FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

Automata, Computability and Complexity

THEORY AND APPLICATIONS

Elaine Rich

PEARSON
Prentice Hall

Upper Saddle River NJ 07458

Library of Congress Cataloging-in-Publication Data on File

Vice President and Editorial Director, ECS: *Marcia J. Horton*
Executive Editor: *Tracy Dunkelberger*
Assistant Editor: *Carole Snyder*
Editorial Assistant: *ReeAnne Davis*
Managing Editor: *Scott Disanno*
Production Editor: *Rose Kernan*
Director of Creative Services: *Paul Belfanti*
Creative Director: *Juan Lopez*
Cover Designer: *Mavreen Eide*
Managing Editor, AV Management and Production: *Patricia Burns*
Art Editor: *Gregory Dulles*
Director, Image Resource Center: *Melinda Reo*
Manager, Rights and Permissions: *Zina Arabia*
Manager, Visual Research: *Beth Brenzel*
Manager, Cover Visual Research and Permissions: *Karen Sanatar*
Manufacturing Manager, ESM: *Alexis Heydt-Long*
Manufacturing Buyer: *Lisa McDowell*
Marketing Manager: *Mack Patterson*

Pearson Prentice Hall
Pearson Education, Inc.
Upper Saddle River, NJ 07458

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN: 0-13-228806-0
ISBN: 978-0-13-228806-4

Pearson Education Ltd., *London*
Pearson Education Australia Pty. Ltd., *Sydney*
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., *Hong Kong*
Pearson Education Canada, Inc., *Toronto*
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education—Japan, *Tokyo*
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., *Upper Saddle River, New Jersey*

CONTENTS

Preface xiii
Acknowledgments xvii
Credits xix

PART I: INTRODUCTION 1

1 Why Study the Theory of Computation? 2
1.1 The Shelf Life of Programming Tools 2
1.2 Applications of the Theory Are Everywhere 5

2 Languages and Strings 8
2.1 Strings 8
2.2 Languages 10
Exercises 19

3 The Big Picture: A Language Hierarchy 21
3.1 Defining the Task: Language Recognition 21
3.2 The Power of Encoding 22
3.3 A Machine-Based Hierarchy of Language Classes 28
3.4 A Tractability Hierarchy of Language Classes 34
Exercises 34

4 Computation 36
4.1 Decision Procedures 36
4.2 Determinism and Nondeterminism 41
4.3 Functions on Languages and Programs 48
Exercises 52

PART II: FINITE STATE MACHINES AND REGULAR LANGUAGES 53

5 Finite State Machines 54
5.1 Deterministic Finite State Machines 56
5.2 The Regular Languages 60
5.3 Designing Deterministic Finite State Machines 63

5.4 Nondeterministic FSMs 66
5.5 From FSMs to Operational Systems 79
5.6 Simulators for FSMs ● 80
5.7 Minimizing FSMs ● 82
5.8 A Canonical Form for Regular Languages 94
5.9 Finite State Transducers ● 96
5.10 Bidirectional Transducers ● 98
5.11 Stochastic Finite Automata: Markov Models and HMMs ● 101
5.12 Finite Automata, Infinite Strings: Büchi Automata ● 115
Exercises 121

6 Regular Expressions 127
6.1 What is a Regular Expression? 128
6.2 Kleene's Theorem 133
6.3 Applications of Regular Expressions 147
6.4 Manipulating and Simplifying Regular Expressions 149
Exercises 151

7 Regular Grammars ● 155
7.1 Definition of a Regular Grammar 155
7.2 Regular Grammars and Regular Languages 157
Exercises 161

8 Regular and Nonregular Languages 162
8.1 How Many Regular Languages Are There? 162
8.2 Showing That a Language Is Regular 163
8.3 Some Important Closure Properties of Regular Languages 165
8.4 Showing That a Language is Not Regular 169
8.5 Exploiting Problem-Specific Knowledge 178
8.6 Functions on Regular Languages 179
Exercises 182

9 Algorithms and Decision Procedures for Regular Languages 187
9.1 Fundamental Decision Procedures 187
9.2 Summary of Algorithms and Decision Procedures for Regular Languages 194
Exercises 196

10 Summary and References 198
References 199

PART I

INTRODUCTION

CHAPTER 1

Why Study the Theory of Computation?

In this book, we present a theory of what can be computed and what cannot. We also sketch some theoretical frameworks that can inform the design of programs to solve a wide variety of problems. But why do we bother? We don't we just skip ahead and write the programs that we need? This chapter is a short attempt to answer that question.

1.1 The Shelf Life of Programming Tools

Implementations come and go. In the somewhat early days of computing, programming meant knowing how to write code like:[1]

```
ENTRY     SXA     4,RETURN
          LDQ     X
          FMP     A
          FAD     B
          XCA
          FMP     X
          FAD     C
          STO     RESULT
RETURN    TRA     0

A         BSS     1
B         BSS     1
C         BSS     1
X         BSS     1
TEMP      BSS     1
STORE     BSS     1
          END
```

[1]This program was written for the IBM 7090. It computes the value of a simple quadratic $ax^2 + bx + c$.

In 1957, Fortran appeared and made it possible for people to write programs that looked more straightforwardly like mathematics. By 1970, the IBM 360 series of computers was in widespread use for both business and scientific computing. To submit a job, one keyed onto punch cards a set of commands in OS/360 JCL (Job Control Language). Guruhood attached to people who actually knew what something like this meant:[2]

```
//MYJOB     JOB (COMPRESS),'VOLKER BANDKE', CLASS=P,COND=(0,NE)
//BACKUP   EXEC PGM=IEBCOPY
//SYSPRINT DD   SYSOUT=*
//SYSUT1    DD  DISP=SHR,DSN=MY.IMPORTNT.PDS
//SYSUT2    DD  DISP=(,CATLG),DSN=MY.IMPORTNT.PDS.BACKUP,
//              UNIT=3350,VOL=SER=DISK01,
//              DCB=MY.IMPORTNT.PDS,SPACE=(CYL,(10,10,20))
//COMPRESS EXEC PGM=IEBCOPY
//SYSPRINT DD   SYSOUT=*
//MYPDS     DD  DISP=OLD,DSN=*.BACKUP.SYSUT1
//SYSIN     DD  *
COPY INDD=MYPDS,OUTDD=MYPDS
//DELETE2 EXEC PGM=IEFBR14
//BACKPDS   DD  DISP=(OLD,DELETE,DELETE),DSN=MY.IMPORTNT.PDS.BACKUP
```

By the turn of the millennium, gurus were different. They listened to different music and had never touched a keypunch machine. But many of them did know that the following Java method (when compiled with the appropriate libraries) allows the user to select a file, which is read in and parsed using whitespace delimiters. From the parsed file, the program builds a frequency map, which shows how often each word occurs in the file:

```
public static TreeMap<String, Integer> create() throws IOException
    public static TreeMap<String, Integer> create() throws
                  IOException
    {  Integer freq;
       String word;
       TreeMap<String, Integer> result = new TreeMap<String, Integer>();
       JFileChooser c = new JFileChooser();
       int retval = c.showOpenDialog(null);
       if (retval == JFileChooser.APPROVE_OPTION)
            {   Scanner s = new Scanner( c.getSelectedFile());
                while( s.hasNext() )
                {   word = s.next().toLowerCase();
                    freq = result.get(word);
                    result.put(word, (freq == null ? 1 : freq + 1));
                }
            }
        return result;
    }
}
```

[2]It safely reorganizes and compresses a partitioned dataset.

Along the way, other programming languages became popular, at least within some circles. There was a time when some people bragged that they could write code like:[3]

```
(⌈/V)>(+/V)-⌈/V
```

Today's programmers can't read code from 50 years ago. Programmers from the early days could never have imagined what a program of today would look like. In the face of that kind of change, what does it mean to learn the science of computing?

The answer is that there are mathematical properties, both of problems and of algorithms for solving problems, that depend on neither the details of today's technology nor the programming fashion *du jour*. The theory that we will present in this book addresses some of those properties. Most of what we will discuss was known by the early 1970s (barely the middle ages of computing history). But it is still useful in two key ways:

- It provides a set of abstract structures that are useful for solving certain classes of problems. These abstract structures can be implemented on whatever hardware/software platform is available.
- It defines provable limits to what can be computed, regardless of processor speed or memory size. An understanding of these limits helps us to focus our design effort in areas in which it can pay off, rather than on the computing equivalent of the search for a perpetual motion machine.

In this book our focus will be on analyzing problems, rather than on comparing solutions to problems. We will, of course, spend a lot of time solving problems. But our goal will be to discover fundamental properties of the problems themselves:

- Is there any computational solution to the problem? If not, is there a restricted but useful variation of the problem for which a solution does exist?
- If a solution exists, can it be implemented using some fixed amount of memory?
- If a solution exists, how efficient is it? More specifically, how do its time and space requirements grow as the size of the problem grows?
- Are there groups of problems that are equivalent in the sense that if there is an efficient solution to one member of the group there is an efficient solution to all the others?

[3] An expression in the programming language APL ⌨. It returns 1 if the largest value in a three element vector is greater than the sum of the other two elements, and 0 otherwise [Gillman and Rose 1984, p. 326]. Although APL is not one of the major programming languages in use today, its inventor, Kenneth Iverson, received the 1979 Turing Award for its development.

1.2 Applications of the Theory Are Everywhere

Computers have revolutionized our world. They have changed the course of our daily lives, the way we do science, the way we entertain ourselves, the way that business is conducted, and the way we protect our security. The theory that we present in this book has applications in all of those areas. Throughout the main text, you will find notes that point to the more substantive application-focused discussions that appear in Appendices G–Q. Some of the applications that we'll consider are:

- Languages, the focus of this book, enable both machine/machine and person/machine communication. Without them, none of today's applications of computing could exist.

> Network communication protocols are languages. (I. 1) Most web pages are described using the Hypertext Markup Language, HTML. (Q.1.2) The Semantic Web, whose goal is to support intelligent agents working on the Web, exploits additional layers of languages, such as RDF and OWL, that can be used to describe the content of the Web. (I. 3) Music can be viewed as a language, and specialized languages enable composers to create new electronic music. (N.1) Even very unlanguage-like things, such as sets of pictures, can be viewed as languages by, for example, associating each picture with the program that drew it. (Q.1.3)

- Both the design and the implementation of modern programming languages rely heavily on the theory of context-free languages that we will present in Part III. Context-free grammars are used to document the languages' syntax and they form the basis for the parsing techniques that all compilers use.

> The use of context-free grammars to define programming languages and to build their compilers is described in Appendix G.

- People use natural languages, such as English, to communicate with each other. Since the advent of word processing, and then the Internet, we now type or speak our words to computers. So we would like to build programs to manage our words, check our grammar, search the World Wide Web, and translate from one language to another. Programs to do that also rely on the theory of context-free languages that we present in Part III.

> A sketch of some of the main techniques used in natural language processing can be found in Appendix L.

- Systems as diverse as parity checkers, vending machines, communication protocols, and building security devices can be straightforwardly described as finite state machines, which we'll describe in Chapter 5.

> A vending machine is described in Example 5.1. A family of network communication protocols is modeled as finite state machines in I.1. An example of a simple building security system, modeled as a finite state machine, can be found in J.1. An example of a finite state controller for a soccer-playing robot can be found in P.4.

- Many interactive video games are (large, often nondeterministic) finite state machines.

> An example of the use of a finite state machine to describe a role playing game can be found in N.3.1.

- DNA is the language of life. DNA molecules, as well as the proteins that they describe, are strings that are made up of symbols drawn from small alphabets (nucleotides and amino acids, respectively). So computational biologists exploit many of the same tools that computational linguists use. For example, they rely on techniques that are based on both finite state machines and context-free grammars.

> For a very brief introduction to computational biology see Appendix K.

- Security is perhaps the most important property of many computer systems. The undecidability results that we present in Part IV show that there cannot exist a general purpose method for automatically verifying arbitrary security properties of programs. The complexity results that we present in Part V serve as the basis for powerful encryption techniques.

> For a proof of the undecidability of the correctness of a very simple security model, see J.2. For a short introduction to cryptography, see J.3.

- Artificial intelligence programs solve problems in task domains ranging from medical diagnosis to factory scheduling. Various logical frameworks have been proposed for representing and reasoning with the knowledge that such programs exploit. The undecidability results that we present in Part IV show that there cannot exist a general theorem prover that can decide, given an arbitrary statement in first order logic, whether or not that statement follows from the system's axioms. The complexity results that we present in Part V show that, if we back off to the far less expressive system of Boolean (propositional) logic, while it becomes possible to decide the validity of a given statement, it is not possible to do so, in general, in a reasonable amount of time.

> For a discussion of the role of undecidability and complexity results in artificial intelligence, see Appendix M. The same issues plague the development of the Semantic Web. (I.3)

- Clearly documented and widely accepted standards play a pivotal role in modern computing systems. Getting a diverse group of users to agree on a single standard is never easy. But the undecidability and complexity results that we present in Parts IV and V mean that, for some important problems, there is no single right answer for all uses. Expressively weak standard languages may be tractable and decidable, but they may simply be inadequate for some tasks. For those tasks, expressively powerful languages, that give up some degree of tractability and possibly decidability, may be required. The provable lack of a one-size-fits-all language makes the standards process even more difficult and may require standards that allow alternatives.

> We'll see one example of this aspect of the standards process when we consider, in I.3, the design of a description language for the Semantic Web.

- Many natural structures, including ones as different as organic molecules and computer networks, can be modeled as graphs. The theory of complexity that we present in Part V tells us that, while there exist efficient algorithms for answering some important questions about graphs, other questions are "hard", in the sense that no efficient algorithm for them is known nor is one likely to be developed.

> We'll discuss the role of graph algorithms in network analysis in I.2.

- The complexity results that we present in Part V contain a lot of bad news. There are problems that matter yet for which no efficient algorithm is likely ever to be found. But practical solutions to some of these problems exist. They rely on a variety of approximation techniques that work pretty well most of the time.

> An almost optimal solution to an instance of the traveling salesman problem with 1,904,711 cities has been found, as we'll see in Section 27.1. Randomized algorithms can find prime numbers efficiently, as we'll see in Section 30.2.4. Heuristic search algorithms find paths in computer games (N.3.2) and move sequences for champion chess-playing programs. (N.2.5)

CHAPTER 2

Languages and Strings

In the theory that we are about to build, we are going to analyze problems by casting them as instances of the more specific question, "Given some string s and some language L, is s in L?" Before we can formalize what we mean by that, we need to define our terms.

An ***alphabet***, often denoted Σ, is a finite set. We will call the members of Σ ***symbols*** or ***characters***.

2.1 Strings

A ***string*** is a finite sequence, possibly empty, of symbols drawn from some alphabet Σ. Given any alphabet Σ, the shortest string that can be formed from Σ is the empty string, which we will write as ε. The set of all possible strings over an alphabet Σ is written Σ^*. This notation exploits the Kleene star operator, which we will define more generally below.

EXAMPLE 2.1 Alphabets

Alphabet name	Alphabet symbols	Example strings
The English alphabet	{a, b, c, . . . , z}	ε, aabbcg, aaaaa
The binary alphabet	{0, 1}	ε, 0, 001100
A star alphabet	{★, ✪, ☆, ✭, ✶, ✡, ☆}	ε, ✪✪, ✪✶✶☆★☆
A music alphabet	{𝅝, 𝅗𝅥, ♩, ♪, 𝅘𝅥𝅯, 𝅘𝅥𝅰}	ε, 𝅝𝅗𝅥𝅗𝅥♩♩𝅘𝅥𝅯𝅘𝅥𝅯

In running text, we will indicate literal symbols and strings by writing them `like this`.

2.1.2 Functions on Strings

The ***length*** of a string s, which we will write as $|s|$, is the number of symbols in s. For example:

$$|\varepsilon| = 0$$
$$|1001101| = 7$$

For any symbol c and string s, we define the function $\#_c(s)$ to be the number of times that the symbol c occurs in s. So, for example, $\#_a(\text{abbaaa}) = 4$.

The ***concatenation*** of two strings s and t, written $s \| t$ or simply st, is the string formed by appending t to s. For example, if $x = \text{good}$ and $y = \text{bye}$, then $xy = \text{goodbye}$. So $|xy| = |x| + |y|$.

The empty string, ε, is the identity for concatenation of strings. So $\forall x\ (x\varepsilon = \varepsilon x = x)$.

Concatenation, as a function defined on strings, is associative. So $\forall s, t, w\ ((st)w = s\,(tw))$.

Next we define string ***replication***. For each string w and each natural number i, the string w^i is defined as:

$$w^0 = \varepsilon$$
$$w^{i+1} = w^i w$$

For example:

$\text{a}^3 = \text{aaa}$
$(\text{bye})^2 = \text{byebye}$
$\text{a}^0\text{b}^3 = \text{bbb}$

Finally we define string ***reversal***. For each string w, the reverse of w, which we will write w^R, is defined as:

If $|w| = 0$ then $w^R = w = \varepsilon$.
If $|w| \geq 1$ then $\exists a \in \Sigma\ (\exists u \in \Sigma^*\ (w = ua))$, (i.e., the last character of w is a.)
Then define $w^R = au^R$.

THEOREM 2.1 Concatenation and Reverse of Strings

Theorem: If w and x are strings, then $(wx)^R = x^R w^R$.
For example, $(\text{nametag})^R = (\text{tag})^R(\text{name})^R = \text{gateman}$.

Proof: The proof is by induction on $|x|$:

Base case: $|x| = 0$. Then $x = \varepsilon$, and $(wx)^R = (w\varepsilon)^R = (w)^R = \varepsilon w^R = \varepsilon^R w^R = x^R w^R$.

Prove: $\forall n \geq 0\ (((|x| = n) \rightarrow ((wx)^R = x^R w^R)) \rightarrow ((|x| = n+1) \rightarrow ((wx)^R = x^R w^R)))$.

Consider any string x, where $|x| = n+1$. Then $x = ua$ for some character a and $|u| = n$. So:

$$
\begin{aligned}
(wx)^R &= (w(ua))^R && \text{rewrite } x \text{ as } ua \\
&= ((wu)a)^R && \text{associativity of concatenation} \\
&= a(wu)^R && \text{definition of reversal} \\
&= a(u^R w^R) && \text{induction hypothesis} \\
&= (au^R)w^R && \text{associativity of concatenation} \\
&= (ua)^R w^R && \text{definition of reversal} \\
&= x^R w^R && \text{rewrite } ua \text{ as } x
\end{aligned}
$$

2.1.3 Relations on Strings

A string s is a ***substring*** of a string t iff s occurs contiguously as part of t. For example:

aaa	is a substring of	aaabbbaaa
aaaaaa	is not a substring of	aaabbbaaa

A string s is a ***proper substring*** of a string t iff s is a substring of t and $s \neq t$. Every string is a substring (although not a proper substring) of itself. The empty string, ε, is a substring of every string.

A string s is a ***prefix*** of t iff $\exists x \in \Sigma^* \ (t = sx)$. A string s is a ***proper prefix*** of a string t iff s is a prefix of t and $s \neq t$. Every string is a prefix (although not a proper prefix) of itself. The empty string, ε, is a prefix of every string. For example, the prefixes of abba are: ε, a, ab, abb, abba.

A string s is a ***suffix*** of t iff $\exists x \in \Sigma^* \ (t = xs)$. A string s is a ***proper suffix*** of a string t iff s is a suffix of t and $s \neq t$. Every string is a suffix (although not a proper suffix) of itself. The empty string, ε, is a suffix of every string. For example, the suffixes of abba are: ε, a, ba, bba, abba.

2.2 Languages

A ***language*** is a (finite or infinite) set of strings over a finite alphabet Σ. When we are talking about more than one language, we will use the notation Σ_L to mean the alphabet from which the strings in the language L are formed.

EXAMPLE 2.2 Defining Languages Given an Alphabet

Let $\Sigma = \{$a, b$\}$. $\Sigma^* = \{\varepsilon$, a, b, aa, ab, ba, bb, aaa, aab, $\ldots\}$.

Some examples of languages over Σ are:

$\varnothing$, $\{\varepsilon\}$, $\{$a, b$\}$, $\{\varepsilon$, a, aa, aaa, aaaa, aaaaa$\}$,
$\{\varepsilon$, a, aa, aaa, aaaa, aaaaa, $\ldots\}$

2.2.2 Techniques for Defining Languages

We will use a variety of techniques for defining the languages that we wish to consider. Since languages are sets, we can define them using any of the set-defining techniques that are described in A.2. For example, we can specify a characteristic function, i.e., a predicate that is *True* of every element in the set and *False* of everything else.

EXAMPLE 2.3 All a's Precede All b's

Let $L = \{w \in \{\mathtt{a},\mathtt{b}\}^* : \text{all a's precede all b's in } w\}$. The strings ε, a, aa, aabbb, and bb are in L. The strings aba, ba, and abc are not in L. Notice that some strings trivially satisfy the requirement for membership in L. The rule says nothing about there having to be any a's or any b's. All it says is that any a's there are must come before all the b's (if any). If there are no a's or no b's, then there can be none that violate the rule. So the strings ε, a, aa, and bb trivially satisfy the rule and are in L.

EXAMPLE 2.4 Strings That End in a

Let $L = \{x : \exists y \in \{\mathtt{a},\mathtt{b}\}^* \ (x = y\mathtt{a})\}$. The strings a, aa, aaa, bbaa, and ba are in L. The strings ε, bab, and bca are not in L. L consists of all strings that can be formed by taking some string in $\{\mathtt{a},\mathtt{b}\}^*$ and concatenating a single a onto the end of it.

EXAMPLE 2.5 The Perils of Using English to Describe Languages

Let $L = \{x\#y : x, y \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$ and, when x and y are viewed as the decimal representations of natural numbers, $square(x) = y\}$. The strings 3#9 and 12#144 are in L. The strings 3#8, 12, and 12#12#12 are not in L. But what about the string #? Is it in L? It depends on what we mean by the phrase, "when x and y are viewed as the decimal representations of natural numbers." Is ε the decimal representation of some natural number? It is possible that an algorithm that converts strings to numbers might convert ε to 0. In that case, since 0 is the square of 0, # is in L. If, on the other hand, the string-to-integer converter fails to accept ε as a valid input, # is not in L. This example illustrates the dangers of using English descriptions of sets. They are sometimes ambiguous. We will strive to use only unambiguous terms. We will also, as we discuss below, develop other definitional techniques that do not present this problem.

EXAMPLE 2.6 The Empty Language

Let $L = \{\} = \emptyset$. L is the language that contains no strings.

EXAMPLE 2.7 The Empty Language is Different From the Empty String

Let $L = \{\varepsilon\}$, the language that contains a single string, ε. Note that L is different from $\emptyset$.

All of the examples we have considered so far fit the definition that we are using for the term *language*: a set of strings. They're quite different, though, from the everyday use of the term. Everyday languages are also languages under our definition.

EXAMPLE 2.8 English Isn't a Well-Defined Language

Let $L = \{w : w \text{ is a sentence in English}\}$.

Examples:	Kerry hit the ball.	/* Clearly in L.
	Colorless green ideas sleep furiously.[4]	/* The syntax is correct but what could it mean?
	The window needs fixed.	/* In some dialects of L.
	Ball the Stacy hit blue.	/* Clearly not in L.

The problem with languages like English is that there is no clear agreement on what strings they contain. We will not be able to apply the theory that we are about to build to any language for which we cannot first produce a formal specification. Natural languages, like English or Spanish or Chinese, while hard to specify, are of great practical importance, though. As a result, substantial effort has been expended in creating formal and computationally effective descriptions of them that are good enough to be used as the basis for applications such as grammar checking and text database retrieval.

> To the extent that formal descriptions of natural languages like English can be created, the theory that we are about to develop can be applied, as we will see in Parts II and III and Appendix L.

[4] This classic example of a syntactically correct but semantically anomalous sentence is from [Chomsky 1957].

EXAMPLE 2.9 A Halting Problem Language

Let $L = \{w : w \text{ is a C program that halts on all inputs}\}$. L is substantially more complex than, for example, $\{x \in \{a,b\}^*$: all a's precede all b's$\}$. But, unlike English, there does exist a clear formal specification of it. The theory that we are about to build will tell us something very useful about L.

We can use the relations that we have defined on strings as a way to define languages.

EXAMPLE 2.10 Using the Prefix Relation

We define the following languages in terms of the prefix relation on strings:

$L_1 = \{w \in \{a, b\}^* : \text{no prefix of } w \text{ contains b}\}$
$= \{\varepsilon, a, aa, aaa, aaaa, aaaaa, aaaaaa, \ldots\}$.

$L_2 = \{w \in \{a, b\}^* : \text{no prefix of } w \text{ starts with b}\}$
$= \{w \in \{a, b\}^* : \text{the first character of } w \text{ is a}\} \cup \{\varepsilon\}$.

$L_3 = \{w \in \{a, b\}^* : \text{every prefix of } w \text{ starts with b}\}$
$= \varnothing$.

L_3 is equal to $\varnothing$ because ε is a prefix of every string. Since ε does not start with b, no strings meet L_3's requirement.

Recall that we defined the replication operator on strings: For any string s and integer n, $s^n = n$ copies of s concatenated together. For example, $(bye)^2 = byebye$. We can use replication as a way to define a language, rather than a single string, if we allow n to be a variable, rather than a specific constant.

EXAMPLE 2.11 Using Replication to Define a Language

Let $L = \{a^n : n \geq 0\}$. $L = \{\varepsilon, a, aa, aaa, aaaa, aaaaa, \ldots\}$.

Languages are sets. So, if we want to provide a computational definition of a language, we could specify either:

- a language generator, which enumerates (lists) the elements of the language, or
- a language recognizer, which decides whether or not a candidate string is in the language and returns *True* if it is and *False* if it isn't.

For example, the logical definition, $L = \{x : \exists y \in \{\mathtt{a}, \mathtt{b}\}^* \ (x = y\mathtt{a})\}$ can be turned into either a language generator (enumerator) or a language recognizer.

In some cases, when considering an enumerator for a language L, we may care about the order in which the elements of L are generated. If there exists a total order D of the elements of Σ_L (as there does, for example, on the letters of the Roman alphabet or the symbols for the digits 0 – 9), then we can use D to define on L a useful total order called ***lexicographic order*** (written $<_L$):

- Shorter strings precede longer ones: $\forall x\ (\forall y\ ((|x| < |y|) \rightarrow (x <_L y)))$, and
- Of strings that are the same length, sort them in dictionary order using D.

When we use lexicographic order in the rest of this book, we will assume that D is the standard sort order on letters and numerals. If D is not obvious, we will state it.

We will say that a program ***lexicographically enumerates*** the elements of L iff it enumerates them in lexicographic order.

EXAMPLE 2.12 Lexicographic Enumeration

Let $L = \{x \in \{\mathtt{a}, \mathtt{b}\}^* : \text{all a's precede all b's}\}$. The lexicographic enumeration of L is:

ε, a, b, aa, ab, bb, aaa, aab, abb, bbb, aaaa, aaab, aabb, abbb, bbbb, aaaaa, …

In Parts II, III, and IV of this book, we will consider a variety of formal techniques for specifying both generators (enumerators) and recognizers for various classes of languages.

2.2.3 What is the Cardinality of a Language?

How large is a language? The smallest language over any alphabet is $\varnothing$, whose cardinality is 0. The largest language over any alphabet Σ is Σ^*. What is $|\Sigma^*|$? Suppose that $\Sigma = \varnothing$. Then $\Sigma^* = \{\varepsilon\}$ and $|\Sigma^*| = 1$. But what about the far more useful case in which Σ is not empty?

THEOREM 2.2 The Cardinality of Σ^*

Theorem: If $\Sigma \neq \varnothing$ then Σ^* is countably infinite.

Proof: The elements of Σ^* can be lexicographically enumerated by a straightforward procedure that:

- Enumerates all strings of length 0, then length 1, then length 2, and so forth.
- Within the strings of a given length, enumerates them in dictionary order.

This enumeration is infinite since there is no longest string in Σ^*. By Theorem A.1, since there exists an infinite enumeration of Σ^*, it is countably infinite.

Since any language over Σ is a subset of Σ^*, the cardinality of every language is at least 0 and at most $\aleph_0$. So all languages are either finite or countably infinite.

2.2.4 How Many Languages Are There?

Let Σ be an alphabet. How many different languages are there that are defined on Σ? The set of languages defined on Σ is $\mathcal{P}(\Sigma^*)$, the power set of Σ^*, or the set of all subsets of Σ^*. If $\Sigma = \varnothing$ then Σ^* is $\{\varepsilon\}$ and $\mathcal{P}(\Sigma^*)$ is $\{\varnothing, \{\varepsilon\}\}$. But, again, what about the useful case in which Σ is not empty?

THEOREM 2.3 An Uncountably Infinite Number of Languages

Theorem: If $\Sigma \neq \varnothing$ then the set of languages over Σ is uncountably infinite.

Proof: The set of languages defined on Σ is $\mathcal{P}(\Sigma^*)$. By Theorem 2.2, Σ^* is countably infinite. By Theorem A.4, if S is a countably infinite set, $\mathcal{P}(S)$ is uncountably infinite. So $\mathcal{P}(\Sigma^*)$ is uncountably infinite.

2.2.5 Functions on Languages

Since languages are sets, all of the standard set operations are well-defined on languages. In particular, we will find union, intersection, difference, and complement to be useful. Complement will be defined with Σ^* as the universe unless we explicitly state otherwise.

EXAMPLE 2.13 Set Functions Applied to Languages

Let: $\Sigma = \{\text{a}, \text{b}\}$.
$L_1 = \{$strings with an even number of a's$\}$.
$L_2 = \{$strings with no b's$\} = \{\varepsilon, \text{a}, \text{aa}, \text{aaa}, \text{aaaa}, \text{aaaaa}, \text{aaaaaa}, \ldots\}$.

$L_1 \cup L_2 = \{$all strings of just a's plus strings that contain b's and an even number of a's$\}$.

$L_1 \cap L_2 = \{\varepsilon, \text{aa}, \text{aaaa}, \text{aaaaaa}, \text{aaaaaaaa}, \ldots\}$.

$L_2 - L_1 = \{\text{a}, \text{aaa}, \text{aaaaa}, \text{aaaaaaa}, \ldots\}$.

$\neg(L_2 - L_1) = \{$strings with at least one b $\} \cup \{$strings with an even number of a's$\}$.

Because languages are sets of strings, it makes sense to define operations on them in terms of the operations that we have already defined on strings. Three useful ones to consider are concatenation, Kleene star, and reverse.

Let L_1 and L_2 be two languages defined over some alphabet Σ. Then their ***concatenation***, written L_1L_2 is:

$$L_1L_2 = \{w \in \Sigma^* : \exists s \in L_1 (\exists t \in L_2 (w = st))\}.$$

EXAMPLE 2.14 Concatenation of Languages

Let: $L_1 = \{\texttt{cat}, \texttt{dog}, \texttt{mouse}, \texttt{bird}\}$.
$L_2 = \{\texttt{bone}, \texttt{food}\}$.

$L_1L_2 = \{\texttt{catbone}, \texttt{catfood}, \texttt{dogbone}, \texttt{dogfood}, \texttt{mousebone}, \texttt{mousefood}, \texttt{birdbone}, \texttt{birdfood}\}$.

The language $\{\varepsilon\}$ is the identity for concatenation of languages. So, for all languages L, $L\{\varepsilon\} = \{\varepsilon\}L = L$.

The language $\varnothing$ is a zero for concatenation of languages. So, for all languages L, $L\varnothing = \varnothing L = \varnothing$. That $\varnothing$ is a zero follows from the definition of the concatenation of two languages as the set consisting of all strings that can be formed by selecting some string s from the first language and some string t from the second language and then concatenating them together. There are no ways to select a string from the empty set.

Concatenation, as a function defined on languages, is associative. So, for all languages L_1, L_2, and L_3:

$$(L_1L_2)L_3 = L_1(L_2L_3).$$

It is important to be careful when concatenating languages that are defined using replication. Recall that we used the notation $\{\texttt{a}^n : n \geq 0\}$ to mean the set of strings composed of zero or more a's. That notation is a shorthand for a longer, perhaps clearer expression, $\{w : \exists n \geq 0\ (w = \texttt{a}^n)\}$. In this form, it is clear that n is a variable bound by an existential quantifier. We will use the convention that the scope of such quantifiers is the entire expression in which they occur. So multiple occurrences of the same variable letter are the same variable and must take on the same value. Suppose that $L_1 = \{\texttt{a}^n : n \geq 0\}$ and $L_2 = \{\texttt{b}^n : n \geq 0\}$. By the definition of language concatenation, $L_1L_2 = \{w : w$ consists of a (possibly empty) a region followed by a (possibly empty) b region$\}$. $L_1L_2 \neq \{\texttt{a}^n\texttt{b}^n : n \geq 0\}$, since every string in $\{\texttt{a}^n\texttt{b}^n : n \geq 0\}$ must have the same number of b's as a's. The easiest way to avoid confusion is simply to rename conflicting variables before attempting to concatenate the expressions that contain them. So $L_1L_2 = \{\texttt{a}^n\texttt{b}^m : n, m \geq 0\}$. In Chapter 6 we will define a convenient notation that will let us write this as $\texttt{a}^*\texttt{b}^*$.

Let L be a language defined over some alphabet Σ. Then the ***Kleene star*** of L, written L^* is:

$$L^* = \{\varepsilon\} \cup \{w \in \Sigma^* : \exists k \geq 1\ (\exists w_1, w_2, \ldots w_k \in L\ (w = w_1w_2 \ldots w_k))\}.$$

In other words, L^* is the set of strings that can be formed by concatenating together zero or more strings from L.

EXAMPLE 2.15 Kleene Star

Let $L = \{\texttt{dog}, \texttt{cat}, \texttt{fish}\}$. Then:

$L^* = \{\varepsilon, \texttt{dog}, \texttt{cat}, \texttt{fish}, \texttt{dogdog}, \texttt{dogcat}, \ldots,$
$\texttt{fishdog}, \ldots, \texttt{fishcatfish}, \texttt{fishdogfishcat}, \ldots\}$.

