there is some c for which

$$g(n)=\sum_{k=1}^{n} t(k) \ge \sum_{k=1}^{n} \frac{1}{c}\, k = \frac{1}{c}\sum_{k=1}^{n} k = \frac{1}{c} \cdot \frac{n^2+n}{2}$$

It follows that $g(n)$, the total worst-case time at line 6, is $\Omega(n^2)$. That gives a total lower bound of $\Omega(n^2)$. That matches the upper bound, so *worst-case-space*(`matchingAs`, n) is $\Theta(n^2)$.

The worst-case space analysis is more straightforward:

```
                                            Space Cost
1.  String matchingAs(String s) {
2.    String result = "";
3.    for (int i = 0;
4.         i < s.length();
5.         i++)                              Θ(log n)
6.      r += 'a';                             Θ(n)
7.    return r;
8.  }                                     +___________
                                              Θ(n)
```

The largest value constructed at line 6 has the same length as the input string `s`.

Example: Exponential Time, Linear Space

Consider this decision method for the set of prime numbers:

```
boolean isPrime(int x) {
  if (x < 2) return false;
  for (int i = 2; i < x; i++)
    if (x % i == 0) return false;
  return true;
}
```

The analysis must be carried out in terms of the input size n, which is the amount of space required to represent the input integer x. The relation between n and x is determined by the representation. If we're using decimal, n is roughly $log_{10}\, x$; if we're using binary, it's roughly $log_2\, x$, and so on. Let's just say that for some constant base b, x is $\Theta(b^n)$. Then we have this for an upper bound on the worst-case time:

		Time Cost	Iterations	Total
1.	`boolean isPrime(int x) {`			
2.	`  if (x < 2) return false;`	$O(n)$	$O(1)$	$O(n)$
3.	`  for (int i = 2;`			
4.	`       i < x;`	$O(n)$	$O(b^n)$	$O(nb^n)$
5.	`       i++)`	$O(n)$	$O(b^n)$	$O(nb^n)$
6.	`    if (x % i`	$O(n)$	$O(b^n)$	$O(nb^n)$
7.	`          ==`	$O(1)$	$O(b^n)$	$O(b^n)$
8.	`        0)`	$O(1)$	$O(b^n)$	$O(b^n)$
9.	`      return false;`			
10.	`  return true;`			
11.	`}`			+ ________
__				$O(nb^n)$

An analysis of the lower bounds, which we omit, gives a weaker result for line 5 but the same total, $\Omega(nb^n)$. Thus, *worst-case-time*(`isPrime`, n) is $\Theta(nb^n)$: exponential time.

If we performed our analysis in terms of the *value* of x, instead of the length of the string representing x, we would get a much less intimidating result: *worst-case-time*(`isPrime`, x) is $O(x\ log\ x)$. But in the study of formal language, we always treat complexity as a function of the size of the input string, whether it's a string of letters or, as in this case, a string of digits.

Turning to the space complexity, we have:

```
                                                   Space Cost
1.   boolean isPrime(int x) {
2.     if (x < 2) return false;
3.     for (int i = 2;
4.          i < x;
5.          i++)                                     Θ(n)
6.       if (x % i                                   Θ(n)
7.             ==
8.           0)
9.         return false;
10.    return true;
11.  }                                            +__________
                                                     Θ(n)
```

The largest value constructed at line 5 is x, and the largest remainder at line 6 is $\lceil x/2 \rceil$-1. Thus the total space requirement, *worst-case-space*(isPrime, n), is $\Theta(n)$.

Exercises

EXERCISE 1

True or false:

a. $2n$ is $O(n)$
b. $2n$ is $\Omega(n)$
c. $2n$ is $\Theta(n)$
d. $3n^2 + 3n$ is $O(n^3)$
e. $3n^2 + 3n$ is $\Omega(n^3)$
f. $3n^2 + 3n$ is $\Theta(n^3)$
g. n is $O(2n)$
h. n^2 is $\Omega(n^2 + 2n + 1)$
i. $\log n^2$ is $\Omega(\log n^3)$
j. n^2 is $O(n (\log n)^2)$
k. 2^n is $O(2^{2^n})$
l. 3^n is $\Omega(5^n)$

EXERCISE 2

Simplify each of the following as much as possible. For example, $O(3a^2 + 2a)$ simplifies to $O(a^2)$. (*Hint:* A few are already as simple as possible.)

a. $O(5)$
b. $O(\log_3 a)$

c. $O(\mathit{log}\ a + 1)$
d. $O(\mathit{log}\ a^2)$
e. $O((\mathit{log}\ a)^2)$
f. $\Omega(a + \mathit{log}\ a)$
g. $\Omega(\mathit{log}\ a + 1)$
h. $\Omega(\max(a^2, a))$
i. $\Omega(2^n + n^{100})$
j. $\Theta(a^2 + 2a)$
k. $\Theta(3n^2 + 2n + \mathit{log}\ n + 1)$
l. $\Theta(n2^n)$
m. $\Theta(3^n + 2^n)$

EXERCISE 3

Using the definitions only, without using the simplification theorems proved in Section 19.2, prove the following:

a. $n^2 + 5$ is $O(n^2)$
b. $n^3 + 2n^2$ is $\Omega(n)$
c. $a^2 + 2a + 12$ is $\Theta(a^2)$

EXERCISE 4

Give a detailed proof of all three cases for Theorem 19.1.3.

EXERCISE 5

Give a detailed proof of Theorem 19.1.4.

EXERCISE 6

Give a detailed proof of Theorem 19.1.1, for the Ω and Θ versions.

EXERCISE 7

Give a detailed proof of Theorem 19.1.2, for the Ω and Θ versions.

EXERCISE 8

These questions concern the TM M for the language $L(M) = \{a^n b^n\}$, discussed in Section 19.4.

a. Derive an exact formula for $\mathit{time}(M, a^n b^n)$, and show that it is $\Theta(n^2)$.
b. Show that $\mathit{space}(M, a^n b^n)$ is $\Theta(n)$.
c. Show that $\mathit{worst\text{-}case\text{-}space}(M, m)$ is $O(m)$, and conclude (using the result from Part b) that it is $\Theta(m)$.
d. Show that $\mathit{worst\text{-}case\text{-}time}(M, m)$ is $O(m^2)$, and conclude (using the result from Part a) that it is $\Theta(m^2)$. (Note that this m is the total length of the string; your proof must encompass strings that are not in $L(M)$.)

EXERCISE 9

These questions concern the TM from Section 16.3 (for the language $\{a^n b^n c^n\}$).

a. Show that *worst-case-space*(M, m) is $O(m)$.
b. Show that *worst-case-space*(M, m) is $\Omega(m)$, and conclude that it is $\Theta(m)$.
c. Show that *worst-case-time*(M, m) is $O(m^2)$.
d. Show that *worst-case-time*(M, m) is $\Omega(m^2)$, and conclude that it is $\Theta(m^2)$.

EXERCISE 10

These questions concern the TM from Section 16.7 (for the language $\{xcx \mid x \in \{a, b\}^*\}$).

a. Show that *worst-case-space*(M, n) is $O(n)$.
b. Show that *worst-case-space*(M, n) is $\Omega(n)$, and conclude that it is $\Theta(n)$.
c. Show that *worst-case-time*(M, n) is $O(n^2)$.
d. Show that *worst-case-time*(M, n) is $\Omega(n^2)$, and conclude that it is $\Theta(n^2)$.

EXERCISE 11

These questions concern the TM from Section 17.2 (the *stacker* machine). Assume that all input strings are in the required form $x\#y$, where $x \in \{0, 1\}^*$ and $y \in \{0, 1\}^*$.

a. Show that *space*$(M, x\#y)$ is $\Theta(|x\#y|)$, and thus *worst-case-space*$(M, n) = \Theta(n)$.
b. Derive a Θ bound on *worst-case-time*(M, n).
c. Derive a Θ bound on *best-case-time*(M, n).

EXERCISE 12

These questions concern the TM from Section 17.4 (the *ith* machine). Assume the input string is in the required form.

a. Show that *worst-case-time*(M, n) is $\Omega(n2^n)$.
b. Show that *worst-case-space*(M, n) is $\Omega(2^n)$.

EXERCISE 13

Derive tight asymptotic bounds on the worst-case time and space used by this `contains` method. Show enough detail to make a convincing case that your bounds are correct.

```
boolean contains(String s, char c) {
  for (int i = 0; i < s.length(); i++)
    if (s.charAt(i)==c) return true;
  return false;
}
```

EXERCISE 14

Derive tight asymptotic bounds on the worst-case time and space used by this decision method for the language $\{xx^R\}$. Show enough detail to make a convincing case that your bounds are correct.

```
boolean xxR(String s) {
  if (s.length() % 2 != 0) return false;
  for (int i = 0; i < s.length() / 2; i++)
    if (s.charAt(i) != s.charAt(s.length()-i-1))
      return false;
  return true;
}
```

EXERCISE 15

a. Let $g(n)$ be a function with

$$g(n)=\sum_{k=1}^{n} t(k)$$

where $t(k)$ is $\Theta(\log k)$. Show that $g(n)$ is $\Theta(n \log n)$. (*Hint:* Use this asymptotic version of Stirling's approximation: $\log n!$ is $\Theta(n \log n)$.)

b. Using that result, derive a tight asymptotic bound on the worst-case runtime of the following method. Show your derivation for the upper bound first, then your derivation for the matching lower bound.

```
boolean containsC(String s) {
  int i = s.length();
  while (i > 0) {
    i--;
    if (s.charAt(i)=='c') return true;
  }
  return false;
}
```

EXERCISE 16

Derive an asymptotic tight bound on the worst-case space used by this `fact` method, and then derive an asymptotic upper bound on the worst-case time. (*Hint:* Use the result of Exercise 15, Part a.) Make sure your results are expressed in terms of the length of the input, not in terms of its integer value. Show enough detail to make a convincing case that your bounds are correct.

```
int fact(int p) {
  int sofar = 1;
  while (n > 0) sofar *= p--;
  return sofar;
}
```

20

Deterministic Complexity Classes

Have you ever thought about what you would do if you had all the money in the world? It's fun to imagine. It can also be instructive, when you realize that some of the things you want, money can't buy. When we considered questions of computability in previous chapters, we were asking the same kind of question: "What languages can be decided, with unlimited time and space?" Of course, we never have unlimited time and space, but it's fun to think about. It can also be instructive, when you realize that some of the languages you want to decide can't be decided, no matter how much time and space you have to spend.

Now we turn to a different, more pragmatic kind of question: what languages can be decided on a limited budget? We'll use the worst-case-time and worst-case-space measures developed in the previous chapter to classify languages according to the resource budgets required to decide them.

20.1 Time Complexity Classes

Any function $g(n)$ over $\mathcal{N}$ can serve as an asymptotic upper bound on runtime and so can be used to define a class of formal languages:

$$\textit{TIME}(g(n)) = \{L \mid L \text{ is decided by some decision method } \texttt{p}, \text{ where } \textit{worst-case-time}(\texttt{p}, n) \text{ is } O(g(n))\}.$$

The resulting class of languages is called a *complexity class*: languages classified according to the resources required to decide them.

The previous chapter analyzed the worst-case time requirements of individual methods. Now, we are considering the time requirements of whole languages—of all possible decision methods for a given language. To prove that a language L is in $\textit{TIME}(g(n))$ can be straightforward. We need only give an example of a decision method `p`, show that it decides L, and show (using the techniques of the previous chapter) that its worst-case time is $O(g(n))$. For example, in the previous chapter we analyzed a decision method `anbn` for the language $\{a^nb^n\}$. We showed that *worst-case-time*(`anbn`, n) is $O(n \log n)$. That proves that the language can be decided in $O(n \log n)$ time:

$$\{a^nb^n\} \in \textit{TIME}(n \log n).$$

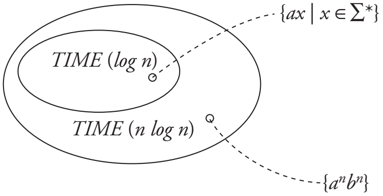
Since the language can be decided within that time budget, it can certainly also be decided within any more generous budget, so we can also conclude that $\{a^nb^n\} \in \textit{TIME}(n^2)$, $\{a^nb^n\} \in \textit{TIME}(n^3)$, and so on. In general, if $f(n)$ is $O(g(n))$, then $\textit{TIME}(f(n)) \subseteq \textit{TIME}(g(n))$.

On the other hand, to prove that a language L is *not* in $\textit{TIME}(g(n))$ is usually much more difficult. It isn't enough to say that, although you tried really hard, you just couldn't think of an $O(g(n))$ decision method. You have to prove that no such method can exist. Here's an example:

Theorem 20.1: $\{a^nb^n\} \notin \textit{TIME}(\log n)$.

Proof: Suppose by way of contradiction that $\{a^nb^n\}$ has some decision method `p` for which *worst-case-time*(`p`, n) is $O(\log n)$. This is not enough time even to access every symbol in the input string, which would take $\Omega(n)$ time. So it must be true that, for some some n_0, `p` leaves at least one symbol uninspected in every input string x with $|x| \geq n_0$. In particular, there must be at least one symbol in the string $a^{n_0}b^{n_0}$ that `p` does not examine. Because it never examines that symbol, `p` must execute identically on a mutated version of $a^{n_0}b^{n_0}$ with the unexamined symbol flipped (from a to b or from b to a). Thus `p` must accept both $a^{n_0}b^{n_0}$ and the mutated version of $a^{n_0}b^{n_0}$. But the mutated version of $a^{n_0}b^{n_0}$ is not in $\{a^nb^n\}$. So $L(\texttt{p}) \neq \{a^nb^n\}$, contradicting our assumption. By contradiction, no such decision method `p` exists, and $\{a^nb^n\} \notin \textit{TIME}(\log n)$.

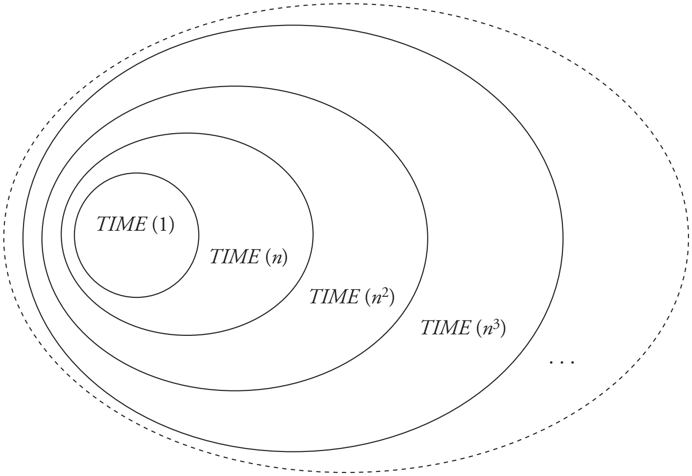
We have now established the existence of two distinct complexity classes. The class *TIME*($\log n$) is not empty; we've seen that it includes the language $\{ax \mid x \in \Sigma^*\}$. *TIME*($n \log n$) is not the same as *TIME*($\log n$); we've seen that it includes the language $\{a^n b^n\}$, which is not in the smaller class. So we have



It is not surprising to observe that the more time you have for the decision, the more languages you can decide.

A *time-hierarchy theorem* states this observation more rigorously. We know that if $f(n)$ is $O(g(n))$, then *TIME*($f(n)$) $\subseteq$ *TIME*($g(n)$). A time-hierarchy theorem establishes some additional conditions on f and g that allow you to conclude that the inclusion is strict—that *TIME*($f(n)$) $\subset$ *TIME*($g(n)$). It's a very useful kind of theorem: it often allows you to conclude that two complexity classes are distinct, without the need to find and prove special theorems like Theorem 20.1, above.

In fact, our time-hierarchy theorem automatically establishes an infinite hierarchy of complexity classes. For example, it can be used to show that if $a < b$ then *TIME*(n^a) $\subset$ *TIME*(n^b), producing an infinite hierarchy of polynomial complexity classes:



This polynomial time hierarchy is just one simple consequence of our time-hierarchy theorem. Roughly speaking, our time-hierarchy theorem says that anything more than an extra log factor is enough to decide additional languages. So there is actually a far richer structure of time-complexity classes than those pictured above. For example, there is an infinite hierarchy of distinct classes like $\mathit{TIME}(n \log^2 n)$ and $\mathit{TIME}(n \log^4 n)$ that all lie inside the diagram above, between $\mathit{TIME}(n)$ and $\mathit{TIME}(n^2)$.

Unfortunately, this rich and powerful theorem is rather difficult to state and prove in detail. Interested readers should consult Appendix B.

20.2 Polynomial Time

We have seen the notion of Turing equivalence: many different models of computation, from lambda calculus to modern programming languages, turn out to be interconvertible with Turing machines. The concepts of computability are robust with respect to such conversions. The class of recursive languages and the class of RE languages do not change when the underlying computational model is changed. By contrast, some of the concepts of complexity are more fragile.

For example, $\mathit{TIME}(f(n))$ defined using Java is not generally the same as $\mathit{TIME}(f(n))$ defined using TMs. It isn't hard to see why. Some operations (like `s.charAt(i)`) are cheaper under our cost model for Java, and other operations (like sequential access to all the symbols of the input string) are cheaper for TMs. The good news is that conversions among the standard models of computation produce only a polynomial change in runtime. For example, a TM can simulate a digital processor in such a way that the number of TM moves is some polynomial function of the number of processor clock cycles. So any language that can be decided in polynomial time, measured in processor clock cycles, can also be decided in polynomial time measured in TM moves—though not, generally, the same polynomial. If you lump all the polynomial-time complexity classes together, you get a combined class that is robust with respect to the underlying model of computation. *Polynomial time* on a Turing machine is the same as *polynomial time* in Java. This robust class of polynomial-time-decidable languages is called P:

$$P = \bigcup_k \mathit{TIME}(n^k)$$

The robustness of P allows decision problems to be described more naturally and informally. Usually, we omit any mention of the model of computation, with the understanding that we are looking for an algorithm that could be implemented for any of the standard Turing-equivalent models. Likewise, the exact encoding of the decision algorithm's input can often be omitted. For example, we used a lot of ink in Chapter 16 showing exactly how to encode a DFA as a string over the alphabet {0, 1}, then sketching how to make a TM to simulate that DFA. Sometimes that level of detail is necessary, but if we're only interested in knowing whether the language is in P or not it may be preferable to describe the decision problem more informally, like this:

<u>*DFA-acceptance*</u>

Instance: a DFA *M* and a string *x*.
Question: does *M* accept *x*?

To show that *DFA-acceptance* can be solved in polynomial time, it suffices to sketch an algorithm, using any reasonable model of computation:

```
boolean dfaAcceptance(DFA M, String x) {
  State s = M.startState();
  for (int i = 0; i < x.length(); i++)
    s = M.delta(s,x.charAt(i));
  return s.isAccepting();
}
```

This is too high-level to satisfy our definition of a decision method; in particular, it takes something other than a string as input, and it suppresses the details of how a DFA could be represented as a string. Justifiably so: any competent programmer can imagine implementations for the classes `DFA` and `State`, implementations in which every method runs in polynomial time. Given such an implementation, the whole decision method clearly takes polynomial time. In fact, it would probably have been enough merely to describe the algorithm in English: "Simulate *M* on *x*, making one move for each symbol, and return true if and only if *M* ends in an accepting state." With this algorithm in mind, we conclude that DFA-acceptance is in P—an informal way of saying that, for any reasonable string encoding and model of computation,

$$\{x \mid x \text{ encodes a DFA } M \text{ and a string } x \text{ with } x \in L(M)\} \in P.$$

Informal descriptions are more readable because they suppress details. They are also more susceptible to error, for the same reason. When you read or write these descriptions, you should be vigilant, carefully thinking about how you might represent each problem instance as a string. For example, consider this decision problem:

<u>*Primality*</u>

Instance: a number $x \in \mathcal{N}$.
Question: is *x* prime?

How is the natural number *x* represented? The informal description doesn't say, but it is conventional to assume that natural numbers are represented compactly, using a string of numerals with a positional notation like binary or decimal. That's like our cost model for Java. In fact, we've already seen a Java decision method for this problem:

```
boolean isPrime(int x) {
  if (x < 2) return false;
  for (int i = 2; i < x; i++)
    if (x % i == 0) return false;
  return true;
}
```

As we showed in Section 19.5, this takes exponential time—$\Theta(nb^n)$, if integers are represented in base b and n is the length of the string of base b numerals used to represent x. That is *not* polynomial time; when $b > 1$, b^n is not $O(n^k)$ for any fixed k. So this decision method does not show that *Primality* is in *P*. (It turns out that it *is* in *P*, but proving it requires a much more subtle algorithm. See the Further Reading at the end of this chapter.)

P includes some of the major classes we've already studied:

Theorem 20.2: $\{L \mid L \text{ is context free}\} \subset P$.

Proof: Polynomial-time parsing algorithms for CFGs, like the CKY algorithm (see Section 15.3), are well known. But not all languages in *P* are context free; for example, it is easy to decide $\{a^n b^n c^n\}$ in polynomial time.

Because *P* includes all context-free languages, we know it includes all regular languages. Because it includes all regular languages, we know it includes all finite languages—a fact we'll make use of in the next section.

People sometimes characterize the complexity class *P* as the set of *tractable* decision problems. That's probably an overstatement, because almost all the algorithms used in real computer systems have not just polynomial complexity, but *very low* polynomial complexity. For most applications that work with large data-sets, useful algorithms can't take much more than linear time, and it's hard to think of practical examples in which problems requiring even, say, $O(n^4)$ time are considered tractable. (For that matter, it's hard to think of any commercial setting in which the constant factor hidden by the big-O notation is actually immaterial. If you slow down a program by a factor of ten, customers don't say, "Who cares? It's only a constant factor!") *P* certainly includes all the tractable decision problems, but also includes many whose decision computations are, by most everyday measures, prohibitively expensive.

But if in practice the word *tractable* isn't a perfect description for the inside of *P*, the word *intractable* certainly seems like a good description for the outside of *P*. That's another reason for the importance of this class. Languages that are recursive, yet beyond the polynomial-time boundary, have an interesting intermediate status. The time required to decide such a language grows more rapidly than n^4, more rapidly than n^{4000}, more rapidly than n^k for any fixed value of k. These are languages that can be decided in theory but not in practice;

languages whose computation we can specify but cannot afford to carry out, except on the very shortest input strings.

20.3 Exponential Time

The time-hierarchy theorem guarantees the existence of languages beyond the border of *P*. For example, it can be used to prove that $P \subset TIME(2^n)$. This class $TIME(2^n)$, like the individual, polynomial-time complexity classes, is not robust with respect to changes in the underlying computational model; what takes $O(2^n)$ time with one model might take $O(2^{n^2})$ time with another. To make a robust, exponential-time complexity class, we define

$$EXPTIME = \bigcup_k TIME(2^{n^k})$$

This is what it means to say that a language requires *exponential time* to decide: the decision algorithm takes time proportional to some constant base raised to a power that is a polynomial function of the length of the input string.

The time-hierarchy theorem demonstrates that two classes are distinct by constructing a language that is provable in one but not in the other. In this way, it shows that there is at least one language in *EXPTIME* that is not in *P*. But the language it constructs is highly artificial—it's provably hard to decide, but it isn't clear why you would ever want to decide it in the first place! Luckily, we don't have to rely on the time-hierarchy theorem to illustrate *EXPTIME*. In fact, we've already seen one example of a language that is in *EXPTIME* but not in *P*. It's the language decided by our `runWithTimeLimit` method. Recall that we defined `runWithTimeLimit(p,in,j)` so that it returns true if and only if the recognition method `p` returns true for the input string `in` within `j` steps. As an informal decision problem, this is

Run-with-time-limit
Instance: a recognition method `p`, a string `in`, and a natural number `j`.
Question: does `p` accept `in` within `j` moves?

This decision problem can be solved in exponential time, as shown in the following theorem:

Theorem 20.3.1: *Run-with-time-limit* is in *EXPTIME*.

Proof: Our decision method `runWithTimeLimit` is described in more detail in Appendix B, which shows that evaluating `runWithTimeLimit(p,in,j)` takes Θ(`j` *log* `j`) time. This is log-linear in `j`—but of course the magnitude of `j` is exponential in the size of the problem instance. Restating this in terms of the size *n* of the problem instance, `runWithTimeLimit` takes $O(nb^n)$ time. This is still within *EXPTIME*, however, so we conclude that *Run-with-time-limit* is in *EXPTIME*.

We can also prove that the decision problem cannot be solved using polynomial time:

Theorem 20.3.2: *Run-with-time-limit* is not in *P*.

Proof: Suppose by way of contradiction that *Run-with-time-limit* is in *P*. Then there is some decision method `rWTL` that does what `runWithTimeLimit` does, but in polynomial time. We will show that `rWTL` could be used to decide any *EXPTIME* language in polynomial time.

Let L be any language in *EXPTIME*. By definition there exists some decision method string `Ldec` that decides L, with some exponent k, so that `Ldec`'s worst-case time on inputs of length n is $O(2^{n^k})$. This means that there exist some c and n_0 for which, for all strings x with $|x| \geq n_0$, `Ldec` decides x within $b = c2^{n^k}$ steps. The set of strings $x \in L$ with $|x| < n_0$ is finite, and therefore in *P*. Therefore, there exists a polynomial-time decision method `LdecShort` for these strings.

Using these definitions for `k`, `c`, `n0`, and `LdecShort`, the following method `LdecPoly` decides L:

```
1      boolean LdecPoly(String x) {
2        int n = x.length();
3        if (n < n0) return LdecShort(x);
4        int b = c * power(2,power(n,k));
5        return rWTL(Ldec,x,b);
6      }
```

Line 3, using `LdecShort`, takes polynomial time. The exponentiations at line 4 can be done in polynomial time (this is left as an exercise). The `rWTL` method called at line 5 takes polynomial time. Therefore `LdecPoly` takes polynomial time, and it follows that $L \in$ *P*. Because L is an arbitrary language in *EXPTIME*, we must conclude that *EXPTIME* $\subseteq$ *P*. But this contradicts the hierarchy theorem. By contradiction, *Run-with-time-limit* is not in *P*.

This proof uses the technique of reduction, which we first encountered in Chapter 18. In that chapter it didn't matter how efficient the reduction was, as long as it was computable. The proof above is different; it depends critically on the fact that the reduction itself requires only polynomial time. Lines 2 through 4 accomplish the reduction; you can think of them as defining a function f that converts any instance of L into an equivalent instance of *Run-with-time-limit*. Such reductions are very useful for proving things about complexity classes; they even have their own special notation.

> $A \leq_{\mathrm{P}} B$ (read as "A is polynomial-time reducible to B") if there exists some polynomial-time computable function f such that for all strings x over the alphabet of A, $x \in A$ if and only if $f(x) \in B$.

Our proof of Theorem 20.3.2 shows that for all languages $L \in$ EXPTIME, $L \leq_{\mathrm{P}}$ *Run-with-time-limit*. In that sense, *Run-with-time-limit* is at least as hard as any EXPTIME problem. There is a term for languages that have this property:

> A is EXPTIME-hard if, for all $L \in$ EXPTIME, $L \leq_{\mathrm{P}} A$.

As the proof of Theorem 20.3.2 shows, *Run-with-time-limit* is EXPTIME-hard. If any EXPTIME-hard problem were found to be in P, we could use the polynomial-time reduction to show that all EXPTIME languages are in P. That would contradict the hierarchy theorem, which guarantees that not all EXPTIME problems are in P. Conclusion: If you know that a language is EXPTIME-hard, you can conclude that it is not in P.

Combining the results of Theorems 20.3.1 and 20.3.2, we see that *Run-with-time-limit* is in EXPTIME and is at least as hard as any other problem in EXPTIME. There is a term for languages that have both of these two properties:

> A is EXPTIME-complete if A is in EXPTIME and A is EXPTIME-hard.

EXPTIME-complete problems are, in a sense, the hardest problems in EXPTIME. The proofs of Theorems 20.3.1 and 20.3.2 together show that *Run-with-time-limit* is EXPTIME-complete.

Beyond EXPTIME lie other, still more expensive, complexity classes:

$$2EXPTIME = \bigcup_k TIME\left(b = c2^{2^{n^k}}\right) \quad \text{(doubly exponential time)}$$

$$3EXPTIME = \bigcup_k TIME\left(b = c2^{2^{2^{n^k}}}\right) \quad \text{(triply exponential time)}$$

and so on. The time-hierarchy theorem guarantees that each of these classes is occupied by languages that cannot be decided in less time. There's even a union of all the classes in this exponential-time hierarchy:

$$ELEMENTARY\ TIME = \bigcup_n nEXPTIME$$

We'll look at some examples of languages in these super-high-complexity classes in the last section of this chapter.

20.4 Space-Complexity Classes

Up to this point we have dealt with time complexity; the whole topic of complexity classes can be revisited with a focus on space complexity instead. Any $g(n)$ can serve as an asymptotic upper bound on space, and so can define another class of formal languages:

$$\mathit{SPACE}(g(n)) = \{L \mid L \text{ is decided by some decision method } \mathtt{p}, \text{ where } \mathit{worst\text{-}case\text{-}space}(\mathtt{p}, n) \text{ is } O(g(n))\}.$$

Basic results for space complexity parallel those for time complexity, so we'll just summarize them here.

- There is a separate hierarchy theorem for space. From basic definitions we know that if $f(n)$ is $O(g(n))$, then $\mathit{SPACE}(f(n)) \subseteq \mathit{SPACE}(g(n))$. The space hierarchy theorem establishes some additional conditions on f and g that allow you to conclude that the inclusion is strict—that $\mathit{SPACE}(f(n)) \subset \mathit{SPACE}(g(n))$.
- The space-hierarchy theorem is strong enough to show, for example, that if $a < b$ then $\mathit{SPACE}(n^a) \subset \mathit{SPACE}(n^b)$, producing an infinite hierarchy of polynomial-complexity classes for space.
- Corresponding to the time-complexity class *P* (polynomial time) there is a space-complexity class *PSPACE* (polynomial space) that is robust with respect to changes in the underlying computational model:

  $$\mathit{PSPACE} = \bigcup_k \mathit{SPACE}(n^k)$$
- Corresponding to the time-complexity class *EXPTIME* (exponential time) there is a space-complexity class *EXPSPACE* (exponential space) that provably contains problems not solvable in *PSPACE*:

  $$\mathit{EXPSPACE} = \bigcup_k \mathit{SPACE}(2^{n^k})$$
- Corresponding to the time-complexity classes *EXPTIME*-hard and *EXPTIME*-complete are the space-complexity classes *EXPSPACE*-hard and *EXPSPACE*-complete. A language A is *EXPSPACE*-hard if for all $L \in \mathit{EXPSPACE}$, $L \leq_p A$. (A polynomial-time reduction is still used.) A language is *EXPSPACE*-complete if it is both *EXPSPACE*-hard and an element of *EXPSPACE*.
- Corresponding to the exponential-time hierarchy, there is an exponential-space hierarchy, in which each class provably contains languages not decidable with less space: *2EXPSPACE*, *3EXPSPACE*, and so on.

The space-complexity classes are related to the time-complexity classes we've already seen. Some of these relations are clear:

Theorem 20.4.1: For any $g(n)$, $\textit{TIME}(g(n)) \subseteq \textit{SPACE}(g(n))$.

Proof: According to our cost model, simply creating values taking $\Theta(g(n))$ space must take $\Omega(g(n))$ time.

This holds, not just for our Java decision methods, but for any reasonable cost model and for a variety of models of computation. For example, simply visiting $g(n)$ cells of a Turing machine's tape requires making at least $g(n)$ moves.

As a direct consequence of Theorem 20.4.1, we have

Theorem 20.4.2: $\textit{P} \subseteq \textit{PSPACE}$ and $\textit{EXPTIME} \subseteq \textit{EXPSPACE}$.

To find inclusions in the other direction—a space-complexity class included in a time-complexity class—we have to reach up to a higher exponential:

Theorem 20.4.3: For any $g(n)$, there exists some constant k such that $\textit{SPACE}(g(n)) \subseteq \textit{TIME}(2^{k \cdot g(n)})$.

Proof sketch: This proof is easier to do using the TM model. The basic idea is to show that a TM that operates within an m-cell region of its tape cells cannot have more than 2^{km} distinct IDs, for some constant k. So a modified version of the machine, limited to 2^{km} steps, must produce the same decision as the original.

Again, this is true for any reasonable cost model and for a variety of models of computation.

As a direct consequence of Theorem 20.4.3, we have

Theorem 20.4.4: $\textit{PSPACE} \subseteq \textit{EXPTIME}$ and $\textit{EXPSPACE} \subseteq \textit{2EXPTIME}$.

Some of the relations among time- and space-complexity classes are fascinating puzzles—puzzles that have withstood many years of research effort by some of the most brilliant computer scientists on the planet. For example, most researchers believe that $\textit{P} \neq \textit{PSPACE}$ and $\textit{EXPTIME} \neq \textit{EXPSPACE}$, but no one has been able to prove either.

20.5 Higher-Complexity Classes

This section offers a guided tour of some of the higher-complexity classes. We state these results without proof, providing instead some pointers into the literature for those who are interested.

PSPACE-complete Problems

These are the hardest problems in *PSPACE*. They seem to be intractable, requiring exponential time, but no one has been able to prove this. A classic, *PSPACE*-complete problem is *QBF*: the Quantified Boolean Formulas problem.

QBF: Quantified Boolean Formulas
Instance: a formula that uses only Boolean variables, the logical connectives $\neg$, $\wedge$, and $\vee$, and the quantifiers $\forall$ and $\exists$.
Question: is the formula true?

For example, the *QBF* instance $\forall x \, \exists y \, (x \vee y)$ is true: for all Boolean values that can be assigned to x, there exists an assignment for y that makes $x \vee y$ true. That's a positive instance, an element of the *QBF* language. A negative instance is $\forall x \, \exists y \, (x \wedge y)$; it's a well-formed instance, but it is not true that for all x there exists a y that makes $x \wedge y$ true.

Imagine yourself facing *QBF* as a programming assignment: write a program that lets the user type in a *QBF* formula, evaluates it, and tells whether it is true or false. There's a simple kind of *PSPACE* solution to this problem: you just evaluate the input formula using all possible combinations of truth assignments for the variables and collect the results. (To work out the details, try Exercise 8.) But because there are exponentially many different combinations of truth assignments for the variables, this approach takes exponential time. It's a simple, brute-force solution, yet no solution with better asymptotic worst-case time complexity is known.

A more familiar *PSPACE*-complete program (at least to the readers of this book) is the problem of deciding whether two regular expressions are equivalent.

Regular-expression-equivalence
Instance: two regular expressions r_1 and r_2, using only the three basic operations: concatenation, +, and Kleene star.
Question: is $L(r_1) = L(r_2)$?

Regular-expression-equivalence is *PSPACE*-complete, as is NFA-equivalence:

NFA-equivalence
Instance: two NFAs M_1 and M_2.
Question: is $L(M_1) = L(M_2)$?

The same result does *not* hold for DFAs. Two DFAs can be converted to minimum-state DFAs and compared for equivalence in polynomial time.

EXPTIME-complete Problems

We have already shown that *Run-with-time-limit* is *EXPTIME*-complete. That might seem like a rather artificial problem, but many more natural *EXPTIME*-complete problems have been found. An example is the game of checkers:

Generalized-checkers
Instance: an n-by-n checkerboard with red and black pieces.
Question: can red guarantee a win from this position?

Notice that the game is generalized to boards of unbounded size. Of course, even for $n = 8$ (as in American checkers) the number of problem instances—the number of possible configurations of pieces on the 8-by-8 checkerboard—is large enough to make the game challenging for humans. The language of winning positions is too large for human players to memorize. But it is still a finite number, and that makes the fixed-size game uninteresting from the perspective of formal language; it's just a finite (though very large) language, decidable by a simple (though very large) DFA. Generalized to n-by-n checkerboards, the language becomes infinite, and deciding it turns out to be intractable. The exact rules of checkers (or draughts) vary around the world, but under one reasonable definition of the rules, *Generalized-checkers* has been shown to be *EXPTIME*-complete. Generalized versions of chess and go have also been shown to be *EXPTIME*-complete.

EXPSPACE-complete Problems

In Section 9.4, several extensions to regular expressions were discussed. One was squaring, defining $L((r)^2) = L(rr)$. The question of whether two regular expressions are equivalent, when squaring is allowed, becomes more difficult to answer. You can of course convert all the squaring into explicit duplication; then you've got two basic regular expressions that can be compared as already discussed. But that first step, eliminating the squaring, can blow up the expression exponentially—not a promising first step, expecially considering that the rest of the algorithm needs space that is polynomial in that new, blown-up length! In fact, the regular-expression-equivalence problem is known to be *EXPSPACE*-complete when the expressions can use these four operations: concatenation, +, Kleene star, and squaring.

Instead of squaring, regular expressions can be extended with an intersection operator &, so that $L(r_1 \& r_2) = L(r_1) \cap L(r_2)$. This again makes the equivalence problem *EXPSPACE*-complete. Note that neither of these extensions changes the set of languages you can define using regular expressions; it remains the set of regular languages. The only change is that, using the extensions, you can define some regular languages much more compactly.

A Problem Requiring *2EXPTIME*

In Section 18.13 we saw that number theory—the theory of natural numbers with multiplication and addition—is undecidable. A similar, but simpler, theory arises if you make just one small change: leave out multiplication. This theory is called Presburger arithmetic.

Presburger-arithmetic

Instance: a formula that uses only the constants 0 and 1, variables over the natural numbers, the + operator, the = comparison, the logical connectives ¬, ∧, and ∨, and the quantifiers ∀ and ∃.

Question: is the formula true?

Mojzesz Presburger explored this theory in the late 1920s. He gave a set of axioms for this arithmetic and proved that they are consistent (not leading to any contradictions) and complete (capable of proving all true theorems). He proved that this arithmetic is decidable: an algorithm exists to decide it, so the language is recursive. But it is known that no algorithm that decides *Presburger-arithmetic* can take less than doubly exponential time. So this is an example of a language that is in *2EXPTIME*, but not in *EXPTIME*.

A Problem Requiring Nonelementary Time

Beyond *2EXPTIME* lies an infinite hierarchy of higher exponentials: *3EXPTIME*, *4EXPTIME*, and so on, each one occupied by problems that cannot be solved in less time. The union of all the classes in this exponential-time hierarchy is the class called *ELEMENTARY TIME*: the class of languages decidable in k-fold exponential time, for any k. That's an impressively generous time budget—but it still isn't enough to handle all the recursive languages. There are languages that are recursive, but provably not in *ELEMENTARY TIME*.