EXAMPLE 2.16 Kleene Star, Again

Let $L = \{w \in \{a,b\}^* : \#_a(w) \text{ is odd and } \#_b(w) \text{ is even}\}$. Then $L^* = \{w \in \{a,b\}^* : \#_b(w) \text{ is even}\}$. The constraint on the number of a's disappears in the description of L^* because strings in L^* are formed by concatenating together any number of strings from L. If an odd number of strings are concatenated together, the result will contain an odd number of a's. If an even number are used, the result will contain an even number of a's.

L^* always contains an infinite number of strings as long as L is not equal to either $\emptyset$ or $\{\varepsilon\}$ (i.e., as long as there is at least one nonempty string any number of which can be concatenated together). If $L = \emptyset$, then $L^* = \{\varepsilon\}$, since there are no strings that could be concatenated to ε to make it longer. If $L = \{\varepsilon\}$, then L^* is also $\{\varepsilon\}$.

It is sometimes useful to require that at least one element of L be selected. So we define:

$$L^+ = LL^*.$$

Another way to describe L^+ is that it is the closure of L under concatenation. Note that $L^+ = L^* - \{\varepsilon\}$ iff $\varepsilon \notin L$.

EXAMPLE 2.17 L^+

Let $L = \{0,1\}^+$ be the set of binary strings. L does not include ε.

Let L be a language defined over some alphabet Σ. Then the ***reverse*** of L, written L^R is:

$$L^R = \{w \in \Sigma^* : w = x^R \text{ for some } x \in L\}.$$

In other words, L^R is the set of strings that can be formed by taking some string in L and reversing it.

Since we have defined the reverse of a language in terms of the definition of reverse applied to strings, we expect it to have analogous properties.

THEOREM 2.4 Concatenation and Reverse of Languages

Theorem: If L_1 and L_2 are languages, then $(L_1L_2)^R = L_2^R L_1^R$.

Proof: If x and y are strings, then $\forall x\ (\forall y\ ((xy)^R = y^R x^R))$ Theorem 2.1

$(L_1L_2)^R = \{(xy)^R : x \in L_1 \text{ and } y \in L_2\}$	Definition of concatenation of languages
$= \{y^R x^R : x \in L_1 \text{ and } y \in L_2\}$	Lines 1 and 2
$= L_2^R L_1^R$	Definition of concatenation of languages

We have now defined the two important data types, string and language, that we will use throughout this book. In the next chapter, we will see how we can use them to define a framework that will enable us to analyze computational problems of all sorts (not just ones you may naturally think of in terms of strings).

2.2.6 Assigning Meaning to the Strings of a Language

Sometimes we are interested in viewing a language just as a set of strings. For example, we'll consider some important formal properties of the language we'll call $A^nB^n = \{a^nb^n : n \geq 0\}$. In other words, A^nB^n is the language composed of all strings of a's and b's such that all the a's come first and the number of a's equals the number of b's. We won't attempt to assign meanings to any of those strings.

But some languages are useful precisely because their strings do have meanings. We use natural languages like English and Chinese because they allow us to communicate ideas. A program in a language like Java or C^{++} or Perl also has a meaning. In the case of a programming language, one way to define meaning is in terms of some other (typically closer to machine architecture) language. So, for example, the meaning of a Java program can be described as a Java Virtual Machine program. An alternative is to define a program's meaning in a logical language.

Philosophers and linguists (and others) have spent centuries arguing about what sentences in natural languages like English (or Sanskrit or whatever) mean. We won't attempt to solve that problem here. But if we are going to work with formal languages, we need a precise way to map each string to its meaning (also called its ***semantics***). We'll call a function that assigns meanings to strings a ***semantic interpretation function***. Most of the languages we'll be concerned with are infinite because there is no bound on the length of the strings that they contain. So it won't, in general, be possible to define meanings by a table that pairs each string with its meaning.

We must instead define a function that knows the meanings of the language's basic units and can combine those meanings, according to some fixed set of rules, to build meanings for larger expressions. We call such a function, which can be said to "compose" the meanings of simpler constituents into a single meaning for a larger expression, a ***compositional semantic interpretation function***. There arguably exists a mostly compositional semantic interpretation function for English. Linguists fight about the gory details of what such a function must look like. Everyone agrees that words have meanings and that one can build a meaning for a simple sentence by combining the meanings of the subject and the verb. For example, speakers of English would have no trouble assigning a meaning to the sentence, "I gave him the fizding," provided that they are told what the meaning of the word "fizding" is. Everyone also agrees that the meaning of idioms, like "I'm going to give him a piece of my mind," cannot be derived compositionally. Some other issues are more subtle.

> Languages whose strings have meaning pervade computing and its applications. Boolean logic and first-order logic are languages. Programming languages are languages. (G.1) Network protocols are languages. (I.1) Database query languages are languages. (Q.1.1) HTML is a language for defining

> Web pages. (Q.1.2) XML is a more general language for marking up data. (Q.1.2) OWL is a language for defining the meaning of tags on the Web. (I.3.6) BNF is a language that can be used to specify the syntax of other languages. (G.1.1) DNA is a language for describing proteins. (K.1.2) Music is a language based on sound. (N.1)

When we define a formal language for a specific purpose, we design it so that there exists a compositional semantic interpretation function. So, for example, there exist compositional semantic interpretation functions for programming languages like Java and C++. There exists a compositional semantic interpretation function for the language of Boolean logic. It is specified by the truth tables that define the meanings of whichever operators (e.g., $\wedge$, $\vee$, $\neg$ and $\rightarrow$) are allowed.

One significant property of semantic interpretation functions for useful languages is that they are generally not one-to-one. Consider:

- English: The sentences, "Chocolate, please," "I'd like chocolate," "I'll have chocolate," and "I guess chocolate today," all mean the same thing, at least in the context of ordering an ice cream cone.
- Java: The following chunks of code all do the same thing:

```
int x = 4;     int x = 4;     int x = 4;     int x = 4;
x++;           ++x;           x = x + 1;     x = x --1;
```

The semantic interpretation functions that we will describe later in this book, for example for the various grammar formalisms that we will introduce, will not be one-to-one either.

Exercises

1. Consider the language $L = \{1^n2^n : n > 0\}$. Is the string 122 in L?
2. Let $L_1 = \{a^nb^n : n > 0\}$. Let $L_2 = \{c^n : n > 0\}$. For each of the following strings, state whether or not it is an element of L_1L_2:
   a. ε.
   b. aabbcc.
   c. abbcc.
   d. aabbcccc.
3. Let $L_1 = \{$peach, apple, cherry$\}$ and $L_2 = \{$pie, cobbler, $\varepsilon\}$. List the elements of L_1L_2 in lexicographic order.
4. Let $L = \{w \in \{a, b\}^* : |w| \equiv_3 0\}$. List the first six elements in a lexicographic enumeration of L.
5. Consider the language L of all strings drawn from the alphabet $\{a, b\}$ with at least two different substrings of length 2.

a. Describe L by writing a sentence of the form $L = \{w \in \Sigma^* : P(w)\}$, where Σ is a set of symbols and P is a first-order logic formula. You may use the function $|s|$ to return the length of s. You may use all the standard relational symbols (e.g., $=$, $\neq$, $<$, etc.), plus the predicate *Substr*(s, t), which is *True* iff s is a substring of t.

b. List the first six elements of a lexicographic enumeration of L.

6. For each of the following languages L, give a simple English description. Show two strings that are in L and two that are not (unless there are fewer than two strings in L or two not in L, in which case show as many as possible).

a. $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : \text{exactly one prefix of } w \text{ ends in } \mathtt{a}\}$.

b. $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : \text{all prefixes of } w \text{ end in } \mathtt{a}\}$.

c. $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : \exists x \in \{\mathtt{a}, \mathtt{b}\}^+ (w = \mathtt{a}x\mathtt{a})\}$.

7. Are the following sets closed under the following operations? If not, what are their respective closures?

a. The language $\{\mathtt{a}, \mathtt{b}\}$ under concatenation.

b. The odd length strings over the alphabet $\{\mathtt{a}, \mathtt{b}\}$ under Kleene star.

c. $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^*\}$ under reverse.

d. $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : w \text{ starts with } \mathtt{a}\}$ under reverse.

e. $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : w \text{ ends in } \mathtt{a}\}$ under concatenation.

8. For each of the following statements, state whether it is *True* or *False*. Prove your answer.

a. $\forall L_1, L_2 (L_1 = L_2 \text{ iff } L_1^* = L_2^*)$.

b. $(\varnothing \cup \varnothing^*) \cap (\neg\varnothing - (\varnothing\varnothing^*)) = \varnothing$ (where $\neg\varnothing$ is the complement of $\varnothing$).

c. Every infinite language is the complement of a finite language.

d. $\forall L\, ((L^R)^R = L)$.

e. $\forall L_1, L_2 ((L_1 L_2)^* = L_1^* L_2^*)$.

f. $\forall L_1, L_2 ((L_1^* L_2^* L_1^*)^* = (L_2 \cup L_1)^*)$.

g. $\forall L_1, L_2 ((L_1 \cup L_2)^* = L_1^* \cup L_2^*)$.

h. $\forall L_1, L_2, L_3 ((L_1 \cup L_2) L_3 = (L_1 L_3) \cup (L_2 L_3))$.

i. $\forall L_1, L_2, L_3 ((L_1 L_2) \cup L_3 = (L_1 \cup L_3)(L_2 \cup L_3))$.

j. $\forall L\, ((L^+)^* = L^*)$.

k. $\forall L\, (\varnothing L^* = \{\varepsilon\})$.

l. $\forall L\, (\varnothing \cup L^+ = L^*)$.

m. $\forall L_1, L_2 ((L_1 \cup L_2)^* = (L_2 \cup L_1)^*)$.

CHAPTER 3

The Big Picture: A Language Hierarchy

Our goal, in the rest of this book, is to build a framework that lets us examine a new problem and be able to say something about how intrinsically difficult it is. In order to do this, we need to be able to compare problems that appear, at first examination, to be wildly different. Apples and oranges come to mind. So the first thing we need to do is to define a single framework into which any computational problem can be cast. Then we will be in a position to compare problems and to distinguish between those that are relatively easy to solve and those that are not.

3.1 Defining the Task: Language Recognition

The unifying framework that we will use is language recognition. Assume that we are given:

- The definition of a language L. (We will consider about half a dozen different techniques for providing this definition.)
- A string w.

Then we must answer the question: "Is w in L?" This question is an instance of a more general class that we will call decision problems. A ***decision problem*** is simply a problem that requires a yes or no answer.

In the rest of this book, we will discuss programs to solve decision problems specifically of the form, "Is w in L?" We will see that, for some languages, a very simple program suffices. For others, a more complex one is required. For still others, we will prove that no program can exist.

3.2 The Power of Encoding

The question that we are going to ask, "Is w in L?" may seem, at first glance, way too limited to be useful. What about problems like multiplying numbers, sorting lists, and retrieving values from a database? And what about real problems like air traffic control or inventory management? Can our theory tell us anything interesting about them?

The answer is yes and the key is encoding. With an appropriate encoding, other kinds of problems can be recast as the problem of deciding whether a string is in a language. We will show some examples to illustrate this idea. We will divide the examples into two categories:

- Problems that are already stated as decision problems. For these, all we need to do is to encode the inputs as strings and then define a language that contains exactly the set of inputs for which the desired answer is yes.
- Problems that are not already stated as decision problems. These problems may require results of any type. For these, we must first reformulate the problem as a decision problem and then encode it as a language recognition task.

3.2.1 Everything is a String

Our stated goal is to build a theory of computation. What we are actually about to build is a theory specifically of languages and strings. Of course, in a computer's memory, everything is a (binary) string. So, at that level, it is obvious that restricting our attention to strings does not limit the scope of our theory. Often, however, we will find it easier to work with languages with larger alphabets.

Each time we consider a new problem, our first task will be to describe it in terms of strings. In the examples that follow, and throughout the book, we will use the notation $<X>$ to mean a string encoding of some object X. We'll use the notation $<X, Y>$ to mean the encoding, into a single string, of the two objects X and Y.

The first three examples we'll consider are of problems that are naturally described in terms of strings. Then we'll look at examples where we must begin by constructing an appropriate string encoding.

EXAMPLE 3.1 Pattern Matching on the Web

- Problem: Given a search string w and a web document d, do they match? In other words, should a search engine, on input w, consider returning d?
- The language to be decided: $\{<w, d> : d \text{ is a candidate match for the query } w\}$.

EXAMPLE 3.2 Question-Answering on the Web

- Problem: Given an English question q and a web document d (which may be in English or Chinese), does d contain the answer to q?
- The language to be decided: $\{<q, d> : d \text{ contains the answer to } q\}$.

The techniques that we will describe in the rest of this book are widely used in the construction of systems that work with natural language (e. g., English or Spanish or Chinese) text and speech inputs. (Appendix L)

EXAMPLE 3.3 Does a Program Always Halt?

- Problem: Given a program p, written in some standard programming language, is p guaranteed to halt on all inputs?
- The language to be decided: $HP_{ALL} = \{p : p \text{ halts on all inputs}\}$.

A procedure that could decide whether or not a string is in HP_{ALL} could be an important part of a larger system that proves the correctness of a program. Unfortunately, as we will see in Theorem 21.3, no such procedure can exist.

EXAMPLE 3.4 Primality Testing

- Problem: Given a nonnegative integer n, is it prime? In other words, does it have at least one positive integer factor other than itself and 1?
- An instance of the problem: Is 9 prime?
- Encoding of the problem: We need a way to encode each instance. We will encode each nonnegative integer as a binary string.
- The language to be decided: PRIMES $= \{w : w \text{ is the binary encoding of a prime number}\}$.

Prime numbers play an important role in modern cryptography systems. (J.3) We'll discuss the complexity of PRIMES in Section 28.1.7 and again in Section 30.2.4.

EXAMPLE 3.5 Verifying Addition

- Problem: Verify the correctness of the addition of two numbers.
- Encoding of the problem: We encode each of the numbers as a string of decimal digits. Each instance of the problem is a string of the form:

$$<integer_1> + <integer_2> = <integer_3>.$$

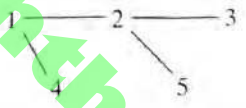
EXAMPLE 3.5 (*Continued*)

- The language to be decided:

  INTEGERSUM = {w of the form: <$integer_1$>+ <$integer_2$> = <$integer_3$> : each of the substrings <$integer_1$>, <$integer_2$> and <$integer_3$> is an element of {0, 1, 2, 3, 4, 5, 6, 8, 9} and $integer_3$ is the sum of $integer_1$ and $integer_2$}.
- Examples of strings in L: `2 + 4 = 6` `23 + 47 = 70`.
- Examples of strings not in L: `2 + 4 = 10` `2 + 4`.

EXAMPLE 3.6 Graph Connectivity

- Problem: Given an undirected graph G, is it connected? In other words, given any two distinct vertices x and y in G, is there a path from x to y?
- Instance of the problem: Is the following graph connected?



- Encoding of the problem: Let V be a set of binary numbers, one for each vertex in G. Then we construct <G> as follows:
  - Write $|V|$ as a binary number.
  - Write a list of edges, each of which is represented by a pair of binary numbers corresponding to the vertices that the edge connects.
  - Separate all such binary numbers by the symbol /.

For example, the graph shown above would be encoded by the following string, which begins with an encoding of 5 (the number of vertices) and is followed by four pairs corresponding to the four edges:

`101/1/10/10/11/1/100/10/101.`

- The language to be decided:

  CONNECTED = {$w \in \{0, 1, /\}^* : w = n_1/n_2/\ldots n_i$, where each n_i is a binary string and w encodes a connected graph, as described above}.

EXAMPLE 3.7 Protein Sequence Alignment

- Problem: Given a protein fragment f and a complete protein molecule p, could f be a fragment from p?

- Encoding of the problem: Represent each protein molecule or fragment as a sequence of amino acid residues. Assign a letter to each of the 20 possible amino acids. So a protein fragment might be represented as AGHTYWDNR.
- The language to be decided: $\{<f, p> f$ could be a fragment from $p\}$.

> The techniques that we will describe in the rest of this book are widely used in computational biology. (Appendix K)

In each of these examples, we have chosen an encoding that is expressive enough to make it possible to describe all of the instances of the problem we are interested in. But have we chosen a good encoding? Might there be another one? The answer to this second question is yes. And it will turn out that the encoding we choose may have a significant impact on what we can say about the difficulty of solving the original problem. To see an example of this, we need look no farther than the addition problem that we just considered. Suppose that we want to write a program to examine a string in the addition language that we proposed above. Suppose further that we impose the constraint that our program reads the string one character at a time, left to right. It has only a finite (bounded in advance, independent of the length of the input string) amount of memory. These restrictions correspond to the notion of a finite state machine, as we will see in Chapter 5. It turns out that no machine of this sort can decide the language that we have described. We'll see how to prove results such as this in Chapter 8.

But now consider a different encoding of the addition problem. This time we encode each of the numbers as a binary string, and we write the digits, from lowest order to highest order, left to right (i.e., backwards from the usual way). Furthermore, we imagine the three numbers aligned in the way they often are when we draw an addition problem. So we might encode 10 + 4 = 14 as:

```
 0101    writing 1010 backwards
+0010    writing 0100 backwards
 0111    writing 1110 backwards
```

We now encode each column of that sum as a single character. Since each column is a sequence of three binary digits, it may take on any one of 8 possible values. We can use the symbols a, b, c, d, e, f, g, and h to correspond to 000, 001, 010, 011, 100, 101, 110, and 111, respectively. So we could encode the 10 + 4 = 14 example as afdf.

It is easy to design a program that reads such a string, left to right, and decides, as each character is considered, whether the sum so far is correct. For example, if the first character of a string is c, then the sum is wrong, since 0 + 1 cannot be 0 (although it could be later if there were a carry bit from the previous column).

> This idea is the basis for the design of binary adders, as well as larger circuits, like multipliers, that exploit them. (P.3)

In Part V of this book we will be concerned with the efficiency (stated in terms of either time or space) of the programs that we write. We will describe both time and space requirements as functions of the length of the program's input. When we do that, it may matter what encoding scheme we have picked since some encodings produce longer strings than others do. For example, consider the integer 25. It can be encoded:

- In decimal as: 25,
- In binary as: 11001, or
- In unary as: 1111111111111111111111111.

We'll return to this issue in Section 27.3.1.

3.2.2 Casting Problems as Decision Questions

Problems that are not already stated as decision questions can be transformed into decision questions. More specifically, they can be reformulated so that they become language recognition problems. The idea is to encode, into a single string, both the inputs and the outputs of the original problem P. So, for example, if P takes two inputs and produces one result, we could construct strings of the form $i_1; i_2; r$. Then a string $s = x; y; z$ is in the language L that corresponds to P iff z is the result that P produces given the inputs x and y.

EXAMPLE 3.8 Casting Addition as Decision

- Problem: Given two nonnegative integers, compute their sum.
- Encoding of the problem: We transform the problem of adding two numbers into the problem of checking to see whether a third number is the sum of the first two. We can use the same encoding that we used in Example 3.5.
- The language to be decided:
  INTEGERSUM = {w of the form: <$integer_1$>+<$integer_2$>=<$integer_3$>, where each of the substrings <$integer_1$>, <$integer_2$>, and <$integer_3$> is an element of $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^+$ and $integer_3$ is the sum of $integer_1$ and $integer_2$}.

EXAMPLE 3.9 Casting Sorting as Decision

- Problem: Given a list of integers, sort it.
- Encoding of the problem: We transform the problem of sorting a list into the problem of examining a pair of lists and deciding whether the second corresponds to the sorted version of the first.

- The language to be decided:

$$L = \{w_1 \# w_2 : \exists n \geq 1 \ (w_1 \text{ is of the form } int_1, int_2, \ldots int_n, \\ w_2 \text{ is of the form } int_1, int_2, \ldots int_n, \text{ and} \\ w_2 \text{ contains the same objects as } w_1 \text{ and } w_2 \text{ is sorted})\}.$$

- Example of a string in L: `1,5,3,9,6#1,3,5,6,9.`
- Example of a string not in L: `1,5,3,9,6#1,2,3,4,5,6,7.`

EXAMPLE 3.10 Casting Database Querying as Decision

- Problem: Given a database and a query, execute the query against the database.
- Encoding of the problem: We transform the task of executing the query into the problem of evaluating a reply to see if it is correct.
- The language to be decided:

$$L = \{d \# q \# a : d \text{ is an encoding of a database,} \\ q \text{ is a string representing a query, and} \\ a \text{ is the correct result of applying } q \text{ to } d\}.$$

- Example of a string in L:

```
(name, age, phone), (John, 23, 567-1234) (Mary, 24, 234-9876 )#
(select name age=23) #
(John).
```

Given each of the problems that we have just considered, there is an important sense in which the encoding of the problem as a decision question is equivalent to the original formulation of the problem: Each can be reduced to the other. We'll have a lot more to say about the idea of reduction in Chapter 21. But, for now, what we mean by ***reduction*** of one problem to another is that, if we have a program to solve the second, we can use it to build a program to solve the first. For example, suppose that we have a program P that adds a pair of integers. Then the following program decides the language INTEGERSUM, which we described in Example 3.8:

Given a string of the form $<integer_1>+<integer_2>=<integer_3>$ do:

1. Let x = *convert-to-integer* ($<integer_1>$).
2. Let y = *convert-to-integer* ($<integer_2>$).
3. Let $z = P(x, y)$.
4. If z = *convert-to-integer* ($<integer_3>$) then accept. Else reject.

Alternatively, if we have a program T that decides INTEGERSUM, then the following program computes the sum of two integers x and y:

1. Lexicographically enumerate the strings that represent decimal encodings of nonnegative integers.
2. Each time a string s is generated, create the new string $<x>+<y>=s$.
3. Feed that string to T.
4. If T accepts $<x>+<y>=s$, halt and return *convert-to-integer*(s).

3.3 A Machine-Based Hierarchy of Language Classes

In Parts II, III, and IV, we will define a hierarchy of computational models, each more powerful than the last. The first model is simple: Programs written for it are generally easy to understand, they run in linear time, and algorithms exist to answer almost any question we might wish to ask about such programs. The second model is more powerful, but still limited. The last model is powerful enough to describe anything that can be computed by any sort of real computer. All of these models will allow us to write programs whose job is to accept some language L. In this section, we sketch this machine hierarchy and provide a short introduction to the language hierarchy that goes along with it.

3.3.1 The Regular Languages

The first model we will consider is the ***finite state machine*** or ***FSM***. Figure 3.1 shows a simple FSM that accepts strings of a's and b's, where all a's come before all b's.

The input to an FSM is a string, which is fed to it one character at a time, left to right. The FSM has a start state, shown in the diagram with an unlabelled arrow leading to it, and some number (zero or more) of accepting states, which will be shown in our diagrams with double circles. The FSM starts in its start state. As each character is read, the FSM changes state based on the transitions shown in the figure. If an FSM M is in an accepting state after reading the last character of some input string s, then M accepts s. Otherwise it rejects it. Our example FSM stays in state 1 as long as it is reading a's. When it sees a b, it moves to state 2, where it stays as long as it continues seeing b's. Both state 1 and state 2 are accepting states. But if, in state 2, it sees an a, it goes to state 3, a nonaccepting state, where it stays until it runs out of input. So, for example, this machine will accept aab, aabbb, and bb. It will reject ba.

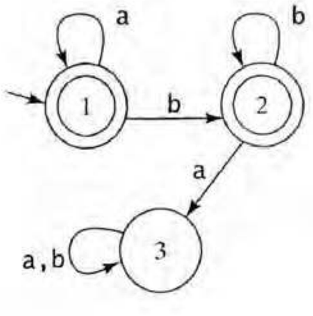


FIGURE 3.1 A simple FSM.

We will call the class of languages that can be accepted by some FSM ***regular***. As we will see in Part II, many useful languages are regular, including binary strings with even parity, syntactically well-formed floating point numbers, and sequences of coins that are sufficient to buy a soda.

3.3.2 The Context-Free Languages

But there are useful simple languages that are not regular. Consider, for example, Bal, the language of balanced parentheses. Bal contains strings like (()) and ()(); it does not contain strings like ()))(. Because it's hard to read strings of parentheses, let's consider instead the related language $A^nB^n = \{a^nb^n : n \geq 0\}$. In any string in A^nB^n, all the a's come first and the number of a's equals the number of b's. We could try to build an FSM to accept A^nB^n. But the problem is, "How shall we count the a's so that we can compare them to the b's?" The only memory in an FSM is in the states and we must choose a fixed number of states when we build our machine. But there is no bound on the number of a's we might need to count. We will prove in Chapter 8 that it is not possible to build an FSM to accept A^nB^n.

But languages like Bal and A^nB^n are important. For example, almost every programming language and query language allows parentheses, so any front end for such a language must be able to check to see that the parentheses are balanced. Can we augment the FSM in a simple way and thus be able to solve this problem? The answer is yes. Suppose that we add one thing, a single stack. We will call any machine that consists of an FSM, plus a single stack, a ***pushdown automaton*** or ***PDA***.

We can easily build a PDA M to accept A^nB^n. The idea is that, each time it sees an a, M will push it onto the stack. Then, each time it sees a b, it will pop an a from the stack. If it runs out of input and stack at the same time and it is in an accepting state, it will accept. Otherwise, it will reject. M will use the same state structure that we used in our FSM example above to guarantee that all the a's come before all the b's. In diagrams of PDAs, read an arc label of the form $x/y/z$ to mean, "if the input is an x, and it is possible to pop y off the stack, then take the transition, do the pop of y, and push z". If the middle argument is ε, then don't bother to check the stack. If the third argument is ε, then don't push anything. Using those conventions, the PDA shown in Figure 3.2 accepts A^nB^n.

Using a very similar sort of PDA, we can build a machine to accept Bal and other languages whose strings are composed of properly nested substrings. For example, a ***palindrome*** is a string that reads the same right-to-left as it does left-to right. We can easily build a PDA to accept the language PalEven = $\{ww^R : w \in \{a, b\}^*\}$, the

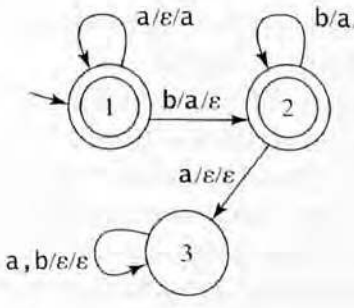


FIGURE 3.2 A simple PDA that accepts A^nB^n.

language of even-length palindromes of a's and b's. The PDA for PalEven simply pushes all the characters in the first half of its input string onto the stack, guesses where the middle is, and then starts popping one character for each remaining input character. If there is a guess that causes the pushed string (which will be popped off in reverse order) to match the remaining input string, then the input string is in PalEven.

But we should note some simple limitations to the power of the PDA. Consider the language WW = $\{ww : w \in \{a, b\}^*\}$, which is just like PalEven except that the second half of each of its strings is an exact copy of the first half (rather than the reverse of it). Now, as we'll prove in Chapter 13, it is not possible to build an accepting PDA (although it would be possible to build an accepting machine if we could augment the finite state controller with a first-in, first-out queue rather than a stack).

We will call the class of languages that can be accepted by some PDA ***context-free***. As we will see in Part III, many useful languages are context-free, including most programming languages, query languages, and markup languages.

3.3.3 The Decidable and Semidecidable Languages

But there are useful straightforward languages that are not context-free. Consider, for example, the language of English sentences in which some word occurs more than once. As an even simpler (although probably less useful) example, consider another language to which we will give a name. Let $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$, i.e., the language composed of all strings of a's, b's, and c's such that all the a's come first, followed by all the b's, then all the c's, and the number of a's equals the number of b's equals the number of c's. We could try to build a PDA to accept $A^nB^nC^n$. We could use the stack to count the a's, just as we did for A^nB^n. We could pop the stack as the b's come in and compare them to the a's. But then what shall we do about the c's? We have lost all information about the a's and the b's since, if they matched, the stack will be empty. We will prove in Chapter 13 that it is not possible to build a PDA to accept $A^nB^nC^n$.

But it is easy to write a program to accept $A^nB^nC^n$. So, if we want a class of machines that can capture everything we can write programs to compute, we need a model that is stronger than the PDA. To meet this need, we will introduce a third kind of machine. We will get rid of the stack and replace it with an infinite tape. The tape will have a single read/write head. Only the tape square under the read/write head can be accessed (for reading or for writing). The read/write head can be moved one square in either direction on each move. The resulting machine is called a ***Turing machine***. We will also change the way that input is given to the machine. Instead of streaming it, one character at a time, the way we did for FSMs and PDAs, we will simply write the input string onto the tape and then start the machine with the read/write head just to the left of the first input character. We show the structure of a Turing machine in Figure 3.3. The arrow under the tape indicates the location of the read/write head.

At each step, a Turing machine M considers its current state and the character that is on the tape directly under its read/write head. Based on those two things, it chooses its next state, chooses a character to write on the tape under the read/write head, and chooses whether to move the read/write head one square to the right or one square to the left. A finite segment of M's tape contains the input string. The rest is blank, but M may move the read/write head off the input string and write on the blank squares of the tape.

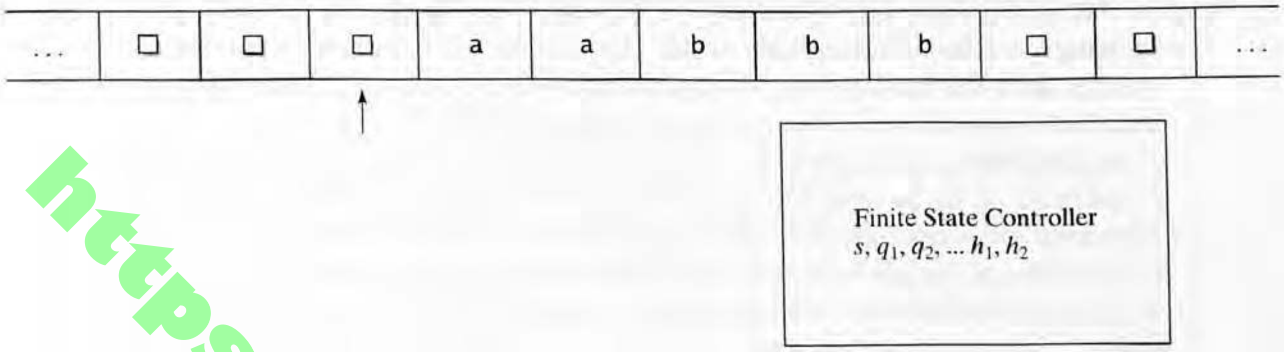


FIGURE 3.3 The structure of a Turing machine.

There exists a simple Turing machine that accepts $A^nB^nC^n$. It marks off the leftmost a, scans to the right to find a b, marks it off, continues scanning to the right, finds a c, and marks it off. Then it goes back to the left, marks off the next a, and so forth. When it runs out of a's, it makes one final pass to the right to make sure that there are no extra b's or c's. If that check succeeds, the machine accepts. If it fails, or if at any point the machine failed to find a required b or c, it rejects. For the details of how this machine operates, see Example 17.8.

Finite state machines and pushdown automata (with one technical exception that we can ignore for now) are guaranteed to halt. They must do so when they run out of input. Turing machines, on the other hand, carry no such guarantee. The input simply sits on the tape. A Turing machine may (and generally does) move back and forth across its input many times. It may move back and forth forever. Or it may simply move in one direction, off the input onto the blank tape, and keep going forever. Because of its flexibility in using its tape to record its computation, the Turing machine is a more powerful model than either the FSM or the PDA. In fact, we will see in Chapter 18 that any computation that can be written in any programming language or run on any modern computer can be described as a Turing machine. However, when we work with Turing machines, we must be aware of the fact that they cannot be guaranteed to halt. And, unfortunately we can prove (as we will do in Chapter 19) that there exists no algorithm that can examine a Turing machine and tell whether or not it will halt (on any one input or on all inputs). This fundamental result about the limits of computation is known as the undecidability of the halting problem.

We will use the Turing machine to define two new classes of languages:

- A language L is ***decidable*** iff there exists a Turing machine M that halts on all inputs, accepts all strings that are in L, and rejects all strings that are not in L. In other words, M can always say yes or no, as appropriate.
- A language L is ***semidecidable*** iff there exists a Turing machine M that accepts all strings that are in L and fails to accept every string that is not in L. Given a string that is not in L, M may reject or it may loop forever. In other words, M can recognize a solution and then say yes, but it may not know when it should give up looking for a solution and say no.

Bal, A^nB^n, PalEven, WW, and $A^nB^nC^n$ are all decidable languages. Every decidable language is also semidecidable (since the requirement for semidecidability is strictly weaker than the requirement for decidability). But there are languages that are semidecidable yet not decidable. As an example, consider $L = \{<p, w>: p$ is a Java program that halts on input $w\}$. L is semidecidable by a Turing machine that simulates p running on w. If the simulation halts, the semidecider can halt and accept. But, if the simulation does not halt, the semidecider will not be able to recognize that it isn't going to. So it has no way to halt and reject. Just as there exists no algorithm that can examine a Turing machine and decide whether or not it will halt, there is no algorithm to examine a Java program (without having to run it) and make that determination. So L is semidecidable but not decidable.

3.3.4 The Computational Hierarchy and Why It Is Important

We have now defined four language classes:

1. Regular languages, which can be accepted by some finite state machine.
2. Context-free languages, which can be accepted by some pushdown automaton.
3. Decidable (or simply D) languages, which can decided by some Turing machine that always halts.
4. Semidecidable (or SD) languages, which can be semidecided by some Turing machine that halts on all strings in the language.

Each of these classes is a proper subset of the next class, as illustrated in the diagram shown in Figure 3.4.