A simple example is the regular-expression-equivalence problem, when regular expressions are extended with a complement operator ^{C}, so that $L((r)^C) = \{x \in \Sigma^* \mid x \notin L(r)\}$. We know that the regular languages are closed for complement, so this extension does not add the ability to define new languages. But, once again, we get the ability to define some languages much more compactly; and, once again, that makes the problem of deciding the equivalence of two regular expressions harder, in this case, *much* harder. When complement is allowed, the problem of deciding the equivalence of two regular expressions provably requires nonelementary time.

20.6 Further Reading

The question of whether *Primality* is in *P* was an open problem for many years. Practical algorithms for primality testing were known, and widely used in cryptography, but they were probabilistic; they involved the use of random numbers and with very small probability took a very long time. No deterministic polynomial-time algorithm was known until

Agrawal, M., N. Kayal, and N. Saxena. "PRIMES in P." *Annals of Mathematics* 160, no. 2 (2004) 781–93.

On *QBF*, regular-expression equivalence, and NFA equivalence, see

Garey, Michael, and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W. H. Freeman and Company, 1979.

The original results on the complexity of generalized board games can be found in

Robson, J. M. "N by N Checkers is EXPTIME complete." *SIAM Journal on Computing* 13, no. 2 (1984): 252–67.

Robson, J. M. "The Complexity of Go." *Proc. International Federation of Information Processing* (1983): 13–417.

Fraenkel, S., and D. Lichtenstein. "Computing a perfect strategy for n*n chess requires time exponential in n." *Proceedings of the 8th International Colloquium on Automata, Languages, and Programming* (1981): 278–93.

The equivalence problem for regular expressions with exponentiation is discussed in

Sipser, Michael. *Introduction to the Theory of Computation*. Belmont, CA: PWS Publishing, 1997, 313–17.

For regular expressions with intersection, see

Fuerer, Martin. "The Complexity of the Inequivalence Problem for Regular Expressions with Intersection." *ICALP* (1980): 234-45.

For regular expressions with complement, see

Meyer, A. R., and L. Stockmeyer. "Nonelementary word problems in automata and logic." *Proceedings of the AMS Symposium on Complexity of Computation*. (1973).

Exercises

EXERCISE 1

a. For fixed k, is $O(n^k)$ the same as $O(n^k \log n)$? Prove your answer.

b. Is *P* the same as $\bigcup_k TIME(n^k \log n)$? Prove your answer.

EXERCISE 2

a. For fixed k, is $O(2^{n^k})$ the same as $O(3^{n^k})$? Prove your answer.

b. Is *EXPTIME* the same as $\bigcup_k TIME(3^{n^k})$? Prove your answer.

EXERCISE 3

Suppose that an incredibly fast computer began executing a decision method at the time of the big bang. Assume that the big bang was 15 billion years ago, and that the computer has executed one step per yoctosecond ever since—that is, 10^{24} steps per second. What is the largest input of the size n for which it could have reached a decision by now, assuming that the decision required

a. n steps
b. n^2 steps
c. n^{13} steps
d. 2^n steps
e. 10^n steps
f. 2^{2^n} steps
g. $2^{2^{2^n}}$ steps

EXERCISE 4

(This exercise refers to concepts discussed with the time-hierarchy theorem in Appendix B.) Give an implementation and analysis to prove that each of the following functions over $\mathcal{N}$ is time constructible.

a. n^3
b. 2^n
c. $n \log_2 n$ (Implement this as $n\lfloor \log_2 n \rfloor$ and assume $n > 0$.)
d. $2^{2n} / \log_2 n$ (Implement this as $\dfrac{2^{2n}}{n\lfloor \log_2 n \rfloor}$, and assume $n > 0$.)

EXERCISE 5

(Some parts require the time-hierarchy theorem from Appendix B. You may assume that all the functions you need are time constructible.)

a. Is $TIME(n^3)$ the same as $TIME(n^3 (\log n)^3)$? Prove your answer.
b. Is $TIME(n^3)$ the same as $TIME(3n^3)$? Prove your answer.
c. Is $TIME(2^n)$ the same as $TIME(2^{n+2})$? Prove your answer.
d. Is $TIME(2^n)$ the same as $TIME(n2^n)$? Prove your answer.
e. Is $TIME(n \log n)$ the same as $TIME(n\sqrt{n})$? Prove your answer.

EXERCISE 6

In the proof of Theorem 20.3.1, the decision method `LdecPoly` has this statement:

```
int b = c * power(2,power(n,k));
```

Give an implementation of the `power` method, and using it show that the runtime for this statement is polynomial in `n`.

EXERCISE 7

For each of the following *QBF* instances, decide whether it is true or false, and prove it.

a. $\forall x(x)$
b. $\exists x(x)$
c. $\forall x(x \vee \overline{x})$
d. $\exists x(x \vee \overline{x})$
e. $\forall x \exists y((x \vee y) \wedge (\overline{x} \vee \overline{y}))$
f. $\exists x \forall y((x \vee y) \wedge (\overline{x} \vee \overline{y}))$
g. $\exists x \forall y \exists z(x \vee y \vee z)$
h. $\exists x \forall y \exists z((\overline{x} \wedge y) \vee (y \vee z))$

EXERCISE 8

In this exercise, you will write a Java application `DecideQBF` that takes a *QBF* instance from the command line and prints `true` or `false` depending on whether the input string is in the language of true, syntactically legal, quantified Boolean formulas.

Start with the syntax, the `QBF` classes, and the parser from Chapter 15, Exercise 8. Then add an `eval` method to the `QBF` interface and to each of the `QBF` classes, so that each formula knows how to decide whether it is true or not. (You will need to pass the context of current bindings in some form as a parameter to the `eval` method.) The `eval` method should throw an `Error` if it encounters scoping problems—that is, if there is a use of a free variable, as in `Ax(y)`, or a redefinition of a bound variable, as in `Ax(Ex(x))`. Of course, the parser will also throw an `Error` if the string cannot be parsed. All these `Errors` should be caught, and your `DecideQBF` should decide false for such inputs. It should decide true only for inputs that are legal, true, quantified Boolean formulas. For example, you should have

```
> java DecideQBF 'Ey(y)'
true
> java DecideQBF 'ExAy((x+y)*(~x+~y))'
false
> java DecideQBF 'x'
false -- free variable x
> java DecideQBF 'hello'
false -- syntax error
```

EXERCISE 9

The following decision problems involving basic regular expressions can all be solved by algorithms that begin by converting the regular expression into an NFA, as in the construction of Lemma 7.1.

a. Give an algorithm and analysis demonstrating that *Regular-expression-equivalence* is in *EXPTIME*.

b. Give an algorithm and analysis demonstrating that the following decision problem is in *P*.

Regular-expression-emptiness
Instance: a regular expression r using only the three basic
operations: concatenation, +, and Kleene star.
Question: is $L(r) = \{\}$?

c. Give an algorithm and analysis demonstrating that the following decision problem is in *EXPTIME*.

Regular-expression-universality
Instance: a regular expression r using only the three basic
operations: concatenation, +, and Kleene star.
Question: is $L(r) = \Sigma^*$?

d. Give an algorithm and analysis demonstrating that the following decision problem is in *P*.

Regular-expression-finiteness
Instance: a regular expression r using only the three basic
operations: concatenation, +, and Kleene star.
Question: is $L(r)$ finite?

EXERCISE 10

Consider the following decision problems:

Extended-regular-expression-equivalence
Instance: extended regular expressions r_1 and r_2 that may use
concatenation, +, Kleene star, and complement.
Question: is $L(r_1) = \mathrm{L}(r_2)$?

Extended-regular-expression-emptiness
Instance: an extended regular expression r that may use
concatenation, +, Kleene star, and complement.
Question: is $L(r) = \{\}$?

This chapter stated (without proof) that *Extended-regular-expression-equivalence* requires nonelementary time. Now, prove that *Extended-regular-expression-emptiness* requires nonelementary time if and only if *Extended-regular-expression-equivalence* requires nonelementary time.

EXERCISE 11

State the following assertions about natural numbers as formulas in Presburger arithmetic. Then state whether the assertion is true.

a. The successor of any number is not equal to that number.
b. There is a number that is greater than or equal to any other number.
c. Adding zero to any number gives you that number again.
d. No number is less than three.
e. Zero is not the successor of any number.
f. Between every number and its successor there lies another number.
g. For any $x \neq 0$, $3x > 2$.

EXERCISE 12

Using Java decision methods, show that *P* is closed for

a. complement
b. union
c. intersection
d. reversal
e. concatenation
f. Kleene star (This last is more difficult than the others. Try for a solution using dynamic programming.)

EXERCISE 13

Using Java decision methods, show that *PSPACE* is closed for

a. complement
b. union
c. intersection
d. reversal
e. concatenation
f. Kleene star

EXERCISE 14

Prove or disprove each of the following assertions.

a. The $\leq_P$ relation is transitive: if $A \leq_P B$ and $B \leq_P C$ then $A \leq_P C$.
b. The $\leq_P$ relation is reflexive: for all languages L, $L \leq_P L$.
c. The $\leq_P$ relation is symmetric: if $A \leq_P B$ then $B \leq_P A$.
d. If $A \leq_P B$ and $B \in P$ then $A \in P$.
e. If $A \leq_P B$ and $B \leq_P A$ then both A and B are in P.
f. If A is *EXPTIME*-hard and $A \leq_P B$ then B is *EXPTIME*-hard.

g. If A is *PSPACE*-complete and $A \leq_P B$ then B is *PSPACE*-complete.

h. If $L \leq_P$ *Run-with-time-limit* then $L \in$ *EXPTIME*.

i. If $L \leq_P$ *Run-with-time-limit* then L is *EXPTIME*-complete.

j. *Generalized-chess* $\in$ *2EXPTIME*.

EXERCISE 15

Prove that any language L requires nonelementary time if and only if it requires nonelementary space.

21 Complexity Classes

In this chapter we focus on the class NP*—a class defined using a model of computation that, as far as we know, cannot possibly be realized as a physical computer. This might seem, at first, like an arbitrary construction—another puzzle divised by professors for the torment of students. It does not seem like the kind of thing that belongs in a book whose subtitle is "A Practical Introduction." Where's the practical value in learning about* NP*? But we ask for your indulgence for a few pages.* NP *turns out to have great commercial significance. It is also at the heart of the most important open problem in computer science—for the solution of which there is, on standing offer, a reward of $1 million.*

If that's not practical, what is?

21.1 Verification Methods

In this chapter we consider nondeterministic complexity classes and in particular the class *NP*. But we start with a purely deterministic computation called a *verification method*. (In Section 21.5 we'll see how nondeterminism fits into the picture.)

In form, a verification method is like a decision method, but with two string parameters.

A verification method takes two `String` parameters, the *instance* and the *certificate*, and returns a `boolean` value.

We have already stretched our definition of a decision method to allow multiple string parameters, on the understanding that if we wanted to use the strict definition, we could rewrite any method that takes two strings `(x,y)` to make it take a single string `x+c+y`, for some separator `char c` not occurring in `x`. So, in that sense, a verification method is no different from a decision method, and we can continue to use all our old definitions.

The important difference between verification methods and decision methods lies in how we use them to define languages. We use a verification method to define, not the set of instance/certificate pairs, but the set of positive instances: the set of instances for which some certificate exists.

The language defined by a verification method `v(x,p)` is

$L(\texttt{v}) = \{\texttt{x} \mid \text{for some certificate } \texttt{p}, \texttt{v(x,p)}\}$.

The instance string `x` is the string being tested for language membership; it is in the language if there exists at least one string `p` that the verification method accepts as a certificate for `x`.

For example, this is a verification method for the language $\{zz \mid z \in \Sigma^*\}$:

```
boolean xxVerify(String x, String p) {
  return x.equals(p+p);
}
```

In this example, the certificate is a string p such that $x = pp$. The instance string x is in the language $\{zz \mid z \in \Sigma^*\}$ if and only if such a string p exists. In this case, of course, the certificate is telling us something we could easily have figured out anyway; a decision method for the language is not that much more difficult to write. But in other cases, the information in the certificate appears to make writing a verification method for a language much simpler than writing a decision method. Stretching the definition to allow `int` inputs, consider this verification method:

```
boolean compositeVerify(int x, int p) {
  return p > 1 && p < x && x % p == 0;
}
```

This is a verification method for the language of composite numbers; a composite number is a positive integer with an integer factor other than one or itself. Here, verification seems to be much easier than decision; checking a proposed factor of *x* seems to be much easier than deciding from scratch whether such a factor exists.

As these examples show, a verification method by itself is not a computational procedure for determining language membership. It does not tell you how to find the certificate; it only checks it after you've found it. Of course, you could search for it blindly. Given any verification method `aVerify` for a language *A*, you could make a recognition method for *A* by enumerating the possible certificates `p` and trying them all. Suppose `SigmaStar` is an enumerator class that enumerates Σ^*—then we could implement a recognition method this way:

```
boolean aRecognize(String x) {
  SigmaStar e = new SigmaStar();
  while (true) {
    String p = e.next();
    if (aVerify(x,p)) return true;
  }
}
```

But this is not a decision method; if there is no string `p` for which `aVerify(x,p)` returns true, it simply runs forever. To make this into a decision method, we need some finite bound on the set of certificates that must be checked.

21.2 *NP*, *NP*-Hard, and *NP*-Complete

A finite bound is part of the definition of the language class called *NP*. Roughly speaking, *NP* is the set of languages that have polynomial-size certificates verified by polynomial-time verifiers.

NP is the set of all languages *L* for which there exists a polynomial-time verification method `v` and a constant *k*, so that

$$L = \{\texttt{x} \mid \text{for some certificate } \texttt{p} \text{ with } |\texttt{p}| \leq |\texttt{x}|^k, \texttt{v(x,p)}\}$$

This is a restriction on our general definition of a verification method: not only must it run in polynomial time, but if it accepts any certificate for a string `x`, it must accept at least one whose size is polynomial in x. We'll call this an *NP verification method*, to distinguish it from the unrestricted kind. The time and space requirements of a verification method are measured in terms of the input size $n = |x| + |p|$. But here, because $|p|$ itself is polynomial in $|x|$, we can say that the worst-case time and space of an *NP* verification method are simply polynomial in $n = |x|$.

Let's examine this class *NP*. Where does it fit among our other complexity classes? For one thing, it clearly contains all of *P*.

Theorem 21.1.1: $P \subseteq NP$.

Proof: Let *A* be any language in *P*. Then *A* has a polynomial-time decision method aDec. We can use this to construct a polynomial-time verification method for *A*:

```
boolean aVerify(String x, String p) {
  return aDec(x);
}
```

This method ignores the certificate string and decides whether $x \in A$ using aDec. If $x \notin A$, it will return false in polynomial time, no matter what certificate is given. If $x \in A$, it will return true in polynomial time, for any certificate p, and in particular for the empty string. So it meets our definition for an *NP* verifier using $k = 0$.

With a slightly more elaborate construction, we can show that *NP* is contained in *EXPTIME*.

Theorem 21.1.2: $NP \subseteq EXPTIME$.

Proof: Let *A* be any language in *NP*. Then *A* has an *NP* verification method aVerify. Let *k* be the constant from the definition, so that if $x \in A$, there is some certificate p with $|p| \le |x|^k$ for which aVerify(x,p). Let SigmaStar be an enumerator class that enumerates Σ^* in order of length, starting with ε. Then, using our definition for k, the following method aDecide decides *A*:

```
boolean aDecide(String x) {
  int b = power(x.length(),k);
  SigmaStar e = new SigmaStar();
  while (true) {
    String p = e.next();
```

```
            if (p.length() > b) return false;
            if (aVerify(x,p)) return true;
          }
        }
```

If there is a certificate for x, there must be some certificate p with $|\mathrm{p}| \leq |\mathrm{x}|^k$, so this method will find it and return true. If not, this method will exhaust all the certificates with $|\mathrm{p}| \leq |\mathrm{x}|^k$ and return false. The number of iterations of the loop—the number of calls to the polynomial-time verifier—in the worst case for an input of size n is the number of possible certificates p with $|\mathrm{p}| \leq n^k$, which is $|\Sigma|^{n^k}$. Thus, $A \in$ *EXPTIME*.

In this proof, the verifier for *A* runs in polynomial time, so the language of instance/certificate pairs is in *P*. But what about the language *A* itself—the language of instances for which a certificate exists? It's still possible that *A* is in *P*, using a decision method with a better strategy than searching though all possible certificates. But our construction above shows only that *A* is in *EXPTIME*.

We can make hardness and completeness definitions for *NP* as usual:

A is *NP*-hard if, for all $L \in$ *NP*, $L \leq_{\mathrm{p}} A$.

We have already seen quite a few *NP*-hard problems. For example, *Generalized-checkers* is *EXPTIME*-hard and *NP* $\subseteq$ *EXPTIME*, so *Generalized-checkers* is *NP*-hard. But *Generalized-checkers* is not known to be in *NP*. Of particular interest are those problems that are both *NP*-hard and in *NP*.

A is *NP*-complete if *A* is in *NP* and *A* is *NP*-hard.

NP-complete problems are, in a sense, the hardest problems in *NP*. We have not yet seen any examples of *NP*-complete problems, but we'll see several in the following pages.

Putting Theorems 21.1.1 and 21.1.2 together gives us a rough idea of where *NP* fits in among our other complexity classes: *P* $\subseteq$ *NP* $\subseteq$ *EXPTIME*. But that's a very wide range; problems in *P* are (at least nominally) tractable, while *EXPTIME*, as we have seen, provably contains intractable problems. So what about those problems in *NP*? Are they all tractable, or are the harder ones (in particular, the *NP*-complete problems) intractable? That's the heart of the matter—and the most important open question in computer science today. Most researchers believe that *P* $\neq$ *NP*, so that the *NP*-complete problems are intractable, requiring more than polynomial time. But no one has been able to prove it. No one has been able either to find polynomial-time algorithms for these problems or to prove that no such algorithms exist.

21.3 A Tour of Six Decision Problems

In this section we will explore six different *NP*-complete problems. These decision problems form a chain, with each reduced in polynomial time to the next.

SAT

This is the classic *NP*-complete problem: the first problem proved to be *NP*-complete, and still one of the most widely used and studied.

Satisfiability (SAT)
Instance: a formula that uses only Boolean variables, with operators
for logical AND, logical OR, and logical NOT.
Question: is there some assignment of values to the variables that
makes the formula true?

This is related to the *QBF* problem we've already seen, but simpler; a *SAT* instance is like a *QBF* instance with no quantifiers. (In *SAT*, all the variables are implicitly existentially quantified.) Mathematicians often use the operator symbols ¬, ∧, and ∨ and with precedence in that order and parentheses as needed; for example:

$a \wedge b$	true, under the assignment *a*=true and *b*=true
$a \wedge \neg a$	false, under all assignments
$(a \vee b) \wedge \neg(a \wedge b)$	true, under the assignment *a*=true and *b*=false

In most programming languages we use symbols that are easier to type. In Java, we might give the same examples as

```
a & b
a & !a
(a | b) & !(a & b)
```

But the concrete syntax isn't important here. Both the mathematical notation and the Java notation are just context-free languages, for which simple grammars can be given, and a string that encodes a *SAT* instance in one of these languages can be parsed and checked syntactically in polynomial time. We might say that the language of syntactically correct *SAT* instances is in *P*. But that still leaves the satisfiability question: *SAT* is a subset of the syntactically correct instances, containing those that have at least one satisfying truth assignment. This subset is not context free, and it is not believed to be in *P*.

It is, however, in *NP*. We can show this by outlining a verification method for it. Here, the certificate is a truth assignment; a verification method need only verify that it makes the formula true.

Lemma 21.1.1: *SAT* ∈ *NP*.