As we move outward in the language hierarchy, we have access to tools with greater and greater expressive power. So, for example, we can define A^nB^n as a context-free language but not as a regular one. We can define $A^nB^nC^n$ as a decidable language but not as a context-free or a regular one. This matters because expressiveness generally comes at a price. The price may be:

- ***Computational efficiency***: Finite state machines run in time that is linear in the length of the input string. A general context-free parser based on the idea of a pushdown automaton requires time that grows as the cube of the length of the input string. A Turing machine may require time that grows exponentially (or faster) with the length of the input string.
- ***Decidability***: There exist procedures to answer many useful questions about finite state machines. For example, does an FSM accept some particular string? Is an FSM minimal (i.e., is it the simplest machine that does the job it does)? Are two FSMs identical? A subset of those questions can be answered for pushdown automata. None of them can be answered for Turing machines.
- ***Clarity***: There exist tools that enable designers to draw and analyze finite state machines. Every regular language can also be described using the (often very convenient) regular expression pattern language that we will define in Chapter 6. Every context-free language, in addition to being recognizable by some pushdown

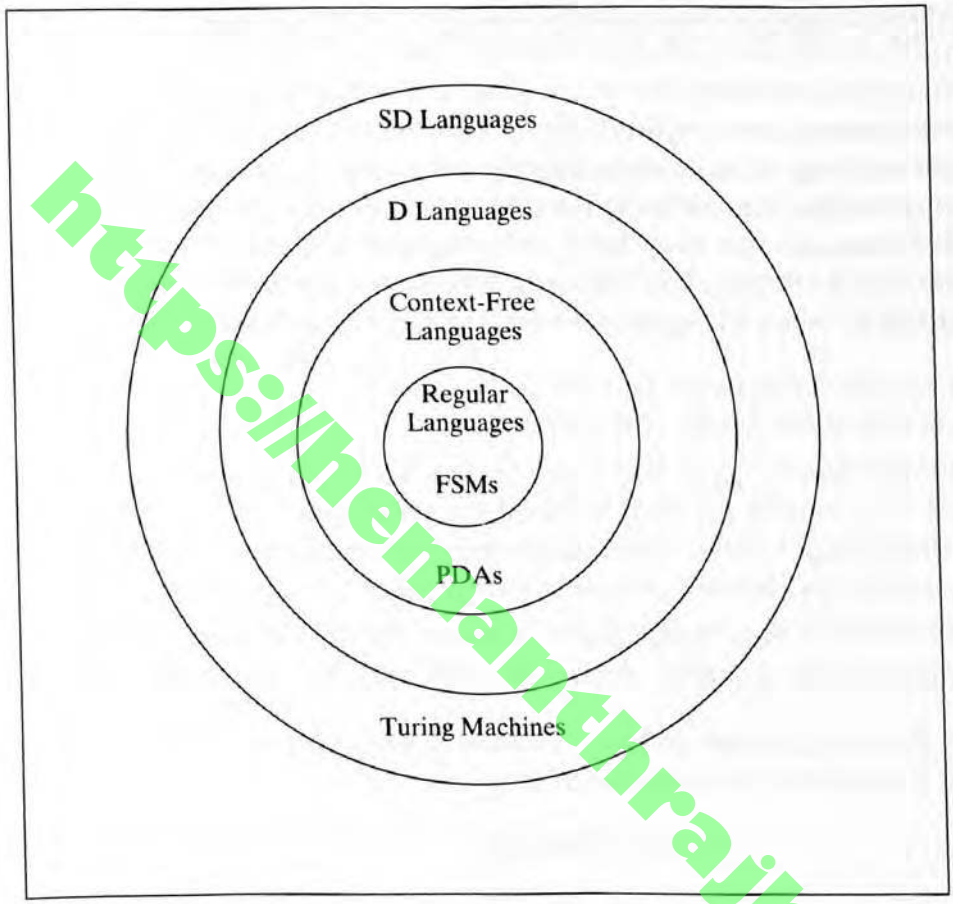


FIGURE 3.4 A hierarchy of language classes.

automaton, can (as we will see in Chapter 11) be described with a context-free grammar. For many important kinds of languages, context-free grammars are sufficiently natural that they are commonly used as documentation tools. No corresponding tools exist for the broader classes of decidable and semidecidable languages.

So, as a practical as well as a theoretical matter, it makes sense, given a particular problem, to describe it using the simplest (i.e., expressively weakest) formalism that is adequate to the job.

> ***The Rule of Least Power***[5]: "Use the least powerful language suitable for expressing information, constraints or programs on the World Wide Web."

Although stated in the context of the World Wide Web, the Rule of Least Power applies far more broadly. We're appealing to a generalization of it here. We'll return to a discussion of it in the specific context of the Semantic Web in I.3.

In Parts II, III, and IV of this book, we explore the language hierarchy that we have just defined. We will start with the smallest class, the regular languages, and move outwards.

[5]Quoted from [Berners-Lee and Mendelsohn 2006].

3.4 A Tractability Hierarchy of Language Classes

The decidable languages, as defined above, are those that can, *in principle*, be decided. Unfortunately, in the case of some of them, any procedure that can decide whether or not a string is in the language may require, on reasonably large inputs, more time steps than have elapsed since the Big Bang. So it makes sense to take another look at the class of decidable languages, this time from the perspective of the resources (time, space, or both) that may be required by the best decision procedures we can construct. We will do that in Part V. So, for example, we will define the classes:

- ***P***, which contains those languages that can be decided in time that grows as some polynomial function of the length of the input,
- ***NP***, which contains those languages that can be decided by a nondeterministic machine (one that can conduct a search by guessing which move to make) with the property that the amount of time required to explore one sequence of guesses (one path) grows as some polynomial function of the length of the input, and
- ***PSPACE***, which contains those languages that can be decided by a machine whose space requirement grows as some polynomial function of the length of the input.

These classes, like the ones that we defined in terms of particular kinds of machines, can be arranged in a hierarchy. For example, it is the case that:

$$P \subseteq NP \subseteq PSPACE$$

Unfortunately, as we will see, less is known about the structure of this hierarchy than about the structure of the hierarchy we drew in the last section. For example, perhaps the biggest open question of theoretical computer science is whether P = NP. It is possible, although generally thought to be very unlikely, that every language that is in NP is also in P. For this reason, we won't draw a picture here. Any picture we could draw might suggest a situation that will eventually turn out not to be true.

Exercises

1. Consider the following problem: Given a digital circuit C, does C output 1 on all inputs? Describe this problem as a language to be decided.
2. Using the technique we used in Example 3.8 to describe addition, describe square root as a language recognition problem.
3. Consider the problem of encrypting a password, given an encryption key. Formulate this problem as a language recognition problem.
4. Consider the optical character recognition (OCR) problem: Given an array of black and white pixels and a set of characters, determine which character best matches the pixel array. Formulate this problem as a language recognition problem.
5. Consider the language $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$, discussed in Section 3.3.3. We might consider the following design for a PDA to accept $A^nB^nC^n$: As each a

is read, push two a's onto the stack. Then pop one a for each b and one a for each c. If the input and the stack come out even, accept. Otherwise reject. Why doesn't this work?

6. Define a PDA-2 to be a PDA with two stacks (instead of one). Assume that the stacks can be manipulated independently and that the machine accepts iff it is in an accepting state and both stacks are empty when it runs out of input. Describe the operation of a PDA-2 that accepts $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$. (*Note*: We will see, in Section 17.5.2, that the PDA-2 is equivalent to the Turing machine in the sense that any language that can be accepted by one can be accepted by the other.)

CHAPTER 4

Computation

Our goal in this book is to be able to make useful claims about problems and the programs that solve them. Of course, both problem specifications and the programs that solve them take many different forms. Specifications can be written in English, or as a set of logical formulas, or as a set of input/output pairs. Programs can be written in any of a wide array of common programming languages. As we said in the last chapter, in this book we are, for the most part, going to depart from those standard methods and, instead

- Define problems a languages to be decided, and
- Define programs as state machines whose input is a string and whose output is *Accept* or *Reject*.

Both because of this change in perspective and because we are going to introduce two ideas that are not common in everyday programming practice, we will pause, in this chapter, and look at what we mean by computation and how we are going to go about it. In particular, we will examine three key ideas:

1. Decision procedures.
2. Nondeterminism.
3. Functions on languages (alternatively, programs that operate on other programs).

Once we have finished this discussion, we will begin our examination of the language classes that we outlined in Chapter 3.

4.1 Decision Procedures

Recall that a ***decision problem*** is one for which we must make a yes/no decision. An ***algorithm*** is a detailed procedure that accomplishes some clearly specified task. A ***decision procedure*** is an algorithm to solve a decision problem. Put another way, it is a program whose result is a Boolean value. Note that, in order to be guaranteed to return a Boolean value, a decision procedure must be guaranteed to halt on all inputs.

This book is about decision procedures. We will spend most of our time discussing decision procedures to answer questions of the form:

- Is string s in language L?

But we will also attempt to answer other questions, in particular ones that ask about the machines that we will build to answer the first group of questions. So we may ask questions such as:

- Given a machine (an FSM, a PDA, or a Turing machine), does it accept any strings?
- Given two machines, do they accept the same strings?
- Given a machine, is it the smallest (simplest) machine that does its job?

If we have in mind a decision problem to which we want an answer, there are three things we may want to know:

1. Does there exist a decision procedure (i.e., an algorithm) to answer the question? A decision problem is ***decidable*** iff the answer to this question is yes. A decision problem is ***undecidable*** iff the answer to this question is no. A decision problem is ***semidecidable*** iff there exists an algorithm that halts and returns *True* iff *True* is the answer. When *False* is the answer, it may either halt and return *False* or it may loop. Some undecidable problems are semidecidable; some are not even that.
2. If any decision procedures exist, find one.
3. Again, if any decision procedures exist, what is the most efficient one and how efficient is it?

In the early part of this book, we will ask questions for which decision procedures exist and we will often skip directly to question 2. But, as we progress, we will begin to ask questions for which, provably, no decision procedure exists. It is because there are such problems that we have articulated question 1.

Decision procedures are programs. They must possess two correctness properties:

1. The program must be guaranteed to halt on all inputs.
2. When the program halts and returns an answer, it must be the correct answer for the given input.

Let's consider some examples.

EXAMPLE 4.1 Checking for Even Numbers

Is the integer x even? This one is easy. Assume that / performs (truncating) integer division. Then the following program answers the question:

even (x : integer)=
    If $(x/2)*2 = x$ then return *True* else return *False*.

EXAMPLE 4.2 Checking for Prime Numbers

Is the positive integer x prime? Given an appropriate string encoding, this problem corresponds to the language PRIMES that we defined in Example 3.4. Defining a procedure to answer this question is not hard, although it will require a loop and so it will be necessary to prove that the loop always terminates. Several algorithms that solve this problem exist. Here's an easy one:

prime (x: positive integer) =
  For $i = 2$ to *ceiling* (*sqrt*(x)) do:
    If $(x/i) * i = x$ then return *False*.
  Return *True*.

The function *ceiling*(x), also written $\lceil x \rceil$ returns the smallest integer that is greater than or equal to x. This program is guaranteed to halt. The natural numbers between 0 and *ceiling* (*sqrt*(x))−2 form a well-ordered set under $\leq$. Let *index* correspond to *ceiling* (*sqrt*(x))−i. At the beginning of the first pass through the loop, the value of *index* is *ceiling* (*sqrt*(x))−2. The value of *index* decreases by one each time through the loop. The loop ends when that value becomes 0. It's worth pointing out that, while this program is simple and it is easy to prove that it is correct, it is not the most efficient program that we could write. We'll have more to say about this problem in Sections 28.1.7 and 30.2.4.

For our next few examples we need a definition. The sequence:

$$F_n = 2^{2^n} + 1, n \geq 0,$$

defines the ***Fermat numbers*** 🖳. The first few Fermat numbers are:

$$F_0 = 3, F_1 = 5, F_2 = 17, F_3 = 257, F_4 = 65{,}537, F_5 = 4{,}294{,}967{,}297.$$

EXAMPLE 4.3 Checking for Small Prime Fermat Numbers

Are there any prime Fermat numbers less than 1,000,000? There exists a simple decision procedure to answer this question:

fermatSmall() =
  $i = 0$.
  Repeat:
    candidate = (2**(2**i)) + 1.
    If *candidate* is prime then return *True*.

$i = i + 1$.

until *candidate* $\geq$ 1,000,000.

Return *False*.

This algorithm is guaranteed to halt because the value of *candidate* increases each time through the loop and the loop terminates when its value exceeds a fixed bound. We will skip the proof that the correct answer is returned.

EXAMPLE 4.4 Checking for Large Prime Fermat Numbers

Are there any prime Fermat numbers greater than 1,000,000? This question is different in one important way from the previous one. Does there exist a decision procedure to answer this question? What about:

fermatLarge ()=

$i = 0$.

Repeat:

candidate = (2 ** (2 ** i)) + 1.

If *candidate* > 1,000,000 and is prime then return *True*.

$i = i + 1$.

Return *False*.

What can we say about this program? If there is a prime Fermat number greater than 1,000,000, *fermatLarge* will find it and will halt. But suppose that there is no such number. Then the program will loop forever. *FermatLarge* is not capable of returning *False* even if *False* is the correct answer. So, is *fermatLarge* a decision procedure? No. A decision procedure must halt and return the correct answer, whatever that is.

Can we do better? Is there a decision procedure to answer this question? Yes. Since this question takes no arguments, it has a simple answer, either *True* or *False*. So either

fermatYes ()=

Return *True*,

or

fermatNo ()=

Return *False*.

correctly answers the question. Our problem now is, "Which one?" No one knows. Fermat himself was only able to generate the first five Fermat numbers, and, on

EXAMPLE 4.4 *(Continued)*

that basis, conjectured that all Fermat numbers are prime. If he had been right, then *fermatYes* answers the question. However, it now seems likely that there are no prime Fermat numbers greater than 65,537. A substantial effort continues to be devoted to finding one, but so far the only discoveries have been larger and larger composite Fermat numbers. But there is also no proof that a larger prime one does not exist nor is there an algorithm for finding one. We simply do not know.

EXAMPLE 4.5 Checking for Programs That Halt on a Particular Input

Now consider a problem that is harder and that cannot be solved by a simple constant function such as *fermatYes* or *fermatNo*. Given an arbitrary Java program p that takes a string w as an input parameter, does p halt on some particular value of w? Here's a candidate for a decision procedure:

haltsOnw (p: program, w: string) =

1. Simulate the execution of p on w.
2. If the simulation halts return *True* else return *False*.

Is *haltsOnw* a decision procedure? No, because it can never return the value *False*. Yet *False* is sometimes the correct answer (since there are (p, w) pairs such that p fails to halt on w). When *haltsOnw* should return *False*, it will loop forever in step 1. Can we do better? No. It is possible to prove, as we will do in Chapter 19, that no decision procedure for this question exists.

Define a ***semidecision procedure*** to be a procedure that halts and returns *True* whenever *True* is the correct answer. But, whenever *False* is the correct answer, it may return *False* or it may loop forever. In other words, a semidecision procedure knows when to say yes but it is not guaranteed to know when to say no. A ***semidecidable problem*** is a problem for which a semidecision procedure exists. Example 4.5 is a semidecidable problem. While some semidecidable problems are also decidable, that one isn't.

EXAMPLE 4.6 Checking for Programs That Halt on All Inputs

Now consider an even harder problem: Given an arbitrary Java program that takes a single string as an input parameter, does it halt on all possible input values? Here's a candidate for a decision procedure:

haltsOnAll (*program*) =

1. For $i = 1$ to infinity do:
   Simulate the execution of *program* on all possible input strings of length i.
2. If all of the simulations halt return *True* else return *False*.

HaltsOnAll will never halt on any program since, to do so, it must try running the program on an infinite number of strings. And there is not a better procedure to answer this question. We will show, in Chapter 21, that it is not even semidecidable.

The bottom line is that there are three kinds of questions:

- Those for which a decision procedure exists.
- Those for which no decision procedure exists but a semidecision procedure exists.
- Those for which not even a semi-decision procedure exists.

As we move through the language classes that we will consider in this book, we will move from worlds in which there exist decision procedures for just about every question we can think of to worlds in which there exist some decision procedures and perhaps some semidecision procedures, all the way to worlds in which there do not exist even semidecision procedures.

But keep in mind throughout that entire progression what a *decision* procedure is. It is an algorithm that is *guaranteed* to halt on all inputs.

4.2 Determinism and Nondeterminism

Imagine adding to a programming language the function *choose*, which may be written in either of the following forms:

- *choose* (action 1;;
  action 2;;
  ...
  action n)
- *choose* (x from S: $P(x)$)

In the first form, *choose* is presented with a finite list of alternatives, each of which will return either a successful value or the value *False*. *Choose* will:

- Return some successful value, if there is one.
- If there is no successful value, then choose will:

- Halt and return *False* if all the actions halt and return *False*.
- Fail to halt if any of the actions fails to halt. We want to define *choose* this way since any path that has not halted still has the potential to return a successful value.

In the second form, *choose* is presented with a set S of values. S may be finite or it may be infinite if it is specified by a generator. *Choose* will:

- Return some element x of S such that $P(x)$ halts with a value other than *False*, if there is one.
- If there is no such element, then *choose* will:
  - Halt and return *False* if it can be determined that, for all elements x of S, $P(x)$ is not satisfied. This will happen if S is finite and there is a procedure for checking P that always halts. It may also happen, even if S is infinite, if there is some way, short of checking all the elements, to determine that no elements that satisfy P exist.
  - Fail to halt if there is no mechanism for determining that no elements of S that satisfy P exist. This may happen either because S is infinite or because there is no algorithm, guaranteed to halt on all inputs, that checks for P and returns *False* when necessary.

In both forms, the job of *choose* is to find a successful value (which we will define to be any value other than *False*) if there is one. When we don't care which successful value we find (or how we find it), *choose* is a useful abstraction, as we will see in the next few examples.

We will call programs that are written in our new language, which includes *choose*, ***nondeterministic***. We will call programs that are written without using *choose* ***deterministic***.

Real computers are, of course, deterministic. So, if *choose* is going to be useful, there must exist a way to implement it on a deterministic machine. For now, however, we will be noncommittal as to how that is done. It may try the alternatives one at a time, or it may pursue them in parallel. If it tries them one at a time, it may try them in the order listed, in some random order, or in some order that is carefully designed to maximize the chances of finding a successful value without trying all the others. The only requirement is that it must pursue the alternatives in some fashion that is guaranteed to find a successful value if there is one. The point of the *choose* function is that we can separate the design of the choosing mechanism from the design of the program that needs a value and calls *choose* to find it one.

EXAMPLE 4.7 Nondeterministically Choosing a Travel Plan

Suppose that we regularly plan medium length trips. We are willing to drive or to fly and rent a car or to take a train and use public transportation if it is available when we get there, as long as the total cost of the trip and the total time required are reasonable. We don't care about small differences in time or cost enough to

make it worth exhaustively exploring all the options every time. We can define the function *trip-plan* to solve our problem:

trip-plan (*start*, *finish*)=
  Return (*choose* (*fly-major-airline-and-rent-car* (*start*, *finish*));;
    fly-regional-airline-and-rent-car (*start*, *finish*);;
    take-train-and-use-public-transportation (*start*, *finish*);;
    drive (*start*,*finish*))).

Each of the four functions *trip-plan* calls returns with a successful value iff it succeeds in finding a plan that meets the cost and time requirements. Probably the first three of them are implemented as an Internet agent that visits the appropriate Web sites, specifies the necessary parameters, and waits to see if a solution can be found. But notice that *trip-plan* can return a result as soon as at least one of the four agents finds an acceptable solution. It doesn't care whether the four agents can be run in parallel or are tried sequentially. It just wants to know if there's a solution and, if so, what it is.

A good deal of the power of choose comes from the fact that it can be called recursively. So it can be used to describe a search process, without having to specify the details of how the search is conducted.

EXAMPLE 4.8 Nondeterministically Searching a Space of Puzzle Moves

Suppose that we want to solve the 15-puzzle 💻. We are given two configurations of the puzzle, for example the ones shown here labeled (a) and (b). The goal is to begin in configuration (a) and, through a sequence of moves, reach configuration (b). The only allowable move is to slide a numbered tile into the blank square.

5	2	15	9
7	8	4	12
13	1	6	11
10	14	3	

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

EXAMPLE 4.8 (*Continued*)

Using *choose*, we can easily write *solve-15*, a program that finds a solution if there is one. The idea is that *solve*-15 will guess at a first move. From the board configuration that results from that move, it will guess at a second move. From there, it will guess at a third move, and so on. If it reaches the goal configuration, it will report the sequence of moves that got it there.

Using the second form of *choose* (in which values are selected from a set that can be generated each time a new choice must be made), we can define *solve*-15 so that it returns an ordered list of board positions. The first element of the list corresponds to the initial configuration. Following that, in order, are the configurations that result from each of the moves. The final configuration will correspond to the goal. So the result of a call to *solve*-15 will describe a move sequence that corresponds to a solution to the original problem. We'll invoke *solve*-15 with a list that contains just the initial configuration. So we define:

solve-15 (*position-list*)=

/* Explore moves available from the last board configuration to have been generated.

current = *last* (*position-list*).

If *current* = *solution* then return (*position-list*).

/* Assume that *successors* (*current*) returns the set of configurations that can be generated by one legal move from *current*. Then *choose* picks one with the property that, once it has been appended to *position-list*, *solve-15* can continue and find a solution. We assume that *append* destructively modifies its first argument.

choose (*x* from *successors* (*current*): *solve-15* (*append* (*position-list*, *x*))).

Return *position-list*.

If there is a solution to a particular instance of the 15-puzzle, *solve-15* will find it. If we care about how efficiently the solution is found, then we can dig inside the implementation of *choose* and try various strategies, including:

- Checking to make sure we don't generate a board position that has already been explored, or
- Sorting the successors by how close they are to the goal.

But if we don't care about how *choose* works, we don't have to.

15-puzzle configurations can be divided into two equivalence classes. Every configuration can be transformed into every other configuration in the same class and into none of the configurations in the other class 💻.

Many decision problems can be solved straightforwardly using *choose*.

EXAMPLE 4.9 Nondeterministically Searching for a Satisfying Assignment

A wff in Boolean logic is ***satisfiable*** iff it is true for at least one assignment of truth values to the literals it contains. Now consider the following problem, which we'll call SAT: Given a Boolean wff w, decide whether or not w is satisfiable.

To see how we might go about designing a program to solve the SAT problem, consider an example wff $w = P \wedge (Q \vee R) \wedge \neg (R \vee S) \rightarrow Q$. We can build a program that considers the predicate symbols (in this case P, Q, R, and S) in some order. For each one, it will pick one of the two available values, *True* or *False*, and assign it to all occurrences of that predicate symbol in w. When no predicate symbols remain, all that is necessary is to use the truth table definitions of the logical operators to simplify w until it has evaluated to either *True* or *False*. If *True*, then we have found an assignment of values to the predicates that makes w true; w is satisfiable. If *False*, then this path fails to find such an assignment and it fails. This procedure must halt because w contains only a finite number of predicate symbols, one is eliminated at each step, and there are only two values to choose from at each step. So either some path will return *True* or all paths will eventually halt and return *False*.

The following algorithm returns *True* if the answer to the question is yes and *False* if the answer to the question is no:

```
decideSAT (w : Boolean wff)=
    If there are no predicate symbols in w then:
       Simplify w until it is either True or False.
       Return w.
    Else:
       Find P, the first predicate symbol in w.
       /* Let w/P/x mean the wff w with every instance of P replaced
          by x.
       Return choose (decideSAT (w/P/True);;
                      decideSAT (w/P/False)).
```

One way to envision the execution of a program like *solve*-15 or *decideSAT* is as a search tree. Each node in the tree corresponds to a snapshot of *solve*-15 or *decideSAT* and each path from the root to a leaf node corresponds to one computation that *solve*-15 or *decideSAT* might perform. For example, if we invoke *decideSAT* on the input $P \wedge \neg R$, the set of possible computations can be described by the tree in Figure 4.1. The first level in the tree corresponds to guessing a value for P and the second level corresponds to guessing a value for R.

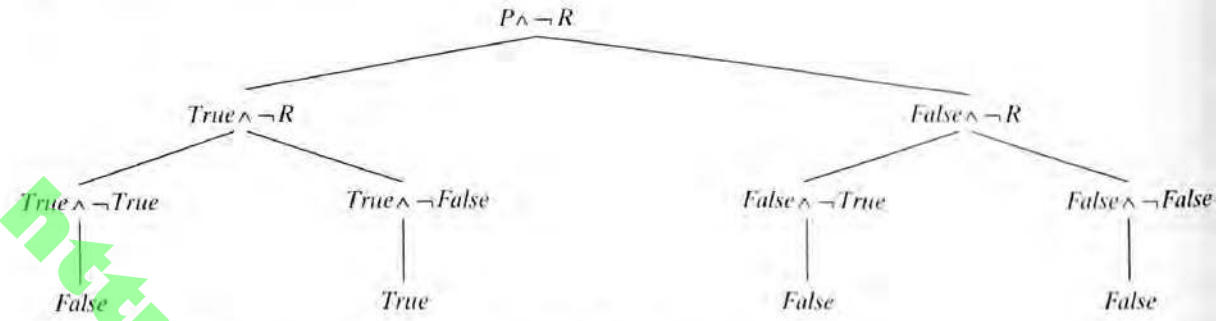


FIGURE 4.1 A search tree created by *decideSAT* on the input $P \wedge \neg R$.

Since there exists at least one computational path that succeeds (i.e., returns a value other than *False*), *decideSAT* will pick the value returned by one such path and return it. So *decideSAT* will return *True*. It may do so after exploring all four of the paths shown above (if it is unlucky choosing an order in which to explore the paths). Or it may guess correctly and find the successful path without considering any of the others.

> Efficient algorithms for solving Boolean satisfiability problems are important in a wide variety of domains. No general and efficient algorithms are known. But, in B.1.3, we'll describe ordered binary decision diagrams (OBDDs), which are used in SAT solvers that work, in practice, substantially more efficiently than *decideSAT* does.

One of the most important properties of programs that exploit *choose* is clear from the simple tree that we just examined: Guesses that do not lead to a solution can be effectively ignored in any analysis that is directed at determining the program's result.

Does adding *choose* to our programming language let us solve any problems that we couldn't solve without it? The answer to that question turns out to depend on what else the programming language already lets us do.

Suppose, for example, that we are describing our programs as finite state machines (FSMs). One way to add *choose* to the FSM model is to allow two or more transitions, labeled with the same input character, to emerge from a single state. We show a simple example in Figure 4.2.

We'll say that a nondeterministic FSM M (i.e., one that may exploit *choose*) accepts iff at least one of its paths accepts. It will reject iff all of its paths reject. So M's job is to

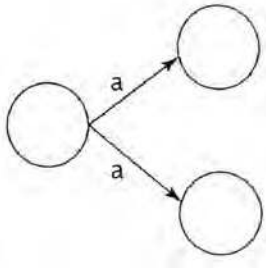


FIGURE 4.2 A nondeterministic FSM with two competing transitions labeled a.

find an accepting path if there is one. If it succeeds, it can ignore all other paths. If M exploits *choose* and does contain competing transitions, then one way to view its behavior is that it makes a guess and chooses an accepting path if it can.

While we will find it very convenient to allow nondeterminism like this in finite state machines, we will see in Section 5.4 that, whenever there is a nondeterministic FSM to accept some language L, there is also a (possibly much larger and more complicated) deterministic FSM that accepts L. So adding *choose* doesn't change the class of languages that can be accepted.

Now suppose that we are describing our programs as pushdown automata (PDAs). Again we will add *choose* to the model by allowing competing transitions coming out of a state. As we will see in Chapter 13, now the answer is that adding *choose* adds power. There are languages that can be accepted by PDAs that exploit *choose* that cannot be accepted by any PDA that does not exploit it.

Lastly, suppose that we are describing our programs as Turing machines or as code in a standard, modern programming language. Then, as we will see in Chapter 17, we are back to the situation we were in with FSMs. Nondeterminism is a very useful design tool that lets us specify complex programs without worrying about the details of how the search is managed. But, if there is a nondeterministic Turing machine that solves a problem, then there is a deterministic one (one that does not exploit *choose*) that also solves the problem.

In the two cases (FSMs and Turing machines) in which adding *choose* does not add computational power to our model, we will see that it does add descriptive power. We'll see examples for which a very simple nondeterministic machine can do the work of a substantially more complex deterministic one. We'll present algorithms, for both FSMs and Turing machines, that construct, given an arbitrary nondeterministic machine, an equivalent deterministic one. Thus we can use nondeterminism as an effective design tool and leave the job of building a deterministic program to a compiler.

In Part V, we will take a different look at analyzing problems and the programs that solve them. There we will be concerned with the complexity of the solution: How much running time does it take or how much memory does it require? In that analysis, nondeterminism will play another important role. It will enable us to separate our solution to a problem into two parts:

1. The complexity of an individual path through the search tree that *choose* creates. Each such path will typically correspond to checking one complete guess to see if it is a solution to the problem we are trying to solve.
2. The total complexity of the entire search process.

So, although nondeterminism may at first seem at odds with our notion of effective computation, we will find throughout this book that it is a very useful tool in helping us to analyze problems and see how they fit into each of the models that we will consider.

For some problems, it is useful to extend *choose* to allow probabilities to be associated with each of the alternatives. For example, we might write:

choose ((.5) action 1;;
(.3) action 2;;
(.2) action 3)

For some applications, the semantics we will want for this extended form of *choose* will be that exactly one path should be pursued. Let Pr(n) be the probability associated with alternative n. Then *choose* will select alternative n with probability Pr(n). For other applications, we will want a different semantics: All paths should be pursued and a total probability should be associated with each path as a function of the set of probabilities associated with each step along the path. We will have more to say about how these probabilities actually work when we talk about specific applications.

4.3 Functions on Languages and Programs

In Chapter 2, we described some useful functions on languages. We considered simple functions such as complement, concatenation, union, intersection, and Kleene star. All of those were defined by straightforward extension of the standard operations on sets and strings. Functions on languages are not limited to those, however. In this section, we mention a couple of others, which we'll come back to at various points throughout this book.

EXAMPLE 4.10 The Function *chop*

Define $chop(L) = \{w : \exists x \in L\ (x = x_1 c x_2 \wedge x_1 \in \Sigma_L{}^* \wedge x_2 \in \Sigma_L{}^* \wedge c \in \Sigma_L \wedge |x_1| = |x_2| \wedge w = x_1 x_2)\}$. In other words, *chop*(L) is all the odd length strings in L with their middle character chopped out.

Recall the language $A^nB^n = \{a^n b^n : n \geq 0\}$. What is *chop* ($A^nB^n$)? The answer is ∅, since there are no odd length strings in A^nB^n.

What about $A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$? What is *chop* ($A^nB^nC^n$)? Approximately half of the strings in $A^nB^nC^n$ have odd length and so can have their middle character chopped out. Strings in $A^nB^nC^n$ contribute strings to *chop* ($A^nB^nC^n$) as follows:

n	in $A^nB^nC^n$	in *chop* $A^nB^nC^n$
0	ε	
1	abc	ac
2	aabbcc	
3	aaabbbccc	aaabbccc
4	aaaabbbbcccc	
5	aaaaabbbbbccccc	aaaaabbbbccccc

So, $chop(A^nB^nC^n) = \{a^{2n+1} b^{2n} c^{2n+1} : n \geq 0\}$.

EXAMPLE 4.11 The Function *firstchars*

Define $firstchars(L) = \{w : \exists y \in L\ (y = cx \wedge c \in \Sigma_L \wedge x \in \Sigma_L^* \wedge w \in c^*)\}$. So we could determine $firstchars(L)$ by looking at all the strings in L, finding all the characters that start such strings, and then, for each such character c, adding to $firstchars(L)$ all the strings in c^*. Let's look at *firstchars* applied to some languages:

L	$firstchars(L)$
$\varnothing$	$\varnothing$
$\{\varepsilon\}$	$\varnothing$
{a}	{a}*
A^nB^n	{a}*
{a, b}*	{a}* ∪ {b}*

Given some function f on languages, we may want to ask the question, "If L is a member of some language class C, what can we say about $f(L)$? Is it too a member of C? Alternatively, is the class C closed under f?"

EXAMPLE 4.12 Are Language Classes Closed Under Various Functions?

Consider two classes of languages, INF (the set of infinite languages) and FIN (the set of finite languages). And consider four of the functions we have discussed: union, intersection, *chop* and *firstchars*. We will ask the question, "Is class C closed under function f?" The answers are (with the number in each cell pointing to an explanation below for the corresponding answer):

	FIN	INF
union	yes (1)	yes (5)
intersection	yes (2)	no (6)
chop	yes (3)	no (7)
firstchars	no (4)	yes (8)

1. For any sets A and B, $|A \cup B| \leq |A| + |B|$.
2. For any sets A and B, $|A \cap B| \leq min\,(|A|, |B|)$.
3. Each string in L can generate at most one string in $chop\ (L)$, so $|chop\ (L)| \leq |L|$.

EXAMPLE 4.12 (*Continued*)

4. To show that any class C is not closed under some function f it is sufficient to show a single counter example: a language L where $L \in C$ but $f(L) \notin C$. We showed such a counter example above: *firstchars* $(\{\mathsf{a}\}) = \{\mathsf{a}\}^*$.
5. For any sets A and B, $|A \cup B| \geq |A|$.
6. We show one counterexample: Let $L_1 = \{\mathsf{a}\}^*$ and $L_2 = \{\mathsf{b}\}^*$. L_1 and L_2 are infinite. But $L_1 \cap L_2 = \{\varepsilon\}$, which is finite.
7. We have already shown a counterexample: $\mathrm{A^nB^n}$ is infinite. But *Chop* $(\mathrm{A^nB^n}) = \varnothing$, which is finite.
8. If L is infinite, then it contains at least one string of length greater than 0. That string has some first character c. Then $\{c\}^* \subseteq$ *firstchars* (L) and $\{c\}^*$ is infinite.

In the rest of this book, we will discuss the four classes of languages: regular, context-free, decidable, and semidecidable, as described in Chapter 3. One of the questions we will ask for each of them is whether they are closed under various operations.

Given some function f on languages, how can we:

1. Implement f?
2. Show that some class of languages is closed under f?

The answer to question 2 is generally by construction. In other words, we will show an algorithm that takes a description of the input language(s) and constructs a description of the result of applying f to that input. We will then use that constructed description to show that the resulting language is in the class we care about. So our ability to answer both questions 1 and 2 hinges on our ability to define an algorithm that computes f, given a description of its input (which is one or more languages).

In order to define an algorithm A to compute some function f, we first need a way to define the input to A. Defining A is going to be very difficult if we allow, for example, English descriptions of the language(s) on which A is supposed to operate. What we need is a formal model that is exactly powerful enough to describe the languages on which we would like A to be able to run. Then A could use the description(s) of its input language(s) to build a new description, using the same model, of the result of applying f.

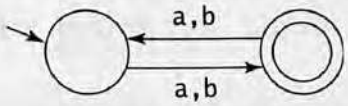
EXAMPLE 4.13 Representing Languages So That Functions Can Be Applied

Suppose that we wish to compute the function union. It will be very hard to implement union if we allow input language description such as:

- $\{w \in \{\mathsf{a}, \mathsf{b}\}^* : w$ has an odd number of characters$\}$.

- $\{w \in \{a, b\}^* : w$ has an even number of a's$\}$.
- $\{w \in \{a, b\}^* :$ all a's in w precede all b's$\}$.

Suppose, on the other hand, that we describe each of these languages as a finite state machine that accepts them. So, for example, language 1 would be represented as



In Chapter 8, we will show an algorithm that, given two FSMs, corresponding to two regular languages, L_1 and L_2, constructs a new FSM that accepts the union of L_1 and L_2.

If we use finite state machines (or pushdown automata or Turing machines) as input I to an algorithm A that computes some function f, then what A will do is to manipulate those FSMs (or PDAs or Turing machines) and produce a new one that accepts the language $f(I)$. If we think of the input FSMs (or PDAs or Turing machines) as programs, then A is a program whose input and output are other programs.

> Lisp is a programming language that makes it easy to write programs that manipulate programs. (G.5)

Programs that write other programs are not particularly common, but they are not fundamentally different from programs that work with any other data type. Programs in any conventional programming language can be expressed as strings, so any program that can manipulate strings can manipulate programs. Unfortunately, the syntax of most programming languages makes it relatively difficult to design programs that can effectively manipulate other programs. As we will see later, the FSM, PDA, and Turing machine formalisms that we are going to focus on are reasonably easy to work with. Programs that perform functions on FSMs, PDAs, and Turing machines will be an important part of the theory that we are about to build.

> Programs that write other programs play an important role in some application areas, including mathematical modeling of such things as oil wells and financial markets. (G. 8)

Exercises

1. Describe in clear English or pseudocode a decision procedure to answer the question, "Given a list of integers N and an individual integer n, is there any element of N that is a factor of n?"
2. Given a Java program p and the input 0, consider the question, "Does p ever output anything?"
   a. Describe a semidecision procedure that answers this question.
   b. Is there an obvious way to turn your answer to part a into a decision procedure?
3. Recall the function *chop* (L), defined in Example 4.10. Let $L = \{w \in \{a, b\}^* : w = w^R\}$. What is *chop* (L)?
4. Are the following sets closed under the following operations? Prove your answer. If a set is not closed under the operation, what is its closure under the operation?
   a. $L = \{w \in \{a, b\}^* : w$ ends in a$\}$ under the function *odds*, defined on strings as follows: *odds*(s) = the string that is formed by concatenating together all of the odd numbered characters of s. (Start numbering the characters at 1.) For example, *odds*(ababbbb) = aabb.
   b. FIN (the set of finite languages) under the function *oddsL*, defined on languages as follows:

$$oddsL\ (L) = \{w : \exists x \in L\ (w = odds\ (x))\}.$$

   c. INF (the set of infinite languages) under the function *oddsL*.
   d. FIN under the function *maxstring*, defined in Example 8.22.
   e. INF under the function *maxstring*.
5. Let $\Sigma = \{a, b\}$. Let S be the set of all languages over Σ. Let f be a binary function defined as follows:

$$f : S \times S \rightarrow S.$$

$$f(x, y) = x - y.$$

   Answer each of the following questions and justify your answer:
   a. Is f one-to-one?
   b. Is f onto?
   c. Is f commutative?
6. Describe a program, using *choose*, to:
   a. Play Sudoku 💻, described in N. 2.2.
   b. Solve Rubik's Cube® 💻.

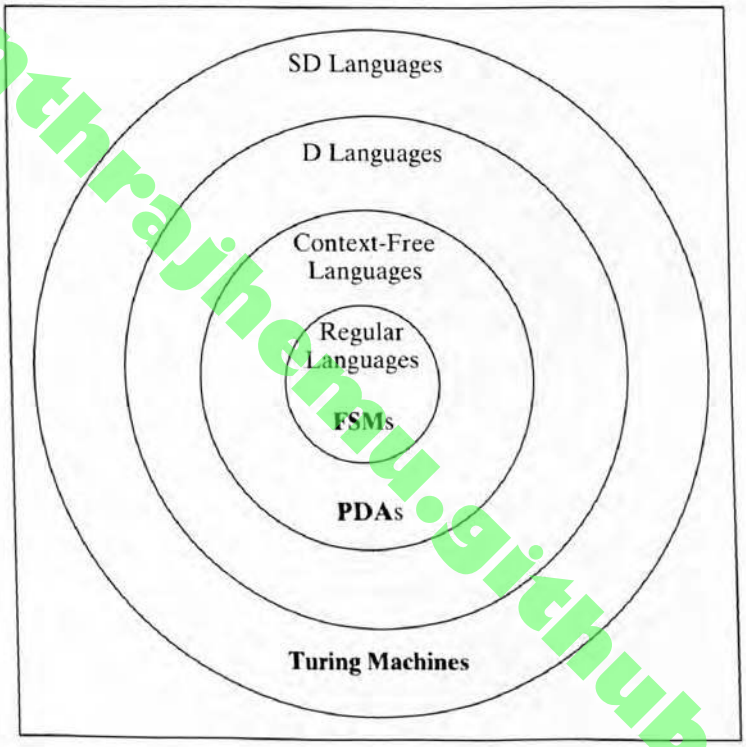
PART II

FINITE STATE MACHINES AND REGULAR LANGUAGES

In this section, we begin our exploration of the language hierarchy. We will start in the inner circle, which corresponds to the class of regular languages.

We will explore three techniques, which we will prove are equivalent, for defining the regular languages:

- Finite state machines.
- Regular languages.
- Regular grammars.



CHAPTER 5

Finite State Machines

The simplest and most efficient computational device that we will consider is the finite state machine (or FSM).

EXAMPLE 5.1 A Vending Machine

Consider the problem of deciding when to dispense a drink from a vending machine. To simplify the problem a bit, we'll pretend that it were still possible to buy a drink for $.25 and we will assume that vending machines do not take pennies. The solution that we will present for this problem can straightforwardly be extended to modern, high-priced machines.

The vending machine controller will receive a sequence of inputs, each of which corresponds to one of the following events:

- A coin is deposited into the machine. We can use the symbols N (for nickel), D (for dime), and Q (for quarter) to represent these events.
- The coin return button is pushed. We can use the symbol R (for return) to represent this event.
- A drink button is pushed and a drink is dispensed. We can use the symbol S (for soda) for this event.

After any finite sequence of inputs, the controller will be in either:

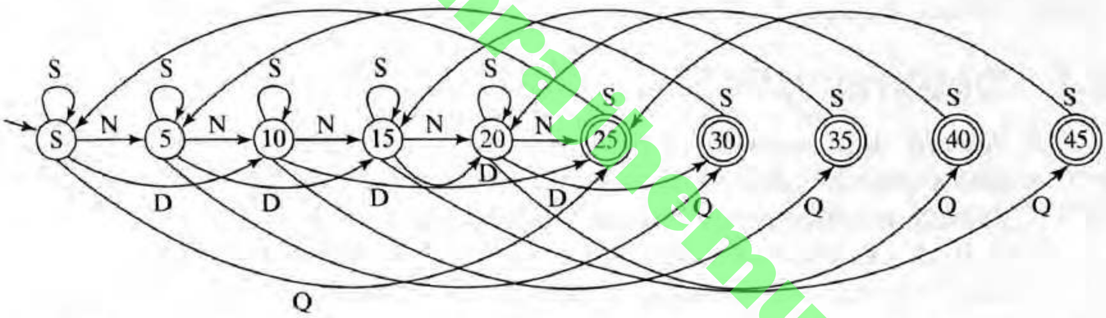
- A dispensing state, in which it is willing to dispense a drink if a drink button is pushed.
- A nondispensing state, in which not enough money has been inserted into the machine.

While there is no bound on the length of the input sequence that a drink machine may see in a week, there is only a finite amount of history that its controller must remember in order to do its job. It needs only to be able to answer

the question, "Has enough money been inserted, since the last time a drink was dispensed, to purchase the next drink?" It is of course possible for someone to keep inserting money without ever pushing a dispense-drink button. But we can design a controller that will simply reject any money that comes in after the amount required to buy a drink has been recorded and before a drink has actually been dispensed. We will however assume that our goal is to design a customer-friendly drink machine. For example, the thirsty customer may have only dimes. So we'll build a machine that will accept up to $.45. If more than the necessary $.25 is inserted before a dispensing button is pushed, our machine will remember the difference and leave a "credit" in the machine. So, for example, if a customer inserts three dimes and then asks for drink, the machine will remember the balance of $.05.

Notice that the drink controller does not need to remember the actual sequence of coins that it has received. It need only remember the total *value* of the coins that have been inserted since the last drink was dispensed.

The drink controller that we have just described needs 10 states, corresponding to the possible values of the credit that the customer has in the machine: 0, 5, 10, 15, 20, 25, 30, 35, 40, and 45 cents. The main structure of the controller is then:



The state that is labeled *S* is the start state. Transitions from one state to the next are shown as arrows and labeled with the event that causes them to take place. As coins are deposited, the controller's state changes to reflect the amount of money that has been deposited. When the drink button is pushed (indicated as *S* in the diagram) and the customer has a credit of less than $.25, nothing happens. The machine's state does not change. If the drink button is pushed and the customer has a credit of $.25 or more, the credit is decremented by $.25 and a drink is dispensed. The drink-dispensing states, namely those that correspond to "enough money", can be thought of as goal or accepting states. We have shown them in the diagram with double circles.

Not all of the required transitions have been shown in the diagram. It would be too difficult to read. We must add to the ones shown all of the following:

- From each of the accepting states, a transition back to itself labeled with each coin value. These transitions correspond to our decision to reject additional coins once the machine has been fed the price of a drink.

EXAMPLE 5.1 *(Continued)*

- From each state, a transition back to the start state labeled R. These transitions will be taken whenever the customer pushes the coin return button. They correspond to the machine returning all of the money that it has accumulated since the last drink was dispensed.

The drink controller that we have just described is an example of a finite state machine. We can think of it as a device to solve a problem (dispense drinks). Or we can think of it as a device to recognize a language (the "enough money" language that consists of the set of strings, such as NDD, that drive the machine to an accepting state in which a drink can be dispensed). In most of the rest of this chapter, we will take the language recognition perspective. But it does also make sense to imagine a finite state machine that actually acts in the world (for example, by outputting a coin or a drink). We will return to that idea in Section 5.9.

The history of finite state machines substantially predates modern computers. (P. 1)

5.1 Deterministic Finite State Machines

A ***finite state machine*** (or FSM) is a computational device whose input is a string and whose output is one of two values that we can call *Accept* and *Reject*. FSMs are also sometimes called finite state automata or FSAs.

If M is an FSM, an input string is fed to M one character at a time, left to right. Each time it receives a character, M considers its current state and the new character and chooses a next state. One or more of M's states may be marked as accepting states. If M runs out of input and is in an accepting state, it accepts. If, however, M runs out of input and is not in an accepting state, it rejects. The number of steps that M executes on input w is exactly equal to $|w|$, so M always halts and either accepts or rejects.

We begin by defining the class of FSMs whose behavior is deterministic. In such machines, there is always exactly one move that can be made at each step; that move is determined by the current state and the next input character. In Section 5.4, we will relax this restriction and introduce nondeterministic FSMs (also called NDFSMs), in which there may, at various points in the computation, be more than one move from which the machine may choose. We will continue to use the term FSM to include both deterministic and nondeterministic FSMs.

A telephone switching circuit can easily be modeled as a DFSM.

Formally, a ***deterministic FSM*** (or ***DFSM***) M is a quintuple $(K, \Sigma, \delta, s, A)$, where:

- K is a finite set of states,
- Σ is the input alphabet,

- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states, and
- δ is the transition function. It maps from:

$$\underset{\text{state}}{K} \times \underset{\text{input symbol}}{\Sigma} \text{ to } \underset{\text{state}}{K}.$$

A ***configuration*** of a DFSM M is an element of $K \times \Sigma^*$. Think of it as a snapshot of M. It captures the two things that can make a difference to M's future behavior:

- Its current state.
- The input that is still left to read.

The ***initial configuration*** of a DFSM M, on input w, is (s_M, w), where s_M is the start state of M. (We can use the subscript notation to refer to components of a machine M's definition, although, when the context makes it clear what machine we are talking about, we may omit the subscript.)

The transition function δ defines the operation of a DFSM M one step at a time. We can use it to define the sequence of configurations that M will enter. We start by defining the relation *yields-in-one-step*, written $|-_M$. *Yields-in-one-step* relates *configuration*$_1$ to *configuration*$_2$ iff M can move from *configuration*$_1$ to *configuration*$_2$ in one step. Let c be any element of Σ and let w be any element of Σ^*. Then,

$$(q_1, cw) \mid-_M (q_2, w) \text{ iff } ((q_1, c), q_2) \in \delta.$$

We can now define the relation *yields*, written $|-_M^*$ to be the reflexive, transitive closure of $|-_M$. So configuration C_1 yields configuration C_2 iff M can go from C_1 to C_2 in zero or more steps. In this case, we will write:

$$C_1 \mid-_M^* C_2.$$

A ***computation*** by M is a finite sequence of configurations $C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that:

- C_0 is an initial configuration,
- C_n is of the form (q, ε), for some state $q \in K_M$ (i.e., the entire input string has been read), and
- $C_0 \mid-_M C_1 \mid-_M C_2 \mid-_M \ldots \mid-_M C_n$.

Let w be an element of Σ^*. Then we will say that:

- M ***accepts*** w iff $(s, w) \mid-_M^* (q, \varepsilon)$, for some $q \in A_M$. Any configuration (q, ε), for some $q \in A_M$, is called an ***accepting configuration*** of M.
- M ***rejects*** w iff $(s, w) \mid-_M^* (q, \varepsilon)$, for some $q \notin A_M$. Any configuration (q, ε), for some $q \notin A_M$, is called an ***rejecting configuration*** of M.

M halts whenever it enters either an accepting or a rejecting configuration. It will do so immediately after reading the last character of its input.

The ***language accepted by*** M, denoted $L(M)$, is the set of all strings accepted by M.

EXAMPLE 5.2 A Simple Language of a's and b's

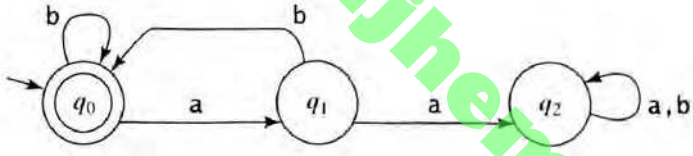
Let $L = \{w \in \{a, b\}^* : \text{every a is immediately followed by a b}\}$. L can be accepted by the DFSM $M = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_0\})$, where:

$$\begin{aligned}\delta = \{&((q_0, a), q_1),\\ &((q_0, b), q_0),\\ &((q_1, a), q_2),\\ &((q_1, b), q_0),\\ &((q_2, a), q_2),\\ &((q_2, b), q_2))\}.\end{aligned}$$

The tuple notation that we have just used for δ is quite hard to read. We will generally find it useful to draw δ as a transition diagram instead. When we do that, we will use two conventions:

1. The start state will be indicated with an unlabeled arrow pointing into it.
2. The accepting states will be indicated with double circles.

With those conventions, a DFSM can be completely specified by a transition diagram. So M is:



We will use the notation a, b as a shorthand for two transitions, one labeled a and one labeled b.

As an example of M's operation, consider the input string abbabab. M's computation is the sequence of configurations: $(q_0, \text{abbabab})$, (q_1, bbabab), (q_0, babab), (q_0, abab), (q_1, bab), (q_0, ab), (q_1, b), (q_0, ε). Since q_0 is an accepting state, M accepts.

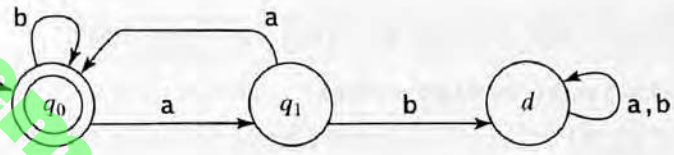
If we look at the three states in M, the machine that we just built, we see that they are of three different sorts:

1. State q_0 is an accepting state. Every string that drives M to state q_0 is in L.
2. State q_1 is not an accepting state. But every string that drives M to state q_1 could turn out to be in L if it is followed by an appropriate continuation string, in this case, one that starts with a b.

3. State q_2 is what we will call a ***dead state***. Once M enters state q_2, it will never leave. State q_2 is not an accepting state, so any string that drives M to state q_2 has already been determined not to be in L, *no matter what comes next*. We will often name our dead states d.

EXAMPLE 5.3 Even Length Regions of a's

Let $L = \{w \in \{a, b\}^* : \text{every a region in } w \text{ is of even length}\}$. L can be accepted by the DFSM M:



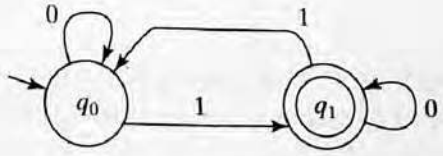
If M sees a b in state q_1, then there has been an a region whose length is odd. So, no matter what happens next, M must reject. So it goes to the dead state d.

> A useful way to prototype a complex system is as a finite state machine. See P. 4 for one example: the controller for a soccer-playing robot.

Because objects of other data types are encoded in computer memories as binary strings, it is important to be able to check key properties of such strings.

EXAMPLE 5.4 Checking for Odd Parity

Let $L = \{w \in \{0, 1\}^* : w \text{ has odd parity}\}$. A binary string has odd parity iff the number of 1's in it is odd. So L can be accepted by the DFSM M:



One of the most important properties of finite state machines is that they are guaranteed to halt on any input string of finite length. While this may seem obvious, it is worth noting since, as we'll see later, more powerful computational models may not share this property.

THEOREM 5.1 DFSMs Halt

Theorem: Every DFSM M, on input w, halts after $|w|$ steps.

Proof: On input w, M executes some computation $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \ldots \vdash_M C_n$, where C_0 is an initial configuration and C_n is of the form (q, ε), for some state $q \in K_M$. C_n is either an accepting or a rejecting configuration, so M will halt when it reaches C_n. Each step in the computation consumes one character of w. So $n = |w|$. Thus M will halt after $|w|$ steps.

5.2 The Regular Languages

We have now built DFSMs to accept four languages:

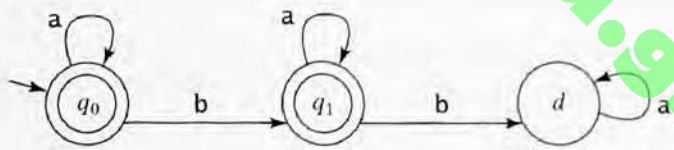
- "enough money to buy a drink",
- $\{w \in \{a, b\}^* :$ every a is immediately followed by a b$\}$,
- $\{w \in \{a, b\}^* :$ every a region in w is of even length$\}$, and
- binary strings with odd parity.

These four languages are typical of a large class of languages that can be accepted by finite state machines.

We define the set of ***regular languages*** to be exactly those that can be accepted by some DFSM.

EXAMPLE 5.5 No More Than One b

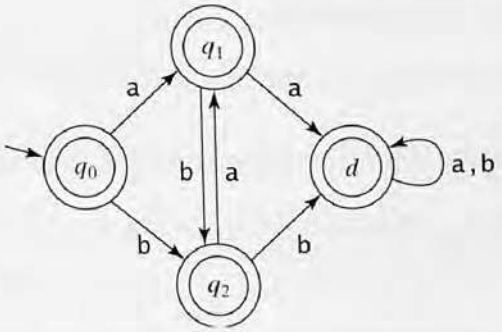
Let $L = \{w \in \{a, b\}^* : w$ contains no more than one b$\}$. L is regular because it can be accepted by the DFSM M:



Any string with more than one b will drive M to the dead state d. All other strings will drive M to either q_0 or q_1, both of which are accepting states.

EXAMPLE 5.6 No Two Consecutive Characters Are the Same

Let $L = \{w \in \{a, b\}^* :$ no two consecutive characters are the same$\}$. L is regular because it can be accepted by the DFSM M:



The start state, q_0, is the only state in which both a and b are legal inputs. M will be in state q_1 whenever the consecutive characters rule has not been violated and the last character it has read was a. At that point, the only legal next character is b. M will be in state q_2 whenever the consecutive characters rule has not been violated and the last character it has read was b. At that point, the only legal next character is a. Any other inputs drive M to d.

Simple languages of a's and b's, like the ones in the last two examples, are useful for practice in designing DFSMs. But the real power of the DFSM model comes from the fact that the languages that arise in many real-world applications are regular.

> The language of universal resource identifiers (URIs), used to describe objects on the World Wide Web, is regular. (L3.1)

To describe less trivial languages will sometimes require DFSMs that are hard to draw if we include the dead state. In those cases, we will omit it from our diagrams. This doesn't mean that it doesn't exist. δ is a function that must be defined for all (state, input) pairs. It just means that we won't bother to draw the dead state. Instead, our convention will be that if there is no transition specified for some (state, input) pair, then that pair drives the machine to a dead state.

EXAMPLE 5.7 Floating Point Numbers

Let FLOAT = $\{w : w$ is the string representation of a floating point number$\}$. Assume the following syntax for floating point numbers:

- A floating point number is an optional sign, followed by a decimal number, followed by an optional exponent.
- A decimal number may be of the form x or $x.y$, where x and y are nonempty strings of decimal digits.

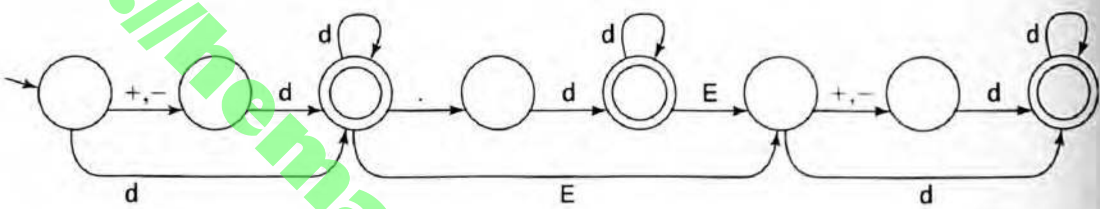
EXAMPLE 5.7 *(Continued)*

- An exponent begins with E and is followed by an optional sign and then an integer.
- An integer is a nonempty string of decimal digits.

So, for example, these strings represent floating point numbers:

+3.0, 3.0, 0.3E1, 0.3E+1, −0.3E+1, −3E8

FLOAT is regular because it can be accepted by the DFSM:



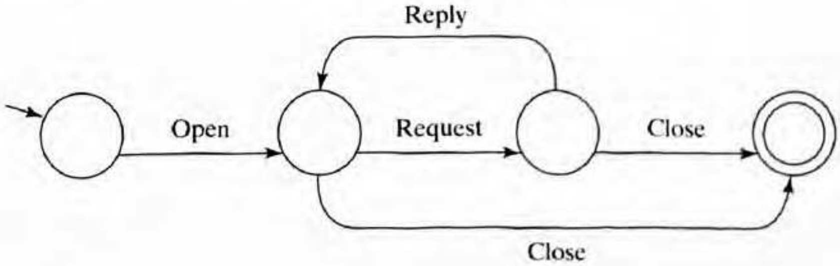
In this diagram, we have used the shorthand d to stand for any one of the decimal digits (0 − 9). And we have omitted the dead state to avoid arrows crossing over each other.

EXAMPLE 5.8 A Simple Communication Protocol

Let L be a language that contains all the legal sequences of messages that can be exchanged between a client and a server using a simple communication protocol. We will actually consider only a very simplified version of such a protocol, but the idea can be extended to a more realistic model.

Let Σ_L = {Open, Request, Reply, Close}. Every string in L begins with Open and ends with Close. In addition, every Request, except possibly the last, must be followed by Reply and no unsolicited Reply's may occur.

L is regular because it can be accepted by the DFSM:



Note that we have again omitted the dead state.

More realistic communication protocols can also be modeled as FSMs. (I.1)

5.3 Designing Deterministic Finite State Machines

Given some language L, how should we go about designing a DFSM to accept L? In general, as in any design task, there is no magic bullet. But there are two related things that it is helpful to think about:

- Imagine any DFSM M that accepts L. As a string w is being read by M, what properties of the part of w that has been seen so far are going to have any bearing on the ultimate answer that M needs to produce? Those are the properties that M needs to record. So, for example, in the "enough money" machine, all that matters is the amount of money since the last drink was dispensed. Which coins came in and the order in which they were deposited make no difference.
- If L is infinite but M has a finite number of states, strings must "cluster". In other words, multiple different strings will all drive M to the same state. Once they have done that, none of their differences matter anymore. If they've driven M to the same state, they share a fate. No matter what comes next, either all of them cause M to accept or all of them cause M to reject. In Section 5.7 we will show that the smallest DFSM for any language L is the one that has exactly one state for every group of initial substrings that share a common fate. For now, however, it helps to think about what those clusters are. We'll do that in our next example.

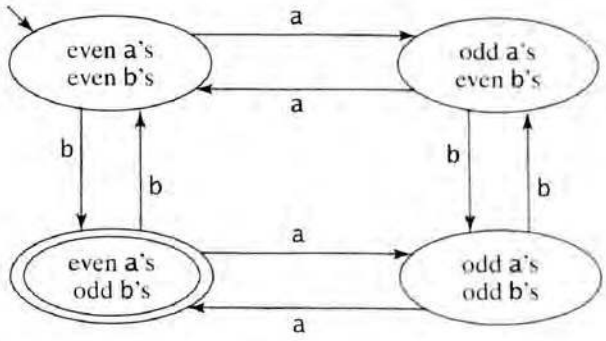
A building security system can be described as a DFSM that sounds an alarm if given an input sequence that signals an intruder. (J.1)

EXAMPLE 5.9 Even a's, Odd b's

Let $L = \{w \in \{a, b\}^* : w$ contains an even number of a's and an odd number of b's$\}$. To design a DFSM M to accept L, we need to decide what history matters. Since M's goal is to separate strings with even a's and odd b's from strings that fail to meet at least one of those requirements, all it needs to remember is whether the count of a's so far is even or odd and whether the count of b's is even or odd. So, since there are two clusters based on the number of a's so far (even and odd) and two clusters based on the number of b's, there are four distinct clusters.

That suggests that we need a four-state DFSM. Often it helps to name the states with a description of the clusters to which they correspond. The following DFSM M accepts L:

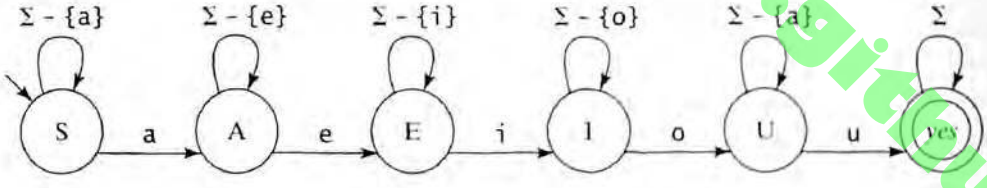
EXAMPLE 5.9 *(Continued)*



Notice that, once we have designed a machine that analyzes an input string with respect to some set of properties we care about, it is relatively easy to build a different machine that accepts strings based on different values of those properties. For example, to change M so that it accepts exactly the strings with both even a's and even b's, all we need to do is to change the accepting state.

EXAMPLE 5.10 All the Vowels in Alphabetical Order

Let $L = \{w \in \{a - z\}^* :$ all five vowels, a, e, i, o, and u, occur in w in alphabetical order$\}$. So L contains words like abstemious, facetious, and sacrilegious. But it does not contain tenacious, which does contain all the vowels, but not in the correct order. It is hard to write a clear, elegant program to accept L. But designing a DFSM is simple. The following machine M does the job. In this description of M, let the label "$\Sigma - \{a\}$" mean "all elements of Σ except a" and let the label "Σ" mean "all elements of Σ":

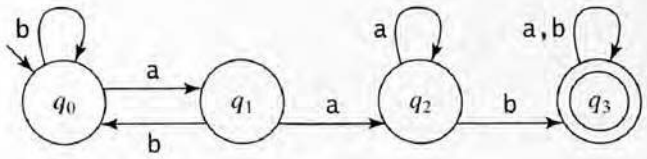


Notice that the state that we have labeled *yes* functions exactly opposite to the way in which the dead state works. If M ever reaches *yes*, it has decided to accept no matter what comes next.

Sometimes an easy way to design an FSM to accept a language L is to begin by designing an FSM to accept the complement of L. Then, as a final step, we swap the accepting and the nonaccepting states.

EXAMPLE 5.11 A Substring that Doesn't Occur

Let $L = \{w \in \{a, b\}^* : w$ does not contain the substring aab$\}$. It is straightforward to design an FSM that looks for the substring aab. So we can begin building a machine to accept L by building the following machine to accept $\neg L$:



Then we can convert this machine into one that accepts L by making states q_0, q_1, and q_2 accepting and state q_3 nonaccepting.

In Section 8.3 we'll show that the regular languages are closed under complement (i.e., the complement of every regular language is also regular). The proof will be by construction and the last step of the construction will be to swap accepting and nonaccepting states, just as we did in the last example.

Sometimes the usefulness of the DFSM model, as we have so far defined it, breaks down before its formal power does. There are some regular languages that seem quite simple when we state them but that can only be accepted by DFSMs of substantial complexity.

EXAMPLE 5.12 The Missing Letter Language

Let $\Sigma = \{a, b, c, d\}$. Let $L_{Missing} = \{w :$ there is a symbol $a_i \in \Sigma$ not appearing in $w\}$. $L_{Missing}$ is regular. We can begin writing out a DFSM M to accept it. We will need the following states:

- The start state: all letters are still missing.

After one character has been read, M could be in any one of:

- a read, so b, c, and d still missing.
- b read, so a, c, and d still missing.
- c read, so a, b, and d still missing.
- d read, so a, b, and c still missing.

After a second character has been read, M could be in any of the previous states or one of:

- a and b read, so c and d still missing.
- a and c read, so b and d still missing.
- and so forth. There are six of these.

EXAMPLE 5.12 *(Continued)*

After a third character has been read, M could be in any of the previous states or one of:

- a and b and c read, so d missing.
- a and b and d read, so c missing.
- a and c and d read, so b missing.
- b and c and d read, so a missing.

After a fourth character has been read, M could be in any of the previous states or:

- All characters read, so nothing is missing.

Every state except the last is an accepting state. M is complicated but it would be possible to write it out. Now imagine that Σ were the entire English alphabet. It would still be possible to write out a DFSM to accept $L_{Missing}$, but it would be so complicated it would be hard to get it right. The DFSM model is no longer very useful.

5.4 Nondeterministic FSMs

To solve the problem that we just encountered in the missing letter example, we will modify our definition of an FSM to allow nondeterminism. Recall our discussion of nondeterminism in Section 4.2. We will now introduce our first specific use of the ideas we discussed there. We'll see that we can easily build a nondeterministic FSM M to accept $L_{Missing}$. Any string in $L_{Missing}$ must be missing at least one letter. We'll design M so that it simply guesses at which letter that is. If there is a missing letter, then at least one of M's guesses will be right and the corresponding path will accept. So M will accept.

5.4.1 What Is a Nondeterministic FSM?

A nondeterministic FSM (or NDFSM) M is a quintuple $(K, \Sigma, \Delta, s, A)$, where:

- K is a finite set of states,
- Σ is an alphabet,
- $s \in K$ is the start state,
- $A \subseteq$ is the set of final states, and
- Δ is the transition relation. It is a finite subset of: $(K \times (\Sigma \cup \{\varepsilon\})) \times K$.

In other words, each element of Δ contains a (state, input symbol or ε) pair, and a new state.

We define configuration, initial configuration, accepting configuration, *yields-in-one-step*, *yields*, and computation analogously to the way that we defined them for DFSMs.

Let w be an element of Σ^*. Then we will say that:

- M ***accepts*** w iff *at least one* of its computations accepts.
- M ***rejects*** w iff none of its computations accepts.

The ***language accepted by*** M, denoted $L(M)$, is the set of all strings accepted by M.

There are two key differences between DFSMs and NDFSMs. In every configuration, a DFSM can make exactly one move. However, because Δ can be an arbitrary relation (that may not also be a function), that is not necessarily true for an NDFSM. Instead:

- An NDFSM M may enter a configuration in which there are still input symbols left to read but from which *no* moves are available. Since any sequence of moves that leads to such a configuration cannot ever reach an accepting configuration, M will simply halt without accepting. This situation is possible because Δ is not a function. So there can be (state, input) pairs for which no next state is defined.
- An NDFSM M may enter a configuration from which *two or more* competing moves are possible. The competition can come from either or both of the following properties of the transition relation of an NDFSM:
  - An NDFSM M may have one or more transitions that are labeled ε, rather than being labeled with a character from Σ. An ε-transition out of state q may (but need not) be followed, without consuming any input, whenever M is in state q. So an ε-transition from a state q competes with all other transitions out of q. One way to think about the usefulness of ε-transitions is that they enable M to guess at the correct path before it actually sees the input. Wrong guesses will generate paths that will fail but that can be ignored.
  - Out of some state q, there may be more than one transition with a given label. These competing transitions give M another way to guess at a correct path.

Consider the fragment, shown in Figure 5.1, of an NDFSM M. If M is in state q_0 and the next input character is an a, then there are three moves that M could make:

1. It can take the ε-transition to q_1 before it reads the next input character,
2. It can read the next input character and take the transition to q_2, or
3. It can read the next input character and take the transition to q_3.

One way to envision the operation of M is as a tree, as shown in Figure 5.2. Each node in the tree corresponds to a configuration of M. Each path from the root corresponds to a sequence of moves that M might make. Each path that leads to a configuration in which the entire input string has been read corresponds to a computation of M.

An alternative is to imagine following all paths through M in parallel. Think of M as being in a *set* of states at each step of its computation. If, when M runs out of input, the set of states that it is in contains at least one accepting state, then M will accept.

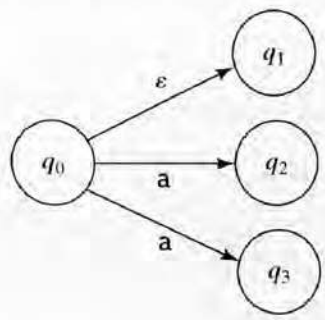


FIGURE 5.1 An NDFSM with two kinds of nondeterminism.