Proof sketch: We need to show that an *NP* verification method `vSAT(x,p)` exists for *Satisfiability*. The instance string is `x`, of course, a *Satisfiability* formula, whose syntax we can check in polynomial time. The certificate `p` can be just a string over the alphabet {T, F}, encoding an assignment of truth values to each of the variables in `x`, in some fixed order (such as in the order of their first appearance in `x`). A formula of length *n* cannot contain more than *n* distinct variables, so the size of the certificate is polynomial (in fact, linear) in the size of the instance. All `vSAT` has to do is to evaluate the formula using the truth assignment given by the certificate. This can easily be done in polynomial time.

The *NP* verification method for *SAT* only needs to check, in polynomial time, whether the particular truth assignment given by the certificate makes the formula true. Checking a truth assignment is easy; finding one seems much harder. There is a simple procedure, of course: a blind search for a certificate, as in the proof of Theorem 21.1.2. But that takes $O(2^n)$ time in the worst case. And although many researchers have sought a polynomial-time solution, none has been found.

If there is a polynomial-time solution, then there is a polynomial-time solution for every problem in *NP*, and so *P* = *NP*. The following theorem was proved by Stephen Cook in 1971.

Lemma 21.1.2 (Cook's Theorem): *SAT* is *NP*-hard.

Proof sketch: For any $L \in NP$, there is a polynomial-time verification method, which we'll take to be in the form of a Turing machine *M*. For any such machine we can give a polynomial-time reduction $f(x)$ that generates a Boolean formula, so that $f(x) \in SAT$ if and only if $x \in L$. The Boolean formula, though large and complicated, is only polynomially larger than *x*. A more detailed sketch of this important proof may be found in Appendix C.

We have now proved that *SAT* is in *NP* and is *NP*-hard. Putting these results together, we conclude

Theorem 21.2.1: *SAT* is *NP*-complete.

CNFSAT

Sometimes, the simplicity of Turing machines makes them more useful than decision methods for proofs—as in the proof of Cook's Theorem. Similarly, simplified versions of *SAT*, as long as they are still *NP*-complete, are often more useful for other proofs than *SAT* itself.

A *SAT* formula is in *conjunctive normal form* (*CNF*) when it is the logical AND of one or more *clauses*, each of which is the logical OR of one or more *literals*, each of which is either a *positive literal* (a single Boolean variable) or a *negative literal* (a logical NOT operator applied to a single Boolean variable).

Restricting instances to CNF produces a different decision problem:

CNF Satisfiability (*CNFSAT*)
Instance: a *SAT* instance in conjunctive normal form.
Question: is there some assignment of values to the variables that
makes the formula true?

CNFSAT instances use all the same operators as *SAT* instances, but restrict the order in which they may be applied: variables may be negated using ¬, then those pieces (the literals) may be combined using ∨, and finally those pieces (the clauses) may be combined using ∧. For example, here are some *CNFSAT* instances, written with a mathematical syntax:

$(a) \wedge (b)$	true, under the assignment a=true and b=true
$(a) \wedge (\neg a)$	false, under all assignments
$(a \vee b) \wedge (\neg a \vee \neg b)$	true, under the assignment a=true and b=false
$(\neg b) \wedge (\neg c) \wedge (a \vee b \vee c) \wedge (\neg a \vee b \vee c)$	false, under all assignments

This concrete syntax is simpler than that concrete syntax we used for *SAT*. (In fact, depending on the details, it might well be regular—see Exercise 10.) But the concrete syntax is not the important question here. For any reasonable representation of *CNFSAT* instances, the important question is the satisfiability question: is there a truth assignment for the variables that makes the whole formula true?

It is easy to see that *CNFSAT* is in NP, because it can use largely the same verification method as *SAT*.

Lemma 21.2.1: *CNFSAT* ∈ NP.

Proof: The restricted syntax is context free and so can be checked in polynomial time. With that change, an NP verification method can be constructed the same way as for *SAT*, in the proof of Lemma 21.1.1.

In fact, checking the certificates could be simpler for *CNFSAT* than for general *SAT*. The verification method only needs to check that, using the given truth assignment, each clause contains at least one true literal. The more interesting result is that, in spite of the restricted syntax, *CNFSAT* is still NP-hard.

Lemma 21.2.2: *CNFSAT* is *NP*-hard.

Proof: By reduction from *SAT*. See Appendix C.

We have now shown that *CNFSAT* is in *NP* and is *NP*-hard. Putting these results together, we have

Theorem 21.2.2: *CNFSAT* is *NP*-complete.

We now have two *NP*-complete problems. Each can be reduced to the other in polynomial time: *SAT* $\leq_P$ *CNFSAT*, and *CNFSAT* $\leq_P$ *SAT*. In that sense, *CNFSAT* is exactly as difficult to decide as *SAT* itself. It's simpler without being any easier.

3SAT

It turns out that we can simplify satisfiability even more, without losing *NP*-completeness. We can require that every clause contain exactly the same number of literals:

A CNF formula is in *k*-CNF if each clause contains exactly *k* distinct literals.

Restricting instances to 3-CNF produces a decision problem that is structurally very simple, yet still *NP*-complete:

3-CNF Satisfiability (3SAT)
Instance: a SAT instance in 3-CNF.
Question: is there some assignment of values to the variables that
makes the formula true?

This is just a restricted syntax for *SAT* instances, so proving membership in *NP* is easy:

Lemma 21.3.1: *3SAT* $\in$ *NP*.

Proof: The restricted syntax is context free and so can be checked in polynomial time. With that change, an *NP* verification method can be constructed the same way as for *SAT*, in the proof of Lemma 21.1.1.

Proving *NP*-hardness is also straightforward. We already know that *CNFSAT* is *NP*-hard, so we only need to show that *CNFSAT* $\leq_P$ *3SAT*.

Lemma 21.3.2: *3SAT* is *NP*-hard.

Proof: By reduction from *CNFSAT*. Each clause *c* in a *CNFSAT* instance consists of one or more literals: $c = (l_1 \vee l_2 \vee \ldots \vee l_n)$. Depending on the number

n of literals in the clause, we can replace each clause c with one or more *3SAT* clauses as follows.

If $n = 1$, $c = (l)$ for some single literal l. Create two new variables x and y, not occurring elsewhere in the formula, and replace c with these four clauses:

$$(l \vee x \vee y) \wedge (l \vee x \vee \neg y) \wedge (l \vee \neg x \vee y) \wedge (l \vee \neg x \vee \neg y)$$

A truth assignment satisfies these if and only if it makes l true, so this transformation preserves satisfiability. (Note that we cannot simply use $(l \vee l \vee l)$, because 3-CNF requires three *distinct* literals in each clause.)

If $n = 2$, $c = (l_1 \vee l_2)$. Create one new variable x, not occurring elsewhere in the formula, and replace c with these two clauses:

$$(l_1 \vee l_2 \vee x) \wedge (l_1 \vee l_2 \vee \neg x)$$

A truth assignment satisfies these if and only if it makes $l_1 \vee l_2$ true, so this transformation preserves satisfiability.

If $n = 3$, c is already a 3-CNF clause, and no transformation is necessary.

If $n \geq 4$, $c = (l_1 \vee l_2 \vee l_3 \vee \ldots \vee l_n)$. Create $n - 3$ new variables, x_3 through x_{n-1}, not occurring elsewhere in the formula, and replace c with these $n - 2$ clauses:

$$\begin{aligned}
&(l_1 \vee l_2 \vee x_3) \wedge \\
&\qquad (\neg x_3 \vee l_3 \vee x_4) \wedge \\
&\qquad (\neg x_4 \vee l_4 \vee x_5) \wedge \\
&\qquad (\neg x_5 \vee l_5 \vee x_6) \wedge \\
&\qquad \ldots \wedge \\
&(\neg x_{n-1} \vee l_{n-1} \vee l_n)
\end{aligned}$$

(Except for the first and last, these clauses are all of the form $(\neg x_i \vee l_i \vee x_{i+1})$.) Suppose the original clause is satisfiable. Then there is some truth assignment that makes at least one of the l_i true. This can be extended to a truth assignment that satisfies all the replacement clauses, by choosing x_k = true for all $k \leq i$, and x_k false for all $k > i$. Conversely, suppose there is some truth assignment that satisfies all the replacement clauses. Such a truth assignment cannot make all the l_i false; if it did, the first replacement clause would require x_3 = true, and so the second would require x_4 = true, and so on, making the last clause false. Since a satisfying assignment for the replacement clauses must make at least one l_i true, it also satisfies the original clause. Therefore, this transformation preserves satisfiability.

Applying this transformation to each clause in the original formula ϕ produces a 3CNF formula ϕ', with $\phi \in$ *CNFSAT* if and only if $\phi' \in$ *3SAT.* The entire construction takes polynomial time. (In fact, the size of ϕ' is linear in the size of ϕ.) Therefore, *CNFSAT* $\leq_P$ *3SAT.* Because *CNFSAT* is *NP*-hard, we conclude that *3SAT* is *NP*-hard.

Incidentally, it turns out that 3 is the smallest value of k for which *kSAT* is *NP*-hard. The satisfiability of 1-CNF and 2-CNF formulas can be decided in linear time.

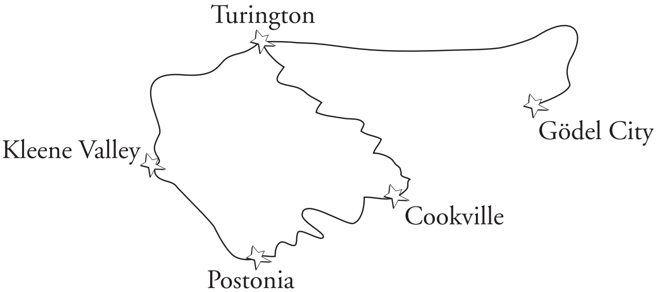
We have now shown that *3SAT* is in *NP* and is *NP*-hard. Putting these results together, we have

Theorem 21.2.3: *3SAT* is *NP*-complete.

Vertex Cover

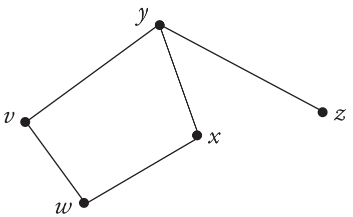
Now for something (apparently) completely different. The *Vertex Cover* problem concerns a property of graphs and does not, at first glance, appear to have anything to do with logic.

Imagine that you are running a courier service, and you have a map of cities connected by roads. You want to establish transfer stations in the cities, enough so that there is at least one at one end of every road. But you may not have enough money to build one in every city. For example, suppose this is your map:



If you can afford to build $k = 5$ stations, there's no problem: just put one in every city. But suppose you can only afford $k = 2$. Is it still possible to place at least one at one end of every road? Yes. If you build one in Turington and one in Postonia, that will suffice. On the other hand, if you can only afford $k = 1$, there is no solution.

This is a concrete example of the *Vertex Cover* problem. In the abstract, a problem instance has two parts: a graph and a goal $k \in \mathcal{N}$. The graph $G = (V, E)$ consists of a set V of vertices and a set E of edges. Each edge is just a pair of vertices. So for the example above, this would be



$G = (V, E)$

$V = \{v, w, x, y, z\}$

$E = \{(v, w), (w, x), (x, y), (v, y), (y, z)\}$

A vertex cover of a graph is a subset of the vertex set, $V' \subseteq V$, such that for all $(u, v) \in E$, $u \in V'$ or $v \in V'$. Of course every graph has a vertex cover $V' = V$, but the decision problem asks, is there a vertex cover V' of size k or less?

Vertex Cover
Instance: a graph $G = (V, E)$ and a number $k \in \mathcal{N}$.
Question: does G have a vertex cover of size k or less?

It is easy to see that *Vertex Cover* is in *NP*. The certificate can just be a proposed vertex cover, and we have only to check that it is a correct cover and has size k or less. As usual, you would have to specify some encoding of the problem instance as a string and check that the input string is a legal encoding. But abstracting away those details, we have

Lemma 21.4.1: *Vertex Cover* $\in$ *NP*.

Proof: We can make an *NP* verification method for *Vertex Cover*. The instance is a graph $G = (V, E)$ and a number $k \in \mathcal{N}$, and the certificate for an instance is some $V' \subseteq V$. Check the syntax of the problem instance and certificate. Then make a pass over the edge set E. For each $(u, v) \in E$, check that at least one of u or v occurs in V'. For any reasonable encoding of the graph, these tests can be completed in polynomial time.

Vertex Cover is also *NP*-hard. The proof, by reduction from *3SAT*, is rather surprising: what could *Vertex Cover* have to do with Boolean satisfiability? But we can show how to take any 3-CNF formula ϕ and convert it into an instance of *Vertex Cover*: a graph G and constant k such that G has a vertex cover of size k or less, if and only if ϕ is satisfiable.

Lemma 21.4.2: *Vertex Cover* is *NP*-hard.

Proof: By reduction from *3SAT*. See Appendix C.

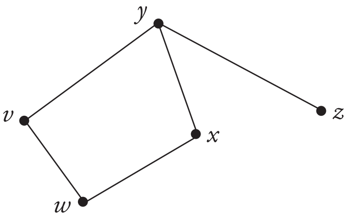
We have now shown that *Vertex Cover* is in *NP* and is *NP*-hard. Putting these results together, we have

Theorem 21.2.4: *Vertex Cover* is *NP*-complete.

There's an interesting relative of *Vertex Cover* called *Edge Cover*. An edge cover is a subset of the edge set, $E' \subseteq E$, such that every $v \in V$ is touched by some edge in E'.

Edge Cover
Instance: a graph $G = (V, E)$ and a number $k \in \mathcal{N}$.
Question: does G have an edge cover of size k or less?

Recall that our first example of *Vertex Cover* used this graph:



Examining the same graph for edge covers, we see that at $k = 5$, we can use every edge, $E' = E$, to make an edge cover; at $k = 3$ there is still a solution at $E' = \{(y, z), (v, w), (x, y)\}$; while at $k = 2$ there is no solution.

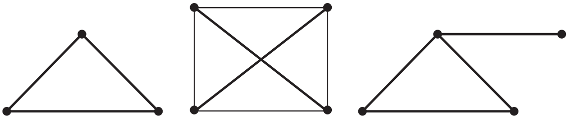
The surprising thing here is that, while *Vertex Cover* is NP-complete, *Edge Cover* is in P. We won't present it here, but there's a fairly simple, polynomial-time algorithm for finding a minimum-size edge cover for any graph. This is a good example of how unreliable intution can be as a guide to decision problems. *Edge Cover* may seem like a close relative of *Vertex Cover*, but it's in a different and (apparently) much simpler complexity class. *Vertex Cover* seems at first glance to be completely unrelated to Boolean-satisfiability problems; yet *Vertex Cover*, *SAT*, *CNFSAT*, and *3SAT* are all NP-complete, so they can all be reduced to each other in polynomial time.

Hamiltonian Circuit

The next is a classic route-finding problem: the problem of finding a path that starts and ends at the same point and visits every vertex in the graph exactly once. (Think of a delivery truck that needs to start at the warehouse, visit a list of destinations, and return to the warehouse.)

Hamiltonian Circuit
Instance: a graph $G = (V, E)$.
Question: is there a path that starts and ends at the same vertex,
arriving at every vertex exactly once?

For example, consider these three graphs:



The first two have Hamiltonian circuits; the third does not.

It is easy to see that *Hamiltonian Circuit* is in *NP*. The certificate can just be a list of vertices in some order. We have only to check that it is a Hamiltonian circuit for the graph. As usual, you would have to specify some encoding of the problem instance as a string and check that the input string is a legal encoding. But abstracting away those details, we have

Lemma 21.5.1: *Hamiltonian Circuit* ∈ *NP*.

Proof: We can make an *NP* verification method for *Hamiltonian Circuit*. The instance is a graph $G = (V, E)$, and the certificate for an instance is some sequence of vertices. Check the syntax of the problem instance and certificate. Then make a pass over the certificate, checking that it is a Hamiltonian circuit: that it visits every vertex exactly once, moves only along edges actually present in the graph, and ends where it began. For any reasonable encoding of the graph, these tests can be completed in polynomial time.

Hamiltonian Circuit is also *NP*-hard. The proof is by reduction from *Vertex Cover*. Given any instance of *Vertex Cover*, consisting of a graph G and a natural number k, we can construct a new graph H such that H has a Hamiltonian circuit if and only if G has a vertex cover of size k or less.

Lemma 21.5.2: *Hamiltonian Circuit* is *NP*-hard.

Proof: By reduction from *Vertex Cover*. See Appendix C.

We have now shown that *Hamiltonian Circuit* is in *NP* and is *NP*-hard. Putting these results together, we have

Theorem 21.2.5: *Hamiltonian Circuit* is *NP*-complete.

Many variations of this problem are also *NP*-complete. For example, the problem remains *NP*-complete if we allow paths that visit every vertex exactly once, but don't end up where they started. It remains *NP*-complete if we consider graphs with one-way edges (*directed* graphs).

On the other hand, there's an interesting relative of *Hamiltonian Circuit* called *Euler Circuit*. A Hamiltonian circuit must visit each vertex exactly once; an Euler circuit must use each *edge* exactly once.

Euler Circuit
Instance: a graph $G = (V, E)$.
Question: is there a path that starts and ends at the same vertex and follows every edge exactly once?

Alhough this sounds like a close relative of *Hamiltonian Circuit*, it is actually much simpler. We won't present it here, but there is a simple algorithm to find an Euler circuit for a graph or prove that none exists. (A graph has an Euler circuit if and only if it is fully connected and every vertex has an even number of incident edges.) So *Euler Circuit* is in *P*.

Traveling Salesman

Imagine you're planning the route for a delivery truck. It leaves the distribution center in the morning, makes a long list of deliveries, and returns to the distribution center at night. You have only enough gas to drive k miles. Is there a route you can use to make all your deliveries without running out of gas?

This is a *Traveling Salesman Problem*.

Traveling Salesman Problem (TSP)
Instance: a graph $G = (V, E)$ with a length for each edge and a total length target $k \in \mathcal{N}$.
Question: is there a path that starts and ends at the same vertex, visits every vertex exactly once, and has a total length $\leq k$?

Lemma 21.6.1: *TSP* $\in$ *NP*.

Proof: We can make an *NP* verification method for *TSP*. The instance is a graph $G = (V, E)$ and a number k; the certificate for an instance is some sequence of vertices. Check the syntax of the problem instance and certificate. Then make a pass over the certificate, checking that it visits every vertex once, moves only along edges actually present in the graph, ends where it began, and has a total length $\leq k$. For any reasonable encoding of the graph, these tests can be completed in polynomial time.

Our *TSP* is so close to *Hamiltonian Circuit* that a polynomial-time reduction is simple.

Lemma 21.6.2: *TSP* is *NP*-hard.

Proof: By reduction from *Hamiltonian Circuit*. Any instance $G = (V, E)$ of *Hamiltonian Circuit* can be converted to an instance of *TSP* in polynomial time by simply assigning a length of 1 to each edge in E and choosing $k = |V|$. With these lengths, a Hamiltonian circuit of G is a *TSP* path of a total length k, and a *TSP* path of total length $\leq k$ must be k edges long and so must be a Hamiltonian circuit. Thus *Hamiltonian Circuit* $\leq_P$ *TSP.* Because *Hamiltonian Circuit* is *NP*-hard, we conclude that *TSP* is *NP*-hard.

We have now shown that *TSP* is in *NP* and is *NP*-hard. Putting these results together, we have

Theorem 21.2.6: *TSP* is *NP*-complete.

There are many variations of this problem. In some, the goal is a closed circuit; in others, the goal is a path from a specified target to a specified destination. In some the graph is undirected; in others the graph is directed. In some, the goal is to visit every vertex at least once; in others, exactly once. In some the edge lengths are Euclidean, or at least are guaranteed to obey the triangle inequality (length(x, z) $\leq$ length(x, y) + length(y, z)); in others the lengths are unconstrained. In some, the graph is always complete, with an edge between every pair of vertices; in others the graph contains some edges but not others. All these variations turn out to be *NP*-complete.

Unfortunately, there's no standard terminology for these problems. In different contexts, *Traveling Salesman Problem* might mean any one of these variations.

21.4 The Wider World of *NP*-Complete Problems

In the previous section we introduced six *NP*-complete problems. For all of them, it was easy to show membership in *NP*; the difficult part for most was showing *NP*-hardness. We started with *SAT* and sketched the proof of Cook's Theorem, showing that *SAT* is *NP*-hard. From there we developed *NP*-hardness results for the others using a chain of polynomial-time reductions:

$$SAT \leq_P CNFSAT \leq_P 3SAT \leq_P Vertex\ Cover \leq_P Hamiltonian\ Circuit \leq_P TSP$$

Through this chain, *SAT* can be reduced in polynomial time to any of the others. Through Cook's Theorem, any of the others can be reduced back to *SAT*. So they all stand or fall together. If a polynomial-time solution is found for any one of them, then all of them have a polynomial-time solution. If a proof is found for any one of them, that it is not in *P*, then none of them have a polynomial-time solution.

This is a remarkable state of affairs. It would be remarkable even if only these six problems were involved. In fact, the number of known *NP*-complete problems is now well into the *thousands*, with a vast tree of polynomial-time reductions rooted at *SAT*. There are *NP*-complete decision problems in many different domains:

- Logic problems like *SAT*
- Mathematical problems involving sets and equation-solving
- Graph problems: covering problems like *Vertex Cover*, colorability problems, and subgraph-finding problems
- Strategy problems for games
- Network problems involving routing, capacity, and connectivity
- Database problems, including compression, allocation, and retrieval
- Compiler problems, including code generation and register allocation
- Scheduling problems: the coordination of limited resources in diverse domains such as business, manufacturing, construction, health care, and education
- Layout problems such as disk allocation, warehouse space allocation, and cutting-machine optimization

An amazing variety of problems in all kinds of domains, some very far removed from computer science, have turned out to be *NP*-complete. All of them stand or fall together.

For many of these decision problems there is a corresponding optimization problem. For example:

Decision Problem	**Optimization Problem**
Does the graph G have a vertex cover of size k or less? (*Vertex Cover*)	Find a minimum-size vertex cover for G.
Is there a circuit of the vertices of G, of total length k or less? (*TSP*)	Find a shortest circuit of G.
Can G be colored using no more than k colors, so that no two adjacent vertices get the same color?	Find a coloring of G using the minimum number of colors, so that no two adjacent vertices get the same color.
Is there a way to fill a knapsack from a given inventory, so that the total value of the items is at least k and the total weight is no more than w?	Find the most valuable way to fill a knapsack from a given inventory, so that the total weight is no more than w.
Is there a way to cut a sheet of metal into a set of required parts, so that less than k square millimeters is wasted?	Find the least wasteful way to cut a sheet of metal into a set of required parts.

When studying formal language we focus on decision problems, but in other contexts you are much more likely to encounter the corresponding optimization problems. People often fail to distinguish between a decision problem and its corresponding optimization problem. This is technically sloppy: *P* and *NP* are sets of languages, so it is not really correct to speak of an optimization problem as being *NP*-complete. However, this casual usage is often justified by the close connection between the two kinds of problems.

Consider, for example, our *Vertex Cover* decision problem and its corresponding optimization problem, *Vertex Cover Optimization*. There is a simple recursive solution for *Vertex Cover Optimization*:

```
minCover($G$)
  if $G$ has no edges, return {}
  else
          let $(v_1, v_2)$ be any edge in $G$
          let $V_1$ = minCover($G - v_1$)
          let $V_2$ = minCover($G - v_2$)
          let $n_1 = |V_1|$
          let $n_2 = |V_2|$
          if $n_1 < n_2$ return $V_1 \cup \{v_1\}$
          else return $V_2 \cup \{v_2\}$
```

Here, the expression $G - v$ refers to the graph G, with the vertex v and all its incident edges removed. In the base case, when there are no edges in the graph, the empty set is clearly the minimum-size vertex cover. Otherwise, we choose any edge (v_1, v_2) in G. At least one of the two ends of this edge must be in any cover for G, so we try both. We remove v_1 and the edges it covers and then recursively find a minimum cover V_1 for the rest of the graph; then $V_1 \cup \{v_1\}$ must have the minimum size of any cover for G that contains v_1. Then we do the same for the other vertex, so that $V_2 \cup \{v_2\}$ has the minimum size of any cover for G that contains v_2. Because we know that at least one of v_1 or v_2 must be part of any vertex cover, we can conclude that the smaller of $V_1 \cup \{v_1\}$ and $V_2 \cup \{v_2\}$ must be a minimum vertex cover for G. (If they are the same size, it would be correct to return either one.)

This is a reasonably simple solution, but very expensive. Our minCover calls itself recursively, *twice*. That makes a binary tree of recursive calls, whose height is (in the worst case) equal to the number of edges in the graph. That will take exponential time. Now, if we could find n_1 and n_2 some other way—find the size of a minimum-size vertex cover, without actually finding the cover—then we could reorganize the code so that minCover calls itself recursively, only *once*:

```
minCover($G$)
  if $G$ has no edges, return {}
  else
          let $(v_1, v_2)$ be any edge in $G$
          let $n_1$ = minCoverSize($G - v_1$)
          let $n_2$ = minCoverSize($G - v_2$)
          if $n_1 < n_2$ return minCover($G - v_1$) $\cup \{v_1\}$
          else return minCover($G - v_2$) $\cup \{v_2\}$
```

This makes just one recursive call for each edge in the graph. In fact, not counting the time spent in minCoverSize, this is now a polynomial-time solution for *Vertex Cover Optimization*.

That's the basis of the following proof:

Theorem 21.3: *Vertex Cover* $\in$ *P* if and only if *Vertex Cover Optimization* can be computed in polynomial time.

Proof: Suppose *Vertex Cover Optimization* can be computed in polynomial time. We can immediately use this to decide any *Vertex Cover* instance (G, k): in polynomial time, compute a minimum-size cover U for G, and return true if $|U| \leq k$, false if $|U| > k$.

In the other direction, suppose *Vertex Cover* $\in$ *P*. We can compute the size of a minimum-size vertex cover like this:

```
minCoverSize($G$)
  for each $i$ from 0 to the number of vertices in $G$
          if $(G, i) \in$ Vertex Cover return $i$
```

This minCoverSize function makes only polynomially many tests for membership in *Vertex Cover*. Therefore, with *Vertex Cover* $\in$ *P*, this is a polynomial-time computation. We can then use it, as already shown, to complete a polynomial-time solution for *Vertex Cover Optimization*.

This is a common situation: either the decision problem and the corresponding optimization problem can be done in polynomial time, or neither can. Such optimization problems stand or fall along with the thousands of decision problems for *NP*-complete languages.

One interesting feature of optimization problems, not shared by decision problems, is that they are natural targets for approximation. For example:

Optimization Problem	Approximation Problem
Find a minimum-size vertex cover for *G*.	Find a reasonably small vertex cover for *G*.
Find a shortest circuit of *G*.	Find a reasonably short circuit of *G*.
Find a coloring of *G* using the minimum number of colors, so that no two adjacent vertices get the same color.	Find a coloring of *G* using a reasonably small number of colors, so that no two adjacent vertices get the same color.
Find the most valuable way to fill a knapsack from a given inventory, so that the total weight is no more than *w*.	Find a reasonably valuable way to fill a knapsack from a given inventory, so that the total weight is no more than *w*.
Find the least wasteful way to cut a sheet of metal into a set of required parts.	Find a way to cut a sheet of metal into a set of required parts, with a reasonably small amount of waste.

In practice, many applications will be satisfied by any reasonably good solution, even if it is not provably optimal. That's good news, because while no polynomial-time solutions are known for the optimization problems, we can sometimes find a reasonably close approximation in polynomial time. (Of course, what makes a solution *reasonably* close to optimal depends on the application.)

A good algorithm for finding an approximate solution for one *NP*-related optimization problem does not necessarily translate to others. Each problem has unique approximation properties. For example, consider *Traveling Salesman Problem*. In some variations, the edge lengths are guaranteed to obey the triangle inequality (length(x, z) ≤ length(x, y) + length(y, z)); in others the lengths are unconstrained. From the formal-language point of view it makes no difference; both versions are *NP*-complete. But if you are trying to find an approximation algorithm, it makes a great deal of difference. There is a good algorithm for finding a circuit that is guaranteed to be no more than 3/2 times the length of the shortest circuit—if the triangle inequality is obeyed. If the triangle inequality is not obeyed, you can prove that no such approximation is possible in polynomial time, unless *P* = *NP*. (See Exercise 16.)

But we will not explore these approximation algorithms further here. Our focus here is on formal language. In the study of formal language we are chiefly concerned with deciding language membership. In this black-and-white world, the *NP*-complete problems are all treated as interchangeable; each may be reduced to the others in polynomial time. Indeed, we tend to think of them as just different faces of the same problem—different avatars of the one, big, abstract, *NP*-complete superproblem.

21.5 Nondeterministic Turing Machines

Our Turing machines are deterministic, but one can also define nondeterministic Turing machines (NDTMs). The big change from the deterministic version is in the transition function. In the deterministic version there is at most one move from any configuration: $\delta(q, a) = (p, b, \mathrm{D})$. In the nondeterministic version, we define $\delta(q, a)$ as a set of zero or more such triples, one for each of the possible moves when in the state q reading a. We also change the definitions of $\mapsto$ and $\mapsto^*$ accordingly. The definition of $L(M)$ can remain the same, using the new $\mapsto^*$ relation. (That is, $L(M) = \{x \in \Sigma^* \mid idfix(q_0x) \mapsto^* ypz \text{ for some } p \in F\}$.)

We've seen this kind of nondeterminism in several other contexts. When we explored it in the context of finite-state automata, we saw that it made no difference in definitional power; a language is definable using a DFA if and only if it is definable using an NFA. On the other hand, in the context of stack machines, nondeterminism made a significant difference. Nondeterminism was built into our basic definition of stack machines, and some of the exercises in Chapter 15 explored the serious limitations imposed by trying to make them deterministic. So how about Turing machines: does making them nondeterministic change the class of languages that we can define? The answer is no: NDTMs can define exactly the same languages that TMs can define.

To prove this, we'll introduce a deterministic way to try any finite path of computation through an NDTM. Say that an NDTM has been *ordered* if, for each state q and symbol a, the set $\delta(q, a)$ has been arbitrarily indexed, starting from $\delta(q, a)_0$. Then we can define a method `runGuided(N,s,guide)` that simulates one finite path of computation of an ordered NDTM `N` on an input string `s`. The third parameter, the guide string, is some string in $\{0, 1\}^*$; on each move, one or more bits of `guide` are used to select the transition. (Specifically, on each transition $\delta(q, a)$, we can choose $k = |\delta(q, a)|$, take the next $max(\lceil log_2\, k \rceil, 1)$ bits from `guide`, treat them as the binary encoding of a number i, and use $\delta(q, a)_{i \bmod k}$ as the next move. Note that even on a deterministic move, at least one bit is used—so `runGuided` always returns, after at most |`guide`| moves.) The method `runGuided` returns one of three string values describing the outcome of that computation: "reject" if it rejects, "accept" if it accepts, or "unfinished" if the guide string is exhausted before `N` halts on `s`. Clearly this can be implemented as a polynomial-time method; the details are left as an exercise.

Using this primitive, we can see that allowing nondeterminism does not change our results concerning computability.

Theorem 21.4.1: A language is RE if and only if it is $L(N)$ for some NDTM N.

Proof: Every TM is (with cosmetic changes in type) an NDTM, so in one direction this is trivial: a recursively enumerable language is by definition $L(M)$ for some TM M, which can be taken as an NDTM that happens to have $|\delta(q, a)| \leq 1$ for all q and a.

In the other direction, suppose L is $L(N)$ for some NDTM N. Let `BinaryStar` be an enumerator class for $\{0, 1\}^*$. We construct a recognition method for L as follows:

```
boolean lrec(String s) {
  BinaryStar e = new BinaryStar();
  while (true) {
    String guide = e.next();
    String result = runGuided(N,s,guide);
    if (result.equals("accept")) return true;
  }
}
```

If there is an accepting computation of `N` on `s`, there is at least one guide string corresponding to that computation, and this method will eventually try that guide string and return true. Otherwise there is no guide string for any accepting computation, and the method will run forever. Therefore, L(`lrec`) $= L(N) = L$. Because L has a recognition method, it is recursively enumerable.

A similar construction, with a little more attention to detail, can be used to make a *decision* method for $L(N)$ from any *total* NDTM N. (See Exercise 17.) So our definition of a recursive language is likewise unaffected by the addition of nondeterminism.

On the other hand, the addition of nondeterminism seems to have a major effect on our complexity classes. To explore this, we'll need to revisit our cost model. We will say that the cost of an NDTM computation is the cost of the most expensive of any of the possible sequences on the given input:

time(M, x) = the *maximum* number of moves made by M on input $x \in \Sigma^*$
space(M, x) = the *maximum* length of tape visited by M on input $x \in \Sigma^*$

Building from that foundation, what kinds of complexity classes do we get? Surprisingly, one at least is already familiar: the class of languages defined by some polynomial-time NDTM is exactly the class *NP*. We prove this in two parts.

Theorem 21.4.2: If $L \in$ *NP*, $L = L(N)$ for some polynomial-time NDTM *N*.

Proof: Let *L* be an arbitrary language in *NP*. Then *L* has a verification method `vL`. Because this is an *NP*-verification method, it runs in polynomial time, and we have a polynomial bound $t(x)$ on the length of any viable certificate *y*. Now construct an NDTM *N* with two phases of computation. In the first phase, *N* skips to the end of its input string *x*, writes a separator symbol #, and then writes a certificate string, choosing up to $t(x)$ symbols to write nondeterministically. Then it returns the head to the starting position, the first symbol of *x*. In the second phase, *N* performs the same computation as `vL`(*x*, *y*), verifying the instance/certificate pair. Now $L(N) = L$, and *N* runs in polynomial time.

The NDTM constructed by this machine does all of its nondeterministic computation up front, capturing it in the certificate. Verifying the certificate is purely deterministic. The same idea works in reverse: given any NDTM *N*, we can construct a verification algorithm that works by capturing all the nondeterminism in the certificate. In fact, we've already seen the construction: it's our `runGuided` method.

Theorem 21.4.3: If *N* is a polynomial-time NDTM, $L(N) \in$ *NP*.

Proof: Suppose *N* is an NDTM. We construct a verification method `vL` using the `runGuided` method previously defined:

```
boolean vL(String x, String p) {
   return runGuided(N,x,p).equals("accept");
}
```

Here, the certificate is a guide string, and verifying it is simply a matter of checking that the computation it specifies is an accepting one. Since *N* is a polynomial-time NDTM, there is some polynomial-length guide string `p` for any `x` $\in L(N)$, and `runGuided` takes polynomial time on it. Thus, `vL` is an *NP*-verification method for *L,* and so $L \in$ *NP*.

Putting those results together, we see that NDTMs and verification methods are interchangeable ways to define the complexity class *NP*. ("There it is again!") In fact, *NP* was originally defined in terms of NDTMs; *NP* stands for Nondeterministic Polynomial time.

NDTMs can be used to define many other nondeterministic complexity classes, both for space and time: *NPSPACE*, *NEXPTIME*, and so on. But of all these, the most important open questions revolve around *NP*.

21.6 The Million-Dollar Question

We are now coming to the end of our study of complexity classes and to the end of this book. We have seen only a few of the most important complexity classes. Researchers have investigated the properties of hundreds of different classes, some more important than others, and there are many open problems concerning them. But a few problems involving the most important classes stand out as particularly tantalizing. We know for example that

$$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$$

And we know that $P \subset EXPTIME$, so at least one of the three inclusions above must be proper. But we don't know which one, or ones!

The one most important open problem in this field today is the question of *P* versus *NP*. Roughly speaking, the *P* class represents problems for which we can *find* the answer in polynomial time; the *NP* class represents problems for which we can *verify* a given answer in polynomial time. Our intuition tells us that finding an answer can be much more difficult than verifying one. Finding a proof feels harder than checking someone else's proof; solving a jigsaw puzzle takes hours of work, while checking someone's solution takes just a glance. (Same thing for Sudoku puzzles—and yes, decision problems related to Sudoku have been shown to be *NP*-complete.) To this intuition, we add a long experience of failure—the failure to find a polynomial-time solution to any of the *NP*-hard problems. Many of these problems are of great commercial importance, and many programmers over the last three decades have struggled with them, but to no avail.

For most researchers, this all adds up to a feeling that these problems are truly intractable, though we can't yet prove it. In fact, modern computing depends on this supposed intractability in many ways. For example, many of the cryptographic techniques that protect financial transactions on the Internet depend on this assumption. If a polynomial-time solution to an *NP*-hard problem were ever found, there would be quite a scramble to find other ways to secure network transactions!

But just because many researchers have this same feeling is no reason to consider the solution inevitable. The history of mathematics and computing contains a number of widely believed hunches and conjectures that turned out to be false. Indeed, we've seen a major one already in this book: in the early 1900s, most mathematicians shared David Hilbert's feeling that a proof of the completeness and correctness of number theory was just around the corner, until Kurt Gödel proved them all wrong. We've learned not to be too complacent; a consensus of hunches is no substitute for a proof.

All in all, it's a dramatic situation. There is a tension that has existed since the problem was first identified over 30 years ago, and it has grown as the list of *NP*-complete problems has grown. To resolve this growing tension a solid proof is needed: ideally, either a polynomial-time decision method for some *NP*-complete problem, or a proof that no such method exists. Certain fame awaits anyone who can solve this problem. Fame, and fortune: the *P* versus *NP* problem is one of seven key problems for which the Clay Mathematics Institute has offered prizes—$1 million for each problem. The prizes were announced in 2000, in Paris, 100 years after Hilbert announced his famous list of problems in that same city. So far, the prizes remain unclaimed.

21.7 Further Reading

An excellent discussion of *NP*, along with a very useful catalog of several hundred (seemingly) intractable problems, can be found in

Garey, Michael, and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W. H. Freeman and Company, 1979.

Exercises

EXERCISE 1

For each of the following verification methods, describe the language it defines, and write a decision method for it.

a.
```
boolean xxVerify(String x, String p) {
  return x.equals(p+p);
}
```

b.
```
boolean compositeVerify(int x, int p) {
 return p > 1 && p < x && x % p == 0;
}
```

c.
```
boolean vc(String x, String p) {
  return x.equals(p+p+p);
}
```

d.
```
boolean vd(String x, String p) {
  return x.equals(p);
}
```

e.
```
boolean ve(int x, int p) {
   return x = 2*p;
}
```

f.
```
boolean vf(int x, int p) {
   return x = p*p;
}
```

EXERCISE 2

Assuming that L is any *NP*-complete language, use the definitions to prove the following statements:

a. For all $A \in NP$, $A \leq_P L$.
b. For all A, if A is *NP*-hard then $L \leq_P A$.
c. For all A, if A is *NP*-complete then $A \leq_P L$.

d. For all A, if A is NP-complete then $L \leq_p A$.

e. If $L \in$ P, then all NP-complete languages are in P, and P = NP.

f. If $L \notin$ P, then no NP-complete language is in P, and P ≠ NP.