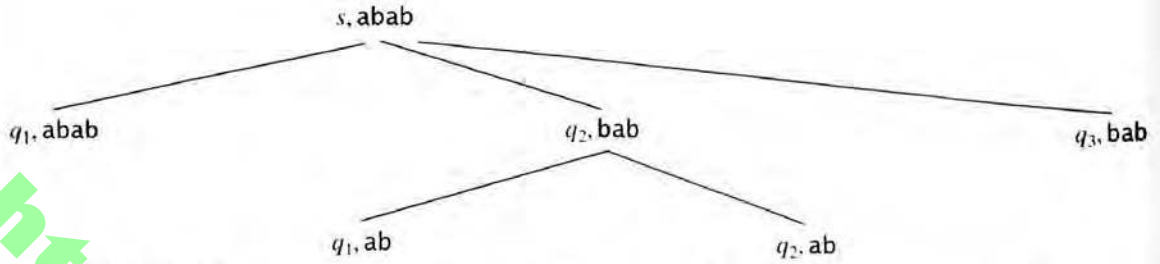
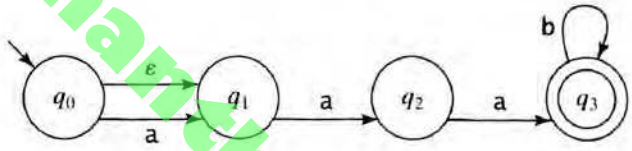


FIGURE 5.2 Viewing nondeterminism as search through a space of computation paths.

EXAMPLE 5.13 An Optional Initial a

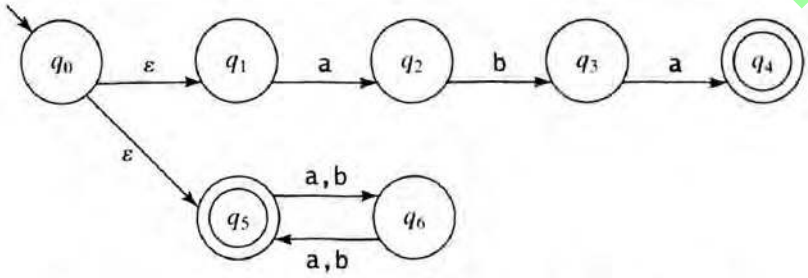
Let $L = \{w \in \{a, b\}^* : w$ is made up of an optional a followed by aa followed by zero or more b's$\}$. The following NDFSM M accepts L:



M may (but is not required to) follow the ε-transition from state q_0 to state q_1 before it reads the first input character. In effect, it must guess whether or not the optional a is present.

EXAMPLE 5.14 Two Different Sublanguages

Let $L = \{w \in \{a, b\}^* : w =$ aba or $|w|$ is even$\}$. An easy way to build an FSM to accept this language is to build FSMs for each of the individual sublanguages and then "glue" them together with ε-transitions. In essence, the machine guesses, when processing a string, which sublanguage the string might be in. So we have:

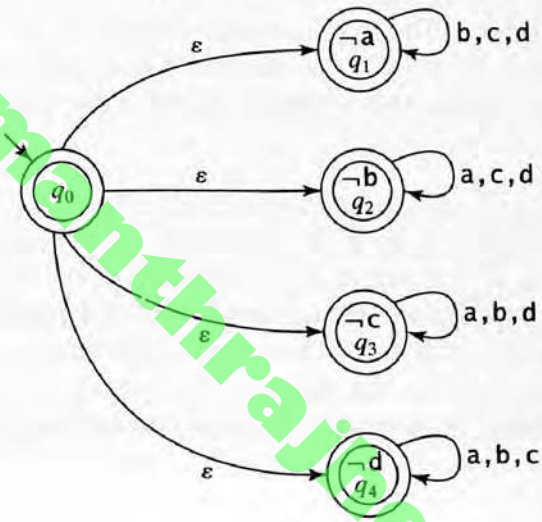


The upper machine accepts $\{w \in \{a, b\}^* : w =$ aba$\}$. The lower one accepts $\{w \in \{a, b\}^* : |w|$ is even$\}$.

By exploiting nondeterminism, it may be possible to build a simple FSM to accept a language for which the smallest deterministic FSM is complex. A good example of a language for which this is true is the missing letter language that we considered in Example 5.12.

EXAMPLE 5.15 The Missing Letter Language, Again

Let $\Sigma = \{a, b, c, d\}$. $L_{Missing} = \{w$: there is a symbol $a_i \in \Sigma$ not appearing in $w\}$. The following simple NDFSM M accepts $L_{Missing}$:



M works by guessing which letter is going to be the missing one. If any of its guesses is right, it will accept. If all of them are wrong, then all paths will fail and M will reject.

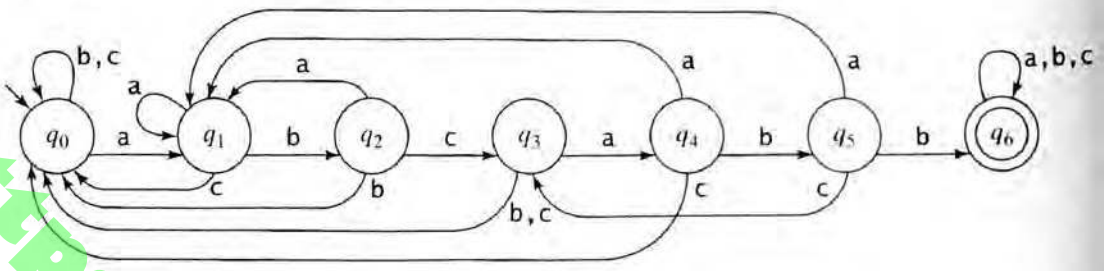
5.4.2 NDFSMs for Pattern and Substring Matching

Nondeterministic FSMs are a particularly effective way to define simple machines to search a text string for one or more patterns or substrings.

EXAMPLE 5.16 Exploiting Nondeterminism for Keyword Matching

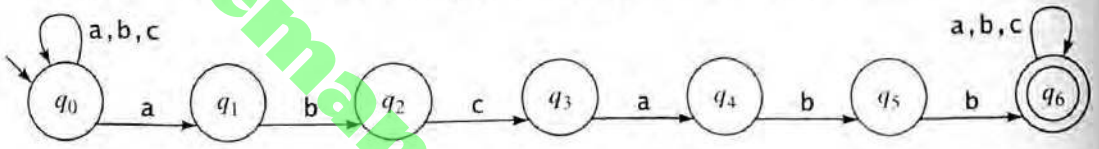
Let $L = \{w \in \{a, b, c\}^* : \exists x, y \in \{a, b, c\}^* \ (w = x \text{ abcabb } y)\}$. In other words, w must contain at least one occurrence of the substring abcabb. The following DFSM M_1 accepts L:

EXAMPLE 5.16 *(Continued)*



While M_1 works, and it works efficiently, designing machines like M_1 and getting them right is hard. The spaghetti-like transitions are necessary because, whenever a match fails, it is possible that another partial match has already been found.

But now consider the following NDFSM M_2, which also accepts L:



The idea here is that, whenever M_2 sees an a, it may guess that it is at the beginning of the pattern abcabb. Or, on any input character (including a), it may guess that it is not yet at the beginning of the pattern (so it stays in q_0). If it ever reaches q_6, it will stay there until it has finished reading the input. Then it will accept.

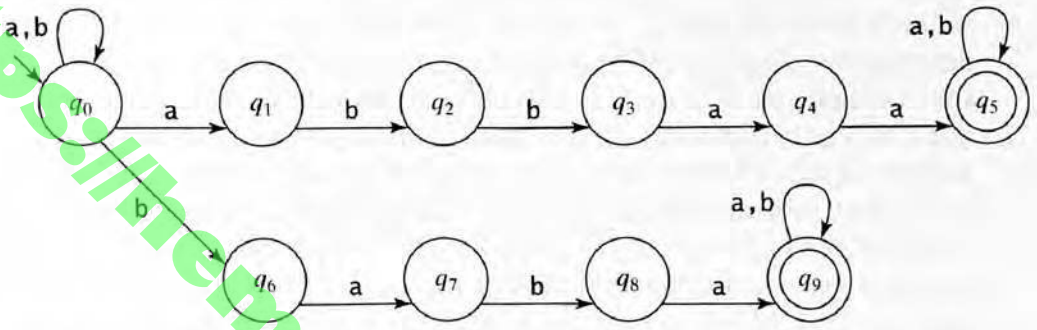
Of course, practical string search engines need to be small and deterministic. But NDFSMs like the one we just built can be used as the basis for constructing such efficient search machines. In Section 5.4.4, we will describe an algorithm that converts an arbitrary NDFSM into an equivalent DFSM. It is likely that that machine will have more states than it needs. But, in Section 5.7, we will present an algorithm that takes an arbitrary DFSM and produces an equivalent minimal one (i.e., one with the smallest number of states). So one effective way to build a correct and efficient string-searching machine is to build a simple NDFSM, convert it to an equivalent DFSM, and then minimize the result. One alternative to this three-step process is the Knuth-Morris-Pratt string search algorithm, which we will present in Example 27.5.

> String searching is a fundamental operation in every word processing or text editing system.

Now suppose that we have not one pattern but several. Hand crafting a DFSM may be even more difficult. One alternative is to use a specialized, keyword-search FSM-building algorithm that we will present in Section 6.2.4. Another is to build a simple NDFSM, as we show in the next example.

EXAMPLE 5.17 Multiple Keywords

Let $L = \{w \in \{a,b\}^* : \exists x, y \in \{a,b\}^* ((w = x\,abbaa\,y) \vee (w = x\,baba\,y))\}$. In other words, w contains at least one occurrence of the substring abbaa or the substring baba. The following NDFSM M accepts L:

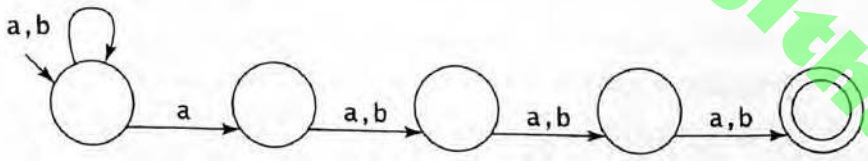


The idea here is that, whenever M sees an a, it may guess that it is at the beginning of the substring abbaa. Whenever it sees a b, it may guess that it is at the beginning of the substring baba. Alternatively, on either a or b, it may guess that it is not yet at the beginning of either substring (so it stays in q_0).

NDFSMs are also a natural way to search for other kinds of patterns, as we can see in the next example.

EXAMPLE 5.18 Other Kinds of Patterns

Let $L = \{w \in \{a, b\}^* :$ the fourth from the last character is a$\}$. The following NDFSM M accepts L:



The idea here is that, whenever it sees an a, one of M's paths guesses that it is the fourth from the last character (and so proceeds along the path that will read the last three remaining characters). The other path guesses that it is not (and so stays in the start state).

It is enlightening to try designing DFSMs for the last two examples. We leave that as an exercise. If you try it, you'll appreciate the value of the NDFSM model as a high-level tool for describing complex systems.

5.4.3 Analyzing Nondeterministic FSMs

Given an NDFSM M, such as any of the ones we have just considered, how can we analyze it to determine what strings it accepts? One way is to do a depth-first search of the paths through the machine. Another is to imagine tracing the execution of the original NDFSM M by following all paths in parallel. To do that, think of M as being in a set of states at each step of its computation. For example, consider again the NDFSM that we built for Example 5.17. You may find it useful to trace the process we are about to describe by using several fingers. Or, when fingers run out, use a coin on each active state. Initially, M is in q_0. If it sees an a, it can loop to state q_0 or go to q_1. So we will think of it as being in the set of states $\{q_0, q_1\}$ (thus we need two fingers or two coins). Suppose it sees a b next. From q_0, it can go to q_0 or q_6. From q_1, it can go to q_2. So, after seeing the string ab, M is in $\{q_0, q_2, q_6\}$ (three fingers or three coins). Suppose it sees a b next. From q_0, it can go to q_0 or q_6. From q_2, it can go to q_3. From q_6, it can go nowhere. So, after seeing abb, M is in $\{q_0, q_3, q_6\}$. And so forth. If, when all the input has been read, M is in at least one accepting state (in this case, q_5 or q_9), then it accepts. Otherwise it rejects.

Handling ε-Transitions

But how shall we handle ε-transitions? The construction that we just sketched assumes that all paths have read the same number of input symbols. But if, from some state q, one transition is labeled ε and another is labeled with some element of Σ, M consumes no input as it takes the first transition and one input symbol as it takes the second transition. To solve this problem, we introduce the function eps: $K_M \rightarrow \mathcal{P}(K_M)$. We define $eps(q)$, where q is some state in M, to be the set of states of M that are reachable from q by following zero or more ε-transitions. Formally:

$$eps(q) = \{p \in K : (q, w) \vdash_M^* (p, w)\}.$$

Alternatively, $eps(q)$ is the closure of $\{q\}$ under the relation $\{(p, r)$: there is a transition $(p, \varepsilon, r) \in \Delta\}$. The following algorithm computes eps:

$eps(q$: state) =

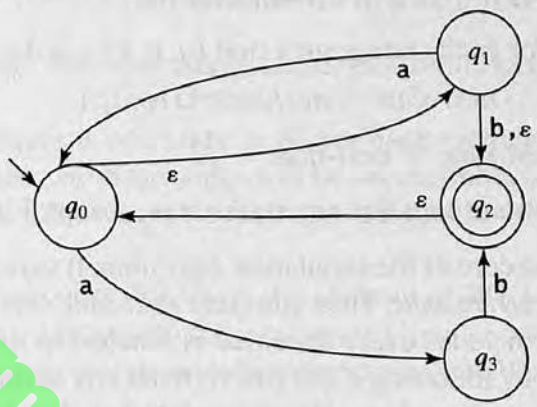
1. $result = \{q\}$.
2. While there exists some $p \in result$ and some $r \notin result$ and some transition $(p, \varepsilon, r) \in \Delta$ do: Insert r into $result$.
3. Return $result$.

This algorithm is guaranteed to halt because, each time through the loop, it adds an element to $result$. It must halt when there are no elements left to add. Since there is only a finite number of candidate elements, namely the finite set of states in M, and no element can be added more than once, the algorithm must eventually run out of elements to add, at which point it must halt. It correctly computes $eps(q)$ because, by the condition associated with the while loop:

- It can add no element that is not reachable from q following only ε-transitions.
- It will add all elements that are reachable from q following only ε-transitions.

EXAMPLE 5.19 Computing *eps*

Consider the following NDFSM M:



To compute $eps(q_0)$, we initially set *result* to $\{q_0\}$. Then q_1 is added, producing $\{q_0, q_1\}$. Then q_2 is added, producing $\{q_0, q_1, q_2\}$. There is an ε-transition from q_2 to q_0, but q_0 is already in *result*. So the computation of $eps(q_0)$ halts.

The result of running *eps* on each of the states of M is:

$$eps(q_0) = \{q_0, q_1, q_2\}.$$
$$eps(q_1) = \{q_0, q_1, q_2\}.$$
$$eps(q_2) = \{q_0, q_1, q_2\}.$$
$$eps(q_3) = \{q_3\}.$$

Example 5.19 illustrates clearly why we chose to define the *eps* function, rather than treating ε-transitions like other transitions and simply following them whenever we could. The machine we had to consider in that example contains what we might choose to call an ***ε-loop***: a loop that can be traversed by following only ε-transitions. Since such transitions consume no input, there is no limit to the number of times the loop could be traversed. So, if we were not careful, it would be easy to write a simulation algorithm that did not halt. The algorithm that we presented for *eps* halts whenever it runs out of unvisited states to add, which must eventually happen since the set of states is finite.

A Simulation Algorithm

With the *eps* function in hand, we can now define an algorithm for tracing all paths in parallel through an NDFSM M:

ndfsmsimulate (M: NDFSM, w: string) =

1. *current-state* = $eps(s)$. /*Start in the set that contains M's start state and any other states that can be reached from it following only ε-transitions.

2. While any input symbols in w remain to be read do:
   2.1. c = get-next-symbol(w).
   2.2. *next-state* = $\varnothing$.
   2.3. For each state q in *current-state* do:
      For each state p such that $(q, c, p) \in \Delta$ do:
         next-state = *next-state* $\cup$ *eps* (p).
   2.4. *current-state* = *next-state*.
3. If *current-state* contains any states in A, accept. Else reject.

Step 2.3 is the core of the simulation algorithm. It says: Follow every arc labeled c from every state in *current-state*. Then compute *next-state* (and thus the new value of *current-state*) so that it includes every state that is reached in that process, plus every state that can be reached by following ε-transitions from any of those states. For more on how this step can be implemented, see the more detailed description of *ndfsmsimulate* that we present in Section 5.6.2.

5.4.4 The Equivalence of Nondeterministic and Deterministic FSMs

In this section, we explore the relationship between the DFSM and NDFSM models that we have just defined.

THEOREM 5.2 If There is a DFSM for L, There is an NDFSM for L

Theorem: For every DFSM there is an equivalent NDFSM.

Proof: Let M be a DFSM that accepts some language L. M is also an NDFSM that happens to contain no ε-transitions and whose transition relation happens to be a function. So the NDFSM that we claim must exist is simply M.

But what about the other direction? The nondeterministic model that we have just introduced makes it substantially easier to build FSMs to accept some kinds of languages, particularly those that involve looking for instances of complex patterns. But real computers are deterministic. What does the existence of an NDFSM to accept a language L tell us about the existence of a deterministic program to accept L? The answer is given by the following theorem:

THEOREM 5.3 If There is an NDFSM for L, There is a DFSM for L

Theorem: Given an NDFSM $M = (K, \Sigma, \Delta, s, A)$ that accepts some language L, there exists an equivalent DFSM that accepts L.

Proof: The proof is by construction of an equivalent DFSM M'. The construction is based on the function *eps* and on the simulation algorithm that we described in the last section. The states of M' will correspond to sets of states in M. So $M' = (K', \Sigma, \delta', s', A')$, where:

- K' contains one state for each element of $\mathcal{P}(K)$.
- $s' = eps(s)$.
- $A' = \{Q \subseteq K : Q \cap A \neq \emptyset\}$.
- $\delta'(Q, c) = \cup \{eps(p) : \exists q \in Q ((q, c, p) \in \Delta)\}$.

We should note the following things about this definition:

- In principle, there is one state in K' for each element of $\mathcal{P}(K)$. However, in most cases, many of those states will be unreachable from s' (and thus unnecessary). So we will present a construction algorithm that creates states only as it needs to.
- We'll name each state in K' with the element of $\mathcal{P}(K)$ to which it corresponds. That will make it relatively straightforward to see how the construction works. But keep in mind that those labels are just names. We could have called them anything.
- To decide whether a state in K' is an accepting state, we see whether it corresponds to an element of $\mathcal{P}(K)$ that contains at least one element of A, i.e., one accepting state from K.
- M' accepts whenever it runs out of input and is in a state that contains at least one accepting state of M. Thus it implements the definition of an NDFSM, which accepts iff at least one path through it accepts.
- The definition of δ' corresponds to step 2.3 of the simulation algorithm we presented above.

The following algorithm computes M' given M:

ndfsmtodfsm(M: NDFSM) =

1. For each state q in K do:
  Compute $eps(q)$. /* These values will be used below.
2. $s' = eps(s)$.
3. Compute δ':
  a. *active-states* $= \{s'\}$. /* We will build a list of all states that are reachable from the start state. Each element of *active-states* is a set of states drawn from K.
  b. $\delta' = \emptyset$.
  c. While there exists some element Q of *active-states* for which δ' has not yet been computed do:
    For each character c in Σ do:
      new-state $= \emptyset$.
      For each state q in Q do:

For each state p such that $(q, c, p) \in \Delta$ do:

$new\text{-}state = new\text{-}state \cup eps(p)$.

Add the transition $(Q, c, new\text{-}state)$ to δ'.

If $new\text{-}state \notin active\text{-}states$ then insert it into *active-states*.

4. $K' = active\text{-}states$.
5. $A' = \{Q \in K' : Q \cap A \neq \emptyset\}$.

The core of *ndfsmtodfsm* is the loop in step 3.3. At each step through it, we pick a state that we know is reachable from the start state but from which we have not yet computed transitions. Call it Q. Then compute the paths from Q for each element c of the input alphabet as follows: Q is a set of states in the original NDFSM M. So consider each element q of Q. Find all transitions from q labeled c. For each state p that is reached by such a transition, find all additional states that are reachable by following only ε-transitions from p. Let *new-state* be the set that contains all of those states. Now we know that whenever M' is in Q and it reads a c, it should go to *new-state*.

The algorithm *ndfsmtodfsm* halts on all inputs and constructs a DFSM M' that accepts exactly $L(M)$, the language accepted by M.

> A rigorous construction proof requires a proof that the construction algorithm is correct. We will generally omit the details of such proofs. But we show them for this case as an example of what these proofs look like. (Appendix C)

The algorithm *ndfsmtodfsm* is important for two reasons:

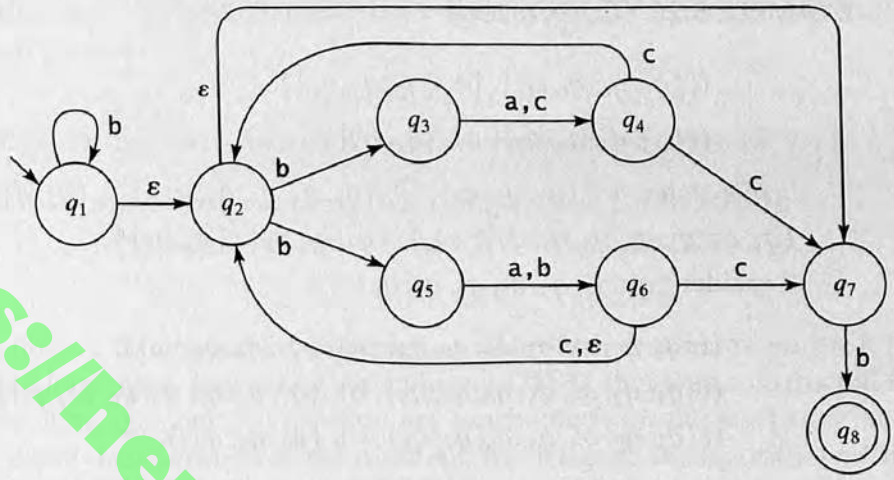
- It proves the theorem that, for every NDFSM there exists an equivalent DFSM.
- It lets us use nondeterminism as a design tool, even though we may ultimately need a deterministic machine. If we have an implementation of *ndfsmtodfsm*, then, if we can build an NDFSM to solve our problem, *ndfsmtodfsm* can easily construct an equivalent DFSM.

EXAMPLE 5.20 Using *ndfsmtodfsm* to Build a Deterministic FSM

Consider the NDFSM M shown on the next page. To get a feel for M, simulate it on the input string bbbacb, using coins to keep track of the states it enters.

We can apply *ndfsmtodfsm* to M as follows:

1. Compute $eps(q)$ for each state q in K_M:



$eps\ (q_1) = \{q_1, q_2, q_7\}$, $eps\ (q_2) = \{q_2, q_7\}$, $eps\ (q_3) = \{q_3\}$, $eps\ (q_4) = \{q_4\}$, $eps\ (q_5) = \{q_5\}$, $eps\ (q_6) = \{q_2, q_6, q_7\}$, $eps\ (q_7) = \{q_7\}$, $eps\ (q_8) = \{q_8\}$.

2. $s' = eps\ (s) = \{q_1, q_2, q_7\}$.

3. Compute δ':

active-states $= \{\{q_1, q_2, q_7\}\}$. Consider $\{q_1, q_2, q_7\}$:

$((\{q_1, q_2, q_7\}, \mathsf{a}), \varnothing)$.
$((\{q_1, q_2, q_7\}, \mathsf{b}), \{q_1, q_2, q_3, q_5, q_7, q_8\})$.
$((\{q_1, q_2, q_7\}, \mathsf{c}), \varnothing)$.

active-states $= \{\{q_1, q_2, q_7\}, \varnothing, \{q_1, q_2, q_3, q_5, q_7, q_8\}\}$. Consider $\varnothing$:

$((\varnothing, \mathsf{a}), \varnothing)$. /* $\varnothing$ is a dead state and we will generally omit it.
$((\varnothing, \mathsf{b}), \varnothing)$.
$((\varnothing, \mathsf{c}), \varnothing)$.

active-states $= \{\{q_1, q_2, q_7\}, \varnothing, \{q_1, q_2, q_3, q_5, q_7, q_8\}\}$. Consider $\{q_1, q_2, q_3, q_5, q_7, q_8\}$:

$((\{q_1, q_2, q_3, q_5, q_7, q_8\}, \mathsf{a}), \{q_2, q_4, q_6, q_7\})$.
$((\{q_1, q_2, q_3, q_5, q_7, q_8\}, \mathsf{b}), \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\})$.
$((\{q_1, q_2, q_3, q_5, q_7, q_8\}, \mathsf{c}), \{q_4\})$.

active-states $= \{\{q_1, q_2, q_7\}, \varnothing, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}\}$. Consider $\{q_2, q_4, q_6, q_7\}$:

$((\{q_2, q_4, q_6, q_7\}, \mathsf{a}), \varnothing)$.

EXAMPLE 5.20 (*Continued*)

$((\{q_2, q_4, q_6, q_7\}, \mathsf{b}), \{q_3, q_5, q_8\})$.

$((\{q_2, q_4, q_6, q_7\}, \mathsf{c}), \{q_2, q_7\})$.

active-states $= \{\{q_1, q_2, q_7\}, \varnothing, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}\}$.

Consider $\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}$:

$((\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \mathsf{a}), \{q_2, q_4, q_6, q_7\})$.

$((\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \mathsf{b}), \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\})$.

$((\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \mathsf{c}), \{q_2, q_4, q_7\})$.

active-states $= \{\{q_1, q_2, q_7\}, \varnothing, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}, \{q_2, q_4, q_7\}\}$. Consider $\{q_4\}$:

$((\{q_4\}, \mathsf{a}), \varnothing)$.

$((\{q_4\}, \mathsf{b}), \varnothing)$.

$((\{q_4\}, \mathsf{c}), \{q_2, q_7\})$.

active-states did not change. Consider $\{q_3, q_5, q_8\}$:

$((\{q_3, q_5, q_8\}, \mathsf{a}), \{q_2, q_4, q_6, q_7\})$.

$((\{q_3, q_5, q_8\}, \mathsf{b}), \{q_2, q_6, q_7\})$.

$((\{q_3, q_5, q_8\}, \mathsf{c}), \{q_4\})$.

active-states $= \{\{q_1, q_2, q_7\}, \varnothing, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}, \{q_2, q_4, q_7\}, \{q_2, q_6, q_7\}\}$.

Consider $\{q_2, q_7\}$:

$((\{q_2, q_7\}, \mathsf{a}), \varnothing)$.

$((\{q_2, q_7\}, \mathsf{b}), \{q_3, q_5, q_8\})$.

$((\{q_2, q_7\}, \mathsf{c}), \varnothing)$.

active-states did not change. Consider $\{q_2, q_4, q_7\}$:

$((\{q_2, q_4, q_7\}, \mathsf{a}), \varnothing)$.

$((\{q_2, q_4, q_7\}, \mathsf{b}), \{q_3, q_5, q_8\})$.

$((\{q_2, q_4, q_7\}, \mathsf{c}), \{q_2, q_7\})$.

active-states did not change. Consider $\{q_2, q_6, q_7\}$:

$((\{q_2, q_6, q_7\}, \mathsf{a}), \varnothing)$.

$((\{q_2, q_6, q_7\}, \mathsf{b}), \{q_3, q_5, q_8\})$.

$((\{q_2, q_6, q_7\}, \mathsf{c}), \{q_2, q_7\})$.

active-states did not change. δ has been computed for each element of *active-states*.

4. $K' = \{\{q_1, q_2, q_7\}, \varnothing, \{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}, \{q_2, q_4, q_7\}, \{q_2, q_6, q_7\}\}$.
5. $A' = \{\{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_3, q_5, q_8\}\}$.

Notice that, in Example 5.20, the original NDFSM had 8 states. So $|\mathcal{P}(K)| = 256$. There could have been that many states in the DFSM that was constructed from the original machine. But only 10 of those are reachable from the start state and so can play any role in the operation of the machine. We designed the algorithm *ndfsmtodfsm* so that only those 10 would have to be built.

Sometimes, however, all or almost all of the possible subsets of states are reachable. Consider again the NDFSM of Example 5.15, the missing letter machine. Let's imagine a slight variant that considers all 26 letters of the alphabet. That machine M has 27 states. So, in principle, the corresponding DFSM could have 2^{27} states. And, this time, all subsets are possible except that M can not be in the start state, q_0, at any time except before the first character is read. So the DFSM that we would build if we applied *ndfsmtodfsm* to M would have $2^{26} + 1$ states. In Section 5.6, we will describe a technique for interpreting NDFSMs without converting them to DFSMs first. Using that technique, highly nondeterministic machines, like the missing letter one, are still practical.

What happens if we apply *ndfsmtodfsm* to a machine that is already deterministic? It must work, since every DFSM is also a legal NDFSM. You may want to try it on one of the machines in Section 5.3. What you will see is that the machine that *ndfsmtodfsm* builds, given an input DFSM M, is identical to M except for the names of the states.

5.5 From FSMs to Operational Systems

An FSM is an abstraction. We can describe an FSM that solves a problem without worrying about many kinds of implementation details. In fact, we don't even need to know whether it will be etched into silicon or implemented in software.

> Statecharts, which are based on the idea of hierarchically structured transition networks, are widely used in software engineering precisely because they enable system designers to work at varying levels of abstraction. (H.2)

FSMs for real problems can be turned into operational systems in any of a number of ways:

- An FSM can be translated into a circuit design and implemented directly in hardware. For example, it makes sense to implement the parity checking FSM of Example 5.4 in hardware.

- An FSM can be simulated by a general purpose interpreter. We will describe designs for such interpreters in the next section. Sometimes all that is required is a simulation. In other cases, a simulation can be used to check a design before it is translated into hardware.
- An FSM can be used as a specification for some critical aspect of the behavior of a complex system. The specification can then be implemented in software just as any specification might be. And the correctness of the implementation can be shown by verifying that the implementation satisfies the specification (i.e., that it matches the FSM).

> Many network communication protocols, including the Alternating Bit protocol and TCP, are described as FSMs. (I.1)

5.6 Simulators for FSMs ●

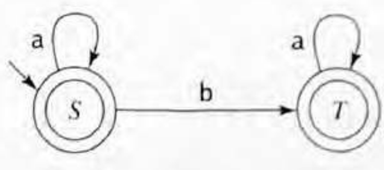
Once we have created an FSM to solve a problem, we may want to simulate its execution. In this section, we consider techniques for doing that, starting with DFSMs, and then extending our ideas to handle nondeterminism.

5.6.1 Simulating Deterministic FSMs

We begin by considering only deterministic FSMs. One approach is to think of an FSM as the specification for a simple, table-driven program and then proceed to write the code.

EXAMPLE 5.21 Hardcoding a Deterministic FSM

Consider the following deterministic FSM M that accepts the language $L = \{w \in \{a, b\}^* : w$ contains no more than one b$\}$.



We could view M as a specification for the following program:

```
Until accept or reject do:
    S:    s = get-next-symbol.
          If s = end-of-file then accept.
          Else if s = a then go to S.
          Else if s = b then go to T.
```

```
T:    s = get-next-symbol.
      If s = end-of-file then accept.
      Else if s = a then go to T.
      Else if s = b then reject.

      End.
```

Given an FSM M with states K, this approach will create a program of length $= 2 + (|K| \cdot (|\Sigma| + 2))$. The time required to analyze an input string w is $\mathcal{O}(|w| \cdot |\Sigma|)$. The biggest problem with this approach is that we must generate new code for every FSM that we wish to run. Of course, we could write an FSM compiler that did that for us. But we don't need to. We can, instead, build an interpreter that executes the FSM directly.

Here's a simple interpreter for a deterministic FSM $M = (K, \Sigma, \delta, s, A)$:

dfsmsimulate(M: DFSM, w: string) =

1. $st = s$.

2. Repeat:

  2.1. c = get-next-symbol(w).

  2.2. If $c \neq$ end-of-file then:

    2.2.1. $st = \delta(st, c)$.

  until c = end-of-file.

3. If $st \in A$ then accept else reject.

The algorithm *dfsmsimulate* runs in time approximately $\mathcal{O}(|w|)$, if we assume that the lookup in step 2.2.1 can be implemented in constant time.

5.6.2 Simulating Nondeterministic FSMs

Now suppose that we want to execute an NDFSM M. One solution is:

ndfsmconvertandsimulate(M : NDFSM) =

  dfsmsimulate(*ndfsmtodfsm*(M)).

But, as we saw in Section 5.4, converting an NDFSM to a DFSM can be very inefficient in terms of both time and space. If M has k states, it could take time and space equal to $\mathcal{O}(2^k)$ just to do the conversion, although the simulation, after the conversion, would take time equal to $\mathcal{O}(|w|)$. So we would like a better way. We would like an algorithm that directly simulates an NDFSM M without converting it to a DFSM first.

We sketched such an algorithm *ndfsmsimulate* in our discussion leading up to the definition of the conversion algorithm *ndfsmtodfsm*. The idea is to simulate being in sets of states at once. But, instead of generating all of the reachable sets of states right

away, as *ndfsmtodfsm* does, it generates them on the fly, as they are needed, being careful not to get stuck chasing ε-loops.

We give here a more detailed description of *ndfsmsimulate*, which simulates an NDFSM $M = (K, \Sigma, \Delta, s, A)$ running on an input string w:

ndfsmsimulate(M: NDFSM, w: string) =

1. Declare the set st. /* st will hold the current state (a set of states from K).
2. Declare the set $st1$. /* $st1$ will be built to contain the next state.
3. $st = eps(s)$. /* Start in all states reachable from s via only ε-transitions.
4. Repeat:

   c = get-next-symbol (w).

   If $c \neq$ end-of-file then do:

   $st1 = \emptyset$.

   For all $q \in st$ do: /* Follow paths from all states M is currently in.

   For all r: $(q, c, r) \in \Delta$ do: /* Find all states reachable from q via a transition labeled c.

   $st1 = st1 \cup eps(r)$. /* Follow all ε-transitions from there.