EXERCISE 3

Consider the following statement: For any NP-complete language L, and for all $A \in$ NP, $L \leq_p A$. Can you prove it true? Can you prove it false? How or why not?

EXERCISE 4

Here is an example of a concrete syntax for *SAT*.

```
S → &(S,S) | |(S,S) | !(S) | V
V → AV | A
A → a | b | … | z
```

Write a Java application `DecideSAT` that decides *SAT*. Your program should parse a Boolean formula from the command line using the syntax above. (The operators are all prefix and fully parenthesized in this syntax, so writing a recursive descent parser from this grammar is straightforward.) Then your program should decide whether the formula is satisfiable. If so, it should print true, followed by a satisfying assignment. If not (or if the formula is illegal) it should print false. For example:

```
> java DecideSAT '&(fred,ginger)'
True with [(fred=true)(ginger=true)]
> java DecideSAT '&(way,!(way))'
False
```

Feel free to implement this with a reasonable limit on the number of distinct variables—the exponential-time search will make giant formulas useless anyway.

EXERCISE 5

Convert these formulas to CNF. (That is, make an equivalent formula using exactly the same variables, but in conjunctive normal form.)

a. $\neg\neg a$

b. $\neg(a \vee b)$

c. $(a \vee b) \wedge \neg(a \wedge b)$

d. $\neg(a \wedge b \wedge \neg c)$

e. $a \vee (b \wedge c)$

f. $(a \wedge b) \vee (c \wedge d)$

g. $a \vee b \vee (c \wedge d)$

EXERCISE 6

(This exercise uses material from Appendix C, Section C.2.) Write a Java application `SAT2CNF` to convert *SAT* formulas to CNF. Your program should parse a Boolean formula from the command line using the syntax from Exercise 4. Then your program should convert the formula to CNF, using the transformations in the proof of Theorem C.1, and print out the resulting CNF formula. For example:

```
> java SAT2CNF '!(&(fred,ginger))'
|(!(fred),!(ginger))
> java SAT2CNF '|(&(a,b),c)'
&(|(a,c),|(b,c))
```

You should be able to make this work using only the transformations in the proof of the theorem: the two DeMorgan's laws, the double negative law, the distributive law, and the commuted version of the distributive law.

EXERCISE 7

(This exercise uses material from Appendix C, Section C.2.) Apply the transformation used in the proof of Lemma 21.2.2, converting the formulas from Exercise 5 into formulas in CNF. (Unlike Exercise 5, this will involve the addition of new variables.)

EXERCISE 8

(This exercise uses material from Appendix C, Section C.2.) Write a Java application `SAT2CNFP` to generate a *CNFSAT* instance from a *SAT* instance. Your program should parse a Boolean formula from the command line using the syntax from Exercise 4. Then your program should convert the formula to a *CNFSAT* instance, using the transformations in the proof of Lemma 21.2.2, and print out the resulting CNF formula. For example:

```
> java SAT2CNFP '!(&(fred,ginger))'
|(!(fred),!(ginger))
> java SAT2CNFP '|(&(a,b),c)'
&(&(|(#0,a),|(#0,b)),|(!(#0),c))
```

This is similar to Exercise 6, but not the same; you should not use the standard distributive law, but should use instead the special transformation from the proof of Lemma 21.2.2 and the commuted version of it, which introduce extra variables as necessary but achieve a polynomial-time reduction. (Our sample output above names the extra variables `#i`, but you can name them whatever you like.)

EXERCISE 9

Use the construction from the proof of Lemma 21.3.2 to convert these CNF formulas to 3-CNF. (This may involve the introduction of new variables.)

a. $\neg a$

b. $\neg(a \wedge \neg b \wedge c)$

c. $\neg(a \wedge b)$

d. $\neg(a \vee b)$

e. $(a \vee b) \wedge (c \vee d)$

f. $(a \vee b \vee c \vee d)$

g. $(a \vee b \vee c \vee d \vee e \vee f \vee g)$

EXERCISE 10

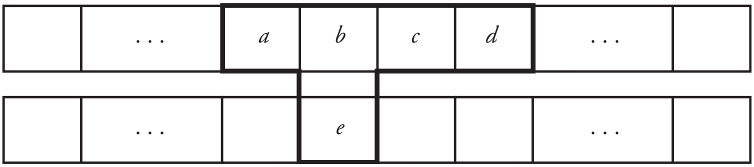
We define a concrete syntax for CNF formulas using the operator symbols $\vee$, $\wedge$, and $\neg$. There is exactly one pair of parentheses around each clause and no other parentheses in the formula. Variable names are strings of one or more lowercase letters.

a. Show that this language is regular by giving a left-linear grammar for it.

b. Show that the restriction of this language to 3-CNF is regular by giving a left-linear grammar for it.

EXERCISE 11

(This exercise uses material from Appendix C, Section C.1.) The proof of Cook's Theorem defines a function $e = f(a, b, c, d)$. This function defines a symbol in an ID for a Turing machine in terms of the previous-step symbols at and near that position:



Give pseudocode for computing this function f using the transition function δ for the TM.

EXERCISE 12

(This exercise uses material from Appendix C, Section C.3.) Convert $a \wedge \neg a$ to a *3SAT* instance, show the *Vertex Cover* instance constructed for it as in the proof of Lemma 21.4.2, and show a minimum-size vertex cover for it. (Because the formula is not satisfiable, you should find that there is no vertex cover of size $\leq 2m + n$.)

EXERCISE 13

Consider the following decision problem:

<u>*Monotone 3SAT*</u>

Instance: a formula in 3-CNF with the property that in each clause,
either all literals are positive or all are negative.

Question: is there some assignment of values to the variables that
makes the formula true?

For example, $(a \vee b \vee d) \wedge (\neg a \vee \neg b \vee \neg d)$ is a monotone formula, a legal instance of *Monotone 3SAT*. By contrast, $(\neg a \vee \neg b \vee d)$ is a *3SAT* formula but not a monotone one.

Prove that *Monotone 3SAT* is *NP*-complete. (*Hint:* To prove *NP*-hardness, use a reduction from *3SAT*. Your reduction will need to convert any 3-CNF formula ϕ into a monotone 3-CNF formula ϕ' such that ϕ is satisfiable if and only if ϕ' is. This will require the introduction of new variables.)

EXERCISE 14

Consider the following decision problem:

1-in-3SAT
Instance: a formula in 3-CNF.
Question: is there some assignment of values to the variables that
makes exactly one literal in each clause true?

Here, the instances are the same as for *3SAT*, but the question is different; we need to decide whether the formula has an assignment that makes *exactly* one literal in every clause true. (Our regular satisfiability problems ask whether there is an assignment that makes *at least* one literal in every clause true.)

Prove that *1-in-3SAT* is *NP*-complete. (*Hint:* To prove *NP*-hardness, use a reduction from *3SAT*. For each clause in the original formula, come up with a set of new clauses that have a 1-in-3 satisfying assignment if and only if the original clause has a plain satisfying assignment. This will require the introduction of new variables.)

EXERCISE 15

The pseudocode in the proof of Theorem 21.3 is inefficient. (We were only interested in showing polynomial time.) Show an implementation of minCoverSize that makes only logarithmically many calls to the decision method for *Vertex Cover*.

EXERCISE 16

The *Traveling Salesman Optimization* problem is this: given a graph G with a length for each edge, find a path that starts and ends at the same vertex, visits every vertex exactly once, and has a minimum total length. An approximate solution to this optimization problem would be an algorithm for finding a reasonably short circuit. We'll say that a circuit is reasonably short if it is no more than k times longer than the minimum-length path, for some constant $k > 1$. (Such a constant k is called a *ratio bound*.)

Now consider the following construction. Given a graph $G = (V, E)$ and a constant $k > 1$, construct a new graph G' with the same vertices. For each edge in G, add the same edge to G' with length 1. For each pair of vertices not connected by an edge in G, add that edge to G' with length $k|V|$. (Thus G' is a complete graph, with an edge of some length between every pair of vertices.)

a. Prove that if G has a Hamiltonian circuit then any *Traveling Salesman Optimization* solution for G' has a total length $|V|$.

b. Prove that if G does not have a Hamiltonian circuit then any *Traveling Salesman Optimization* solution for G' has a total length $> k|V|$.

c. Using these results, prove that for any constant $k > 1$, if there is an approximate solution for *Traveling Salesman Optimization* with a ratio bound k that runs in polynomial time, then *P* = *NP*.

(Note that our construction builds a graph G' with lengths that may violate the triangle inequality. For graphs that obey the triangle inequality, although *TSP* remains *NP*-complete, there are approximation techniques with good ratio bounds.)

EXERCISE 17

An NDTM N is *total* if, for all inputs $x \in \Sigma^*$, all legal sequences of moves either accept (by entering an accepting state) or reject (by entering a configuration from which there is no possible next move).

Using the ideas from the proof of Theorem 21.4.1, show how to construct a decision method `ldec` from any total NDTM N, so that $L($`ldec`$) = L(N)$.

EXERCISE 18

Using your solution to Exercise 17, show that *NP* ⊆ *PSPACE*. (You do not need to show implementations for things like `runGuided` and `BinaryStar`, but you should explain your assumptions about their space requirements.)

EXERCISE 19

Implement the `runGuided` method for NDTMs, as outlined in Section 21.5. Then, using that and your `ldec` method from Exercise 17, write a Java implementation of a universal Turing machine for NDTMs. Your program should read the NDTM to be simulated from a file. You can design your own encoding for storing the NDTM in a file, but it should be in a text format that can be read and written using an ordinary text editor. Your class should have a main method that reads a file name and a text string from the command line, uses your `ldec` method to decide whether the string is in the language defined by the NDTM, and reports the result. For example, if the file `anbncn.txt` contains the encoding of an NDTM for the language $\{a^n b^n c^n\}$, then your program would have this behavior:

```
> java UNDTM anbncn.txt abca
reject
> java UNDTM anbncn.txt abc
accept
```

If you did Exercise 13 from Chapter 17 (a Java implementation of a universal TM for deterministic machines) you should be able to reuse much of that code.

A

From an NFA to a Regular Expression

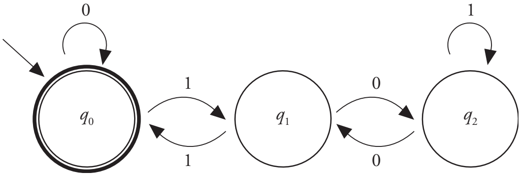
For every NFA, there is an equivalent regular expression. In Chapter 7 we showed an example of how to construct a regular expression for an NFA, but skipped the general construction and proof. In this appendix we tackle the proof. Warning: There are mathematical proofs whose elegance compels both agreement and admiration. They are so simple and concise that they can be understood in their entirety, all at once. Unfortunately, this is not one of them!

A.1 The Internal Languages of an NFA

When you see an NFA, you normally think only of the language it accepts: the language of strings that take it from the start state to end in any accepting state. But to construct a regular expression from an NFA, as demonstrated in Section 7.5, we need to consider other languages relevant to a given NFA.

Let M be any NFA with n states numbered q_0 through q_{n-1}. Define the *internal language* $L(M, i, j, k) = \{x \mid (q_i, x) \mapsto^* (q_j, \varepsilon)$, by some sequence of IDs in which no state other than the first and the last is numbered less than $k\}$.

$L(M, i, j, k)$ is the set of all strings that take the NFA from the state i to the state j, without passing through any state numbered less than k. For example, let M be this DFA:



- $L(M, 2, 2, 3)$ is the set of strings x for which $(q_2, x) \mapsto^* (q_2, \varepsilon)$, where states other than the first and the last are all numbered 3 or higher. Since there are no states in M numbered 3 or higher, there can be no IDs in the sequence other than the first and last. So there are only two strings in this language: ε (which produces the one-ID sequence (q_2, ε)) and 1 (which produces the two-ID sequence $(q_2, 1) \mapsto (q_2, \varepsilon)$). Thus $L(M, 2, 2, 3) = L(1 + \varepsilon)$.
- $L(M, 2, 2, 2)$ is the set of strings x for which $(q_2, x) \mapsto^* (q_2, \varepsilon)$, where states other than the first and the last are all numbered 2 or higher. Since q_2 is the only state that can appear in the sequence, the strings in this language are just strings of any number of 1s. Thus $L(M, 2, 2, 2) = L(1^*)$.
- $L(M, 2, 1, 2)$ is the set of strings x for which $(q_2, x) \mapsto^* (q_1, \varepsilon)$, where states other than the first and the last are all numbered 2 or higher. Since q_2 is the only state other than the last that can appear in the sequence, the strings in this language are just strings of any number of 1s with a single 0 at the end. Thus $L(M, 2, 1, 2) = L(1^*0)$.
- $L(M, 0, 0, 0)$ is the set of strings x for which $(q_0, x) \mapsto^* (q_0, \varepsilon)$, where states other than the first and the last are all numbered 0 or higher. Since all states in M are numbered 0 or higher, this is just the language of strings that take M from its start state to its accepting state. Thus, for this machine, $L(M, 0, 0, 0) = L(M)$.

A.2 The d2r Function

Next we will write a recursive function d2r(M, i, j, k) that constructs a regular expression for any internal language $L(M, i, j, k)$. Later, this will be used in the construction of a regular expression for the entire language $L(M)$.

First, consider the cases of $L(M, i, j, k)$ where k is greater than or equal to the number of states in M. Such internal languages are easy to give regular expressions for because, like the first example above, they do not permit any sequences other than the one-transition sequence q_iq_j and, in the case of $i = j$, the zero-transition sequence $q_{i.}$ So let $r = a_1 + a_2 + \ldots$, for all $a_p \in \Sigma \cup \{\varepsilon\}$ with $q_j \in \delta(q_i, a_p)$. (In case there is no such a_p, let $r = \varnothing$.) Then the regular expression for $L(M, i, j, k)$, where k is greater than or equal to the number of states in M, is either r (if $i \neq j$) or $r + \varepsilon$ (if $i = j$). This will be the base case in our recursive definition of the function d2r. It is a base case that applies to any call with a sufficiently large value of k. As we look at the recursive cases below, bear this in mind: recursive calls that use *increasing* values of k make progress toward this base case.

Now consider the cases of $L(M, i, j, k)$ where k is less than the number of states in M. These are by definition strings that can take M through a sequence of one or more states starting from q_i and ending in q_j, where states other than the first and last are all numbered k or higher. These strings are permitted to take M through state q_k, but are not required to, so they can be divided into two cases: those that have sequences that actually do visit q_k, and those that have sequences that don't. (Because M may be nondeterministic, some strings may fall into both cases.) Those that have sequences that don't visit q_k are in $L(M, i, j, k + 1)$, so we will be able to generate regular expressions for them recursively, as r_1 = d2r(M, i, j, $k + 1$).

It remains to consider those strings that have sequences of moves that actually do visit q_k at least once. These sequences start with q_i, reach q_k one or more times, and eventually end in q_j. Picture such a sequence with all its occurrences of q_k made explicit:

$$q_i \cdots q_k \cdots q_k \cdots \cdots q_k \cdots q_j$$

Such a sequence can be treated in three parts: first a sequence from q_i to q_k; then zero or more sequences from q_k back to q_k; and finally a sequence from q_k to q_j. Because we have made all the occurrences of q_k explicit endpoints of these subsequences, q_k does not occur as an intermediate state in any of the subsequences. *All of the subsequences use only intermediate states numbered* $k + 1$ *or higher*; so we will be able to generate regular expressions for them recursively. For sequences from q_i to q_k, we can use r_2 = d2r(M, i, k, $k + 1$); for sequences from q_k to q_k, we can use r_3 = d2r(M, k, k, $k + 1$); and for sequences from q_k to q_j, we can use r_4 = d2r(M, k, j, $k + 1$). The whole regular expression for $L(M, i, j, k)$, where k is less than the number of states in M, can then be written as $r_1 + (r_2)(r_3)^*(r_4)$. The resulting implementation for d2r is

```
d2r($M$, $i$, $j$, $k$)
    if $k \geq$ (number of states in $M$) then
            let $r = a_1 + a_2 + \ldots$, for all $a_p \in \Sigma \cup \{\varepsilon\}$ with $q_j \in \delta(q_i, a_p)$,
                    or $\varnothing$ if there are no transitions from $q_i$ to $q_j$
            if $i = j$ then return $r + \varepsilon$ else return $r$
    else
            let $r_1$ = d2r($M$, $i$, $j$, $k + 1$)
            let $r_2$ = d2r($M$, $i$, $k$, $k + 1$)
            let $r_3$ = d2r($M$, $k$, $k$, $k + 1$)
            let $r_4$ = d2r($M$, $k$, $j$, $k + 1$)
            return $r_1 + (r_2)(r_3)^*(r_4)$
```

This is a complete procedure for constructing a regular expression for any internal language of an NFA.

A.3 Proof of Lemma 7.2

Now we can restate and prove Lemma 7.2.

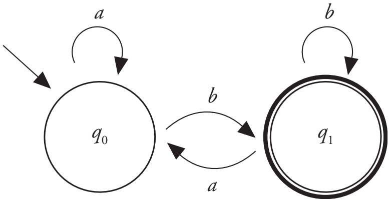
Lemma 7.2: If N is any NFA, there is some regular expression r with $L(N) = L(r)$.

Proof: By construction. Let $M = (Q, \Sigma, \delta, q_0, F)$ be any NFA. We have a recursive function, d2r, that constructs a regular expression for any internal language $L(M, i, j, k)$. $L(M)$ is the language of strings that take M from q_0 to end in any accepting state, using any intermediate states. Thus

$$\mathrm{L}(M) = \bigcup_{q_j \in F} L(M,0,j,0)$$

So to construct a regular expression for $L(M)$, compute d2r(M, 0, j, 0) for each $q_j \in F$, then combine them all with + into one, big, regular expression. (In the case of $F = \{\}$, use $\varnothing$ as the regular expression.) The resulting regular expression r has $L(r) = L(M)$.

For a full example of this construction, consider this NFA:



There is only one accepting state, q_1, so $L(M) = L(M, 0, 1, 0)$. The evaluation of $\mathrm{d2r}(M, 0, 1, 0)$ uses the recursive case, returning $r_1 + (r_2)(r_3)^*(r_4)$, where

$$
\begin{aligned}
r_1 &= \mathrm{d2r}(M, 0, 1, 1)\\
r_2 &= \mathrm{d2r}(M, 0, 0, 1)\\
r_3 &= \mathrm{d2r}(M, 0, 0, 1)\\
r_4 &= \mathrm{d2r}(M, 0, 1, 1)
\end{aligned}
$$

Now the evaluation of $\mathrm{d2r}(M, 0, 1, 1)$, for both r_1 and r_4, uses the recursive case, returning $r_5 + (r_6)(r_7)^*(r_8)$, where

$$
\begin{aligned}
r_5 &= \mathrm{d2r}(M, 0, 1, 2) = b\\
r_6 &= \mathrm{d2r}(M, 0, 1, 2) = b\\
r_7 &= \mathrm{d2r}(M, 1, 1, 2) = b + \varepsilon\\
r_8 &= \mathrm{d2r}(M, 1, 1, 2) = b + \varepsilon
\end{aligned}
$$

Those four are all base cases, as shown. Finally the evaluation of $\mathrm{d2r}(M, 0, 0, 1)$, for both r_2 and r_3, uses the recursive case, returning $r_9 + (r_{10})(r_{11})^*(r_{12})$, where

$$
\begin{aligned}
r_9 &= \mathrm{d2r}(M, 0, 0, 2) = a + \varepsilon\\
r_{10} &= \mathrm{d2r}(M, 0, 1, 2) = b\\
r_{11} &= \mathrm{d2r}(M, 1, 1, 2) = b + \varepsilon\\
r_{12} &= \mathrm{d2r}(M, 1, 0, 2) = a
\end{aligned}
$$

Those are also base cases, so we can now assemble the whole expression, omitting some unnecessary parentheses to make the result slightly more readable:

$$
\begin{aligned}
&\mathrm{d2r}(M, 0, 1, 0)\\
&\quad = r_1 + r_2 r_3{}^* r_4\\
&\quad = r_5 + r_6 r_7{}^* r_8 + r_2 r_3{}^*(r_5 + r_6 r_7{}^* r_8)\\
&\quad = r_5 + r_6 r_7{}^* r_8 + (r_9 + r_{10} r_{11}{}^* r_{12})(\ r_9 + r_{10} r_{11}{}^* r_{12})^*(r_5 + r_6 r_7{}^* r_8)\\
&\quad = b + b(b + \varepsilon)^*(b + \varepsilon)\\
&\qquad\quad + (a + \varepsilon + b(b + \varepsilon)^* a)(a + \varepsilon + b(b + \varepsilon)^* a)^*(b + (b + \varepsilon)^*(b + \varepsilon))
\end{aligned}
$$

The language accepted by the NFA in this example is the language of strings over the alphabet $\{a, b\}$ that end with b; a simple regular expression for that language is $(a + b) * b$. It is true, but not at all obvious, that this short regular expression and the long one generated by the formal construction are equivalent. As you can see, the regular expression given by the construction can be very much longer than necessary. But that is not a concern here. The point of Lemma 7.2 is that the construction can always be done; whether it can be done concisely is another (and much harder) question. It can always be done—so we know that there is a regular expression for every regular language.

B

A Time-Hierarchy Theorem

The more time you allow for a decision, the more languages you can decide. It's not a surprising claim, perhaps, but it isn't easy to prove.

B.1 Another Asymptotic Notation: Little o

To begin, we need to state precisely what "more time" means. To do that, we'll introduce a new asymptotic notation, using little o instead of big O.

Let f and g be any two functions over $\mathcal{N}$. We say

$$f(n) \text{ is } o(g(n))$$

if and only if for every $c > 0$ there exists some n_0, so that for every $n \geq n_0$, $f(n) \leq \frac{1}{c} g(n)$.

To say that $f(n)$ is $o(g(n))$ is to say that f grows more slowly than g. That's stricter than big O; little o rules out the possibility that f and g grow at the same rate. Where big O is like ≤, little o is like <. That will serve as our definition of "more time": $g(n)$ allows more time than $f(n)$ if $f(n)$ is $o(g(n))$.

In Chapter 19, Theorem 19.2 summarized the comparative rates of growth of some common asymptotic functions. It turns out that all the results given there for big O can be tightened to little o. We can state the following:

Theorem B.1:

- For all positive real constants a and b, a is $o((\log n)^b)$.
- For all positive real constants a and b, and for any logarithmic base, $(\log n)^a$ is $o(n^b)$.
- For all real constants $a > 0$ and $c > 1$, n^a is $o(c^n)$.

B.2 Time-Constructible Functions

Next, we'll need to restrict the claim to complexity functions that can be implemented reasonably efficiently.

A function f is *time constructible* if there exists some method

```
      int fval(int n)
```

with `fval(n)` = f(`n`), where $time$(`fval`, n) is $O(f(n))$.

A function f is time constructible if we can compute what $f(n)$ is within $O(f(n))$ time. (This use of a time bound is unusual in that it measures time as a function of the input number, not as a function of the size of the representation of that number.) For example, the function n^2 is time constructible, because we can implement it as

```
int nSquared(int n) {
   return n * n;
}
```

This computes n^2 and takes $\Theta(\log n)$ time, which is certainly $O(n^2)$. Most commonly occurring functions, like n, $n \log n$, n^k, and 2^n, are easily shown to be time constructible. (In fact, it is challenging to come up with a function that is $\Omega(n)$ but *not* time constructible.)

B.3 A Cost Model for `run` and `runWithTimeLimit`

When we introduced the methods `run` and `runWithTimeLimit` back in Chapter 18, we were concerned only with computability, not efficiency. Now, to use these methods to establish a time-hierarchy theorem, we must add some assumptions about them to our cost model.

First, the `run` method: we do not model a separate compilation step, but treat all execution as interpreting the source code. If `m` is a method and `mSource` is a string containing the source code for `m`, it makes no difference whether we write `m(in)` or `run(mSource,in)`—they do the same thing and are assumed to take asymptotically the same space and time.

In Chapter 18 we said that `runWithTimeLimit(p,in,n)` returns the same thing `run(p,in)` returns, if that completes within `n` steps, but returns false otherwise. We left the concept of a runtime "step" undefined; now we stipulate that it must be some constant-time unit of computation, according to our cost model. There is some constant $c \in \mathcal{N}$, $c > 0$, such that, if *time*(`run`, (`p`, `in`)) $\le c$ `n`, then `runWithTimeLimit(p,in,n) = run(p,in).`

Simply running a method `p` for `n` steps takes Θ(`n`) time. However, `runWithTimeLimit(p,in,n)` takes more than Θ(`n`) time, because of the overhead involved in enforcing the time limit. Think of it as a countdown: after each step, `runWithTimeLimit` decrements the original counter `n`, and stops the execution (returning false) if that counter ever reaches 0. This decrement-and-test after each step takes O(*log* `n`) time. Thus, in the worst case, `runWithTimeLimit(p,in,n)` takes Θ(`n` *log* `n`) time.

B.4 A Time-Hierarchy Theorem

Now we can state a time-hierarchy theorem—a formal version of our claim that the more time you allow for a decision, the more languages you can decide.

Theorem B.2 (Time Hierarchy): If g is time constructible and $f(n)$ is $o(g(n))$, then $\textit{TIME}(f(n)) \subset \textit{TIME}(g(n) \log n)$.

Roughly speaking, this says that any more than an extra log factor of time allows you to decide more languages. Of course, you can still decide all the languages you could have decided without that extra log factor; clearly $\textit{TIME}(f(n)) \subseteq \textit{TIME}(g(n) \log n)$. The tricky part is to show that the inclusion is proper—that there exists at least one language

$A \in \textit{TIME}(g(n) \log n)$ that is not in $\textit{TIME}(f(n))$. Such a language A is the one decided by this method:

```
1.  boolean aDec(String p) {
2.     String rejectsItself =
3.       "boolean rejectsItself(String s) {" +
4.       "    return !run(s,s);             " +
5.       "}                                 ";
6.     int t = gVal(p.length());
7.     return runWithTimeLimit(rejectsItself,p,t);
8.  }
```

Here, `gVal` is a method that computes $g(n)$ and does it in $O(g(n))$ time; we know that such a method exists because g is time constructible.

Let A be the language decided by `aDec`. Because `gVal` is the method given by the time constructability of g, line 6 runs in $O(g(n))$ time. Line 7 requires $O(g(n) \log g(n))$ time, so the whole method `aDec` requires $O(g(n) \log g(n))$ time. This shows $A \in \textit{TIME}(g(n) \log g(n))$. It remains to show that $A \notin \textit{TIME}(f(n))$.

Toward that end, we prove a lemma about the method `aDec`:

Lemma B.1: For any language in $\textit{TIME}(f(n))$, there is some decision method string `p` for which `aDec(p) = !run(p,p)`.

Proof: For any decision method string `p`, two conditions are sufficient to establish that `aDec(p) = !run(p,p)`: first, that `p` is $O(f(n))$, and second, that `p` is sufficiently long. For if `p` is $O(f(n))$, and given that $f(n)$ is $o(g(n))$, then *time*(`run, (rejectsItself,p)`) is $o(g(|\texttt{p}|))$. This means that there exists some n_0 such that for every $|\texttt{p}| \geq n_0$,

time(`run, (rejectsItself,p)`) $\leq g(|\texttt{p}|)$.

Because $g(|\texttt{p}|)$ = `t`, we can conclude that when `p` is sufficiently long, the `runWithTimeLimit` at line 7 does not exceed its time limit. Therefore it produces the same result as a plain `run`:

```
runWithTimeLimit(rejectsItself,p,t)
   = run(rejectsItself,p)
   = !run(p,p).
```

Now let B be any language in $\textit{TIME}(f(n))$. By definition, there must exist an $O(f(n))$ decision method for B. Its source code may not have $|\texttt{p}| \geq n_0$, but it is easy to construct a sufficiently long equivalent method, simply by appending spaces. Thus, B always has some decision method string `p` that meets our two conditions, so that `aDec(p) = !run(p,p).`

Lemma B.1 shows that for every decision method for a language in $\mathit{TIME}(f(n))$, there is at least one string `p` about which `aDec` makes the opposite decision. It follows that the language A, the language decided by `aDec`, is not the same as any language in $\mathit{TIME}(f(n))$. Therefore $A \notin \mathit{TIME}(f(n))$, and this completes the proof of Theorem B.2.

The proof of this time-hierarchy theorem uses a variation of the same technique used in Section 18.2 to prove that L_u is not recursive. Note the similarity between the `rejectsItself` method used in this proof and the `nonSelfAccepting` method used there.

B.5 Application

The time-hierarchy theorem is a tool for proving that one complexity class is a proper subset of another. For example, we can use it to prove that $\mathit{TIME}(2^n)$ is a proper subset of $\mathit{TIME}(2^{2n})$. To apply the theorem, we choose $f(n) = 2^n$ and $g(n) = 2^{2n} / \log_2 n$. (Note the separation here: we choose a g that is at least a log factor below the time-complexity function we're ultimately interested in.) First, to invoke the time-hierarchy theorem, we need to know that g is time constructible. Next, we need to know that $f(n)$ is $o(g(n))$. By definition, this is true if for every $c > 0$ there exists some n_0, so that for every $n \geq n_0$, $2^n \leq \frac{1}{c} 2^{2n} \big/ \log_2 n$. Simplifying, we see that this inequality is satisfied whenever we have $2^n / \log_2 n \geq c$, which is clearly true for sufficiently large n, for any constant c. So we can apply the time-hierarchy theorem and conclude that $\mathit{TIME}(f(n)) \subset \mathit{TIME}(g(n) \log n)$. Substituting back in for f and g, we conclude that $\mathit{TIME}(2^n)$ is a proper subset of $\mathit{TIME}(2^{2n})$.

This time-hierarchy theorem cannot be used to prove that time-complexity classes are distinct when they differ by exactly one log factor. For example, though it is true that $\mathit{TIME}(n)$ is a proper subset of $\mathit{TIME}(n \log n)$, our time-hierarchy theorem cannot be used to prove it. The theorem kicks in only when the complexity functions differ by more than a log factor, in the little-o sense.

C

Some NP-Hardness Proofs

This appendix presents the proofs of some of the trickier NP-hardness results from Chapter 21.

C.1 Cook's Theorem: *SAT* Is *NP*-hard

This is one of the most important results in complexity theory. The details of the proof are too hairy even for this appendix, but the following proof sketch should give you a good idea of how it can be done.

Lemma 21.1.2 (Cook's Theorem): *SAT* is *NP*-hard.

Proof sketch: Let L be an arbitrary language in *NP*. We need to show that $L \leq_P SAT$. In other words, we need to show that there is a polynomial-time function f that maps any string x into a Boolean formula $f(x)$, with the property that $x \in L$ if and only if $f(x) \in SAT$. That sounds like a tall order—knowing almost nothing about a language L, we have to show how to map its decision problem to the *SAT* decision problem.

The one thing we do know about L is that it is in *NP*. Therefore, there exists some polynomial-time, verification algorithm for L, using polynomial-size certificates. Now, the proof is simpler if we take this algorithm to be given as a Turing machine M. Then we can be a little more specific about the mapping function f we're looking for; $f(x)$ should be satisfiable if and only if the TM M accepts the string $x\#y$ for some certificate string y.

We know that M is an *NP*-verifier, so if it has not accepted within some polynomially bounded number of steps, it never will. Thus, for each string x we can precompute a number $t(x)$, depending only on the length of x, to serve as a strict bound on the number of steps. Using this information, we can be still more specific about the mapping function f we're looking for; $f(x)$ should be satisfiable if and only if M accepts $x\#y$ for some certificate string y, within $t(x)$ moves.

We can model any such accepting computation as a sequence of exactly $t(x)$ IDs, on the understanding that once M halts, the subsequent IDs in this sequence are unchanged. We can also make each ID in this sequence exactly $2t(x)$ symbols wide, padding with blanks as necessary and placing the initial head position in the center. (We know that no ID needs to be wider than $2t(x)$, because the head can't make more than $t(x)$ moves to the left or right.) If you lay out such a sequence of $t(x)$ IDs, each of size $2t(x)$, you get a grid like this:

	1	2														2t(x)
1	**B**	. . .	**B**	q_0	x_1	. . .	x_2	#	y_1	. . .	y_2	**B**	. . .	**B**
2		. . .				. . .				. . .			. . .	
		. . .				. . .				. . .			. . .	
		. . .				. . .				. . .			. . .	
	⋮													⋮
t(x)		. . .				. . .				. . .			. . .	

In terms of this grid, we can be still more specific about the mapping function *f* we're looking for; *f*(*x*) should be satisfiable if and only if the grid can be filled so that

1. the first line is a starting ID of *M* on *x* (for some *y*), as shown in the illustration above,
2. each line follows from the line before it according to the transition function of *M* (or is the same as the line before it, when *M* has halted), and
3. the final row is an ID in an accepting state.

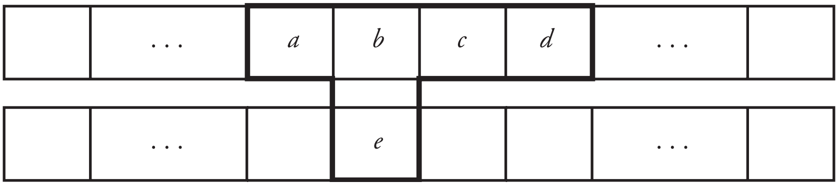
Now we're getting close to something we can express as a Boolean formula. The one remaining snag is the alphabet; cells in the grid contain ID symbols, which are either tape symbols or states, but the only values we can talk about in a Boolean formula are Boolean. No problem: we simply encode each value in $\Gamma \cup Q$ using a fixed-length string over the binary alphabet {T, F}. Let *d* be the length of the string of Ts and Fs used to encode each cell (presumably, $d = \lceil log_2 |Q \cup \Gamma| \rceil$). Then the (*i*, *j*)th cell of the grid can be encoded using *d* Boolean variables, $b_{i,j,1}$ through $b_{i,j,d}$. The whole encoding of the grid requires $d \cdot 2t(x)^2$ such variables—a number which is polynomial in |*x*|.

Now our mapping function *f*(*x*) can construct a big formula using these Boolean variables. The formula is a conjunction of three parts, encoding requirements 1 through 3 above.

$$f(x) = starts \wedge moves \wedge accepts$$

Here's where it gets technical, and we'll skip these details. Roughly speaking:

1. *starts* enforces the requirement that the first line of the grid is a starting ID of M on x. This establishes some of the values $b_{1,j}$ so that the start state is q_0, the tape to the left is all **B**s, and the tape at and to the right of the head contains $x\#$. It leaves the y part, the certificate, unconstrained; ultimately, our formula will be satisfiable if there is some y part that makes the whole system true.
2. *moves* enforces the requirement that each line follow from the next according to M's transition function. In fact, because of the way a Turing machine works, each individual cell $b_{i+1,\mathrm{j}}$ is a simple function of its near neighbors in the previous row, $b_{i,j\text{-}1}$ through $b_{i,j+2}$:



   This function $e = f(a, b, c, d)$ is determined by the transition function of M. (When this move might affect e, one of the inputs is a state, showing the current state and head position. If none of the inputs is a state, the head is not near enough to have any effect on this cell, and so $f(a, b, c, d) = b$.) With the inputs and outputs encoded as Boolean values, this function has a fixed, finite number of Boolean inputs and outputs, and so can be encoded as a Boolean formula. The conjunction of all those Boolean formulas, one for every cell in the grid except for the first row, gives a formula that is satisfiable if and only if the whole grid correctly follows from the first row according to the transition function of M.
3. *accepts* enforces the requirement that the final row shows M in an accepting state. At least one of the $b_{t(x),j}$ encodes an accepting state.

   The resulting formula $f(x)$ is polynomial in the size of the grid, which is polynomial in the size of x. The *starts* part determines every cell of the first row, except the certificate y. For any choice of certificate y, the *moves* part fully determines every other cell in the grid. And the *accepts* part is satisfied if and only if M accepts $x\#y$. Therefore, the whole formula $f(x)$ is satisfiable if and only if there is some certificate y for which M accepts $x\#y$. Therefore, $x \in L$ if and only if $f(x) \in SAT$, and so $L \leq_{\mathrm{P}} SAT$. Because L is an arbitrary language in NP, this proves that SAT is NP-hard.

C.2 *CNFSAT* Is NP-hard

For proving that *CNFSAT* is NP-hard, there's good news and bad news. The good news is that we don't have to repeat all the work of the previous NP-hardness proof. If we can just prove that *SAT* $\leq_P$ *CNFSAT*, that will be enough; we've already proved that for every $L \in NP$, $L \leq_P$ *SAT*, so by transitivity we'll have $L \leq_P$ *CNFSAT*.

The bad news is that the polynomial-time reduction from *SAT* to *CNFSAT* isn't as simple as it first appears. There's a tempting approach that doesn't quite work:

> **Theorem C.1:** For any *SAT* formula, there is an equivalent formula in CNF.
>
> **Proof sketch:** Any *SAT* formula can be transformed to CNF by using simple laws of Boolean logic. First, all the ¬ operators can be moved inward until they apply directly to variables, thus forming the literals for CNF. For this we can use DeMorgan's laws and the law of double negation:
>
> - $\neg(x \wedge y) = (\neg x \vee \neg y)$
> - $\neg(x \vee y) = (\neg x \wedge \neg y)$
> - $\neg\neg x = x$
>
> Then, all the $\vee$ operators can be moved inward until they apply directly to literals, thus forming the clauses for CNF. For this we can use the distributive law:
>
> - $(x \wedge y) \vee z = (x \vee z) \wedge (y \vee z)$
>
> Repeated application of this law (and the commuted version, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$) completes the transformation to CNF.

This is only a proof sketch; a more complete proof would have to define the structure of SAT instances more closely and then use a structural induction to demonstrate that the transformations proposed actually do handle all possible *SAT* instances. (You can experiment with this for yourself; see Exercise 6 in Chapter 21.)