   $st = st1$. /* Done following all paths. So st becomes M's new state.

   If $st = \emptyset$ then exit. /* If all paths have died, quit.

   until c = end-of-file.
5. If $st \cap A \neq \emptyset$ then accept else reject.

Now there is no conversion cost. To analyze a string w requires $|w|$ passes through the main loop in step 4. In the worst case, M is in all states all the time and each of them has a transition to every other one. So one pass could take as many as $\mathcal{O}(|K|^2)$ steps, for a total cost of $\mathcal{O}(w \cdot |K|^2)$.

There is also a third way we could build a simulator for an NDFSM. We could build a depth-first search program that examines the paths through M and stops whenever either it finds a path that accepts or it has tried all the paths there are.

5.7 Minimizing FSMs ✹

If we are going to solve a real problem with an FSM, we may want to find the smallest one that does the job. We will say that a DFSM M is ***minimal*** iff there is no other DFSM M' such that $L(M) = L(M')$ and M' has fewer states than M does.

We might want to be able to ask:

1. Given a language, L, is there a minimal DFSM that accepts L?
2. If there is a minimal machine, is it unique?
3. Given a DFSM M that accepts some language L, can we tell whether M is minimal?
4. Given a DFSM M, can we construct a minimal equivalent DFSM M'?

The answer to all four questions is yes. We'll consider questions 1 and 2 first, and then consider questions 3 and 4.

5.7.1 Building a Minimal DFSM for a Language

Recall that in Section 5.3 we suggested that an effective way to think about the design of a DFSM M to accept some language L over an alphabet Σ is to cluster the strings in Σ^* in such a way that strings that share a future will drive M to the same state. We will now formalize that idea and use it as the basis for constructing a minimal DFSM to accept L.

We will say that x and y are ***indistinguishable*** with respect to L, which we will write as $x \approx_L y$ iff:

$$\forall z \in \Sigma^* \text{ (either both } xz \text{ and } yz \in L \text{ or neither is).}$$

In other words, $\approx_L$ is a relation that is defined so that $x \approx_L y$ precisely in case, if x and y are viewed as prefixes of some longer string, no matter what continuation string z comes next, either both xz and yz are in L or both are not.

EXAMPLE 5.22 How $\approx_L$ Depends on L

If $L = \{\mathtt{a}\}^*$, then $\mathtt{a} \approx_L \mathtt{aa} \approx_L \mathtt{aaa}$. But if $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : |w| \text{ is even}\}$, then $\mathtt{a} \approx_L \mathtt{aaa}$, but it is not the case that $\mathtt{a} \approx_L \mathtt{aa}$ because, if $z = \mathtt{a}$, we have $\mathtt{aa} \in L$ but $\mathtt{aaa} \notin L$.

We will say that x and y are ***distinguishable*** with respect to L, iff they are not indistinguishable. So, if x and y are distinguishable, then there exists at least one string z such that one but not both of xz and yz is in L.

Note that $\approx_L$ is an equivalence relation because it is:

- Reflexive: $\forall x \in \Sigma^* \ (x \approx_L x)$, because $\forall x, z \in \Sigma^* \ (xz \in L \leftrightarrow xz \in L)$.
- Symmetric: $\forall x, y \in \Sigma^* \ (x \approx_L y \rightarrow y \approx_L x)$, because $\forall x, y, z \in \Sigma^* \ ((xz \in L \leftrightarrow yz \in L) \leftrightarrow (yz \in L \leftrightarrow xz \in L))$.
- Transitive: $\forall x, y, z \in \Sigma^* \ (((x \approx_L y) \wedge (y \approx_L w)) \rightarrow (x \approx_L w))$, because: $\forall x, y, z \in \Sigma^* \ (((xz \in L \leftrightarrow yz \in L) \wedge (yz \in L \leftrightarrow wz \in L)) \rightarrow (xz \in L \leftrightarrow wz \in L))$.

We will use three notations to describe the equivalence classes of $\approx_L$:

- [1], [2], etc. will refer to explicitly numbered classes.
- $[x]$ describes the equivalence class that contains the string x.
- [some logical expression P] describes the equivalence class of strings that satisfy P.

Since $\approx_L$ is an equivalence relation, its equivalence classes constitute a partition of the set Σ^*. So:

- No equivalence class of $\approx_L$ is empty, and
- Every string in Σ^* is in exactly one equivalence class of $\approx_L$.

What we will see soon is that the equivalence classes of $\approx_L$ correspond exactly to the states of the minimum DFSM that accepts L. So every string in Σ^* will drive that DFSM to exactly one state.

Given some language L, how can we determine $\approx_L$? Any pair of strings x and y are related via $\approx_L$ unless there exists some z that could follow them and cause one to be in L and the other not to be. So it helps to begin the analysis by considering simple strings and seeing whether they are distinguishable or not. One way to start this process is to begin lexicographically enumerating the strings in Σ^* and continue until a pattern has emerged.

EXAMPLE 5.23 Determining $\approx L$

Let $\Sigma = \{\mathsf{a}, \mathsf{b}\}$. Let $L = \{w \in \Sigma^* : \text{every a is immediately followed by a b}\}$.

To determine the equivalence classes of $\approx_L$, we begin by creating a first class [1] and arbitrarily assigning ε to it. Now consider a. It is distinguishable from ε since $\varepsilon\mathsf{ab} \in L$ but $\mathsf{aab} \notin L$. So we create a new equivalence class [2] and put a in it. Now consider b. $\mathsf{b} \approx_L \varepsilon$ since every string is in L unless it has an a that is not followed by a b. Neither of these has an a that could have that problem. So they are both in L as long as their continuation doesn't violate the rule. If their continuation does violate the rule, they are both out. So b goes into [1].

Next we try aa. It is distinguishable from the strings in [1] because the strings in [1] are in L but aa is not. So, consider ε as a continuation string. Take any string in [1] and concatenate ε. The result is still in L. But $\mathsf{aa}\varepsilon$ is not in L. We also notice that aa is distinguishable from a, and so cannot be in [2], because a still has a chance to become in L if it is followed by a string that starts with a b. But aa is out, no matter what comes next. We create a new equivalence class [3] and put aa in it. We continue in this fashion until we discover the property that holds of each equivalence class.

The equivalence classes of $\approx_L$ are:

[1]	[ε, b, abb, ...]	[all strings in L].
[2]	[a, abbba, ...]	[all strings that end in a and have no prior a that is not followed by a b].
[3]	[aa, abaa, ...]	[all strings that contain at least one instance of aa].

Even this simple example illustrates three key points about $\approx_L$:

- No equivalence class can contain both strings that are in L and strings that are not. This is clear if we consider the continuation string ε. If $x \in L$ then $x\varepsilon \in L$. If $y \notin L$ then $y\varepsilon \notin L$. So x and y are distinguishable by ε.
- If there are strings that would take a DFSM for L to the dead state (in other words, strings that are out of L no matter what comes next), then there will be one equivalence class of $\approx_L$ that corresponds to the dead state.
- Some equivalence class contains ε. It will correspond to the start state of the minimal machine that accepts L.

EXAMPLE 5.24 When More Than One Class Contains Strings in L

Let $\Sigma = \{a, b\}$. Let $L = \{w \in \{a, b\}^* :$ no two adjacent characters are the same$\}$. The equivalence classes of $\approx_L$ are:

[1]	[ε]	[ε].
[2]	[a, aba, ababa, ...]	[all nonempty strings that end in a and have no identical adjacent characters].
[3]	[b, ab, bab, abab, ...]	[all nonempty strings that end in b and have no identical adjacent characters].
[4]	[aa, abaa, ababb ...]	[all strings that contain at least one pair of identical adjacent characters].

From this example, we make one new observation about $\approx_L$:

- While no equivalence class may contain both strings that are in L and strings that are not, there may be more than one equivalence class that contains strings that are in L. For example, in this last case, all the strings in classes [1], [2], and [3] are in L. Only those that are in [4], which corresponds to the dead state, are not in L. That is because of the structure of L: Any string is in L until it violates the rule, and then it is hopelessly out.

Does $\approx_L$ always have a finite number of equivalence classes? It has in the two examples we have considered so far. But let's consider another one.

EXAMPLE 5.25 $\approx_L$ for A^nB^n

Let $\Sigma = \{a, b\}$. Let $L = A^nB^n = \{a^nb^n : n \geq 0\}$.

We can begin constructing the equivalence classes of $\approx_L$:

[1] [ε].
[2] [a].
[3] [aa].
[4] [aaa].

But we seem to be in trouble. Each new string of a's has to go in an equivalence class distinct from the shorter strings because each string requires a different continuation string in order to become in L. So the set of equivalence classes of $\approx_L$ must include at least all of the following classes:

$$\{[n] : n \text{ is a positive integer and } [n] \text{ contains the single string } a^{n-1}\}$$

Of course, classes that include strings that contain b's are also required.

So, if $L = A^nB^n$, then $\approx_L$ has an infinite number of equivalence classes. This should come as no surprise. A^nB^n is not regular, as we will prove in Chapter 8. If the equivalence classes of $\approx_L$ are going to correspond to the states of a machine to accept L, then there will be a finite number of equivalence classes precisely in case L is regular.

We are now ready to talk about DFSMs and to examine the relationship between $\approx_L$ and any DFSM that accepts L. To help do that we will say that a state q of a DFSM M ***contains*** the set of strings s such that M, when started in its start state, lands in q after reading s.

THEOREM 5.4 $\approx_L$ **Imposes a Lower Bound on the Minimum Number of States of a DFSM for** L

Theorem: Let L be a regular language and let $M = (K, \Sigma, \delta, s, A)$ be a DFSM that accepts L. The number of states in M is greater than or equal to the number of equivalence classes of $\approx_L$.

Proof: Suppose that the number of states in M were less than the number of equivalence classes of $\approx_L$. Then, by the pigeonhole principle, there must be at least one state q that contains strings from at least two equivalence classes of $\approx_L$. But then M's future behavior on those strings will be identical, which is not consistent with the fact that they are in different equivalence classes of $\approx_L$.

So now we know a lower bound on the number of states that are required to build an FSM to accept a language L. But is it always possible to find a DFSM M such that $|K_M|$ is exactly equal to the number of equivalence classes of $\approx_L$? The answer is yes.

THEOREM 5.5 There Exists a Unique Minimal DFSM for Every Regular Language

Theorem: Let L be a regular language over some alphabet Σ. Then there is a DFSM M that accepts L and that has precisely n states where n is the number of equivalence classes of $\approx_L$. Any other DFSM that accepts L must either have more states than M or it must be equivalent to M except for state names.

Proof: The proof is by construction of $M = (K, \Sigma, \delta, s, A)$, where:

- K contains n states, one for each equivalence class of $\approx_L$.
- $s = [\varepsilon]$, the equivalence class of ε under $\approx_L$.
- $A = \{[x] : x \in L\}$.
- $\delta([x], a) = [xa]$. In other words, if M is in the state that contains some string x, then, after reading the next symbol a, it will be in the state that contains xa.

For this construction to prove the theorem, we must show:

- K is finite. Since L is regular, it is accepted by some DFSM M'. M' has some finite number of states m. By Theorem 5.4, $n \leq m$. So K is finite.
- δ is a function. In other words, it is defined for all (state, input) pairs and it produces, for each of them, a unique value. The construction defines a value of δ for all (state, input) pairs. The fact that the construction guarantees a unique such value follows from the definition of $\approx_L$.
- $L = L(M)$. In other words, M does in fact accept the language L. To prove this, we must first show that $\forall s, t\ (([\varepsilon], st) \mid\!-_M{}^* ([s], t))$. In other words, when M starts in its start state and has a string that we are describing as having two parts, s and t, to read, it correctly reads the first part s and lands in the state $[s]$, with t left to read. We do this by induction on $|s|$. If $|s| = 0$ then we have $([\varepsilon], \varepsilon t) \mid\!-_M{}^* ([\varepsilon], t)$, which is true since M simply makes zero moves. Assume that the claim is true if $|s| = k$. Then we consider what happens when $|s| = k + 1$. $|s| \geq 1$, so we can let $s = yc$ where $y \in \Sigma^*$ and $c \in \Sigma$. We have:

  /* M reads the first k characters:

  $([\varepsilon], yct) \mid\!-_M{}^* ([y], ct)$ (induction hypothesis, since $|y| = k$).

  /* M reads one more character:

  $([y], ct) \mid\!-_M{}^* ([yc], t)$ (definition of δ_M).

  /* Combining those two, after M has read $k + 1$ characters:

  $([\varepsilon], yct) \mid\!-_M{}^* ([yc], t)$ (transitivity of $\mid\!-_M{}^*$).

  $([\varepsilon], st) \mid\!-_M{}^* ([s], t)$ (definition of s as yc).

Now let t be ε. (In other words, we are examining M's behavior after it reads its entire input string.) Let s be any string in Σ^*. By the claim we just proved, $([\varepsilon], s) \vdash_M^* ([s], \varepsilon)$. M will accept s iff $[s] \in A$, which, by the way in which A was constructed, it will be if the strings in $[s]$ are in L. So M accepts precisely those strings that are in M.

- There exists no smaller machine $M\#$ that also accepts L. This follows directly from Theorem 5.4, which says that the number of equivalence classes of $\approx_L$ imposes a lower bound on the number of states in any DFSM that accepts L.
- There is no different machine $M\#$ that also has n states and that accepts L. Consider any DFSM $M\#$ with n states. We show that either $M\#$ is identical to M (up to state names) or $L(M\#) \neq L(M)$.

  Since we do not care about state names, we can standardize them. Call the start state of both M and $M\#$ state 1. Define a lexicographic ordering on the elements of Σ. Number the rest of the states in both M and $M\#$ as follows:

  Until all states have been numbered do:

  Let q be the lowest numbered state from which there are transitions that lead to an as yet unnumbered state.

  List the transitions that lead out from q to any unnumbered state. Sort those transitions lexicographically by the symbol on them.

  Go through the sorted transitions (q, a, p), in order, and, for each, assign the next unassigned number to state p.

Note that $M\#$ has n states and there are n equivalence classes of $\approx_L$. Since none of those equivalence classes is empty (by the definition of equivalence classes), $M\#$ either wastes no states (i.e., every state contains at least one string) or, if it does waste any states, it has at least one state that contains strings in different equivalence classes of $\approx_L$. If the latter, then $L(M\#) \neq L$. So we assume the former. Now suppose that $M\#$ is different from M. Then there would have to be at least one state q and one input symbol c such that M has a transition (q, c, r) and $M\#$ has a transition (q, c, t) and $r \neq t$. Call the set of strings that r contains $[r]$. Since $M\#$ has no unused states (i.e., states that contain no strings), by the pigeonhole principle, $M\#$'s transition (q, c, t) must send some string s in $[r]$ to a state, t, that also contains strings that are not in $[r]$. All strings in $[t]$ will then share all futures with s. But s is distinguishable from the strings in $[t]$. If two strings that are distinguishable with respect to L share all futures in $M\#$, then $L(M\#) \neq L$. Contradiction.

The construction that we used to prove Theorem 5.5 is useful in its own right: We can us it, if we know $\approx_L$, to construct a minimal DFSM for L.

EXAMPLE 5.26 Building a Minimal DFSM from $\approx_L$

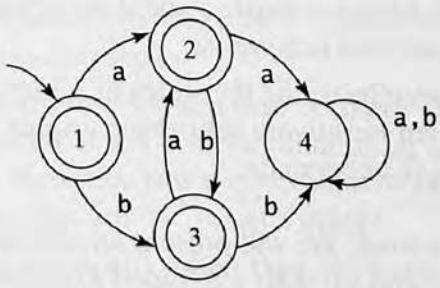
We consider again the language of Example 5.24: Let $\Sigma = \{a, b\}$. Let $L = \{w \in \{a, b\}^* :$ no two adjacent characters are the same$\}$.

The equivalence classes of $\approx_L$ are:

[1]	[ε]	$\{\varepsilon\}$.
[2]	[a, aba, ababa, ...]	{all nonempty strings that end in a and have no identical adjacent characters}.
[3]	[b, ab, bab, abab, ...]	{all nonempty strings that end in b and have no identical adjacent characters}.
[4]	[aa, abaa, ababb ...]	{all strings that contain at least one pair of identical adjacent characters; these strings are not in L, no matter what comes next}.

We build a minimal DFSM M to accept L as follows:

- The equivalence classes of $\approx_L$ become the states of M.
- The start state is [ε] = [1].
- The accepting states are all equivalence classes that contain strings in L, namely [1], [2], and [3].
- $\delta([x], a) = [xa]$. So, for example, equivalence class [1] contains the string ε. If the character a follows ε, the resulting string, a, is in equivalence class [2]. So we create a transition from [1] to [2] labeled a. Equivalence class [2] contains the string a. If the character b follows a, the resulting string, ab, is in equivalence class [3]. So we create a transition from [2] to [3] labeled b. And so forth.



The fact that it is always possible to construct a minimum DFSM M to accept any language L is good news. As we will see later, the fact that that minimal DFSM is unique up to state names is also useful. In particular, we will use it as a basis for an algorithm that checks two DFSMs to see if they accept the same language. The theorem that we have just proven is also useful because it gives us an easy way to prove the following result, which goes by two names, Nerode's theorem and the Myhill-Nerode theorem.

THEOREM 5.6 Myhill-Nerode Theorem

Theorem: A language is regular iff the number of equivalence classes of $\approx_L$ is finite.

Proof: We do two proofs to show the two directions of the implication:

L regular → the number of equivalence classes of $\approx_L$ is finite: If L is regular, then there exists some DFSM M that accepts L. M has some finite number of states m. By Theorem 5.4, the number of equivalence classes of $\approx_L \leq m$. So the number of equivalence classes of $\approx_L$ is finite.

The number of equivalence classes of $\approx_L$ is finite → L regular: If the number of equivalence classes of $\approx_L$ is finite, then the construction that was described in the proof of Theorem 5.5 will build a DFSM that accepts L. So L must be regular.

The Myhill-Nerode theorem gives us our first technique for proving that a language L, such as A^nB^n, is not regular. It suffices to show that $\approx_L$ has an infinite number of equivalence classes. But using the Myhill-Nerode theorem rigorously is difficult. In Chapter 8, we will introduce other methods that are harder to use incorrectly.

5.7.2 Minimizing an Existing DFSM

Now suppose that we already have a DFSM M that accepts L. In fact, possibly M is the only definition we have of L. In this case, it makes sense to construct a minimal DFSM to accept L by starting with M rather than with $\approx_L$. There are two approaches that we could take to constructing a minimization algorithm:

1. Begin with M and collapse redundant states, getting rid of one at a time until the resulting machine is minimal.
2. Begin by overclustering the states of L into just two groups, accepting and nonaccepting. Then iteratively split those groups apart until all the distinctions that L requires have been made.

Both approaches work. We will present an algorithm that takes the second one.

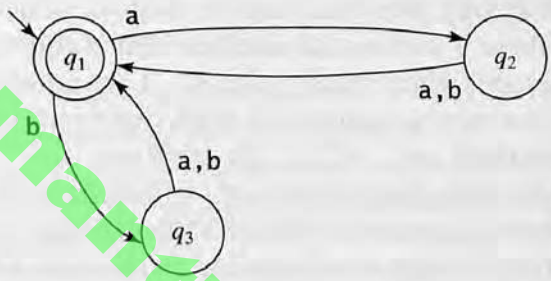
Our goal is to end up with a minimal machine in which all equivalent states of M have been collapsed. In order to do that, we need a precise definition of what it means for two states to be equivalent (and thus collapsible). We will use the following:

We will say that two states q and p in M are ***equivalent***, which we will write $q \equiv p$, iff for all strings $w \in \Sigma^*$, either w drives M to an accepting state from both q and p or it

drives M to a rejecting state from both q and p. In other words, no matter what continuation string comes next, M behaves identically from both states. Note that $\equiv$ is an equivalence relation over states, so it will partition the states of M into a set of equivalence classes.

EXAMPLE 5.27 A Nonminimal DFSM with Two Equivalent States

Let $\Sigma = \{a, b\}$. Let $L = \{w \in \Sigma^* : |w| \text{ is even}\}$. Consider the following FSM that accepts L:



In this machine state $q_2 \equiv$ state q_3.

For two states q and p to be equivalent, they must yield the same outcome for all possible continuation strings. We can't claim an algorithm for finding equivalent states that works by trying all possible continuation strings since there is an infinite number of them (assuming that Σ is not empty). Fortunately, we can show that it is necessary to consider only a finite subset of them. In particular, we will consider them one character at a time, and quit when considering another character has no effect on the machine we are building.

We define a series of equivalence relations $\equiv^n$, for values of $n \geq 0$. For any two states p and q, $p \equiv^n q$ iff p and q yield the same outcome for all strings of length n. So:

- $p \equiv^0 q$ iff they behave equivalently when they read ε. In other words, if they are both accepting or both rejecting states.
- $p \equiv^1 q$ iff they behave equivalently when they read any string of length 1. In other words, if any single character sends both of them to an accepting state or both of them to a rejecting state. Note that this is equivalent to saying that any single character sends them to states that are $\equiv^0$ to each other.
- $p \equiv^2 q$ iff they behave equivalently when they read any string of length 2, which they will do if, when they read the first character they land in states that are $\equiv^1$ to each other. By the definition of $\equiv^1$, they will then yield the same outcome when they read the single remaining character.
- And so forth.

We can state this definition concisely as follows. For all $p, q \in K$:

- $p \equiv^0 q$ iff they are both accepting or both rejecting states.
- For all $n \geq 1$, $q \equiv^n p$ iff:
  - $q \equiv^{n-1} p$, and
  - $\forall a \in \Sigma(\delta(p, a) \equiv^{n-1} \delta(q, a))$.

We will define *minDFSM*, a minimization algorithm that takes as its input a DFSM $M = (K, \Sigma, \delta, s, A)$. *MinDFSM* will construct a minimal DFSM M' that is equivalent to M. It begins by constructing $\equiv^0$, which divides the states of M into at most two equivalence classes, corresponding to A and $K - A$. If M has no accepting states or if all its states are accepting, then there will be only one nonempty equivalence class and we can quit since there is a one-state machine that is equivalent to M. We consider therefore only those cases where both A and $K - A$ are nonempty.

MinDFSM executes a sequence of steps, during which it constructs the sequence of equivalence relations $\equiv^1$, $\equiv^2$, ... To construct $\equiv^{k+1}$, *minDFSM* begins with $\equiv^k$. But then it splits equivalence classes of $\equiv^k$ whenever it discovers some pair of states that do not behave equivalently. *MinDFSM* halts when it discovers that $\equiv^n$ is the same as $\equiv^{n+1}$. Any further steps would operate on the same set of equivalence classes and so would also fail to find any states that need to be split.

We can now state the algorithm:

minDFSM(M: DFSM) =

1. *classes* = $\{A, K-A\}$. /* Initially, just two classes of states, accepting and rejecting.

2. Repeat until a pass at which no change to *classes* has been made:

2.1. *newclasses* = $\varnothing$. /* At each pass, we build a new set of classes, splitting the old ones as necessary. Then this new set becomes the old set, and the process is repeated.

2.2. For each equivalence class e in *classes*, if e contains more than one state, see if it needs to be split:

For each state q in e do: /* Look at each state and build a table of what it does. Then the tables for all states in the class can be compared to see if there are any differences that force splitting.

For each character c in Σ do:

Determine which element of *classes* q goes to if c is read.

If there are any two states p and q such that there is any character c such that, when c is read, p goes to one element of *classes* and q goes to another, then p and q must be split. Create as many new equivalence classes as are necessary so that no state remains in the same class

with a state whose behavior differs from its. Insert those classes into *newclasses*.

If there are no states whose behavior differs, no splitting is necessary. Insert e into *newclasses*.

2.3. $classes = newclasses$.

/* The states of the minimal machine will correspond exactly to the elements of *classes* at this point. We use the notation $[q]$ for the element of *classes* that contains the original state q.

3. Return $M' = (classes, \Sigma, \delta, [s_M], \{[q$: the elements of q are in $A_M]\})$, where $\delta_{M'}$ is constructed as follows:

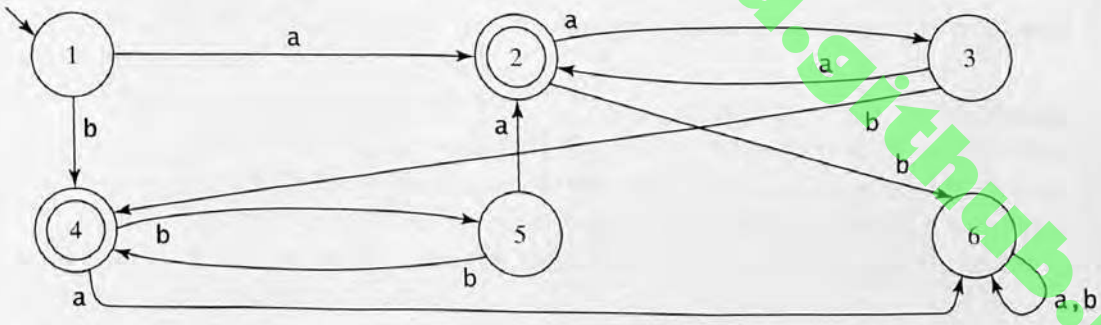
if $\delta_M(q, c) = p$, then $\delta_{M'}([q], c) = [p]$.

Clearly, no class that contains a single state can be split. So, if $|K|$ is k, then the maximum number of times that *minDFSM* can split classes is $k - 1$. Since *minDFSM* halts when no more splitting can occur, the maximum number of times it can go through the loop is $k - 1$. Thus *minDFSM* must halt in a finite number of steps. M' is the minimal DFSM that is equivalent to M since:

- M' is minimal: It splits classes and thus creates new states only when necessary to simulate M, and
- $L(M') = L(M)$: The proof of this is straightforward by induction on the length of the input string.

EXAMPLE 5.28 Using *minDFSM* to Find a Minimal Machine

Let $\Sigma = \{a, b\}$. Let $M =$



We will show the operation of *minDFSM* at each step:

Initially, $classes = \{[2, 4], [1, 3, 5, 6]\}$.

At step 1:

$((2, a), [1, 3, 5, 6])$	$((4, a), [1, 3, 5, 6])$	No splitting required here.
$((2, b), [1, 3, 5, 6])$	$((4, b), [1, 3, 5, 6])$	

EXAMPLE 5.28 (*Continued*)

((1, a), [2, 4])	((3, a), [2, 4])	((5, a), [2, 4])	((6, a), [1, 3, 5, 6])
((1, b), [2, 4])	((3, b), [2, 4])	((5, b), [2, 4])	((6, b), [1, 3, 5, 6])

There are two different patterns, so we must split into two classes, [1, 3, 5] and [6]. Note that, although [6] has the same behavior as [2, 4] after reading a single character, it cannot be combined with [2, 4] because they do not share behavior after reading no characters.

Classes = {[2, 4], [1, 3, 5], [6]}.

At step 2:

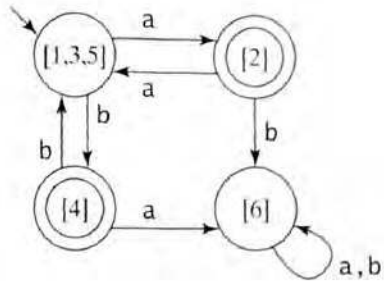
((2, a), [1, 3, 5])	((4, a), [6])		These two must be split.
((2, b), [6])	((4, b), [1, 3, 5])		
((1, a), [2, 4])	((3, a), [2, 4])	((5, a), [2, 4])	No splitting required.
((1, b), [2, 4])	((3, b), [2, 4])	((5, b), [2, 4])	

Classes = {[2], [4], [1, 3, 5], [6]}.

At step 3:

((1, a), [2])	((3, a), [2])	((5, a), [2])	No splitting required.
((1, b), [4])	((3, b), [4])	((5, b), [4])	

So *minDFSM* returns M' =



5.8 A Canonical Form for Regular Languages

A ***canonical form*** for some set of objects C assigns exactly one representation to each class of "equivalent" objects in C. Further, each such representation is distinct, so two objects in C share the same representation iff they are "equivalent" in the sense for which we define the form.

The ordered binary decision diagram (OBDD) is a canonical form for Boolean expressions that makes it possible for model checkers to verify the correctness of very large concurrent systems and hardware circuits. (B.1.3)

Suppose that we had a canonical form for FSMs with the property that two FSMs share a canonical form iff they accept the same language. Further suppose that we had an algorithm that on input M, constructed M's canonical form. Then some questions about FSMs would become easy to answer. For example, we could test whether two FSMs are equivalent (i.e., they accept the same language). It would suffice to construct the canonical form for each of them and test whether the two forms are identical.

The algorithm *minDFSM* constructs, from any DFSM M, a minimal machine that accepts $L(M)$. By Theorem 5.5, all minimal machines for $L(M)$ are identical except possibly for state names. So, if we could define a standard way to name states, we could define a canonical machine to accept $L(M)$ (and thus any regular language). The following algorithm does this by using the state-naming convention that we described in the proof of Theorem 5.5:

buildFSMcanonicalform(M: FSM) =

1. $M' = ndfsmtodfsm(M)$.
2. $M\# = minDFSM(M')$.
3. Create a unique assignment of names to the states of $M\#$ as follows:
   - 3.1. Call the start state q_0.
   - 3.2. Define an order on the elements of Σ.
   - 3.3 Until all states have been named do:

     Select the lowest numbered named state that has not yet been selected. Call it q.

     Create an ordered list of the transitions out of q by the order imposed on their labels.

     Create an ordered list of the as yet unnamed states that those transitions enter by doing the following: If the first transition is (q, c_1, p_1), then put p_1 first. If the second transition is (q, c_2, p_2) and p_2 is not already on the list, put it next. If it is already on the list, skip it. Continue until all transitions have been considered. Remove from the list any states that have already been named.

     Name the states on the list that was just created: Assign to the first one the name q_k, where k is the smallest index that hasn't yet been used. Assign the next name to the next state and so forth until all have been named.
4. Return $M\#$.

Given two FSMs M_1 and M_2, *buildFSMcanonicalform*(M_1) = *buildFSMcanonical form* (M_2) iff $L(M_1) = L(M_2)$. We'll see, in Section 9.1.4, one important use for this canonical form: It provides the basis for a simple way to test whether an FSM accepts any strings or whether two FSMs are equivalent.

5.9 Finite State Transducers ✹

So far, we have used finite state machines as language recognizers. All we have cared about, in analyzing a machine M, is whether or not M ends in an accepting state. But it is a simple matter to augment our finite state model to allow for output at each step of a machine's operation. Often, once we do that, we may cease to care about whether M actually accepts any strings. Many finite state transducers are loops that simply run forever, processing inputs.

One simple kind of finite state transducer associates an output with each state of a machine M. That output is generated whenever M enters the associated state. Deterministic finite state transducers of this sort are called Moore machines, after their inventor Edward Moore. A ***Moore machine*** M is a seven-tuple $(K, \Sigma, O, \delta, D, s, A)$, where:

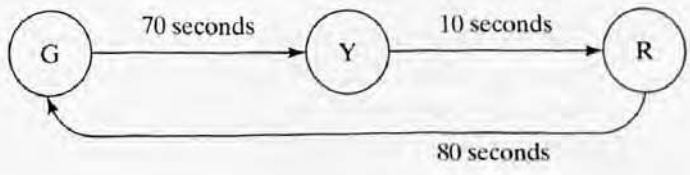
- K is a finite set of states,
- Σ is an input alphabet,
- O is an output alphabet,
- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states (although for some applications this designation is not important),
- δ is the transition function. It is function from $(K \times \Sigma)$ to (K), and
- D is the display or output function. It is a function from (K) to (O^*).

A Moore machine M computes a function $f(w)$ iff, when it reads the input string w, its output sequence is $f(w)$.

EXAMPLE 5.29 A Typical United States Traffic Light

Consider the following controller for a single direction of a very simple U.S. traffic light (which ignores time of day, traffic, the need to let emergency vehicles through, etc.). We will also ignore the fact that a practical controller has to manage all directions for a particular intersection. In Exercise 5.16, we will explore removing some of these limitations.

The states in this simple controller correspond to the light's colors: green, yellow and red. Note that the definition of the start state is arbitrary. There are three inputs, all of which are elapsed time.



A different definition for a deterministic finite state transducer permits each machine to output any finite sequence of symbols as it makes each transition (in other words, as it reads each symbol of its input). FSMs that associate outputs with transitions

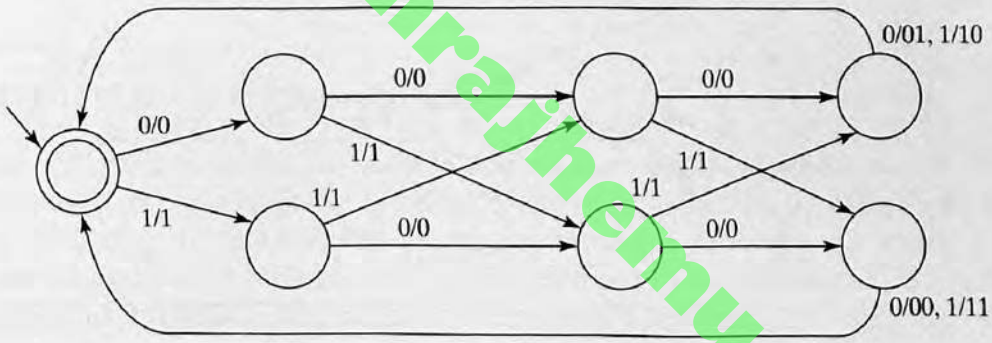
are called Mealy machines, after their inventor George Mealy. A ***Mealy machine*** M is a six-tuple $(K, \Sigma, O, \delta, s, A)$, where:

- K is a finite set of states,
- Σ is an input alphabet,
- O is an output alphabet,
- $s \in K$ is the start state,
- $A \subseteq$ is the set of accepting states, and
- δ is the transition function. It is a function from $(K \times \Sigma)$ to $(K \times O^*)$.

A Mealy machine M computes a function $f(w)$ iff, when it reads the input string w, its output sequence is $f(w)$.

EXAMPLE 5.30 Generating Parity Bits

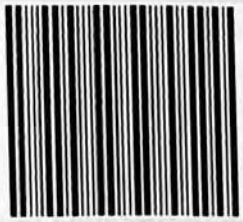
The following Mealy machine adds an odd parity bit after every four binary digits that it reads. We will use the notation a/b on an arc to mean that the transition may be followed if the input character is a. If it is followed, then the string b will be generated.