The theorem shows that any *SAT* instance ϕ can be converted into an equivalent *CNFSAT* instance ϕ'. Then, of course, $\phi \in$ *SAT* if and only if $\phi' \in$ *CNFSAT*. So why doesn't this prove that *SAT* $\leq_P$ *CNFSAT*? The problem is that, although it is a reduction from *SAT* to *CNFSAT*, it is not always a *polynomial-time* reduction. In particular, that distributive law is trouble. It makes duplicate terms; as written above, the right-hand side contains two copies of the term *z*. For formulas that require this operation repeatedly, the result can be an exponential blow-up in the size of the formula (and in the time it takes to construct it). Consider a term of this form:

$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \ldots \vee (a_k \wedge b_k)$$

for some *k*. After the conversion, the *CNF* formula will have one clause for each way to choose one variable from every pair. For example:

$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) =$$

$$(a_1 \vee a_2) \wedge (a_1 \vee b_2) \wedge (b_1 \vee a_2) \wedge (b_1 \vee b_2)$$

$$\begin{aligned}&(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3) = \\ &\quad (a_1 \vee a_2 \vee a_3) \wedge (a_1 \vee a_2 \vee b_3) \wedge (a_1 \vee b_2 \vee a_3) \wedge (a_1 \vee b_2 \vee b_3) \wedge \\ &\quad (b_1 \vee a_2 \vee a_3) \wedge (b_1 \vee a_2 \vee b_3) \wedge (b_1 \vee b_2 \vee a_3) \wedge (b_1 \vee b_2 \vee b_3)\end{aligned}$$

The length of the original *SAT* formula is linear in k, while the number of clauses in the CNF formula is 2^k. So, although our Theorem C.1 shows that every formula has an equivalent CNF formula, it does not give us a polynomial-time reduction from *SAT* to *CNFSAT*.

To show *SAT* $\leq_P$ *CNFSAT*, we need a more subtle reduction, one that does not rely on converting each *SAT* instance ϕ into an equivalent CNF formula ϕ'. Luckily, we don't really need the two formulas to be equivalent. We need something rather more flexible: that ϕ' is satisfiable if and only if ϕ is. As long as that is true, the two formulas can be quite different; in particular, they can use different variables. That's the leverage we need to make a polynomial-time reduction.

Lemma 21.2.2: *CNFSAT* is *NP*-hard.

Proof sketch: We will show *SAT* $\leq_P$ *CNFSAT*. From any *SAT* formula ϕ we can construct a CNF formula ϕ' so that ϕ is satisfiable if and only if ϕ' is. First, all the $\neg$ operators can be moved inward until they apply directly to variables, thus forming the literals for CNF, as in the proof of Theorem C.1. These transformations preserve satisfiability (and in fact produce an equivalent formula).

Next, all the $\vee$ operators can be moved inward until they apply directly to literals, thus forming the clauses for CNF. For this we use a special variation of the distributive law, one that introduces a new variable d_i:

$$(x \wedge y) \vee z = (d_i \vee x) \wedge (d_i \vee y) \wedge (\neg d_i \vee z)$$

(Each application of this law generates a new variable d_i, distinct from any other variable in ϕ.) The result of applying this law is no longer an equivalent formula, because it contains a new variable. However, suppose the left-hand side is satisfiable. Then there is some truth assignment that makes either $x \wedge y$ or z true. We can extend this truth assignment to one that makes the right-hand side true, by making $d_i = z$. Conversely, suppose the right-hand side is satisfiable; then there is some truth assignment that makes all three clauses true. This assignment must make either d_i or $\neg d_i$ false; therefore it must make either $x \wedge y$ or z true. So it is also a satisfying truth assignment for the left-hand side. Therefore, this transformation preserves satisfiability.

Repeated application of this law (and the commuted version, $x \vee (y \wedge z) = (d_i \vee x) \wedge (\neg d_i \vee y) \wedge (\neg d_i \vee z)$) completes the construction of a CNF formula ϕ' that is satisfiable if and only if the original formula ϕ is satisfiable. The entire construction takes polynomial time, so we conclude that *SAT* $\leq_P$ *CNFSAT*. Because *SAT* is NP-hard, we conclude that *CNFSAT* is NP-hard.

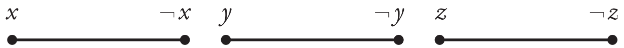
This is only a proof sketch; a more complete proof would have to define the structure of *SAT* instances more closely and then use a structural induction to demonstrate that the transformations proposed actually do convert all possible *SAT* instances and can be implemented to do it in polynomial time. (You can experiment with this for yourself; see Exercise 8 in Chapter 21.)

C.3 *Vertex Cover* Is NP-hard

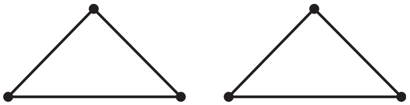
Before we get to this proof, let's look at an example of the construction we'll use. Suppose our formula is this:

$$\phi = (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$$

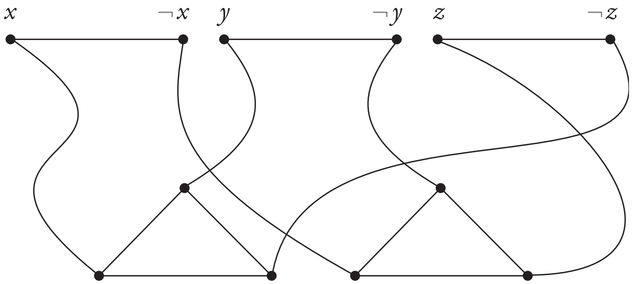
This formula has $n = 3$ variables and $m = 2$ clauses. For each variable, we'll create two vertices, one for the positive literal and one for the negative, with an edge between them, like this:



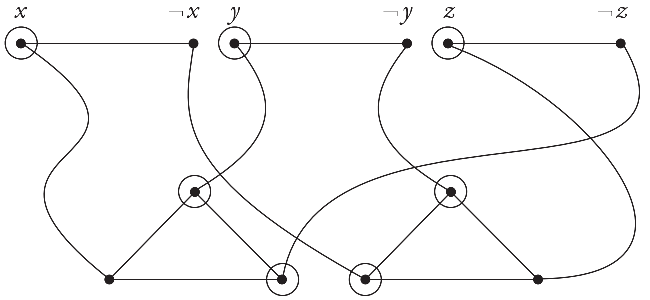
Then, for each clause, we'll add three vertices connected in a triangle. Our example has two clauses, so we'll add this:



Finally, we'll add edges from each triangle vertex to the vertex for the corresponding literal in that clause. For instance, the first clause is $(x \vee y \vee \neg z)$, so our first triangle will get edges from its three corners to the three vertices for x, y, and $\neg z$. We end up with this graph:



Now consider a minimum-size vertex cover for this graph. It must include two of the three vertices from each triangle—there's no other way to cover those three triangle edges. It must also include one vertex for each variable, the positive or negative literal, in order to cover those edges at the top of the illustration. So no cover can contain fewer than $2m + n = 7$ vertices. Here's an example at that minimum size, with the covered vertices circled:

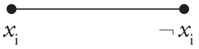


The covered literals give us a satisfying assignment for ϕ: we covered x, y, and z, so let x = true, y = true, and z = true. By construction, that makes at least one literal in each clause true. In fact, any cover of the size $2m + n$ gives a satisfying truth assignment, and any satisfying truth assignment gives a cover of the size $2m + n$. This correspondence is the main idea in the following proof.

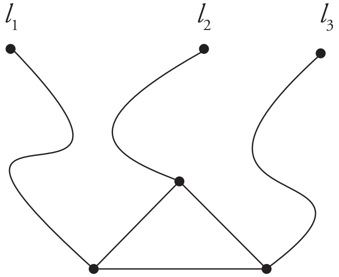
Lemma 21.4.2: *Vertex Cover* is NP-hard.

Proof: By reduction from *3SAT*. Let ϕ be any 3-CNF formula. Let m be the number of clauses in it, and let n be the number of variables. We will construct a graph G_ϕ, as follows.

First, for each of the n variables x_i, create a *variable component*, consisting of two vertices, labeled x_i and $\neg x_i$, with an edge between them:



This gives us a vertex for each possible literal. Next, for each of the m clauses $(l_1 \vee l_2 \vee l_3)$, create a *clause component*: three vertices connected in a triangle, and three *connecting edges* (the curved lines below) from each to the corresponding vertex already created for that literal:



That completes the construction of the graph G_ϕ. To make a complete *Vertex Cover* instance we also need a constant k; choose $k = 2m + n$. The size of this instance and the time to construct it are clearly polynomial in the size of ϕ.

Now we claim that ϕ is satisfiable if and only if G_ϕ has a vertex cover of size k or less. First, suppose ϕ is satisfiable. Then we can use a satisfying assignment to construct a vertex cover V'. First, from each variable component, add either x_i or $\neg x_i$ to V', whichever is true under the satisfying assignment. These vertices cover all the edges in the variable components. They also cover at least one of the connecting edges for each clause component—because, in a satisfying assignment, there is at least one true literal in each clause. For each clause component, select one connecting edge that is already covered (there may be more than one), and add the other two triangle vertices to V'. This covers the three triangle edges and the other two connecting edges. Thus all the edges in the graph are covered. We used a vertex for each variable, plus two for each clause: $|V'| = k = 2m + n$.

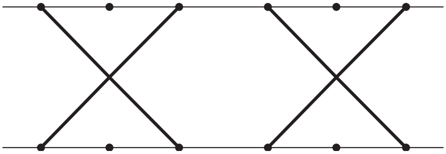
On the other hand, suppose G_ϕ has a vertex cover V' of size k or less. This must contain at least two vertices from each clause component—there's no other way to cover the three edges of the triangle. It must also contain at least one vertex from each variable component—there's no other way to cover the variable component's edge. That makes $k = 2m + n$ vertices already, so we must have $|V'| = k$. There is no room for any other vertices in V'. Because V' contains exactly one of x_i or $\neg x_i$ for each variable, we can use it to make a truth assignment for ϕ: make x_i = true if $x_i \in V'$, and make x_i = false if $\neg x_i \in V'$. Because V' contains exactly two of the triangle vertices from each clause component, one of the connecting edges from each clause component must be covered from the other end; therefore, each clause contains at least one true literal under this truth assignment, so it is a satisfying assignment for ϕ.

We have shown that *3SAT* $\leq_P$ *Vertex Cover.* Because *3SAT* is NP-hard, we conclude that *Vertex Cover* is NP-hard.

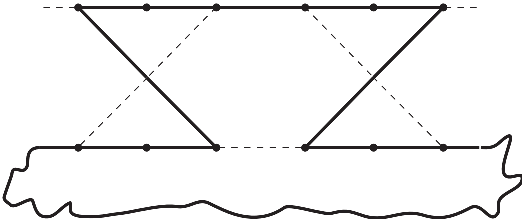
C.4 *Hamiltonian Circuit* Is NP-hard

We will prove this result by reduction from *Vertex Cover*. Given any instance of *Vertex Cover*, consisting of a graph *G* and a natural number *k*, we will show how to construct a new graph *H* such that *H* has a Hamiltonian circuit if and only if *G* has a vertex cover of size *k* or less.

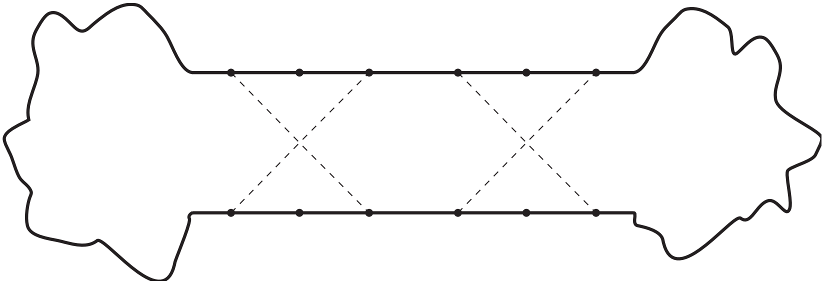
The construction makes repeated use of a special subgraph that we'll call a widget. (Our constructed graph *H* will have one widget for every edge in *G*.) The widget has twelve vertices, connected like this:



The widgets will be connected in the larger graph only by their corners. As such, there are only two ways a widget can participate in a Hamiltonian circuit of that larger graph. The circuit might enter at one corner, visit every vertex of the widget, and leave at the opposite corner on the same side, like this:



We'll call that a *one-pass path*. The alternative is that the circuit might make two separated passes through the widget, visiting one side on one pass and the other side on the other, like this:



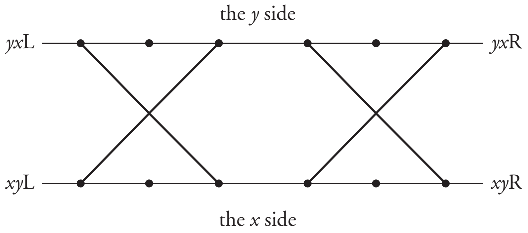
We'll call that a *two-pass path*. Our reduction will use the fact that there is no other way to include a widget in a Hamiltonian path:

> **Lemma C.1:** If a widget graph is connected in a larger graph only at its corners, then a Hamiltonian circuit entering at one of the corners must exit at the opposite corner on the same side, having visited either all the vertices in the widget or all the vertices on that side of the widget and none on the other.
>
> **Proof:** By inspection.

To convince yourself this is true, try making some paths through a widget. You'll quickly see why most of your choices are restricted by the need to visit every vertex exactly once.

In the following construction, there is one widget in the new graph for each edge in the original graph. The widgets are connected together in such a way that Hamiltonian circuits in the new graph correspond to vertex covers of the original graph. If an edge is covered from just one end, our Hamiltonian circuit will visit its widget by a one-pass path; if it is covered from both ends, by a two-pass path.

> **Lemma 21.5.2:** *Hamiltonian Circuit* is *NP*-hard.
>
> **Proof:** By reduction from *Vertex Cover*. Let $G = (V, E)$ and $k \in \mathcal{N}$ be any instance of *Vertex Cover*. If $k \geq |V|$ there is always a vertex cover, so we can reduce these to any small, positive instance of *Hamiltonian Circuit*. Otherwise, $k < |V|$, and we construct a graph H as follows.
>
> First, for each pair of vertices x and y connected by an edge in G, add a widget to H, labeled at the corners as follows:



> Next, add k vertices v_1 through v_k to H.
>
> Next, for each vertex x in G, let $y_1 \ldots y_n$ be any ordering of all the vertices connected to x in G, and add edges $(xy_1\text{R}, xy_2\text{L})$, $(xy_2\text{R}, xy_3\text{L})$, and so on through $(xy_{n-1}\text{R}, xy_n\text{L})$. These edges connect all the x sides (of widgets that have x sides) into one path.

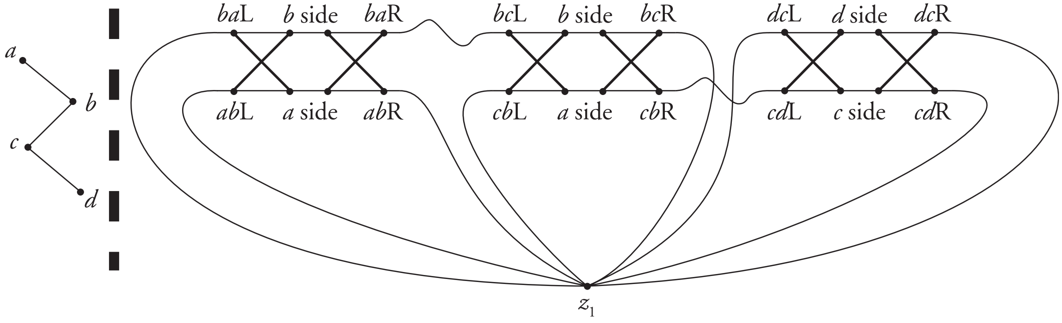
Finally, for each vertex x in G and for each of the new vertices z_i in H, add the edges from z_i to the start of the x-side path (that is, from z_i to xy_1L) and from the end of the x-side path to z_i (that is, from xy_nR to z_i). These edges complete paths leading from any z_i, through any x-side path, to any z_k. That completes the construction. The size of H and the time to construct it are clearly polynomial in the size of G.

Now we claim that the constructed graph H has a Hamiltonian circuit if and only if G has a vertex cover of size k or less. First, suppose H has a Hamiltonian circuit. This must include every z_i. By construction the only way to get from one z_i to the next is by a u_i-side path for some variable u_i, so the Hamiltonian circuit must include k different u_i-side paths. Let $U = \{u_1, \ldots, u_k\}$ be the set of variables whose u_i-side paths are used in the circuit. Every vertex of every widget must be included in the circuit, so every widget must occur on one or two u_i-side paths for some $u_i \in U$. Therefore, every edge in G must have one or two vertices in U. So U is a vertex cover for G.

On the other hand, suppose $U = \{u_1, \ldots, u_k\}$ is some vertex cover for G. Then we can make a circuit in H as follows. Start at z_1; take the u_1-side path to z_2; then take the u_2-side path to z_3; and so on. At the end of the u_k-side path, return to z_1. Because U is a vertex cover for G, every edge has either one or two ends in U; therefore every widget occurs either once or twice in this circuit. If once, use the one-pass path through that widget; if twice, use the two-pass paths. In this way we visit every vertex exactly once, so this is a Hamiltonian circuit.

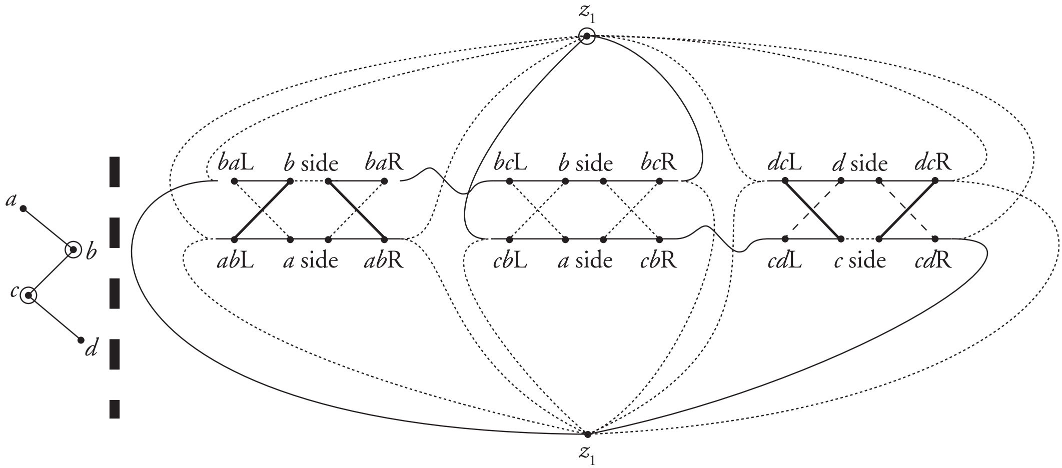
We have shown that *Vertex Cover* $\leq_P$ *Hamiltonian Circuit.* Because *Vertex Cover* is *NP*-hard, we conclude that *Hamiltonian Circuit* is *NP*-hard.

Perhaps a quick example of this construction will help to elucidate the proof. Here is the graph G for an instance of *Vertex Cover*; next to it is the graph H constructed by the proof, with $k = 1$.



As you can see, G has no vertex cover of size $k = 1$, just as H has no Hamiltonian circuit. Any set containing a single vertex fails to cover all the edges in G and corresponds to a path that fails to visit all vertices in H. For example, the set $\{c\}$ covers only two of the three edges in G, just as the path from z_1 through the c sides and back visits only two of the three widgets in H.

On the other hand, if we allow $k = 2$, then G has a cover and H has a Hamiltonian circuit:



The illustration shows the cover for G (with circled vertices) and shows (with solid lines) a corresponding Hamiltonian circuit in H: from z_1 through the b sides to z_2, then through the c sides and back to z_1. Notice how the widgets for the edges (a, b) and (c, d) are visited using the one-pass path, since those edges are covered from only one end, while the widget for (b, c) is visited using the two-pass paths, since it is covered from both ends.

INDEX