Digital circuits can be modeled as transducers using either Moore or Mealy machines. (P. 3)

EXAMPLE 5.31 A Bar Code Reader

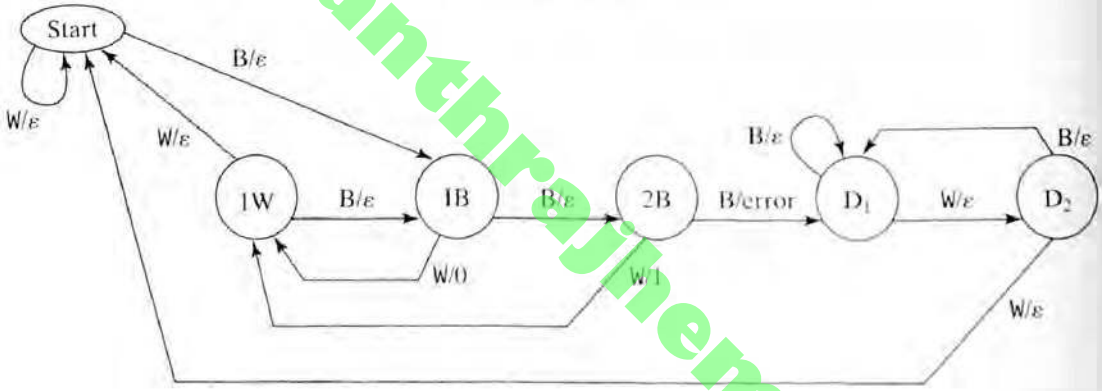
Bar codes are ubiquitous. We consider here a simplification: a bar code system that encodes just binary numbers. Imagine a bar code such as:



EXAMPLE 5.31 (*Continued*)

It is composed of columns, each of the same width. A column can be either white or black. If two black columns occur next to each other, it will look to us like a single, wide, black column, but the reader will see two adjacent black columns of the standard width. The job of the white columns is to delimit the black ones. A single black column encodes 0. A double black column encodes 1.

We can build a finite state transducer to read such a bar code and output a string of binary digits. We'll represent a black bar with the symbol B and a white bar with the symbol W. The input to the transducer will be a sequence of those symbols, corresponding to reading the bar code left to right. We'll assume that every correct bar code starts with a black column, so white space ahead of the first black column is ignored. We'll also assume that after every complete bar code there are at least two white columns. So the reader should, at that point, reset to be ready to read the next code. If the reader sees three or more black columns in a row, it must indicate an error and stay in its error state until it is reset by seeing two white columns.



Interpreters for finite state transducers can be built using techniques similar to the ones that we used in Section 5.6 to build interpreters for finite state machines.

5.10 Bidirectional Transducers ✹

A process that reads an input string and constructs a corresponding output string can be described in a variety of different ways. Why should we choose the finite state transducer model? One reason is that it provides a declarative, rather than a procedural, way to describe the relationship between inputs and outputs. Such a declarative model can then be run in two directions. For example:

- To read an English text requires transforming a word like "liberties" into the root word "liberty" and the affix PLURAL. To generate an English text requires transforming a root word like "liberty" and the semantic marker "PLURAL" into the surface word "liberties". If we could specify, in a single declarative model, the relationship between surface words (the ones we see in text) and underlying root words and affixes, we could use it for either application.

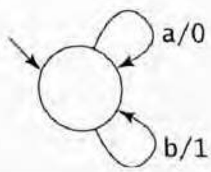
> The facts about English spelling rules and morphological analysis can be described with a bidirectional finite state transducer. (L.1)

- The Soundex system, described below in Example 5.33, groups names that sound alike. To create the Soundex representation of a name requires a set of rules for mapping the spelling of the name to a unique four character code. To find other names that sound like the one that generated a particular code requires running those same rules backwards.
- Many things we call translators need to run in both directions. For example, consider translating between Roman numerals and Arabic ones.

If we expand the definition of a Mealy machine to allow nondeterminism, then any of these bidirectional processes can be represented. A nondeterministic Mealy machine can be thought of as defining a relation between one set of strings (for example, English surface words) and a second set of strings (for example, English underlying root words, along with affixes). It is possible that we will need a machine that is nondeterministic in one or both directions because the relationship between the two sets may not be able to be described as a function.

EXAMPLE 5.32 Letter Substitution

When we define a regular language, it doesn't matter what alphabet we use. Anything that is true of a language L defined over the alphabet $\{a, b\}$ will also be true of the language L' that contains exactly the strings in L except that every a has been replaced by a 0 and every b has been replaced by a 1. We can build a simple bidirectional transducer that can convert strings in L to strings in L' and vice versa.



Of course, the real power of bidirectional finite state transducers comes from their ability to model more complex processes.

EXAMPLE 5.33 Soundex: A Way to Find Similar Sounding Names

People change the spelling of their names. Sometimes the spelling was changed for them when they immigrated to a country with a different language, a different set of sounds, and maybe a different writing system. For various reasons, one

EXAMPLE 5.33 (*Continued*)

might want to identify other people to whom one is related. But because of spelling changes, it isn't sufficient simply to look for people with exactly the same last name. The Soundex ☐ system was patented by Margaret O'Dell and Robert C. Russell in 1918 as a solution to this problem. The system maps any name to a four character code that is derived from the original name but that throws away details of the sort that often get perturbed as names evolve. So, to find related names, one can run the Soundex transducer in one direction, from a starting name to its Soundex code and then, in the other direction, from the code to the other names that share that code. For example, if we start with the name Kaylor, we will produce the Soundex code K460. If we then use that code and run the transducer backwards, we can generate the names Kahler, Kaler, Kaylor, Keeler, Kellar, Kelleher, Keller, Kelliher, Kilroe, Kilroy, Koehler, Kohler, Koller, and Kyler.

The Soundex system is described by the following set of rules for mapping from a name to a Soundex code:

1. If two or more adjacent letters (including the first in the name) would map to the same number if rule 3.1 were applied to them, remove all but the first in the sequence.
2. The first character of the Soundex code will be the first letter of the name.
3. For all other letters of the name do:
   - **3.1.** Convert the letters B, P, F, V, C, S, G, J, K, Q, X, Z, D, T, L, M, N, and R to numbers using the following correspondences:

     B, P, F, V = 1.

     C, S, G, J, K, Q, X, Z = 2.

     D, T = 3.

     L = 4.

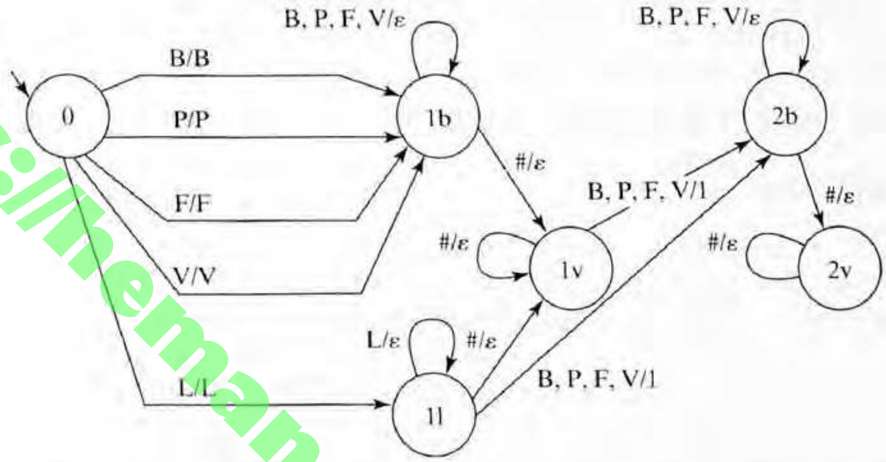
     M, N = 5.

     R = 6.

   - **3.2.** Delete all instances of the letters A, E, I, O, U, Y, H, and W.
4. If the string contains more than three numbers, delete all but the leftmost three.
5. If the string contains fewer than three numbers, pad with 0's on the right to get three.

Here's an initial fragment of a finite-state transducer that implements the relationship between names and Soundex codes. The complete version of this machine can input a name and output a code by interpreting each transition labeled x/y as saying that the transition can be taken on input x and it will output y. Going the other direction, it can input a code and output a name if it interprets each transition the other way: On input y, take the transition and output x. To simplify

the diagram, we've used two conventions: The symbol # stands for any one of the letters A,E,I,O,U,Y,H, or W. And a label of the form *x*, *y*, *z*/*a* is a shorthand for three transitions labeled *x*/*a*, *y*/*a*, and *z*/*a*. Also, the states are named to indicate how many code symbols have been generated/read.



Notice that in one direction (from names to codes), this machine operates deterministically. But, because information is lost in that direction, if we run the machine in the direction that maps from codes to names, it becomes nondeterministic. For example, the ε-transitions can be traversed any number of times to generate vowels that are not represented in the code. Because the goal, in running the machine in the direction from code to names is to generate actual names, the system that does this is augmented with a list of names found in U.S. census reports. It can then follow paths that match those names.

The Soundex system was designed for the specific purpose of matching names in United States census data from the early part of the twentieth century and before. Newer systems, such as Phonix and Metaphone, are attempts to solve the more general problem of identifying words that sound similar to each other. Such systems are used in a variety of applications, including ones that require matching a broader range of proper names (e.g., genealogy and white pages look up) as well as more general word matching tasks (e.g., spell checking).

5.11 Stochastic Finite Automata: Markov Models and HMMs ◆

Most of the finite state transducers that we have considered so far are deterministic. But that is simply a property of the kinds of applications to which they are put. We do not want to live in a world of nondeterministic traffic lights or phone switching circuits. So we typically design controllers (i.e., machines that run things) to be deterministic. For some applications though, nondeterminism can be useful. For example, it can add entertainment value.

> Nondeterministic (possibly stochastic) FSMs can form the basis of video games. (N.3.1)

But now consider problems like the name-evolution one we just discussed. Now we are not attempting to build a controller that drives the world. Instead we are trying to build a model that describes and predicts a world that we are not in control of. Nondeterministic finite state models are often very useful tools in solving such problems. And typically, although we do not know enough to predict with certainty how the behavior of the model will change from one step to the next (thus the need for nondeterminism), we do have some data that enable us to estimate the probability that the system will move from one state to the next. In this section, we explore the use of nondeterministic finite state machines and transducers that have been augmented with probabilistic information.

5.11.1 Markov Models

A Markov model ☐ is an NDFSM in which the state at each step can be predicted by a probability distribution associated with the current state. Steps usually correspond to time intervals, but they may correspond to any ordered discrete sequence. In essence we replace transitions labeled with input symbols by transitions labeled with probabilities. The usual definition of a Markov model is that its behavior at time t depends only on its state at time $t-1$ (although higher-order models may allow any finite number of past states to play a role). Of course, if we eliminate an input sequence, that is exactly the property that characterizes an FSM.

> Markov models have been used in music composition. (N.1.1) They have also been used to model the generation of many other sorts of content, including Web pages ☐.

Formally a ***Markov model*** is a triple $M = (K, \pi, A)$, where:

- K is a finite set of states,
- π is a vector that contains the initial probabilities of each of the states, and
- A is a matrix that represents the transition probabilities. $A[p, q] = \Pr(\mathit{state}\ q \text{ at time } t \mid \text{state } p \text{ at time } t-1)$. In other words $A[p, q]$ is the probability that, if M is in state p, it will go to state q next.

Some definitions specify a unique start state, but this definition is more general. If there is a unique start state, then its initial probability is 1 and the initial probabilities of all other states are 0.

Notice that we have not mentioned any output alphabet. We will assume that the output at each step is simply the name of the state of the machine at that step. The sequence of outputs produced by a Markov model is often called a ***Markov chain***.

> The link structure of the World Wide Web can be modeled as a Markov chain, where the states correspond to Web pages and the probabilities describe the likelihood, in a random walk, of going from one page to the next. Google's PageRank is based on the limits of those probabilities ☐.

Given a Markov model that describes some random process, we can answer either of the following questions:

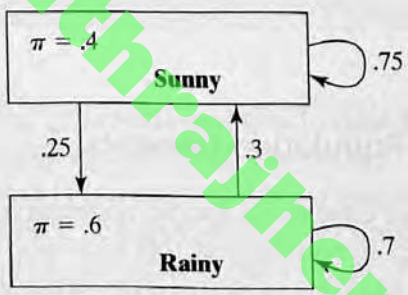
- What is the probability that we will observe a particular sequence $s_1 s_2 \ldots s_n$ of states? We can compute this as follows, using the probability that s_1 is the start state and then multiplying by the probabilities of each of the transitions:

$$\Pr(s_1 s_2 \ldots s_n) = \pi[s_1] \cdot \prod_{i=2}^{n} A[s_{i-1}, s_i].$$

- If the process runs for an arbitrarily long sequence of steps, what is likely to be the result? More specifically, for each state in the system, what is the probability that the system will land in that state?

EXAMPLE 5.34 A Simple Markov Model of the Weather

Suppose that we have the following model for the weather where we live. This model assumes that the weather on day t is influenced only by the weather on day $t - 1$.



We are considering a five day camping trip and want to know the probability of five sunny days in a row. So we want to know the probability of the sequence Sunny Sunny Sunny Sunny Sunny. The model tells us that it is:

$$.4 \cdot (.75)^4 = .1266$$

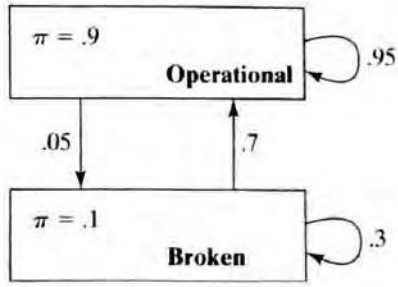
Or we could ask, given that it's sunny today, what is the probability that, if we leave now, it will stay sunny for four more days. Now we assume that the model starts in state Sunny, so we compute:

$$(.75)^4 = .316$$

EXAMPLE 5.35 A Simple Markov Model of System Performance

Markov models are used extensively to model the performance of complex systems of all kinds, including computers, electrical grids, and manufacturing plants. While real models are substantially more complex, we can see how these models work by taking Example 5.34 and renaming the states:

EXAMPLE 5.35 (*Continued*)



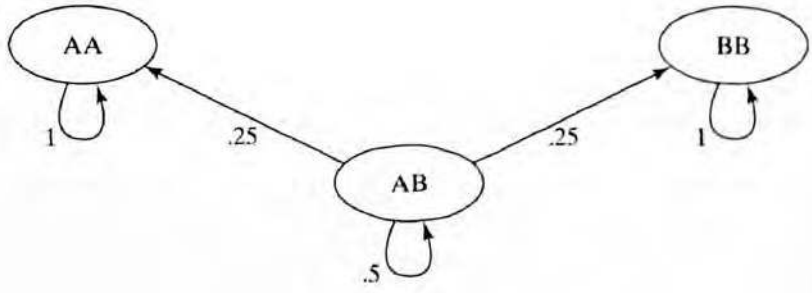
To make it a bit more realistic, we've changed the probabilities so that they describe a system that actually works most of the time. We'll also use smaller time intervals, say seconds. Now we might ask, "Given that the system is now up, what is the probability that the system will stay up for an hour (i.e., for 3600 time steps). The (possibly surprising) answer is:

$$.95^{3600} = 6.3823 \cdot 10^{-81}$$

EXAMPLE 5.36 Population Genetics

In this example we consider a simple problem in population genetics. For a survey of the biological concepts behind this example, see Appendix K. Suppose that we are interested in the effect of inbreeding on the gene pool of a diploid organism (an organism, such as humans, in which each individual has two copies of each gene). Consider the following simple model of the inheritance of a single gene with two alleles (values): A and B. There are potentially three kinds of individuals in the population: the AA organisms, the BB organisms, and the AB organisms. Because we are studying inbreeding, we'll make the assumption that individuals always mate with others who are genetically similar to themselves and so possess the same gene pair.

To simplify our model, we will assume that one couple mates, has two children, and dies. So we can think of each individual as replacing itself and then dying. We can build the following Markov model of a chain of descendents. Each step now corresponds to a generation.



AA pairs can produce only AA offspring. BB pairs can produce only BB offspring. But what about AB pairs? What is their fate? We can answer this question by considering the probability that the model, if it starts in state AB and runs for some number of generations, will land in state AB. That probability is $.5^n$, where n is the number of generations. As n grows, that number approaches 0. We show how quickly it does so in the following table:

n	Pr(AB)
1	.5
5	.03125
10	.0009765625
100	$7.8886 \cdot 10^{-31}$

After only 10 generations, very few heterozygous individuals (i.e., possessing two different alleles) remain. After 100 generations, almost none do. If there is survival advantage in being heterozygous, this could be a disaster for the population. The disaster can be avoided, of course, if individuals mate with genetically different individuals.

Where do the probabilities in a Markov model come from? In some simple cases, they may be computed by hand and added to the system. In most cases, however, they are computed by examining real datasets and discovering the probabilities that best describe those data. So, for example, the probabilities we need for the system performance model of Example 5.35 could be extracted from a log of system behavior over some recent period of time. To see how this can be done, suppose that we have observed the output sequences: T P T Q P Q T and S S P T P Q Q P S T Q P T T P. The correct value for A[P, Q] is the number of times the pair P Q appears in the sequence divided by the total number of times that P appears in any position except the last. Similarly, the correct value for π[P] is the total number of times that P is the first symbol in a sequence divided by the total number of sequences. In realistic problem contexts, the models are huge and they evolve over time. There exist more computationally tractable algorithms for updating the probabilities (and, when necessary the states) of such models.

Substantial work has been done on efficient techniques for updating the huge Markov model of the World Wide Web that is used to compute Google's PageRanks. Note here that both the state set (corresponding to the set of pages on the Web) as well as the probabilities (which depend on the link structure of the Web) must be regularly revised.

All of the Markov models we have presented so far have the property that their behavior at step t is a function only of their state at step $t-1$. Such models are called first-order. To build a first-order model with k states requires that we specify k^2 transition

probabilities. Now suppose that we wish to describe a situation in which what happens next depends on the previous two states. Or the previous three. Using the same techniques that we used to build a first-order model, we can build models that consider the previous n states for any fixed n. Such models are called n^{th} order Markov models. Notice that an n^{th} order model requires k^{n+1} transition probabilities. But if there are enough data available to train a higher-order model (i.e., to assign appropriate probabilities to all of the required transitions), it may be possible to build a system that quite accurately mimics the behavior of a very complex system.

> A third-order Markov model, trained on about half of this book, used word frequencies to generate the text "The Pumping Theorem is a useful way to define a precedence hierarchy for the operators + and *." (L.3.2) A clever application of a higher order Markov model of English is in producing spam that is hard to detect. (L.3.2)

> Early work on the use of Markov models for musical composition suggested that models of order four or less tended to create works that seemed random, while models of order seven or more tended to create works that felt just like copies of works on which the model was trained. (N.1.1)

Whenever we build a Markov model to describe a naturally occurring process, there is a sense in which we are using probabilities to hide an underlying lack of understanding that would enable us to build a deterministic model of the phenomenon. So, for example, if we know that our computer system is more likely to crash in the morning than in the evening, that may show up as a pair of different probabilities in a Markov model, even if we have no clue why the time of day affects system performance. Some Markov models that do a pretty good job of mimicking nature may seem silly to us for exactly that reason. The one that generates random English text is a good example of that. But now suppose that we had a model that did a very good job of predicting earthquakes. Although we might rather have a good structural model that tells us why earthquakes happen, a purely statistical, predictive model would be a very useful tool. It is because of cases like this that Markov models can be extremely valuable tools for anyone studying complex systems (be they naturally occurring ones like plate tectonics or engineering artifacts like computer systems).

5.11.2 Hidden Markov Models

Now suppose that we are interested in analyzing a system that can be described with a Markov model with one important difference: The states of the system are not directly observable. Instead the model has a separate set of output symbols, which are emitted, with specified probabilities, whenever the system enters one of its now "hidden" states. Now we must base our analysis of the system on an observed sequence of

output symbols, from which we can infer, with some probability, the actual sequence of states of the underlying model.

Examples of significant problems that can be described in this way include:

- ***DNA and protein evolution:*** A protein is a sequence of amino acids that is manufactured in living organisms according to a DNA blueprint. Mutations that change the blueprint can occur, with the result that one amino acid may be substituted for another, one or more amino acids may be deleted, or one or more additional amino acids may be inserted. When we examine a DNA fragment or a protein, we'd like to be able to reconstruct the evolutionary process so that we can find other proteins that are functionally related to the current one, even though its details may be different. But the process isn't visible; only its result is.

> HMMs are used for DNA and protein sequence alignment in the face of mutations and other kinds of evolutionary change. (K.3.3)

- ***Speech understanding:*** When we talk, our mouths map from the sentences we want to say into sequences of sounds. The mapping is complex and nondeterministic since multiple words may map to the same sound, words are pronounced differently as a function of the words before and after them, we all form sounds slightly differently, and so forth. All a listener can hear is the sequence of sounds. (S)he would like to reconstruct the mapping (backwards) in order to determine what words we were attempting to say.

> HMMs are used extensively in speech understanding systems. (L.5)

- ***Optical character recognition (OCR):*** When we write, our hands map from an idealized symbol to some set of marks on a page. The marks are observable, but the process that generates them isn't. Imagine that we could describe a probabilistic process corresponding to each symbol that we can write. Then, to interpret the marks, we must select the process that is most likely to have generated the marks we can see.

What is a Hidden Markov Model?

A powerful technique for solving problems such as this is the ***hidden Markov model*** or ***HMM***. An HMM is a nondeterministic finite state transducer that has been augmented with three kinds of probabilistic information:

- Each state is labeled with the probability that the machine will be in that state when it starts.
- Each transition from some state p to some (possibly identical) state q is labeled with the probability that, whenever the machine is in state p, it will go next to state q. We

can specify M's transition behavior completely by defining these probabilities. If it is not possible for M to go from some state p to some other state q, then we simply state the probability of going from p to q as 0.

- Each output symbol c at each state q is labeled with the probability that the machine, if it is in state q, will output c.

Formally, an HMM M is a quintuple (K, O, π, A, B), where:

- K is a finite set of states.
- O is the output alphabet.
- π is a vector that contains the initial probabilities of each of the states.
- A is a matrix that represents the transition probabilities. $A[p, q] = \Pr(\text{state } q \text{ at time } t \mid \text{state } p \text{ at time } t - 1)$.
- B, sometimes called the confusion matrix, represents the output probabilities. $B[q, o] = \Pr(\text{output } o \mid \text{state } q)$. Note that outputs are associated with states (as in Moore machines).

The name "hidden Markov model" derives from the two key properties of such devices:

- They are Markov models. Their state at time t is a function solely of their state at time $t - 1$.
- The actual progression of the machine from one state to the next is hidden from all observers. Only the machine's output string can be observed.

To use an HMM as the basis for an application program, we typically have to solve some or all of the following problems:

- ***The decoding problem:*** Given an observation sequence O and an HMM M, discover the path through M that is most likely to have produced O. For example, O might be a string of words that form a sentence. We might have an HMM that describes the structure of naturally occurring English sentences. Each state in M corresponds to a part of speech, such as noun, verb, or adjective. It's not possible to tell, just by looking at O, what sequence of parts of speech generated it, since many words can have more than one part of speech. (Consider, for example, the simple English sentence, "Hit the fly ball.") But we need to infer the parts of speech (a process called part of speech or POS tagging) before we can parse the sentence. We can do that if we can find the path through the HMM that is the most likely to have generated the observed sentence. This problem can be solved efficiently using a dynamic programming algorithm called the Viterbi algorithm, described below.

HMMs are often used for part of speech tagging. (L.2)

Suppose that the sequences that we observe correspond to original sequences that have been altered in some way. The alteration may have been done intentionally (we'll call this "obfuscation") or it may be the result of a natural phenomenon like evolution or a noisy transmission channel. In either case, if we want to know what the original sequence was, we have an instance of the decoding problem. We seek to find the original sequence that is most likely to have been the one that got transformed into the observed sequence.

> In the Internet era, an important application of obfuscation is the generation of spam. If specific words are known to trigger spam filters, they can be altered, by changing vowels, introducing special characters, or whatever, so that they are still recognizable to people but unrecognizable, at least until the next patch, to the spam filters. HMMs can be used to perform "deobfuscation" in an attempt to foil the obfuscators. 💻.

- ***The evaluation problem:*** Given an observation sequence O and a set of HMMs that describe a collection of possible underlying models, choose the HMM that is most likely to have generated O. For example, O might be a sequence of sounds. We might have one HMM for each of the words that we know. We need to choose the word model that is most likely to have generated O. As another example, consider again the protein problem: Now we have one HMM for each family of related proteins. Given a new sample, we want to find the family to which it is most likely to be related. So we look for the HMM that is most likely to have generated it. This problem can be solved efficiently using the forward algorithm, which is very similar to the Viterbi algorithm except that it considers all paths through a candidate HMM, rather than just the most likely one.
- ***The training problem:*** We typically assume, in crafting an HMM M, that the set K of states is built by hand. But where do all the probabilities in π, A, and B come from? Fortunately, there are algorithms that can learn them from a set of training data (i.e., a set of observed output sequences O). One of the most commonly used algorithms is the Baum-Welch algorithm 💻, also called the forward-backward algorithm. Its goal is to tune π, A, and B so that the resulting HMM M has the property that, out of all the HMMs whose state set is equal to K, M is the one most likely to have produced the outputs that constitute the training set. Because the states cannot be directly observed (as they can be in a standard Markov model), the training technique that we described in Section 5.11.1 won't work here. Instead, the Baum-Welch algorithm employs a technique called ***expectation maximization*** or EM. It is an iterative method, so it begins with some initial set of values for π, A, and B. Then it runs the forward algorithm, along with a related backward algorithm, on the training data. The result of this step is a set of probabilities that describe the likelihood that the existing machine, with the current values of π, A, and B, would have output the training set. Using those probabilities, Baum-Welch updates π, A, and B to increase those probabilities. The process continues until no changes to the parameter values can be made.

The Viterbi Algorithm

Given an HMM M and an observed output sequence O, a solution to the decoding problem is the path through M that is most likely to have produced O. One way to find that most likely path is to explore all paths of length $|O|$, keeping track of the accumulated probabilities, and then report the path whose probability is the highest. This approach is straightforward, but may require searching a tree with $|K_M|^{|O|}$ nodes, so the time required may grow exponentially in the length of O.

A more efficient approach uses a dynamic programming technique in which the most likely path of some length, say t, is computed once and then extended by one more step to find the most likely path of length $t + 1$. The Viterbi algorithm uses this approach. It solves the decoding problem by computing, for each step t and for each state q in M:

- The most likely path to q of all the ones that would have generated $O_1 \ldots O_t$.
- The probability of that path.

Once it has done that for each step for which an output was observed, it traces the path backwards. It assumes that the last state is the one at the end of the overall most likely path. The next to the last state is the one that preceded that one on the most likely path, and so forth.

Assume, at each step t, that the algorithm has already considered all paths of length $t - 1$ that could have generated $O_1 \ldots O_{t-1}$. From those paths, it has selected, for each state p, the most likely path to p and it has recorded the probability of the model taking that path, reaching p, and producing $O_1 \ldots O_{t-1}$. We assume further that the algorithm has also recorded, at each state p, the state that preceded p on that most likely path. Before the first output symbol is observed, the probability that the system has reached some state p is simply $\pi(p)$ and there is no preceding state.

Because the model is Markovian, the only thing that affects the probability of the next state is the previous state. In constructing the model, we assumed that prior history doesn't matter (although that may be only an approximation to reality for some problems). So, at step t, we compute, for each state q, the probability that the best path so far that is consistent with $O_1 \ldots O_t$ ends in q and outputs the first t observed symbols. We do this by considering each state p that the model could have been in at step $t - 1$. We already know the probability that the best path up to step $t - 1$ landed in p and produced the observed output sequence. So, to add one more step, we multiply that probability by $A[p, q]$, the probability that the model, if it were in p, would go next to q. But we have one more piece of information: the next output symbol. So, to compute the probability that the model went through p, landed in q, and output the next symbol o, we multiply by $B[p, o]$. Once these numbers have been computed for all possible preceding states p, we choose the most likely one (i.e., the one with the highest score as described above). We record that score at q and we record at q that the most likely predecessor state is the one that produced that highest score.

Although we've described the output function as a function of the state the model is in, we don't actually consider it until we compute the next step, so it may be easier to think of the outputs as associated with the transitions rather than with the states. In particular, the computation that we have just described will end by choosing the state

in which the model is most likely to land just after it outputs the final observed symbol. That last state will not generate any output.

Once all steps have been considered, we can choose the overall most likely path as follows: Consider all states. The model is most likely to have ended in the one that, at the final time step, has the highest score as described above. Call that highest scoring state the last state in the path. Find the state that was marked as immediately preceding that one. Continue backwards to the start state.

We can summarize this process, known as the ***Viterbi algorithm***, as follows: Given an observed output sequence O, we will consider each time step between 1 and the length of O. At each such step t, we will set $score(q, t)$ to the highest probability associated with any path of length t that lands M in q, having output the first t symbols in O. We will set $backptr(q, t)$ to the state that immediately preceded q along that best path. Once *score* and *backptr* have been computed for each state at each time step t, we can start at the most likely final state and trace backwards to find the sequence *states* that describes the most likely path through M consistent with O. So the Viterbi algorithm is:

Viterbi(M: Markov model, O: output sequence) =

1. For $t = 0$, for each state q, set $score[q, t]$ to $\pi[q]$.

2. /* Trace forward recording the best path at each step:

For $t = 1$ to $|O|$ do:

2.1. For each state q in K do:

2.1.1. For each state p in K that could have immediately preceded q:

$$candidatescore[p] = score[p, t-1] * A[p, q] * B[p, O_t].$$

2.1.2. /* Record score along most likely path:

$$score[q, t] = \max_{p \in K} candidatescore[p].$$

2.1.3. /* Set q's *backptr*. The function argmax returns the value of the argument p that produced the maximum value of $candidatescore[p]$:

$$backptr[q, t] = \operatorname{argmax}_{p \in K} candidatescore[p].$$

/* Retrieve the best path by going backwards from the most likely last state:

3. $states[|O|]$ = the state q with the highest value of $score[q, |O|]$.

4. For $t = |O| - 1$ to 0 do:

4.1. $states[t] = backptr[states[t+1], t+1]$.

5. Return $states[0 : |O| - 1]$. /* Ignore the last state since its output was not observed.

The Forward Algorithm

Now suppose that we want to solve the evaluation problem: Given a set of HMMs and an observed output sequence O, decide which HMM had the highest probability of producing O. This problem can be solved with the ***forward algorithm***,

which is very similar to the Viterbi algorithm except that, instead of finding the single best path through an HMM M, it computes the probability that M could have output O along *any* path. In step 2.1.2, the Viterbi algorithm selects the highest score associated with any one path to q. The forward algorithm, at that point, sums all the scores. The other big difference between the Viterbi algorithm and the forward algorithm is that the forward algorithm does not need to find a particular path. So it will not have to bother maintaining the *backptr* array. We can state the algorithm as follows:

forward(M: Markov model, O: output sequence) =

1. For $t = 0$, for each state q, set *forward-score*$[q, t]$ to $\pi[q]$.

2. /* Trace forward recording, at each step, the total probability associated with all paths to each state:

For $t = 1$ to $|O|$ do:

2.1. For each state q in K do:

2.1.1. Consider each state p in K that could have immediately preceded q:

$$candidatescore[p] = forwardscore[p, t-1] * A[p, q] * B[p, O_t].$$

2.1.2. /* Sum scores over all paths:

$$forwardscore[q, t] = \sum_p candidatescore[p].$$

3. /* Find the total probability of going through M along any path, landing in any of M's states, and emitting O. This is simply the sum of the probability of landing in state 1 having emitted O, plus the probability of landing in state 2 having emitted O, and so forth. So:

$$totalprob = \sum_{q \in K} forwardscore[q, |O|].$$

4. Return *totalprob*.

To solve the evaluation problem, we run the forward algorithm on all of the contending HMMs and return the one with the highest final score.

The Complexity of the Viterbi and the Forward Algorithms

Analyzing the complexity of the Viterbi and the forward algorithms is straightforward. In both cases, the outer loop of step 2 is executed once for each observed output, so $|O|$ times. Within that loop, the computation of *candidatescore* is done once for each state pair. So if M has k states, it is done k^2 times. The computation of *score*/*forwardscore* takes $\mathcal{O}(k)$ steps, as does the computation of *backptr* in the Viterbi algorithm. The final operation of the Viterbi algorithm (computing the list of states to be returned) takes $\mathcal{O}(|O|)$ steps. The final operation of the forward algorithm (computing the total probability of producing the observed output) takes $\mathcal{O}(k)$ steps. So, in both cases, the total time complexity is $\mathcal{O}(k^2 \cdot |O|)$.

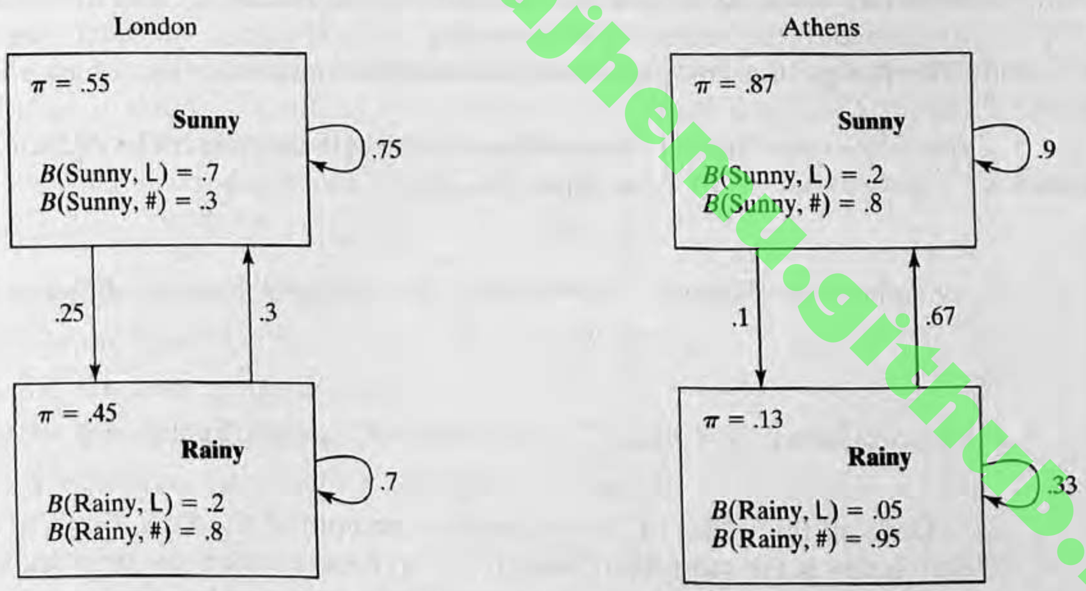
An Example of How These Algorithms Work

The real power of HMMs is in solving complex, real-world problems in which probability estimates can be derived from large datasets. So it is hard to illustrate the effectiveness of HMMs on small problems, but the idea should be clear from the following simple example of the use of the Viterbi algorithm.

EXAMPLE 5.37 Using the Viterbi Algorithm to Guess the Weather

Suppose that you are a state department official in a small country. Each day, you receive a report from each of your consular offices telling you whether or not any of your passports were reported missing that day. You know that the probability of a passport getting lost or stolen is a function of the weather, since people tend to stay inside (and thus manage to keep track of their passports) when the weather is bad. But they tend to go out and thus risk getting their passport lost or stolen if the weather is good. So it amuses you to try to infer the weather in your favorite cities by watching the lost passport reports. We'll use the symbol L to mean that a passport was lost and the symbol # to mean that none was. So, for example, a report for a week might look like LL##L###.

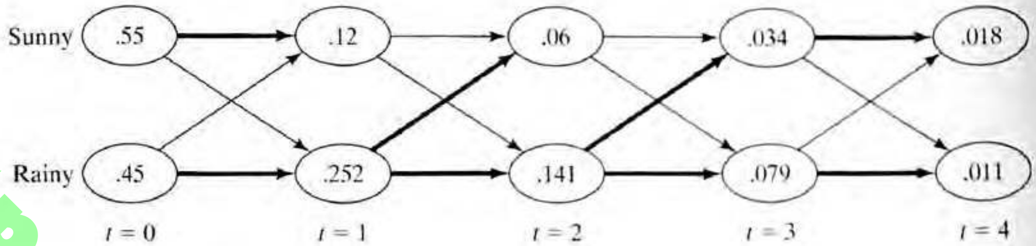
We'll consider just two cities, London and Athens. We can build an HMM for each. Both HMMs have two states, Sunny and Rainy.



Now suppose that you receive the report ###L from London and you want to find out what the most likely sequence of weather reports was for those days. The Viterbi algorithm will solve the problem.

The easiest way to envision the way that *Viterbi* works is to imagine a lattice, in which each column corresponds to a step and each row corresponds to a state in M:

EXAMPLE 5.37 *(Continued)*



The number shown at each point (q, t) is the value that *Viterbi* computes for *score*$[q, t]$. So we can think of Viterbi as creating this lattice left to right, and filling in scores as it goes along. The arrows represent possible transitions in M. The heavy arrows indicate the path that is recorded in the matrix *backptr*.

At $t = 0$, the probabilities recorded in *score* are just the initial probabilities, as given in π. So the sum of the values in column 1 is 1. At later steps, the sum is less than 1 because we are considering only the probabilities of paths through M that result in the observed output sequence. Other paths could have produced other output sequences.

At all times $t > 0$, the values for *score* can be computed by considering the probabilities at the previous time (as recorded in *score*), the probabilities of moving from one state to another (as recorded in the matrix A), and the probabilities (recorded in the vector O) of observing the next output symbol. To see how the Viterbi algorithm computes those values, let's compute the value of *score*[Sunny, 1]:

$$\begin{aligned} \textit{candidate-score}[\text{Sunny}] &= \textit{score}[\text{Sunny}, 0] \cdot A[\text{Sunny}, \text{Sunny}] \cdot B[\text{Sunny}, \#] \\ &= .55 \cdot .75 \cdot .3 \\ &= .12 \end{aligned}$$

$$\begin{aligned} \textit{candidate-score}[\text{Rainy}] &= \textit{score}[\text{Rainy}, 0] \cdot A[\text{Rainy}, \text{Sunny}] \cdot B[\text{Rainy}, \#] \\ &= .45 \cdot .3 \cdot .8 \\ &= .11 \end{aligned}$$

So *score*[Sunny, 1] = *max*(.12, .11) = .12, and *backptr*(Sunny, 1) is set to Sunny.

Once all the values of *score* have been computed, the final step is to observe that Sunny is the most likely state for M to have reached just prior to reading a fifth output symbol. The state that most likely preceded it is Sunny, so we report Sunny as the last state to have produced output. Then we trace the backpointers and report that the most likely sequence of weather reports is Rainy, Rainy, Rainy, Sunny.

Now suppose that the fax machine was broken and the reports for last week came in with the city names chopped off the top. You have received the report ###L and you want to know whether it is more likely that it came from London or from Athens. To solve this problem, you use the forward algorithm. You run the

output sequence ###L through the London model and through the Athens model, this time computing the total probability (as opposed to just the probability along the best path) of reaching each state from any path that is consistent with the output sequence. The most likely source of this report is the model with the highest final probability.

5.12 Finite Automata, Infinite Strings: Büchi Automata ✹

So far, we have considered, as input to our machines, only strings of finite length. Thus we have focused on problems for which we expect to write programs that read an input, compute a result, and halt. Many problems are of that sort, but some are not. For example, consider:

- An operating system.
- An air traffic control system.
- A factory process control system.

Ideally, such systems never halt. They should accept an infinite string of inputs and continue to function. Define Σ^ω to be the set of infinite length strings drawn from the alphabet Σ. For the rest of this discussion, define a language to be a set of such infinite-length strings.

To model the behavior of processes that do not halt, we can extend our notion of an NDFSM to define a machine whose inputs are elements of Σ^ω. Such machines are sometimes called ω-automata (or omega automata).

We'll define one particular kind of ω-automaton: A ***Büchi automaton*** is a quintuple $(K, \Sigma, \Delta, S, A)$, where:

- K is a finite set of states.
- Σ is the input alphabet.
- $S \subseteq K$ is a set of start states.
- $A \subseteq K$ is the set of accepting states.
- Δ is the transition relation. It is a finite subset of:

$$(K \times \Sigma) \times K.$$

Note that, unlike NDFSMs, Büchi automata may have more than one start state. Note also that the definition of a Büchi automaton does not allow ε-transitions.

We define configuration, initial configuration, yields-in-one-step, and yields exactly as we did for NDFSMs. A ***computation*** of a Büchi automaton M is an infinite sequence of configurations $C_0, C_1, \ldots$ such that:

- C_0 is an initial configuration, and
- $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \ldots$

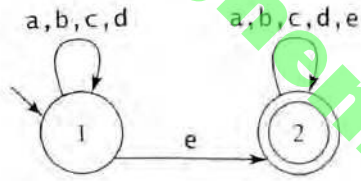
But now we must define what it means for a Büchi automaton M to accept a string. We can no longer define acceptance by the state of M when it runs out of input, since it won't. Instead, we'll say that M accepts a string $w \in \Sigma^\omega$ iff, in at least one of its computations, there is some accepting state q such that, when processing w, M enters q an infinite number of times. So note that it is not required that M enter an accepting state and stay there. But it is not sufficient for M to enter an accepting state just once (or any finite number of times). As before, the language accepted by M, denoted $L(M)$, is the set of all strings accepted by M. A language L is ***Büchi-acceptable*** iff it is accepted by some Büchi automaton.

> Büchi automata can be used to model concurrent systems, hardware devices, and their specifications. Then programs called model checkers can verify that those systems correctly conform to a set of stated requirements. (H.1.2)

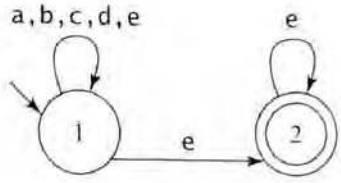
EXAMPLE 5.38 Büchi Automata for Event Sequences

Suppose that there are five kinds of events that can occur in the system that we wish to model. We'll call them a, b, c, d, and e. So let $\Sigma = \{a, b, c, d, e\}$.

We first consider the case in which we require that event e occur at least once. The following (nondeterministic) Büchi automaton accepts all and only the elements of Σ^ω that contain at least one occurrence of e:

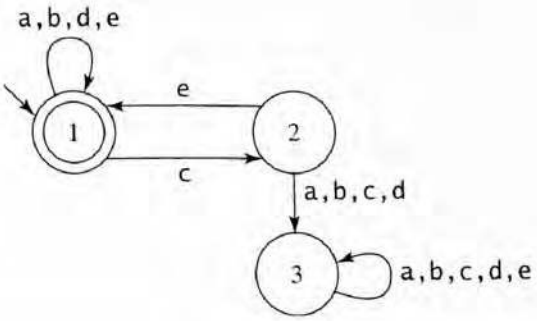


Now suppose that we require that there come a point after which only e's can occur. The following Büchi automaton (described using our convention that the dead state need not be written explicitly) accepts all and only the elements of Σ^ω that eventually reach a point after which no events other than e's occur:



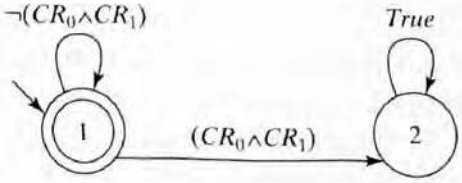
Finally, suppose that we require that every c event be immediately followed by an e event. The following Büchi automaton (this time with the dead state, 3, shown explicitly) accepts all and only the elements of Σ^ω that satisfy that requirement:

EXAMPLE 5.38 (*Continued*)



EXAMPLE 5.39 Mutual Exclusion

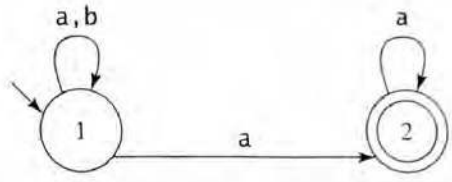
Suppose that we want to model a concurrent system with two processes and enforce the constraint, often called a mutual exclusion property, that it never happens that both processes are in their critical regions at the same time. We could do this in the usual way, using an alphabet of atomic symbols such as {`Both`, `NotBoth`}, where the system receives the input `Both` at any time interval at which both processes are in their critical region and the input `NotBoth` at any other time interval. But a more direct way to model the behavior of complex concurrent systems is to allow inputs that correspond to Boolean expressions that capture the properties of interest. That way, the same Boolean predicates can be combined into different expressions in different machines that correspond to different desirable properties. To capture the mutual exclusion constraint, we'll use two Boolean predicates, CR_0, which will be *True* iff $process_0$ is in its critical region and CR_1, which will be *True* iff $process_1$ is in its critical region. The inputs to the system will then be drawn from a set of three Boolean expressions: $\{(CR_0 \wedge CR_1), \neg(CR_0 \wedge CR_1), True\}$. The following Büchi automaton accepts all and only the input sequences that satisfy the property that $(CR_0 \wedge CR_1)$ never occurs:



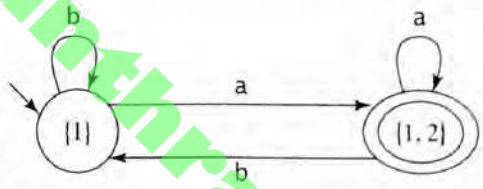
While there is an obvious similarity between Büchi automata and FSMs, and the languages they accept are related, as described below, there is one important difference. For Büchi automata, nondeterminism matters.

EXAMPLE 5.40 For Büchi Automata, Nondeterminism Matters

Let $L = \{w \in \{a, b\}^\omega : \#_b(w) \text{ is finite}\}$. Note that every string in L must contain an infinite number of a's. The following nondeterministic Büchi automaton accepts L:



We can try to build a corresponding deterministic machine by using the construction that we used in the proof of Theorem 5.3 (which says that for every NDFSM there does exist an equivalent DFSM). The states of the new machine will then correspond to subsets of states of the original machine and we'll have:



This new machine is indeed nondeterministic and it does accept all strings in L. Unfortunately, it also accepts an infinite number of strings that are not in L, including $(ba)^\omega$. More unfortunately, we cannot do any better.

THEOREM 5.7 Nondeterministic versus Deterministic Büchi Automata

Theorem: There exist languages that can be accepted by a nondeterministic Büchi automaton (i.e., one that meets the definition we have given), but for which there exists no equivalent deterministic Büchi automaton (i.e., one that has a single start state and whose transitions are defined by a function from $(K \times \Sigma)$ to K).

Proof: The proof is by a demonstration that no deterministic Büchi automaton accepts the language $L = \{w \in \{a, b\}^\omega : \#_b(w) \textit{ is finite}\}$ of Example 5.40. Suppose that there were such a machine B. Then, among the strings accepted by B, would be every string of the form wa^ω, where w is some finite string in $\{a, b\}^*$. This must be true since all such strings contain only a finite number of b's. Remove from B any states that are not reachable from the start state. Now consider any remaining state q in B. Since q is reachable from the start state, there must exist at least one finite string that drives B from the start state to q. Call that string w. Then, as we

just observed, $w\mathsf{a}^\omega$ is in L and so must be accepted by B. In order for B to accept it, there must be at least one accepting state q_a that occurs infinitely often in the computation of B on $w\mathsf{a}^\omega$. That accepting state must be reachable from q (the state of B when just w has been read) by some finite number, which we'll call a_q, of a's (since B has only a finite of states). Compute a_q for every state q in B. Let m be the maximum of the a_q values.

We can now show that B accepts the string $(\mathsf{ba}^m)^\omega$, which is not in L. Since B is deterministic, its transition function is defined on all (state, input) pairs, so it must run forever on all strings including $(\mathsf{ba}^m)^\omega$. From the last paragraph we know that, from any state, there is a string of m or fewer a's that can drive B to an accepting state. So, in particular, after each time it reads a b, followed by a sequence of a's, B must reach some accepting state within m a's. But B has only a finite number of accepting states. So, on input $(\mathsf{ba}^m)^\omega$, B reaches some accepting state an infinite number of times and it accepts.

There is a natural relationship between the languages of infinite strings accepted by Büchi automata and the regular languages (i.e., the languages of finite strings accepted by FSMs). To describe this relationship requires an understanding of the closure properties of the regular languages that we will present in Section 8.3, as well as some of the decision procedures for regular languages that we will present in Chapter 9. It would be helpful to read those sections before continuing to read this discussion of Büchi automata.

Any Büchi-acceptable language can be described in terms of regular languages. To see how, observe that any Büchi automaton B can almost be viewed as an FSM, if we simply consider input strings of finite length. The only reason that that can't quite be done is that Büchi automata may have multiple start states. So, from any Büchi automaton B, we can build what we'll call the ***mirror FSM*** M to B as follows: Let $M = B$ except that, if B has more than one start state, then, in M, create a new start state that has an ε-transition to each of the start states of B. Notice that the set of finite length strings that can drive B from a start state to some state q is identical to the set of finite length strings that can drive M from its start state to state q.

Now consider any Büchi automaton B and any string w that B accepts. Since w is accepted, there is some accepting state in B that is visited an infinite number of times while B processes w. Call that state q. (There may be more than one such state. Pick one.) Then we can divide w into two parts, x and y. The first part, x, has finite length and it drives B from a start state to q for the first time. The second part, y, has infinite length and it simply pushes B through one loop after another, each of which starts and ends in q (although there may be more than one path that does this). The set of possible values for x is regular: It is exactly the set that can be accepted by the FSM M that mirrors B, if we let q be M's only accepting state. Call a path from q back to itself ***minimal*** iff it does not pass through q. Then we also notice that the set of strings that can force B through such a minimal path is also regular. It is the set accepted by the FSM M that mirrors B, if we let q be both M's start state and its only accepting state. These observations lead to the following theorem:

THEOREM 5.8 Büchi-Acceptable and Regular Languages

Theorem: L is a Büchi-acceptable language iff it is the finite union of sets each of which is of the form XY^{ω}, where each X and Y is a regular language.

Proof: Given any Büchi automaton $B = (K, \Sigma, \Delta, S, A)$, let $W_{q_0q_1}$ be the set of all strings that drive B from state q_0 to state q_1. Then, by the definition of what it means for a Büchi automaton to accept a string, we have:

$$L(B) = \bigcup_{s \in S} \bigcup_{q \in A} W_{sq}(W_{qq})^{\omega}.$$

If L is a Büchi-acceptable language, then there is some Büchi automaton B that accepts it. So the only-if part of the claim is true since:

- S and A are both finite,
- For each s and q, W_{sq} is regular since it is the set of strings accepted by B's mirror FSM M with start state s and single accepting state q,
- $W_{qq} = Y^*$, where Y is the set of strings that can force B along a minimal path from q back to q,
- Y is regular since it is the set of strings accepted by B's mirror FSM M with q as its start state and its only accepting state, and
- The regular languages are closed under Kleene star so $W_{qq} = Y^*$ is also regular.

The if part follows from a set of properties of the Büchi-acceptable and regular languages that are described in Theorem 5.9.

THEOREM 5.9 Closure Properties of Büchi Automata

Theorem and Proof: The Büchi-acceptable languages (like the regular languages) are closed under:

- Concatenation with a regular language: If L_1 is a regular language and L_2 is a Büchi-acceptable language, then L_1L_2 is Büchi-acceptable. The proof is similar to the proof that the regular languages are closed under concatenation except that, since ε transitions are not allowed, the machines for the two languages must be "glued together" differently. If q is a state in the FSM that accepts L_1, and there is a transition from q, labeled c, to some accepting state, then add a transition from q, labeled c, to each start state of the Büchi automaton that accepts L_2.
- Union: If L_1 and L_2 are Büchi-acceptable, then $L_1 \cup L_2$ is also Büchi-acceptable. The proof is analogous to the proof that the regular languages are closed under union. Again, since ε transitions are not allowed, we must use a slightly different glue. The new machine we will build will have transitions directly

from a new start state to the states that the original machines can reach after reading one input character.

- Intersection: If L_1 and L_2 are Büchi-acceptable, then $L_1 \cap L_2$ is also Büchi-acceptable. The proof is by construction of a Büchi automaton that effectively runs a Büchi automaton for L_1 in parallel with one for L_2.
- Complement: If L is Büchi-acceptable, then $\neg L$ is also Büchi-acceptable. The proof of this claim is less obvious. It is given in [Thomas 1990].

Further, if L is a regular language, then L^ω is Büchi-acceptable. The proof is analogous to the proof that the regular languages are closed under Kleene star, but we must again use the modification that was used above in the proof of closure under concatenation.

Büchi automata are useful as models for computer systems whose properties we wish to reason about because a set of important questions can be answered about them. In particular, Büchi automata share with FSMs the existence of decision procedures for all of the properties described in the following theorem:

THEOREM 5.10 Decision Procedures for Büchi Automata

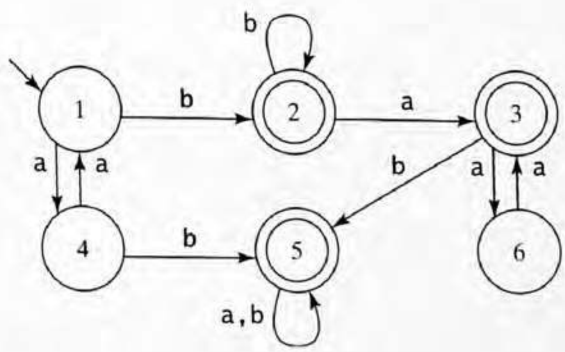
Theorem: There exist decision procedures for all of the following properties:

- Emptiness: Given a Büchi automaton B, is $L(B)$ empty?
- Nonemptiness: Given a Büchi automaton B, is $L(B)$ nonempty?
- Inclusion: Given two Büchi automata B_1 and B_2, is $L(B_1) \subseteq L(B_2)$?
- Equivalence: Given two Büchi automata B_1 and B_2, is $L(B_1) = L(B_2)$?

Proof: The proof of each of these claims can be found in [Thomas 1990].

Exercises

1. Give a clear English description of the language accepted by the following DFSM:



2. Show a DFSM to accept each of the following languages:

a. $\{w \in \{\mathtt{a}, \mathtt{b}\}^* :$ every a in w is immediately preceded and followed by b$\}$.

b. $\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w$ does not end in ba$\}$.

c. $\{w \in \{0, 1\}^* : w$ corresponds to the binary encoding, without leading 0's, of natural numbers that are evenly divisible by 4$\}$.

d. $\{w \in \{0, 1\}^* : w$ corresponds to the binary encoding, without leading 0's, of natural numbers that are powers of 4$\}$.

e. $\{w \in \{0\text{-}9\}^* : w$ corresponds to the decimal encoding, without leading 0's, of an odd natural number$\}$.

f. $\{w \in \{0, 1\}^* : w$ has 001 as a substring$\}$.

g. $\{w \in \{0, 1\}^* : w$ does not have 001 as a substring$\}$.

h. $\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w$ has bbab as a substring$\}$.

i. $\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w$ has neither ab nor bb as a substring$\}$.

j. $\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w$ has both aa and bb as a substrings$\}$.

k. $\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w$ contains at least two b's that are not immediately followed by an a$\}$.

l. $\{w \in \{0, 1\}^* : w$ has no more than one pair of consecutive 0's and no more than one pair of consecutive 1's$\}$.

m. $\{w \in \{0, 1\}^* :$ none of the prefixes of w ends in 0$\}$.

n. $\{w \in \{\mathtt{a}, \mathtt{b}\}^* : (\#_{\mathtt{a}}(w) + 2 \cdot \#_{\mathtt{b}}(w)) \equiv_5 0\}$. ($\#_{\mathtt{a}}(w)$ is the number of a's in w).

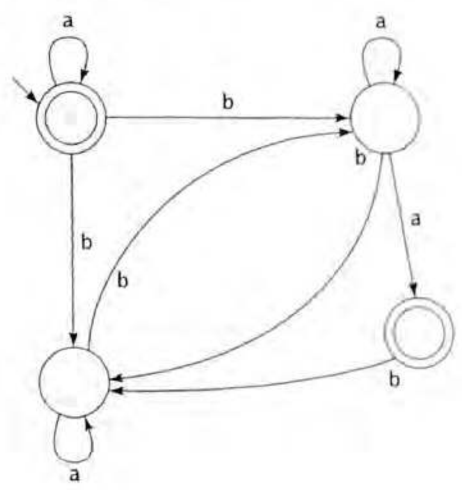
3. Consider the children's game Rock, Paper, Scissors ◻. We'll say that the first player to win two rounds wins the game. Call the two players A and B.

a. Define an alphabet Σ and describe a technique for encoding Rock, Paper, Scissors games as strings over Σ. (*Hint*: Each symbol in Σ should correspond to an ordered pair that describes the simultaneous actions of A and B.)

b. Let L_{RPS} be the language of Rock, Paper, Scissors games, encoded as strings as described in part (a), that correspond to wins for player A. Show a DFSM that accepts L_{RPS}.

4. If M is a DFSM and $\varepsilon \in L(M)$, what simple property must be true of M?

5. Consider the following NDFSM M:



For each of the following strings w, determine whether $w \in L(M)$:

a. aabbba.

b. bab.

c. baba.

6. Show a possibly nondeterministic FSM to accept each of the following languages:

a. $\{a^n ba^m : n, m \geq 0, n \equiv_3 m\}$.

b. $\{w \in \{a, b\}^* : w$ contains at least one instance of aaba, bbb or ababa$\}$.

c. $\{w \in \{0\text{-}9\}^* : w$ corresponds to the decimal encoding of a natural number whose encoding contains, as a substring, the encoding of a natural number that is divisible by 3$\}$.

d. $\{w \in \{0, 1\}^* : w$ contains both 101 and 010 as substrings$\}$.

e. $\{w \in \{0, 1\}^* : w$ corresponds to the binary encoding of a positive integer that is divisible by 16 or is odd$\}$.

f. $\{w \in \{a, b, c, d, e\}^* : |w| \geq 2$ and w begins and ends with the same symbol$\}$.

7. Show an FSM (deterministic or nondeterministic) that accepts $L = \{w \in \{a, b, c\}^* : w$ contains at least one substring that consists of three identical symbols in a row$\}$. For example:

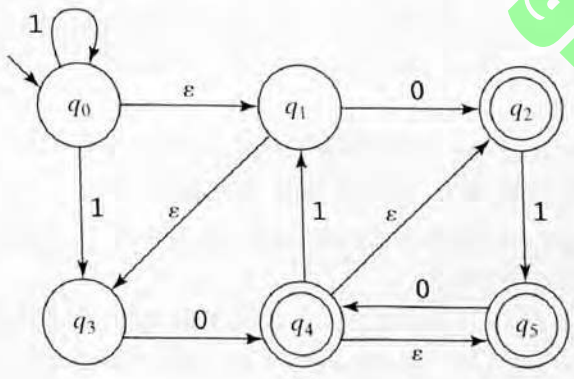
- The following strings are in L: aabbb, baacccbbb.
- The following strings are not in L: ε, aba, abababab, abcbcab.

8. Show a DFSM to accept each of the following languages. The point of this exercise is to see how much harder it is to build a DFSM for tasks like these than it is to build an NDFSM. So do not simply build an NDFSM and then convert it. But do, after you build a DFSM, build an equivalent NDFSM.

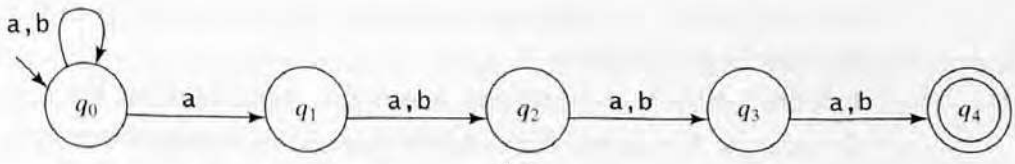
a. $\{w \in \{a, b\}^* :$ the fourth from the last character is a$\}$.

b. $\{w \in \{a, b\}^* : \exists x, y \in \{a, b\}^* : ((w = x \text{ abbaa } y) \vee (w = x \text{ baba } y))\}$.

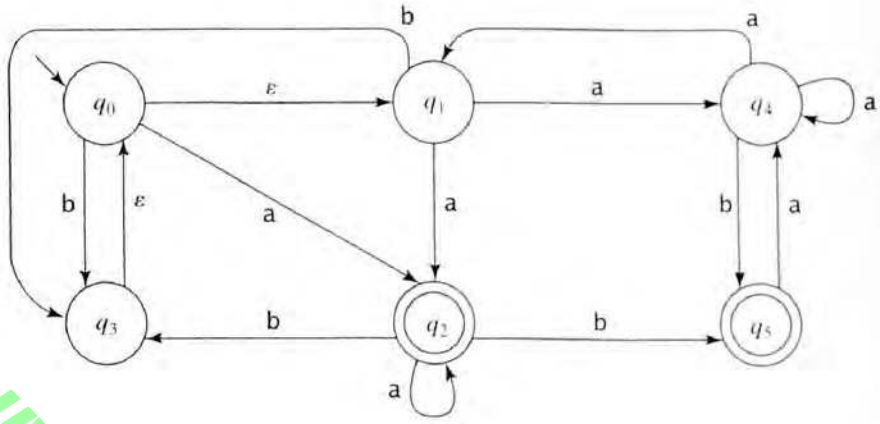
9. For each of the following NDFSMs, use *ndfsmtodfsm* to construct an equivalent DFSM. Begin by showing the value of $eps(q)$ for each state q:



(a)

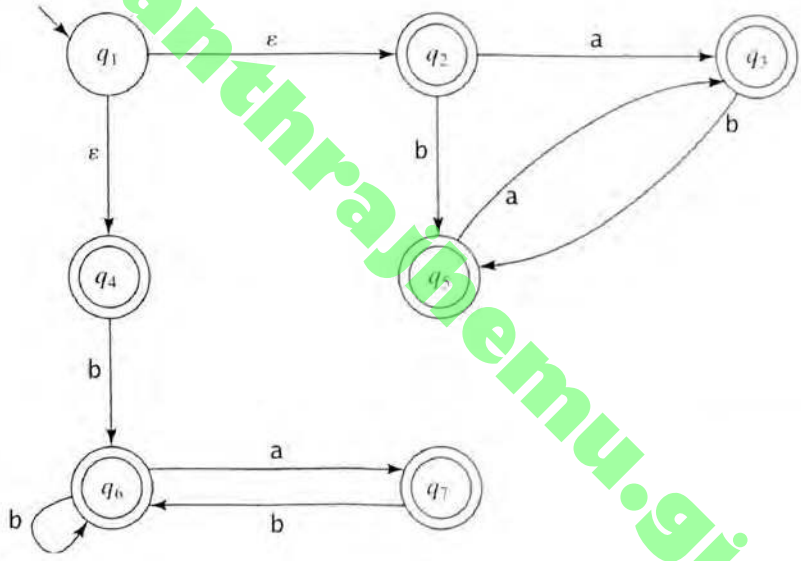


(b)



(c)

10. Let M be the following NDFSM. Construct (using *ndfsmtodfsm*), a DFSM that accepts $\neg L(M)$.



11. For each of the following languages L:

(i) Describe the equivalence classes of $\approx_L$.

(ii) If the number of equivalence classes of $\approx_L$ is finite, construct the minimal DFSM that accepts L.

a. $\{w \in \{0,1\}^* :$ every 0 in w is immediately followed by the string 11$\}$.

b. $\{w \in \{0,1\}^* : w$ has either an odd number of 1's and an odd number of 0's or it has an even number of 1's and an even number of 0's$\}$.

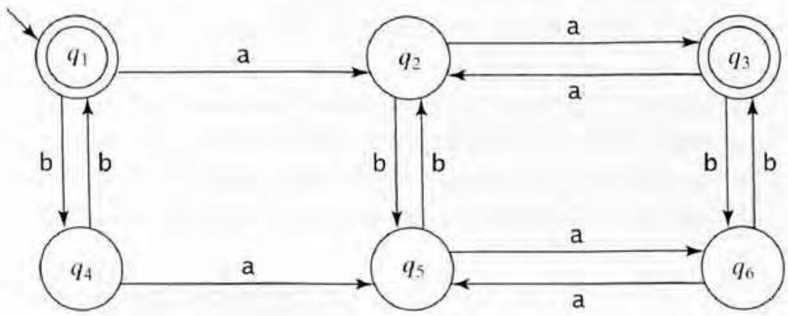
c. $\{w \in \{a,b\}^* : w$ contains at least one occurrence of the string aababa$\}$.

d. $\{ww^R : w \in \{a,b\}^*\}$.

e. $\{w \in \{a,b\}^* : w$ contains at least one a and ends in at least two b's$\}$.

f. $\{w \in \{0,1\}^* :$ there is no occurrence of the substring 000 in $w\}$.

12. Let M be the following DFSM. Use *minDFSM* to minimize M.



13. Construct a deterministic finite state transducer with input alphabet {a, b} for each of the following tasks:

a. On input w, produce 1^n, where $n = \#_a(w)$.

b. On input w, produce 1^n, where $n = \#_a(w)/2$.

c. On input w, produce 1^n, where n is the number of occurrences of the substring aba in w.

14. Construct a deterministic finite state transducer that could serve as the controller for an elevator. Clearly describe the input and output alphabets, as well as the states and the transitions between them.

15. Consider the problem of counting the number of words in a text file that may contain letters plus any of the following non-letter characters:

<blank><linefeed><end-of-file>, . ; : ?!

Define a word to be a string of letters that is preceded by either the beginning of the file or some non-letter character and that is followed by some non-letter character. For example, there are 11 words in the following text:

The *<blank><blank>* cat *<blank><linefeed>*
saw *<blank>* the *<blank><blank><blank>* rat *<linefeed>*
<blank> with
<linefeed> a*<blank>* hat *<linefeed>*
on *<blank>* the *<blank><blank>* mat *<end-of-file>*

Describe a very simple finite-state transducer that reads the characters in the file one at a time and solves the word-counting problem. Assume that there exists an output symbol with the property that, every time it is generated, an external counter gets incremented.

16. Real traffic light controllers are more complex than the one that we drew in Example 5.29.

a. Consider an intersection of two roads controlled by a set of four lights (one in each direction). Don't worry about allowing for a special left-turn signal. Design a controller for this four-light system.

b. As an emergency vehicle approaches an intersection, it should be able to send a signal that will cause the light in its direction to turn green and the light in the cross direction to turn yellow and then red. Modify your design to allow this.

17. Real bar code systems are more complex than the one that we sketched in Example 5.31. They must be able to encode all ten digits, for example. There are several industry-standard formats for bar codes, including the common UPC code 💻 found on nearly everything we buy. Describe a finite state transducer that reads the bars and outputs the corresponding decimal number.

18. Extend the description of the Soundex FSM that was started in Example 5.33 so that it can assign a code to the name Pfifer. Remember that you must take into account the fact that every Soundex code is made up of exactly four characters.

19. Consider the weather/passport HMM of Example 5.37. Trace the execution of the Viterbi and forward algorithms to answer the following questions:

a. Suppose that the report ###L is received from Athens. What was the most likely weather during the time of the report?

b. Is it more likely that ###L came from London or from Athens?

20. Construct a Büchi automaton to accept each of the following languages of infinite length strings:

a. $\{w \in \{a, b, c\}^\omega$: after any occurrence of an a there is eventually an occurrence of a b$\}$.

b. $\{w \in \{a, b, c\}^\omega$: between any two consecutive a's there is an odd number of b's$\}$.

c. $\{w \in \{a, b, c\}^\omega$: there never comes a time after which no b's occur$\}$.

21. In H.2, we describe the use of statecharts as a tool for building complex systems. A statechart is a hierarchically structured transition network model. Statecharts aren't the only tools that exploit this idea. Another is Simulink® 💻, which is one component of the larger programming environment MATLAB® 💻. Use Simulink to build an FSM simulator.

22. In I.1.2, we describe the Alternating Bit protocol for handling message transmission in a network. Use the FSM that describes the sender to answer the question, "Is there any upper bound on the number of times a message may be retransmitted?"

23. In J.1, we show an FSM model of a simple intrusion detection device that could be part of a building security system. Extend the model to allow the system to have two zones that can be armed and disarmed independently of each other.